Machine Intelligence Research
www.mi-research.net

21(3), June 2024, 411-430
DOI: 10.1007/s11633-023-1454-4

Distributed Deep Reinforcement Learning: A Survey and a Multi-player Multi-agent Learning Toolbox

Qiyue Yin[1,2] Tongtong Yu[1] Shengqi Shen[1] Jun Yang[3] Meijing Zhao[1]
Wancheng Ni[1,2] Kaiqi Huang[1,2,4] Bin Liang[3] Liang Wang[1,2,4]

[1] Institute of Automation, Chinese Academy of Sciences, Beijing 100190, China
[2] School of Artificial Intelligence, University of Chinese Academy of Sciences, Beijing 100049, China
[3] Department of Automation, Tsinghua University, Beijing 100084, China
[4] Center for Excellence in Brain Science and Intelligence Technology, Chinese Academy of Sciences, Beijing 100190, China

Abstract: With the breakthrough of AlphaGo, deep reinforcement learning has become a recognized technique for solving sequential decision-making problems. Despite its reputation, data inefficiency caused by its trial and error learning mechanism makes deep reinforcement learning difficult to apply in a wide range of areas. Many methods have been developed for sample efficient deep reinforcement learning, such as environment modelling, experience transfer, and distributed modifications, among which distributed deep reinforcement learning has shown its potential in various applications, such as human-computer gaming and intelligent transportation. In this paper, we conclude the state of this exciting field, by comparing the classical distributed deep reinforcement learning methods and studying important components to achieve efficient distributed learning, covering single player single agent distributed deep reinforcement learning to the most complex multiple players multiple agents distributed deep reinforcement learning. Furthermore, we review recently released toolboxes that help to realize distributed deep reinforcement learning without many modifications of their non-distributed versions. By analysing their strengths and weaknesses, a multi-player multi-agent distributed deep reinforcement learning toolbox is developed and released, which is further validated on Wargame, a complex environment, showing the usability of the proposed toolbox for multiple players and multiple agents distributed deep reinforcement learning under complex games. Finally, we try to point out challenges and future trends, hoping that this brief review can provide a guide or a spark for researchers who are interested in distributed deep reinforcement learning.

Keywords: Deep reinforcement learning, distributed machine learning, self-play, population-play, toolbox.

Citation: Q. Yin, T. Yu, S. Shen, J. Yang, M. Zhao, W. Ni, K. Huang, B. Liang, L. Wang. Distributed deep reinforcement learning: A survey and a multi-player multi-agent learning toolbox. *Machine Intelligence Research*, vol.21, no.3, pp.411–430, 2024. http://doi.org/10.1007/s11633-023-1454-4

1 Introduction

With the breakthrough of AlphaGo[1, 2], an agent that wins many professional Go players in human-computer gaming, deep reinforcement learning (DRL) has come to most researchers' attention, which has become a recognized technique for solving sequential decision making problems. Many algorithms have been developed to solve challenging issues that lie between DRL and the real world applications, such as exploration and exploitation dilemma, data inefficiency, and multi-agent cooperation and competition. Among all these challenges, data inefficiency is the most criticized due to the trial and error learning mechanism of DRL, which requires a huge amount of interactive data.

To alleviate the data inefficiency problem, several research directions have been developed[3]. For example, model-based deep reinforcement learning constructs environment models for generating imaginary trajectories to help reduce times of interaction with the environment. Transfer reinforcement learning mines shared skills, roles, or patterns from source tasks, and then uses the learned knowledge to accelerate reinforcement learning in the target task. Inspired by distributed machine learning techniques, which have been successfully utilized in computer vision and natural language processing[4], distributed deep reinforcement learning (DDRL) was developed, which has shown its potential to train very successful agents, such as Suphx[5], OpenAI Five[6], and AlphaStar[7].

