arXiv:2302.05007v1 [cs.MA] 10 Feb 2023

Scalability Bottlenecks in Multi-Agent Reinforcement Learning Systems

Kailash Gogineni, Peng Wei, Tian Lan and Guru Venkataramani
The George Washington University, Washington, DC, USA
E-mail: {kailashg26, pwei, tlan, guruv}@gwu.edu

Abstract—Multi-Agent Reinforcement Learning (MARL) is a promising area of research that can model and control multiple, autonomous decision-making agents. During online training, MARL algorithms involve performance-intensive computations such as exploration and exploitation phases originating from large observation-action space belonging to multiple agents. In this article, we seek to characterize the scalability bottlenecks in several popular classes of MARL algorithms during their training phases. Our experimental results reveal new insights into the key modules of MARL algorithms that limit the scalability, and outline potential strategies that may help address these performance issues.

Index Terms—C.4 Performance of Systems < C Computer Systems Organization; I.2.11.d Multiagent systems < I.2.11 Distributed Artificial Intelligence < I.2 Artificial Intelligence < I Computing Methodologies

1 Introduction

Reinforcement Learning (RL) algorithms have widespread applications in robotics, aviation, autonomous driving, gaming, recommendation systems and healthcare. RL frameworks optimize AI agent behavior and its interactions with an environment by taking actions based on current observation/state space, evaluating the quality of state-action pairs using a reward function, and then transitioning to a new state [1]. The function that determines the action is known as a policy. The agent aims to find an optimal policy that maximizes the total accumulative (discounted) reward. The function representing the reward estimates is known as the value function.

Multi-agent Reinforcement Learning (MARL [1]) is a rapidly growing research area where there is significant sharing of observations between the agents during training, and joint actions among these agents could affect the environment dynamically. Agents are trained to reach the goals while minimizing interference (obstacles) with each other to perform competitive (e.g., Predator prey) and cooperative (e.g., Cooperative navigation) tasks [2]. In the cooperative setting, all the observations are shared and the training can be performed centrally. A competitive setting differentiates the agent pool- i.e., each agent aims to outperform its opponents. The actions taken by the individual agents affect other agents' behavior and their rewards dynamically in this environment. As a result, MARL training involves several *computationally-challenging* tasks that deal with dynamically changing environments.

In this article, we seek to understand the *key scalability bottlenecks* on well-known model-free MARL frameworks [2], [3], [4] implemented using actor-critic methods with state spaces that are usually very large. We analyze different MARL training phases where the actor and critic networks are responsible for policy and value functions, respectively. As shown in Figure 1, the actor network outputs the actions when a group of agents interact with the environment and each agent learns an individual policy that maps its observations to optimal actions (*Action selection*) to maximize the expected return. During the *mini-batch sampling* phase, each agent i collects the observations, actions, and new observations of all other agents stored within the

