Unlocking the Potential of MAPPO with Asynchronous Optimization

Wei Fu[1,2], Chao Yu[2], Yunfei Li[1,3], and Yi Wu[⋆1,3]

[1] Shanghai Qizhi Institute
[2] Department of Electronics Engineering, Tsinghua University
[3] Institute for Interdisciplinary Information Sciences, Tsinghua University
fuw17@tsinghua.org.cn

Abstract. It almost reaches a consensus that off-policy algorithms dominated research benchmarks of multi-agent reinforcement learning, while recent work [34] demonstrates that on-policy MARL algorithm, Multi-Agent Proximal Policy Optimization (MAPPO), can also attain comparable performance. In this paper, we propose a training framework based on MAPPO, named *async-MAPPO*, which supports scalable asynchronous training. We further re-examine async-MAPPO in StarCraftII micromanagement domain and obtain state-of-the-art performances on several hard and super-hard maps. Finally, we analyze three experimental phenomena and provide hypotheses behind the performance improvement of async-MAPPO.

Keywords: Multi-agent reinforcement learning · Asynchronous training · Distributed computing.

1 Introduction

Recent research progress of multi-agent systems, such as AlphaStar [29], OpenAI Five [20] and hide-and-seek agents [1], indicate the general effectiveness and promising prospect of Multi-Agent Reinforcement Learning (MARL) in building intelligent agents that can behave cooperatively or competitively. It has been a growing trend to design and improve MARL algorithms [6, 12, 23, 32], and to apply MARL to diverse applications, such as full Multiplayer Online Battle Arena (MOBA) game [33], autonomous driving [26] and social dilemmas [14].

Off-policy and value decomposition-based MARL algorithms have been preferred by researchers in recent years [27, 30, 31] since they are thought to be more sample efficient than on-policy ones. However, a recent work [34] demonstrates that with minimal hyperparameter tuning and restricted representation power, Multi-Agent Proximal Policy Optimization (MAPPO), i.e., PPO with centralized value function and decentralized policy, can match or surpass the performance of strong off-policy baselines on 3 categories of cooperative multi-agent benchmarks: Multi-agent Particle Environments (MPE) [17, 18], StarCraftII Multi-Agent Challenge (SMAC) [24] and Hanabi challenge [2].

⋆ Corresponding author

The success of MAPPO indicates that on-policy multi-agent actor-critic algorithms are surprisingly effective and have great potential for MARL applications.

Even though the performance of MAPPO is impressive, there are some deficiencies in the original implementation[4] and experiments.

- **The original implementation of MAPPO is in a serial manner**. The agent sequentially collects data through environment interaction (referred to *rollout* stage) and then uses collected data for optimization (referred to *learning* stage). Rollout and learning need to wait for the completion of the other to enter the next round. If the data to be generated and consumed is vast, both rollout and learning require a longer time to complete and wait, which doubly increases training time and makes large-batch training unendurable.
- **MAPPO still requires carefully selected network architectures and moderate hyperparameter tuning on several maps.** To be more specific, Convolutional Neural Network (CNN) with frame-stacking is used on SMAC maps *3s_vs_4z* and *3s_vs_5z*, while the network architecture on other maps is either Multi-Layer Perceptron (MLP) or MLP with GRU [3]. In terms of hyperparameter tuning, uniquely different mini-batch numbers and initialization gain of the last action layer are utilized on the *MMM2* map.

Can we ameliorate the hyperparameter sensitiveness of MAPPO and accelerate the training procedure simultaneously? Authors of [1] found that batch size plays an imperative role in hide-and-seek training: larger batch size leads to faster convergence and even better sample efficiency, while training with a small batch may never converge. Besides, the aforementioned large-scale MARL applications, such as OpenAI Five and hide-and-seek agent, conformably utilize distributed RL system for fast data collection and large batch for stable training. This acquiescent agreement makes us wonder whether asynchronous training with a large batch is the key element to enhance original MAPPO and to unlock its full potential.

In this paper, we propose ***async-MAPPO***, a MARL framework that integrates MAPPO and the refined SEED (Scalable, EfficiEnt Deep-RL system [4]) architecture. We re-examine async-MAPPO on several hard and super-hard maps in SMAC domain and find that MAPPO algorithm can attain better final performance when training with asynchronous optimization and large batch. Notably, the final performance surpasses all the results reported in [34] and establishes a new state-of-the-art (SOTA). Finally, three experimental phenomena are proposed and discussed, through which we provide enlightenment about the reason behind the performance boost of async-MAPPO.