Generally, training DRL agents consists of two main parts, i.e., pulling policy network parameters to generate data by interacting with the environment, and updating policy network parameters by consuming data. Such a

Review
Manuscript received on December 30, 2022; accepted on May 6, 2023; published online on January 11, 2024
Recommended by Associate Editor Maoguo Gong
Colored figures are available in the online version at https://link.springer.com/journal/11633

structured pattern makes distributed modifications of DRL feasible, and many DDRL algorithms have been developed. For example, the general reinforcement learning architecture[8], likely the first DDRL architecture, divides the training system into four components, i.e., parameter server, learners, actors and replay buffer, which inspires more successive data efficient DDRL architectures. The recently proposed SEEDRL[9], an improved version of IMPALA[10], is claimed to be able to produce and consume millions of frames per second, based on which AlphaStar is successfully trained within 44 days (192 v3 + 12 128 core TPUs, 1 800 CPUs) to beat professional human players.

To make distributed modifications of DRL be able to use multiple machines, several engineering problems should be solved, such as machine communication and distributed storage. Fortunately, several useful toolboxes have been developed and released, and revising DRL codes to a distributed version usually requires a small amount of code modification, which largely promotes the development of DDRL. For example, Horovod[11], released by Uber, makes full use of the ring allreduce technique, and can properly use multiple GPUs for training acceleration by adding only a few lines of codes compared with the single GPU version. Ray[12], a distributed framework of machine learning released by UC Berkeley RISELab, provides RLlib[13] for efficient DDRL, which is easy to use due to its reinforcement learning abstraction and algorithm library.

Considering the great progress of DDRL, it is necessary to comb out the course of DDRL techniques to conclude challenges and opportunities to provide clues for future research. Recently, Samsami and Alimadad[14] gave a brief review of DDRL, but their aim is constructing single agent distributed reinforcement learning framework, and more challenging multiple players and multiple agents DDRL are absent. Czech[15] conducted a short survey on distributed methods for reinforcement learning, but only several classical algorithms were introduced with no key techniques, comparisons and challenges being discussed. Different from previous summaries, this paper aims to provide a more comprehensive survey. Through this paper, we hope to study important components to achieve efficient distributed learning, and use this to provide a new taxonomy. We will compare the classical distributed deep reinforcement learning methods covering single player single agent DDRL to the most complex multiple players multiple agents DDRL. Using the comparison, we hope to provide a guide for beginners and conclude with challenges and opportunities for future study.

The rest of the paper is organized as follows. In Section 2, we briefly describe the background of DRL, distributed learning, and typical testbeds for DDRL. In Section 3, we elaborate on the taxonomy of DDRL, by dividing current algorithms based on the learning frameworks and players and agents participating in. In Section 4, we compare current DDRL toolboxes, which help to achieve efficient DDRL a lot. In Section 5, we introduce a new multi-player multi-agent DDRL toolbox, which provides a useful DDRL tool for complex games. In Section 6, we summarize the main challenges and opportunities for DDRL, hoping to inspire future research. Finally, we conclude the paper in Section 7.

2 Background

2.1 Deep reinforcement learning

Reinforcement learning is a typical kind of machine learning paradigm, the essence of which is learning via interaction. In a general reinforcement learning method, an agent interacts with an environment by providing actions to drive the environment dynamics, and receiving rewards to improve its policy for chasing long-term outcomes. Usually, there are two typical kinds of algorithms to learn the agent, i.e., model-free methods that use no environment models, and model-based approaches that use the pregiven or learned environment models. Many algorithms have been proposed, and readers can refer to [16, 17] for a more thorough review.

In reality, applications naturally involve the participation of multiple agents, making multi-agent reinforcement learning a hot topic. Generally, multi-agent reinforcement learning is modelled as a stochastic game and obeys a learning paradigm similar to that of conventional reinforcement learning. Based on the game setting, agents can be fully cooperative, competitive and a mix of the two, requiring reinforcement learning agents to emerge abilities that can match the goal. Various key problems of multi-agent reinforcement learning have been raised, such as communication and credit assignment. Readers can refer to [18, 19] for a detailed introduction.

With the breakthrough of deep learning, deep reinforcement learning has become a strong learning paradigm by combining the representation learning ability of deep learning and the decision making ability of reinforcement learning, and several successful deep reinforcement learning agents have been proposed. For example, AlphaGo[1, 2], a Go agent that can beat professional human players, is based on single agent deep reinforcement learning. OpenAI Five[6], a dota2 agent that wins champion players in an e-sport for the first time, relies on multi-agent deep reinforcement learning. In the following, unless otherwise stated, we do not distinguish deep reinforcement learning and multi-agent deep reinforcement learning.