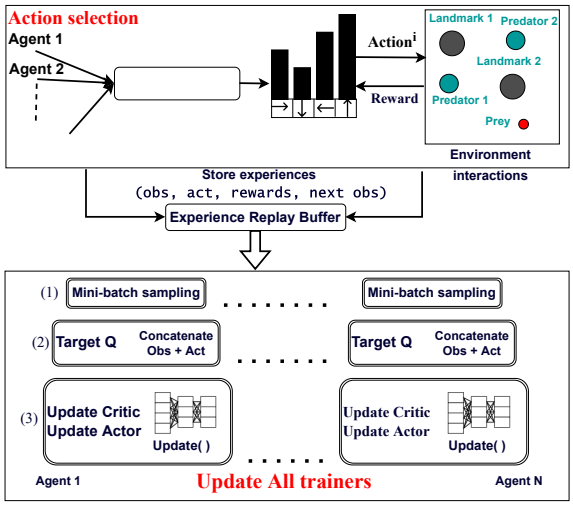
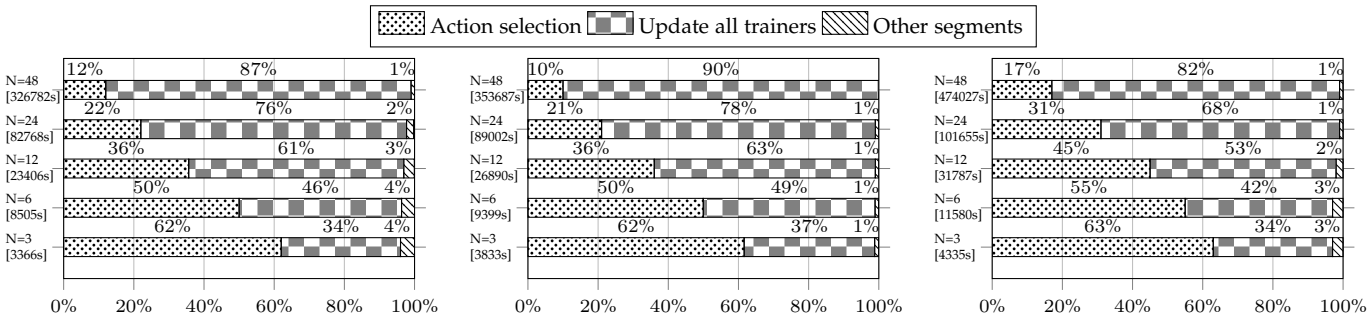


Fig. 1: Overview of our multi-agent decentralized actor, centralized critic approach (Competitive environment).

Experience Replay Buffer. The critic updates the value function using random mini-batch samples of all the agents obtained from the past experiences stored in the replay buffer. During *Update all trainers* phase, the actor network is updated using Q-values computed by the critic [2].

2 Motivation

MARL training is performance-intensive as the agents' policies continually evolve, and the replay buffer samples will be refreshed to find an optimal policy for the inference [2]. Figure 2 shows that *Update all trainers* contributes to ≈35% to ≈90% of the training time as the number of MARL agents grow from 3 to 48. This is primarily due to two reasons: ① In MARL, each agent has its own actor and critic networks since they may have different rewards. Each agent has to randomly sample a mini-batch of transitions from the replay buffer to update both the critic and actor networks. This requires each agent to



(a) MADDPG (b) MATD3 (c) MASAC

Fig. 2: Training time breakdown for three MARL workloads with 3 to 48 agents. The environment is Competitive task (Predator-Prey). The total training time of MARL algorithms (in seconds) is shown on y-axis within square brackets.

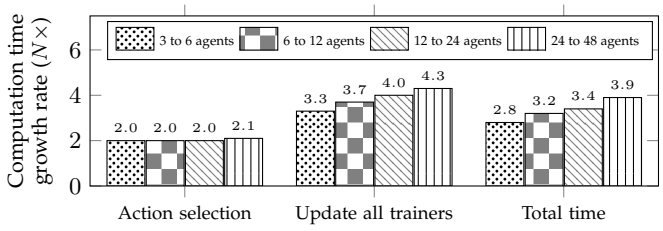


Fig. 3: Computation time growth in MARL modules averaged across three MARL frameworks.

sample experiences from every other agent. ② The dynamic memory requirements of observation and action spaces also grow quadratically due to each agent having to coordinate with other agents towards sharing their observations and actions. We observe that other MARL phases, such as *Action selection* occupy a small portion and scales linearly with the number of agents (Figure 3). This is because, action selection is performed with individual agents' policy using local observations and interactions with the environment. *Other segments* is a combination of experience collection, reward collection and policy initialization and they add a negligible overhead. Note that the agents interact in a shared environment, and the training time is summed over all agents.

Prior studies, like FA3C [5], have focused on accelerating multiple parallel worker scenarios, where each agent is controlled independently within their own environments using single-agent RL algorithms. In contrast, we seek to understand multi-agent learning frameworks, where the agents operate in a single shared environment. Agents in such MARL settings usually have high visibility of one another (leading to large space and action spaces). To the best of our knowledge, this is the first characterization study of MARL scalability bottlenecks.