The contributions of this paper are summarized as follows:

1. We propose *async-MAPPO*, a scalable asynchronous training framework which integrates a refined SEED architecture with MAPPO.
2. We show that async-MAPPO can achieve SOTA performance on several hard and super-hard maps in SMAC domain with significantly faster training speed by tuning only one hyperparameter.

[4] https://github.com/marlbenchmark/on-policy

3. We formulate hypotheses about the effectiveness of large-batch training based on the empirical results.

2 Related Works

Modern multi-agent deep reinforcement learning algorithms mostly follow the paradigm of Centralized-Training-with-Decentralized-Execution (CTDE [7, 8]). Under CTDE paradigm, each agent independently behaves using its policy, while policies are jointly trained given global environment information. Popular MARL algorithms adopting CTDE paradigm can be roughly divided into off-policy and on-policy branches. Off-policy branch includes multi-agent actor-critic algorithms, such as MADDPG [17] and COMA [6], and value-decomposition based algorithms, such as QMIX [23], ROMA [31] and RODE [32]. On-policy branch typically includes MAPPO [34]. While off-policy MARL algorithms attract more attention and are nearly exhaustively developed, on-policy MARL algorithms are rarely studied. However, surprisingly, they are empirically promising and worth further improving [34].

Large-scale MARL projects are always supported by an efficient RL system, such as Rapid framework used in OpenAI Five [20]. RL system design was an early research focus [16]. To address the problem of iterative waiting in serial implementation, IMPALA [5] adopts a scalable actor-learner architecture, where each actor is placed on a CPU core and manages the whole rollout procedure independently, and the learner collects data generated by actors and optimizes model parameters on GPUs. To further utilize GPU/TPU resources in a cluster, SEED [4] decomposes the rollout procedure of IMPALA into a client-server mode. Every client steps through local environments on CPU, and issues action requests to the inference server, while inference server batches observations received from clients and provides action inference on a GPU or TPU. Providing the scalability, efficiency, and resource utilization requirement, SEED is currently one of the best architecture backbones in building a distributed RL system, whose modification has been applied in recent works [33].

Performance improvement with asynchronous training was reported previously in the domain of single-agent RL. Asynchronous training can both facilitate exploration and allow wider hyperparameter sweeping by setting different hyperparameters in every worker node [11]. When integrated with recurrent neural networks and distributed training, deep Q network can achieve state-of-the-art performance on most Atari games and outperform normal Q-learning-based algorithm by a large margin [13]. Considering the above results in single-agent benchmarks and the successful applications of PPO in large-scale multi-agent projects, we could expect a performance boost of MAPPO when combining it with an efficient asynchronous RL system in multi-agent benchmarks.

3 Preliminaries

We consider a decentralized partially observable Markov decision process (Dec-POMDP) [19] with shared rewards among agents defined by the tuple $G =$

Algorithm 1 Trainer Process of MAPPO with Asynchronous Optimization

Input: Parameters θ_0, ψ_0, replay buffer $\mathcal{D}$, parameter queue $\mathcal{Q}$, learning rate α

Send θ_0 and ψ_0 into $\mathcal{Q}$
Launch rollout processes
for update iteration $i = 1, 2, \ldots$ **do**
    wait until there's enough data in $\mathcal{D}$ for optimization
    fetch data batch from $\mathcal{D}$
    compute $A^{\pi}_{\psi_i}$ using GAE [25] with PopArt [10] denormalization
    compute V^{π}_{target} based on A^{π}_{ψ}
    use V^{π}_{target} to update PopArt parameters
    compute loss $-J_{\pi}(\theta_i)$ and $-J_{V^{\pi}}(\psi_i)$ according to Equation (1) and (2)
    $\theta_{i+1} \leftarrow \theta i + \alpha \nabla J_{\pi}(\theta)|_{\theta=\theta_i}$
    $\psi_{i+1} \leftarrow \psi_i + \alpha \nabla J_{V^{\pi}}(\psi_i)|_{\psi=\psi_i}$
    Send θ_{i+1} and ψ_{i+1} into $\mathcal{Q}$
    $i \leftarrow i + 1$
return policy π_{θ}

$\langle I, S, A, P, R, \Omega, O, n, \gamma \rangle$. I defines the set of agents and n is the number of agents. γ is the discount factor. S is the support set of true state in the environment. At each timestep, agent i receives an observation o_i drawn from the observation function O, i.e., $o_i = O(s, i) \in \Omega$. After receiving an observation, agent i infers an available action $a_i \in A$ to execute. A shared reward $R(s, \boldsymbol{a})$ is received by all agents once a joint action $\boldsymbol{a} \in A^n$ is formulated and a transition is triggered according to the transition function $P(s'|s, \boldsymbol{a})$. $\tau_i \in T \equiv (S \times A^n)^t$ denotes the trajectory of agent i of the elapsed t timesteps in one episode.