2.2 Distributed learning

The success of deep learning is inseparable from big data and computing power, which leads to huge demand for distributed learning that can handle data intensive and compute intensive computing. Due to the structured computation pattern of deep learning algorithms, some successful distributed learning methods have been proposed for parallelism in deep learning[20, 21]. An early popular distributed deep learning framework is DistBelief[22], designed by Google, which can train a deep network with billions of parameters using tens of thousands of CPU cores. Based on DistBelief, Google released the second generation of distributed deep learning framework, TensorFlow[23], which has become a widely used tool. Other typical distributed deep learning frameworks[24], such as PyTorch, MXNet and Caffe2, have also been developed and used by the research and industrial communities.

Ben-Num and Hoefler[20] provided an in-depth concurrency analysis of parallel and distributed deep learning. In the survey, the authors gave different types of concurrency for deep neural networks, covering the bottom level operators, and key factors such as network inference and training. Several important topics such as asynchronous stochastic optimization, distributed system architectures and communication schemes are discussed, providing clues for future directions of distributed deep learning. Currently, distributed learning is widely used in various fields, such as wireless networks[25], AIoT service platforms[26] and human-computer gaming[27].

In short, DDRL is a special type of distributed deep learning. Instead of focusing on data parallelism and model parallelism in conventional deep learning, DDRL aims at improving data throughput due to the characteristics of reinforcement learning. To achieve this, several important techniques should be well explored like in distributed deep learning, such as the communication schemes between machines, asynchronous stochastic optimization and distributed storage. Many methods have been proposed. For example, parameter server and its variants, such as shared parameter server and hierarchical parameter server are widely used to store network parameters that may be updated by several processes with synchronous or asynchronous stochastic optimization. RPC, as an efficient remote procedure call for communication, is widely used for various distributed frameworks such as SEED[9] and Ray[12]. Readers can refer to [25] for more details of the bottom techniques for distributed learning.

2.3 Testing environment

With the huge success of AlphaGo[1], DDRL is widely used in games, especially human-computer gaming. Those games provide an ideal testbed for the development of DDRL algorithms or frameworks, from single player single agent DDRL to multiple players multiple agents DDRL.

Atari is a popular reinforcement learning testbed because it has a similar high dimensional visual input compared to humans[28]. In addition, several environments confront challenging issues such as long time horizons and sparse rewards[29]. Many DDRL algorithms are compared in Atari games, showing training acceleration against DRL without parallelism. However, typical Atari games are designed for single player single agent problems.

With the emergence of multi-agent reinforcement learning in multi-agent games, StarCraft multi-agent challenge (SMAC)[30] has become a recognized testbed for single player multi-agent reinforcement learning. Specifically, SMAC is a subtask of StarCraft that focuses on micromanagement challenges, where a team of units is controlled to fight against build-in opponents. Several typical multi-agent reinforcement learning algorithms are released along with SMAC, which support parallel data collection in reinforcement learning.

Apart from the above single player single agent and single player multiple agents testing environments, there are a few multiple players environments for deep reinforcement learning algorithms, such as in OpenSpiel[31]. On the other hand, even though huge success has been made for multiple players games such as Go, StarCraft, dota2, and honor of kings, those environments are used for a few researchers due to the huge game complexity. Researchers from large companies such as Google and OpenAI usually use large computing resources to train human-level AI bots. However, on the whole, those complex multiple player single agent and multiple agents environments largely promote the development of DDRL.

3 Taxonomy of DDRL

3.1 Taxonomic basis

Many DDRL algorithms or frameworks have been developed with representatives such as GORILA[8], A3C[32], APE-X[33], IMPALA[10], distributed PPO[34], R2D2[35] and Seed RL[9], based on which, we can draw the key components of a DDRL, as shown in Fig. 1. We sometimes use the frameworks instead of algorithms or methods because these frameworks are not targeted to a specific reinforcement learning algorithm, and they are more like a distributed framework for various reinforcement learning methods. Generally, there are three main parts for a basic DDRL algorithm, which forms a single player single agent DDRL method:

1) Actors: produce data (trajectories or gradients) by interacting with the environment.