3 Background

Typically, MARL settings with N agents is defined by a set of states, $S = S_1 \times ... \times S_N$, a set of actions $A = A_1 \times ... \times A_N$. Each agent selects its action by using a policy $\pi_{\theta_i} : O_i \times A_i \rightarrow [0, 1]$. The state transition ($T : S \times A_1 \times A_2 \times ... \times A_N$) function produces the next state $S^{'}$, given the current state and actions for each agent. The reward, $R_i : S \times A_i \rightarrow \mathbb{R}$ for each agent is a function of global state and action of *all other agents*, with the aim of maximizing its own expected return $R_i = \sum_{t=0}^{T} \gamma^t r_i^t$, where γ denotes the discount factor and T is the time horizon. For this, we use the actor-critic methods such as MADDPG [2], MATD3 [3], MASAC [4].

In *MADDPG* [2], each agent learns an individual policy that maps the observation to its action to maximize the expected return, which is approximated by the critic. MADDPG lets the critic of agent i to be trained by minimizing the loss with the *target Q-value* and y_i using $\mathcal{L}(\theta_i) = \mathbb{E}_D[(Q_i(S, A_1, ...A_n) - y_i^2]$, and $y_i = r_i + \gamma \overline{Q}_i(S^{'}, A_1^{'}, ...A_n^{'})_{a_j^{'}=\overline{\pi}(o_j^{'})}$, where S and $A_1, ...A_n$ represent the joint observations and actions respectively. D is the experience replay buffer that stores the *observations, actions, rewards, and new observations* samples of all agents obtained after the training episodes. The MARL framework has four networks- actor, critic, target actor, and target critic. $\overline{Q}_i$ and $\overline{\pi}(o_j^{'})$ are the target networks for the stable learning of critic (Q_i) and actor networks. The target actor estimates next action from the policy using the state output by the actor network. The target critic aggregates the output from the target actor to compute the target Q-values, that helps to update the critic network and assess the quality of of actions taken by agents. The target networks are created to achieve training stability. Note that the updating sequence of networks in the back-propagation phase is critics, actors, then the target networks.

MATD3 [3] uses the twin delayed critics to tackle the over-estimation bias problem [3] and incorporates small amounts of noise to the actions sampled from the buffer. As the change of critic values needs to be reflected in the policies of other agents, MATD3 employs delayed policy updates for target networks and the policies to obtain an accurate critic before using it to update the actor. In the domains where it is necessary to learn a winning strategy (e.g., Predator-Prey, Cooperative navigation), MATD3 outperforms MADDPG [3].

MASAC [4] improves the convergence properties over MADDPG and MATD3, and obtains higher returns in the competitive and cooperative environments. MASAC uses the maximum entropy RL, in which the agents are encouraged to maximize the exploration within the policy. MASAC assigns equal probability to nearly-optimal actions which have similar state-action values and avoids repeatedly selecting the same action. This learning trick will increase the stability, policy exploration and the sample efficiency [4], [6].

4 Evaluation Setup

We evaluate three state-of-the-art MARL algorithms, MADDPG, MATD3, and MASAC using Multi-agent Particle Environment (MPE [2]). We outline the behavior of the selected environments in Table 1. The actor and critic networks are paramterized by a two-layer ReLU MLP with 64 units per layer

TABLE 1: Multi-agent Particle environment.

Environment	Details
Cooperative navigation	N agents move in a cooperated manner to reach L landmarks and the rewards encourages the agents get closer to the landmarks.
Predator-Prey	N predators work cooperatively to block the way of M fast paced prey agents. The prey agents are environment controlled and they try to avoid the collision with predators.

and mini-batch size is 1024 for sampling the transitions. In all of our experiments, we use Adam optimizer [7] with a learning rate of 0.01, maximum episode length as 25 (max episodes to reach the terminal state) and τ = 0.01 for updating the target networks. γ is the discount factor which is set to 0.95. The size of replay buffer is 10^6 and entropy coefficient for MASAC is 0.05. The network parameters are updated after every 100 samples added to the replay buffer.