MAPPO follows the CTDE paradigm, where each agent learns a shared policy $\pi_{\theta}(\cdot|\tau_i) : (\Omega \times A)^* \to [0, 1]$ parameterized by θ conditioned on local history observation, and a centralized value function of current policy π, $V^{\pi}_{\psi}(\boldsymbol{\tau}) : S^* \to \mathbb{R}$ parameterized by ψ conditioned on global history states. Here X^* means the Cartesian product of set X in an arbitrary number of timesteps. Note that trajectory $\boldsymbol{\tau}$ together with observation function O contains the history information of both local observation and global state. Hence we use $\boldsymbol{\tau}$ to denote the input of both policy and value function, while the actual input may not be the same.

4 Async-MAPPO

4.1 MAPPO Algorithm

Similar as single-agent PPO, MAPPO simultaneously learns a shared policy $\pi_{\theta}(\cdot|\tau_i)$ and a centralized value function $V_{\psi}(\boldsymbol{\tau}) = \mathbb{E}_{(s_t, \boldsymbol{a}_t) \sim T} \left[\sum_{t=0}^{\infty} R(s_t, \boldsymbol{a}_t)\right]$ by optimizing the following objective:

$$J_{\pi}(\theta) = \sum_{i=1}^{n} \mathbb{E}_{(\tau_t^i, a_t^i) \sim G} \left[\min \left(\text{clip} \left(\frac{\pi_{\theta}(a_t^i|\tau_t^i)}{\pi_{\theta_{old}}(a_t^i|\tau_t^i)} \right) A^{\pi}_{\psi}(\tau_t^i, a_t^i), \frac{\pi_{\theta}(a_t^i|\tau_t^i)}{\pi_{\theta_{old}}(a_t^i|\tau_t^i)} A^{\pi}_{\psi}(\tau_t^i, a_t^i) \right) \right] \quad (1)$$

$$J_{V^{\pi}}(\psi) = -\sum_{i=1}^{n} \mathbb{E}_{\boldsymbol{\tau} \sim G} \left[V^{\pi}_{\text{target}}(\boldsymbol{\tau}) - V^{\pi}_{\psi}(\boldsymbol{\tau}) \right]^2 \quad (2)$$

Algorithm 2 Client

Input: total $k * m$ environments in m groups, remote reference of inference server $\mathcal{S}$

for group $g = 1, \ldots, m$ **do**
  for environment e in g **do**
    o, o_{share}, $a_{\text{avail}} = e$.reset()
  Batch o, o_{share} and a_{avail} into vector
  invoke RPC $\mathcal{S}$.select_action($\boldsymbol{o}$, $\boldsymbol{o}_{\text{share}}$, $\boldsymbol{a}_{\text{avail}}$)
while True **do**
  for group $g = 1, \ldots, m$ **do**
    wait for action response $\boldsymbol{a}$ from $\mathcal{S}$
    for for environment e in g **do**
      o, o_{share}, a_{avail}, r, $d = e$.step(a)
    Batch o, o_{share}, a_{avail}, r, d into vector
    invoke RPC $\mathcal{S}$.select_action($\boldsymbol{o}$, $\boldsymbol{o}_{\text{share}}$, $\boldsymbol{a}_{\text{avail}}$, $\boldsymbol{r}$, $\boldsymbol{d}$)

In the above equations, $A^{\pi}_{\psi}(\tau^i_t, a^i_t)$ denotes the advantage function [28] and $V^{\pi}_{\text{target}}(\boldsymbol{\tau})$ denotes the target value computed by Generalized Advantage Estimation (GAE [25]). Pseudocode and algorithmic details of trainer process can be found in Algorithm 1.