All the workloads are trained and profiled on Nvidia GeForce RTX 3090 Ampere Architecture connected with AMD Ryzen Threadripper PRO 3975WX CPU, which has 32 cores with 128 MiB of Last-Level Cache, 512 Gigabytes of main memory and the CPU's clock speed of 3.5GHz. The server runs on Ubuntu Linux 20.04.5 LTS operating system with CUDA 9.0, cuDNN 7.6.5, PCIe Express® v4.0 with NCCL v2.8.4 communication library. The machine supports python 3.7.15, Tensorflow (v2.11.0), Tensorflow-GPU (v2.1.0) and OpenAI GYM (v0.10.5). We use Perf [8] tool and hardware performance counters for performance analysis. The workloads are trained for 60K episodes using default hyper-parameters recommended by the algorithms.

5 Experimental Evaluation

For deeper analysis, we divide *Update all trainers* into multiple modules: *Mini-batch sampling, Target Q calculation*, and *Q loss & P loss* and present our results in the primarily competitive setting (predator-prey) in order to understand the key factors limiting MARL scalability. We note that the predator agents operate in the cooperative setting to maximize their shared return. Therefore, our test-bed allows us to evaluate both competitive and cooperative agent scenarios.

Overview of Profile. Figure 4 shows the breakdown between the modules, *Mini-batch sampling, Target Q calculation, Q loss and P loss* that contribute 61%, 21%, 10%, and 8% to computation time averaging across different workloads respectively. With increasing number of agents from 3 to 48, we observe that Instructions Per Cycle (IPC) steadily drops by over 21% (1.37 to 1.08), and global cache misses increase by 54%, which indicates that the instruction throughput and cache performance slumps when more agents are involved. A super-linear growth rate of certain key architectural features like branch misses, TLB load misses, and global cache misses further validate the presence of computational bottlenecks across MARL workloads (Figure 5).

5.1 Mini-batch sampling

Our experimental results in Figure 6 shows super-linear increase in computation time with the number of agents during mini-batch sampling, the largest phase within the *Update All Trainers* module. The is also consistently reflected in other related performance metrics: *Total instructions*-4× and *LLC-Load-misses*-3×. The competitive behavior between prey agents and predator agents involves each predator agent receiving the velocity, position relative to all other predator agents and landmarks as observations to hunt the prey. A cooperative behavior also exists between the predator agents to maximize their shared return. Note that the agent replay buffers are kept separate from each other to capture their individual past transitions.

In this module, each agent has to randomly sample a set of mini-batch samples uniformly from others' replay buffers and update the parameters of its critic network. Each agent i performs *lookup-read-write* operations, which grow as a function of number of agents, N and this is repeated on all N agents. The time complexity to collect the transition set is $O(N^2K)$, K being the batch size. In cooperative navigation (*simple spread* [2]), we observe similar scalability bottlenecks since all the agents are trained together to reach the landmarks while avoiding collisions with each other.