4.2 Refined SEED Architecture

As mentioned in 2, SEED [4] is a high-performance distributed RL architecture built upon the gRPC package, which decomposes the rollout stage into client and server calls to fully utilize GPU/TPU resources in a cluster. Although it is scalable and cost-effective, an obvious flaw is that, in the aspect of a remote environment client, after issuing a request to the server, it must wait until the completion of inference and data transportation to continue the next step of environment simulation. This indicates that CPU resources may not be fully utilized due to the idle time of clients. To address this issue and remove the potential bottleneck, similar to the system designed by [22], we propose to store a vector of environments in each client, split the environments into multiple parts, e.g., 2 parts, and alternate stepping across them, which is referred to *Multiple-Buffered Sampling*.

To be more specific, a client possesses m environment splits, and an environment split is composed of k environments. The simulation of each environment in the same split is executed sequentially in a *for* loop. After the request initiation of any environment split, the client will keep stepping through the other environment splits instead of waiting for the response. For example, while the second environment split is stepped through, the actions of the first split are computed on the inference server. Hence, clients can attain full CPU utilization once k is correctly chosen such that the time of inference and data transportation can be overlapped by simulation of other environment splits. Graphical illustration can be found in Figure 1.

Algorithm 3 Inference Server Invocation

Input: replay buffer $\mathcal{D}$, parameter queue $\mathcal{Q}$, inference model π and V, inference batch size B, observation $\boldsymbol{o}$, centralized observation $\boldsymbol{o}_{\text{share}}$, available action $\boldsymbol{a}_{\text{avail}}$, reward $\boldsymbol{r}$, termination indicator $\boldsymbol{d}$

if there's new parameter θ', ψ' in $\mathcal{Q}$ **then**
  $\theta \leftarrow \theta'$, $\psi \leftarrow \psi'$
Store $\boldsymbol{o}$, $\boldsymbol{o}_{\text{share}}$, $\boldsymbol{a}_{\text{avail}}$, $\boldsymbol{r}$, $\boldsymbol{d}$ into $\mathcal{D}$
batching count $\leftarrow$ batching count $+ 1$
Set callback object $\boldsymbol{a}$ to be the slice of $\boldsymbol{a}^{\text{batch}}$
if batching count $\geq B$ **then**
  Get batched $\boldsymbol{o}^{\text{batch}}$, $\boldsymbol{o}_{\text{share}}^{\text{batch}}$, $\boldsymbol{a}_{\text{avail}}^{\text{batch}}$ from $\mathcal{D}$
  Get hidden state $\boldsymbol{h}^{\text{batch}}$ from $\mathcal{D}$
  $\boldsymbol{a}^{\text{batch}}$, $\hat{\boldsymbol{h}}^{\text{batch}}$ = model.inference($\boldsymbol{o}^{\text{batch}}$, $\boldsymbol{o}_{\text{share}}^{\text{batch}}$, $\boldsymbol{a}_{\text{avail}}^{\text{batch}}$,$\boldsymbol{h}^{\text{batch}}$)
  Store $\boldsymbol{a}^{\text{batch}}$, $\hat{\boldsymbol{h}}^{\text{batch}}$ into $\mathcal{D}$
  batching count $\leftarrow 0$
  Trigger callback function on $\boldsymbol{a}$
return $\boldsymbol{a}$

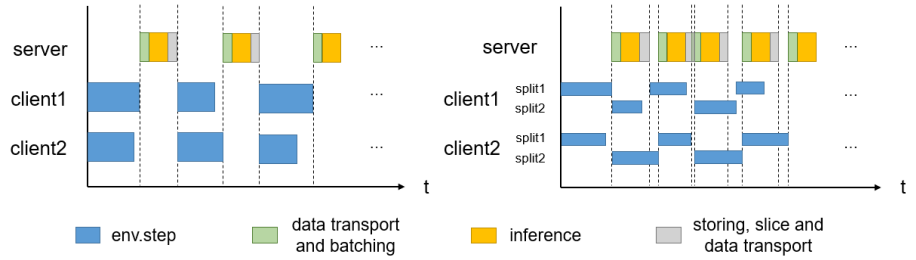

Fig. 1. Graphical illustration of *Multiple-Buffered Sampling*. (Left) original SEED sampling architecture. Servers wait for client requests, batch data, conduct inference, store inference outputs, and then send action slices back to clients. Clients wait for the response from servers after issuing a request. Idle time of client CPU is non-negligible. (Right) *Multiple-Buffered Sampling*. Clients alternate among $m = 2$ environment splits and step through them. Overlapping between client CPU and server GPU makes full use of computation resources.

4.3 Implementation Details

Communication between clients and the inference server is supported by *PyTorch* [21] *torch.distributed* [15] package. Remote reference of inference server is kept by several clients, through which clients can invoke remote procedure calls (RPC) to request actions from the server. After receiving a request from any client, the server stores observations returned in the previous environment step into the replay buffer and returns a *torch.futures.Future* object immediately. Once a certain number of clients invoke RPCs, the server will fetch a data batch from the buffer, conduct inference, and save data returned by inference back into the buffer. At this point, slicing callback function chained in *torch.futures.Future* object is triggered. Finally, clients receive the corresponding slice of the action batch as a response and start a new round of environment steps. Detailed algo-

rithm description of client and server is illustrated in Algorithm 2 and Algorithm 3 respectively.

Replay buffer is based on *NumPy* library [9] and shared memory in Python *multiprocessing* package, i.e., every data segmentation is a *NumPy* array in shared memory, such that the learner can benefit from zero-copy communication when requesting for a data batch stored by inference servers. During optimization, the learner converts a data batch in replay buffer into *PyTorch Tensor* [21], loads it into GPU memory, and optimizes model parameters. After every full update iteration (several PPO update epochs), the updated parameter is pushed into a queue, through which servers synchronize the local model with the latest one to ensure that the rollout procedure is sufficiently on-policy.

5 Experiment

In this section, we examine both system-level and algorithm-level performance of async-MAPPO. In Section 5.1, experiments concerning the effect of *Multiple-Buffered Sampling* and scalability are conducted, which show that refined SEED architecture provides higher system throughput than the original one and still scales well. Algorithm performance is measured in Section 5.2 on selected SMAC maps. These results meet our expectation of performance enhancement with asynchronous optimization and a large batch. We analyze the reason behind algorithm performance gain and formulate two hypotheses derived from experimental observations in Section 5.3.

5.1 System-level Evaluation

We examine the scalability of the refined SEED architecture and the influence of refinement in terms of system throughput, i.e., collected environment Frames Per Second (FPS), on Hanabi learning environment [2]. Table 1 demonstrates the numerical results tested on a single machine and a cluster. In both systems, the first 3 GPUs are used for optimization and the last one is used for rollout inference, where four inference servers are initialized. Detailed hardware description of System #1 and System #2 can be found in the caption of Table 1.

On the one hand, from the first three rows of system #2, we can see that refined SEED architecture can be employed to a cluster with near-linear scaling, i.e., system throughput improves linearly as the number of actors increases, which shows promising scalability. On the other hand, by setting environment from single split into double splits, system throughput is improved in both local and distributed settings (first two rows of system #1, last two rows of system #2), which verifies the necessity of *Multiple-Buffered Sampling* in SEED architecture. If environment splits are further increased (last row of system #1), system throughput will be hurt because there exists a group of environments that are neither stepped through in clients nor waiting for action response from servers. Empirically, splitting environments into two groups is the best practice.

5.2 Algorithm-level Evaluation

Algorithmic Details. Experiments in this section are conducted on System #1 described in the previous section. Only 1 GPU is utilized for both rollout

Table 1. System throughput measurement results and corresponding experiment configuration. System # 1 is a laboratory-level server machine with one physical CPU of 64 cores, 128 GB memory, and 4 NVIDIA 2080Ti GPUs. System #2 is an Ali-Cloud cluster, whose head node is a GPU machine with 48 cores and 4 NVIDIA V100 GPUs and worker nodes are homogeneous CPU machines with 104 virtual cores. Rounded average FPS of 3 independent runs across 30 seconds is presented. The refinement of SEED architecture improves system throughput, with which SEED architecture still scales well.

System	#Actors	#Envs per Actor	#Env Splits	FPS
#1	48	80	1	13.5k
			2	17.5k
			3	16.3k
#2	32	128	2	11.1k
	64			22.0k
	128			44.5k
			1	30.6k

inference and network optimization in consistent with [34]. The number of environment splits is fixed to 2, while the number of actors and environments in each actor varies across selected environments. For StarCraftII environment, the more agents are in the map, the lower FPS and the fewer total environments a single machine can support. The same setting on maps with fewer agents (e.g. *3s_vs_5z*), which initializes a relatively large number of environments, can not be adopted on maps with plenty of agents (e.g. *27m_vs_30m*), otherwise, the game will cause a memory overflow problem. Hence, we report the batch size magnification and FPS acceleration factor instead of specific system configuration and actual FPS. We believe this substitution will make the conclusion more clear.