5.2 Target Q calculation

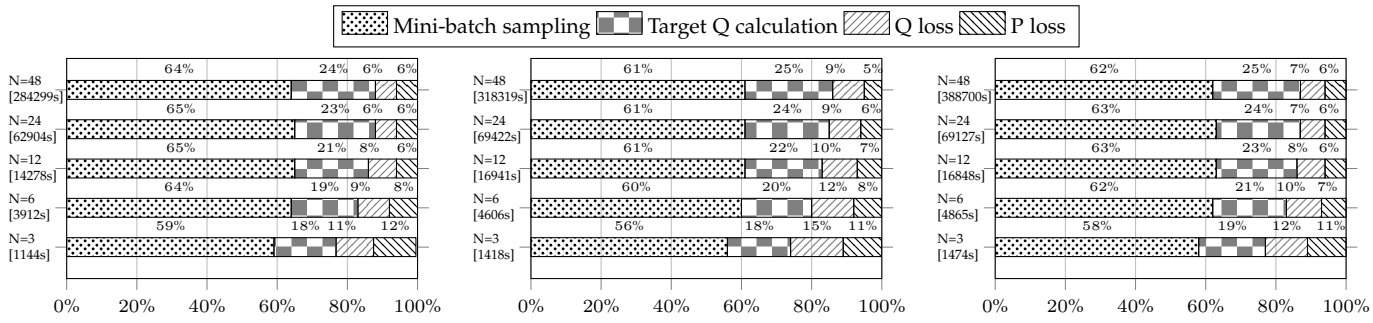
The *Target Q calculation* phase is second largest time consuming phase within *Update All Trainers*. Figure 6 shows that this phase grows by $4\times$ with the number of agents. Note that, in Figure 4, the computation time as a percentage within *Update All Trainers* increases with the number of agents for *target Q*, whereas the execution time proportion of *Q loss and P loss* decrease slightly. This is because, the *target Q* grows by a higher rate compared to *Q loss* (Figure 6). Each agent performs the *next action calculation, target Q next, and target Q values* as a function of all other agents' joint observation-action space. To calculate the *next action*, the agent i uses its policy network to determine *next action-a'* from the *next state-S'*. In this phase, each agent's policy network involves multiplications with input-weight matrix and additions resulting in performance impact. The obtained a' and S' data are aggregated and concatenated into a single vector in order to compute the *target Q next* amongst the cooperating agents. The input space (dimension) for the *Q-function* increases quadratically with the number of agents [9]. The target critic values for each agent i is computed using *target Q next* values from the target actor network. We note that, each agent has to read other agents' policy values; as such for N agents, there is $N \times (N-1)$ memory lookup operations corresponding to the *next action-a'*.

5.3 Back-propagation - Q loss & P loss

Back propagation is the third largest phase of *Update all Trainers*. This phase is dominated by the back-propagation of *critic network* that computes the Mean-Squared Error loss between the target critic and critic networks, and the *actor network* is updated by minimizing the Q values (critic network). As the number of agents increases, the trainable parameters increase, and N policy and N critic networks are built for all N agents, which incurs extra time to update the weights for each agent. The average computation overhead of critic network for each iteration grows by up to 27% for every $2\times$ increase in the agents. For the actor network, the average computation overhead grows by 36%, which can help explain the performance bottlenecks involved in updating the weights for the individual agent networks.

6 Architectural Guidelines

Architectural primitives implementing selective attention [6] may help relieve some of the MARL scalability bottlenecks in *target Q calculation and the critic network* toward reducing the input space for the networks.



(a) MADDPG (b) MATD3 (c) MASAC

Fig. 4: Training time breakdown within *Update all trainers* on three different MARL workloads with 3 to 48 agents under Predator-Prey environment. The total training time of *Update all trainers* (in seconds) is shown on y-axis within square brackets.

To address the data movement-related performance issues, we note that processing in- or near-memory accelerators may help improve the related architectural bottlenecks significantly. For example, recent GDDR6-based Processing-in-Memory (PIM) [10] may be used to accelerate the matrix multiply-accummulate and activation operations for all of the four neural networks in most current MARL frameworks. Future PIM designs may also be augmented to incorporate MARL-specific hardware primitives (e.g., efficient mini-batch sampling) for improved hardware efficiency. Multi-core PIM Neural network accelerators can be used to exploit the higher coarse-grain hardware parallelism offered by multiple neural networks in MARL algorithms.

Code optimizations and data parallelism can also be leveraged during *mini-batch sampling*. Shared memory multi-threading may be used to perform faster sampling (e.g., via OpenMP). This can help reduce the computational bottlenecks with each agent sampling its own transitions with parallel threads [11]. Our hardware performance analysis also shows the potential for improving the branch and TLB behavior, and the need for optimizations in hardware and software that can alleviate these issues.