Two separate networks with the same recurrent structure as [34] are maintained for policy π_θ and value function V_ψ respectively. Hyperparameters, including hidden size, learning rate, and so on, are mostly the same as [34] except for reuse times of each data batch, which is presented in Table 2 in details. We follow the suggestions proposed by [34] and include all the recommended tricks in async-MAPPO, including agent-specific global state, training data usage, value normalization (PopArt) [10], action masking, and death masking, since they are all found to be critical to MAPPO's practical performance. We directly modify the codebase released with [34] for maximum consistency. If not specified, all evaluation procedure is the same as that reported in [34]. To better distinguish original MAPPO implementation from async-MAPPO, we refer to it as *serial-MAPPO* in the remaining part of this section.

StarCraftII Multi-Agent Challenge. Because all the features and tricks of serial-MAPPO are preserved, we expect *no performance drop* in async-MAPPO given that correctly tuned hyperparameters for asynchronous training. Therefore, we omit experiments on easy SMAC maps where the final performance of serial-MAPPO can not be further improved. SMAC maps that meet any of the following conditions are considered:

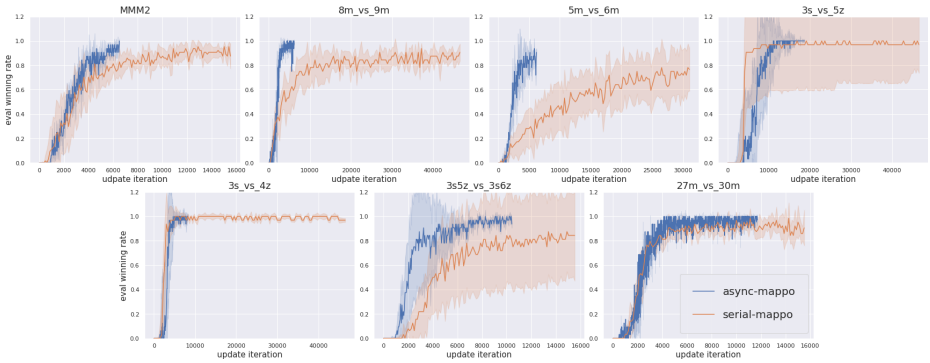

Fig. 2. Learning curves of serial-MAPPO and async-MAPPO on selected SMAC maps. Results of serial-MAPPO is taken from [34] in communication with authors. The Y-axis is the median evaluation winning rate throughout the training procedure. The shaded area indicates the standard deviation of different runs with different random seeds. The X-axis is the number of update iterations. (#update iteration = #total environment steps × sample reuse/batch size)

(1) Serial-MAPPO can not achieve SOTA performance. (*8m_vs_9m, 3s_vs_5z, 27m_vs_30m, 3s5z_vs_3s6z*);
(2) Even though serial-MAPPO achieves SOTA performance, there is plenty of room to improve. (*5m_vs_6m*);
(3) There is uniquely different hyperparameter or network architecture selection in serial-MAPPO. (*MMM2, 3s_vs_4z, 3s_vs_5z*)

Figure 2 demonstrates the median evaluation winning rate and the corresponding standard deviation across the training process. Final evaluation performance, sample reuse[5] times, FPS speedup and batch size magnification are presented in Table 2. When training with asynchronous optimization and large batch, the performance of async-MAPPO matches or exceeds serial-MAPPO, QMIX, and RODE examined in [34], either of which achieved SOTA results on these maps. Besides, training is significantly accelerated compared with serial-MAPPO for about 4∼5 times in terms of wall clock speed.

5.3 Discussion

We provide three experimental phenomena and corresponding analyses in async-MAPPO practice.

Async-MAPPO can endure more reuse times. Note that sample reuse reported in Table 2 is greater or equal to that in [34]. First, a large batch reduces the variance of episode returns and improves the precision of value function and advantage estimation. Second, with more diverse experiences in a data batch, the expectation in Equation 1, i.e., off-policy correction of advantage function, is also more accurate. Consequently, policy improvement direction is more accurate and off-policy correction is not prone to diverge, which is the reason why more reuse times can be applied on the same data batch in async-MAPPO.

[5] Referred to as *ppo_epoch* in [34].

Table 2. Final median evaluation winning rate (standard deviation) of async-MAPPO and serial-MAPPO. Speedup in FPS, magnification in batch size, and sample reuse are presented in the right columns. Runs of async-MAPPO include at least 3 random seeds. Evaluation winning rate of serial-MAPPO is directly taken from [34], while FPS of serial-MAPPO is re-measured in the same machine, system #1. FPS metric applied is the same as in Table 1.