For the input to critic networks, multi-level compression [12] techniques on selected group of agents may be used based on their importance in the environment. Also, LLC-Load-misses during mini-batch sampling are indicative of competition for the LLC cache, that may be addressed through smart cache allocation strategies. Other modules such as *next action calculation*, *environment interactions*, *and action selection* phases may also benefit from parallelization and custom acceleration of key modules.

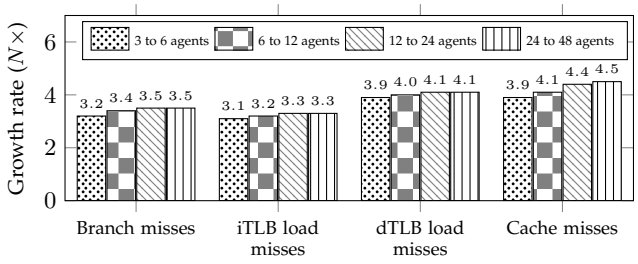


Fig. 5: Hardware performance analysis of *Update all trainers* averaged across three MARL workloads when the number of agents are scaled by 2×. Performance analysis is averaged across three sub-functions of *Update all trainers*.

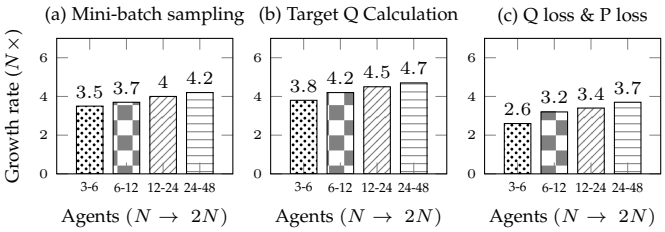


Fig. 6: Computation time growth rate of the sub-functions within *Update all trainers* averaged across three MARL workloads when the number of agents are scaled by 2×.

7 Conclusion

In this work, we present a detailed, end-to-end characterization of several popular Multi-Agent Reinforcement Learning algorithms and in particular, explore the scalability bottlenecks in these workloads. Our experimental analysis present key insights on the modules that are driving factors behind scalability bottlenecks, and outline architectural guidelines to overcome them.

Acknowledgments

This research is based on work supported by the National Science Foundation under grant CCF-2114415.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
[2] R. Lowe, Y. I. Wu, A. Tamar, J. Harb, O. Pieter Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," *NeurIPS*, vol. 30, 2017.
[3] J. Ackermann, V. Gabler, T. Osa, and M. Sugiyama, "Reducing overestimation bias in multi-agent domains using double centralized critics," *NeurIPS Deep RL Workshop*, 2019.
[4] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *ICML*. PMLR, 2018, pp. 1861–1870.
[5] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "Fa3c: Fpga-accelerated deep reinforcement learning," in *ASPLOS*, 2019, pp. 499–513.
[6] S. Iqbal and F. Sha, "Actor-attention-critic for multi-agent reinforcement learning," in *ICML*. PMLR, 2019, pp. 2961–2970.
[7] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
[8] V. Ramos, "Performance counters api for python," https://pypi.org/project/performance-features/, May 2019.
[9] H. U. Sheikh and L. Bölöni, "Multi-agent reinforcement learning for problems with combined individual and team reward," in *IJCNN*. IEEE, 2020, pp. 1–8.

[10] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim *et al.*, "A 1ynm 1.25 V 8Gb, 16Gb/s/pin GDDR6-based Accelerator-in-Memory supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications," in *ISSCC*, vol. 65. IEEE, 2022, pp. 1–3.

[11] T. Kaler, N. Stathas, A. Ouyang, A.-S. Iliopoulos, T. Schardl, C. E. Leiserson, and J. Chen, "Accelerating training and inference of graph neural networks with fast sampling and pipelining," *MLSys*, vol. 4, pp. 172–189, 2022.

[12] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *ISCA*. IEEE, 2018, pp. 776–789.