Map	async-MAPPO	serial-MAPPO	FPS Speedup	Batchsize Magnification	Sample Reuse
3s_vs_4z	**100.0(1.5)**	**100.0(0.9)**	5.50x	7.5x	10
3s_vs_5z	**100.0(2.3)**	96.9(37.5)	5.84x	7.5x	15
5m_vs_6m	**90.6(3.1)**	75.0(18.2)	5.22x	7.5x	10
8m_vs_9m	**96.8(1.5)**	87.5(4.0)	4.66x	7.5x	10
27m_vs_30m	**98.4(3.3)**	93.8(2.4)	3.91x	2x	5
3s5z_vs_3s6z	**96.8(3.3)**	84.4(34.0)	5.20x	7.5x	10
MMM2	**96.8(1.1)**	90.6(2.8)	4.96x	6x	5

Async-MAPPO requires fewer update iterations. Even though sample reuse and total environment steps of async-MAPPO may be larger, the update iterations are fewer than serial-MAPPO, as shown in Figure 2. In the context of advantage normalization, a large batch reduces the variance of advantage estimation and increases the scale of normalized advantage (because the divisor is reduced) while maintaining high accuracy of mean advantage estimation. This means async-MAPPO may step further in the correct direction when using the same learning rate as serial-MAPPO, which explains fewer update iterations are required.

Async-MAPPO is less sensitive to network architectures and hyperparameters. To attain better results, on *MMM2*, serial-MAPPO splits one data batch into two mini-bathes to escape local optima, while async-MAPPO uses the whole data batch. For a similar reason, CNN with frame-stacking was used in maps *3s_vs_4z* and *3s_vs_5z*, while async-MAPPO uses a universal MLP+GRU network architecture. However, async-MAPPO achieves even better performance on these maps. We reckon the analysis in the previous two phenomena also applies in that accurate policy evaluation and fast policy improvement help escaping local optima, causing hyperparameter and network architecture less imperative.

6 Conclusion

In this work, we propose async-MAPPO, the integration of MAPPO algorithm, and refined SEED architecture. Async-MAPPO has promising system-level and algorithm-level performance and establishes a new SOTA result on selected hard and super-hard SMAC maps. We formulate hypotheses about the reason behind based on experimental phenomena. We conjecture that training with asynchronous optimization and a large batch is possibly a generally beneficial choice to use MAPPO. Systematic and theoretical verification of these hypotheses remains in future works.

References

1. Baker, B., Kanitscheider, I., Markov, T., Wu, Y., Powell, G., McGrew, B., Mordatch, I.: Emergent tool use from multi-agent autocurricula. arXiv preprint arXiv:1909.07528 (2019)
2. Bard, N., Foerster, J.N., Chandar, S., Burch, N., Lanctot, M., Song, H.F., Parisotto, E., Dumoulin, V., Moitra, S., Hughes, E., et al.: The hanabi challenge: A new frontier for ai research. Artificial Intelligence **280**, 103216 (2020)
3. Chung, J., Gulcehre, C., Cho, K., Bengio, Y.: Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555 (2014)
4. Espeholt, L., Marinier, R., Stanczyk, P., Wang, K., Michalski, M.: Seed rl: Scalable and efficient deep-rl with accelerated central inference. arXiv preprint arXiv:1910.06591 (2019)
5. Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., et al.: Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In: International Conference on Machine Learning. pp. 1407–1416. PMLR (2018)
6. Foerster, J., Farquhar, G., Afouras, T., Nardelli, N., Whiteson, S.: Counterfactual multi-agent policy gradients. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 32 (2018)
7. Foerster, J.N., Assael, Y.M., De Freitas, N., Whiteson, S.: Learning to communicate with deep multi-agent reinforcement learning. arXiv preprint arXiv:1605.06676 (2016)
8. Gupta, J.K., Egorov, M., Kochenderfer, M.: Cooperative multi-agent control using deep reinforcement learning. In: International Conference on Autonomous Agents and Multiagent Systems. pp. 66–83. Springer (2017)
9. Harris, C.R., Millman, K.J., van der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., et al.: Array programming with numpy. Nature **585**(7825), 357–362 (2020)
10. Hessel, M., Soyer, H., Espeholt, L., Czarnecki, W., Schmitt, S., van Hasselt, H.: Multi-task deep reinforcement learning with popart. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33, pp. 3796–3803 (2019)
11. Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., Van Hasselt, H., Silver, D.: Distributed prioritized experience replay. arXiv preprint arXiv:1803.00933 (2018)
12. Hu, H., Foerster, J.N.: Simplified action decoder for deep multi-agent reinforcement learning. arXiv preprint arXiv:1912.02288 (2019)
13. Kapturowski, S., Ostrovski, G., Quan, J., Munos, R., Dabney, W.: Recurrent experience replay in distributed reinforcement learning. In: International conference on learning representations (2018)
14. Leibo, J.Z., Zambaldi, V., Lanctot, M., Marecki, J., Graepel, T.: Multi-agent reinforcement learning in sequential social dilemmas. arXiv preprint arXiv:1702.03037 (2017)
15. Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al.: Pytorch distributed: Experiences on accelerating data parallel training. arXiv preprint arXiv:2006.15704 (2020)
16. Li, Y., Schuurmans, D.: Mapreduce for parallel reinforcement learning. In: European Workshop on Reinforcement Learning. pp. 309–320. Springer (2011)
17. Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P., Mordatch, I.: Multi-agent actor-critic for mixed cooperative-competitive environments. Neural Information Processing Systems (NIPS) (2017)

18. Mordatch, I., Abbeel, P.: Emergence of grounded compositional language in multi-agent populations. arXiv preprint arXiv:1703.04908 (2017)
19. Oliehoek, F.A., Amato, C.: A concise introduction to decentralized POMDPs. Springer (2016)
20. OpenAI: Openai five. https://blog.openai.com/openai-five/ (2018)
21. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: Pytorch: An imperative style, high-performance deep learning library. arXiv preprint arXiv:1912.01703 (2019)
22. Petrenko, A., Huang, Z., Kumar, T., Sukhatme, G., Koltun, V.: Sample factory: Egocentric 3d control from pixels at 100000 fps with asynchronous reinforcement learning. In: International Conference on Machine Learning. pp. 7652–7662. PMLR (2020)
23. Rashid, T., Samvelyan, M., Schroeder, C., Farquhar, G., Foerster, J., Whiteson, S.: Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. In: International Conference on Machine Learning. pp. 4295–4304. PMLR (2018)
24. 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., Whiteson, S.: The StarCraft Multi-Agent Challenge. CoRR **abs/1902.04043** (2019)
25. Schulman, J., Moritz, P., Levine, S., Jordan, M., Abbeel, P.: High-dimensional continuous control using generalized advantage estimation. arXiv preprint arXiv:1506.02438 (2015)
26. Shalev-Shwartz, S., Shammah, S., Shashua, A.: Safe, multi-agent, reinforcement learning for autonomous driving. arXiv preprint arXiv:1610.03295 (2016)
27. Son, K., Kim, D., Kang, W.J., Hostallero, D.E., Yi, Y.: Qtran: Learning to factorize with transformation for cooperative multi-agent reinforcement learning. In: International Conference on Machine Learning. pp. 5887–5896. PMLR (2019)
28. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
29. Vinyals, O., Babuschkin, I., Czarnecki, W.M., Mathieu, M., Dudzik, A., Chung, J., Choi, D.H., Powell, R., Ewalds, T., Georgiev, P., et al.: Grandmaster level in starcraft ii using multi-agent reinforcement learning. Nature **575**(7782), 350–354 (2019)
30. Wang, J., Ren, Z., Liu, T., Yu, Y., Zhang, C.: Qplex: Duplex dueling multi-agent q-learning. arXiv preprint arXiv:2008.01062 (2020)
31. Wang, T., Dong, H., Lesser, V., Zhang, C.: Roma: Multi-agent reinforcement learning with emergent roles. In: Proceedings of the 37th International Conference on Machine Learning (2020)
32. Wang, T., Gupta, T., Mahajan, A., Peng, B., Whiteson, S., Zhang, C.: Rode: Learning roles to decompose multi-agent tasks. arXiv preprint arXiv:2010.01523 (2020)
33. Ye, D., Chen, G., Zhang, W., Chen, S., Yuan, B., Liu, B., Chen, J., Liu, Z., Qiu, F., Yu, H., et al.: Towards playing full moba games with deep reinforcement learning. arXiv preprint arXiv:2011.12692 (2020)
34. Yu, C., Velu, A., Vinitsky, E., Wang, Y., Bayen, A., Wu, Y.: The surprising effectiveness of mappo in cooperative, multi-agent games. arXiv preprint arXiv:2103.01955 (2021)