2) Learners: consume data (trajectories or gradients) to perform policy neural network parameter updating.

3) Coordinators: coordinate data (parameters or trajectories) to control the communication between learners and actors.

Actors pull neural network parameters from the

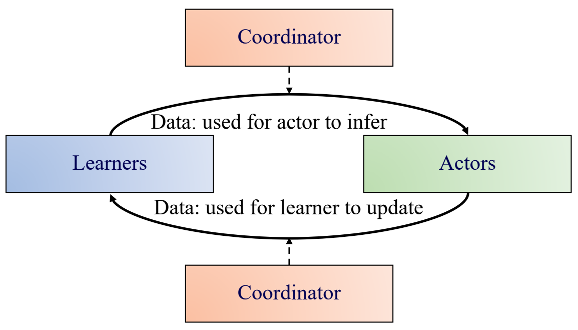

Coordinator
Data: used for actor to infer
Learners
Actors
Data: used for learner to update
Coordinator

Fig. 1 Basic framework of DDRL

learners, receive states from the environments, and perform inference to obtain actions, which drive the dynamics of environments to the next state. By repeating the above process with more than one actor, data throughput can be increased and enough data can be collected. Learners pull data from actors, perform gradient calculation or post-processing, and update the network parameters. More than one learner can alleviate the limited storage of a GPU by utilizing multiple GPUs with tools such as ring allreduce or parameter-server[11]. By repeating the above process, the final reinforcement learning agent can be obtained.

Coordinators are important for DDRL algorithms, which control the communication between learners and actors. For example, when the coordinators are used to synchronize the parameters updating and pulling (by actors), the DDRL algorithm is synchronous. When the parameters updating and pulling (by actors) are not strictly coordinated, the DDRL algorithm is asynchronous. Therefore, a basic classification of DDRL algorithms can be based on the coordinator types.

1) Synchronous based: Global policy parameters updating is synchronized, and pulling policy parameters (by actors) is synchronous, e.g., different actors share the same latest global policy.

2) Asynchronous based: Updating the global policy parameters is asynchronous, or policy updating (by learners) and pulling (by actors) are asynchronous, e.g., actors and learners usually have different policy parameters.

With the above basic framework, a single player single agent DDRL algorithm can be designed. Note that we ignore the bottom techniques used to implement multiple actors, multiple learners and coordinators, such as communication schemes between different jobs in a machine or in multiple machines and the stochastic optimization strategies for parameter updating. These bottom techniques are not the scope of this paper, and we will introduce them in Section 3.2 when specific DDRL methods are presented.

When the number of players or agents is increasing, the above basic framework is unable to train usable RL agents. Based on current DDRL algorithms that support large system level AI such as AlphaStar[7], OpenAI Five[6] and JueWU[36], two key components are essential to build multiple players and multiple agents DDRL, i.e., agent cooperation and player evolution, as shown in Fig. 2.

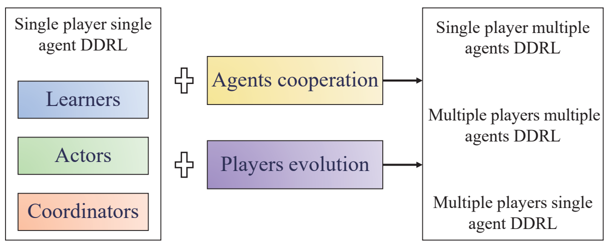

Single player single agent DDRL
Learners
Actors
Coordinators
Agents cooperation
Players evolution
Single player multiple agents DDRL
Multiple players multiple agents DDRL
Multiple players single agent DDRL

Fig. 2 Single player single agent DDRL to multiple players multiple agents DDRL

The module of agents cooperation is used to train multiple agents based on multi-agent reinforcement learning algorithms[18]. Generally, multi-agent reinforcement learning can be classified into two categories, i.e., independent training and joint training, based on how to perform agent relationship modeling.

1) Independent training: Train each agent independently by considering other learning agents as part of the environment.

2) Joint training: Train all the agents as a whole, considering factors such as agent communication, reward assignment and centralized training with distributed execution.

The module of players evolution is designed for agent iteration for each player, where agents of other players are learning at the same time, leading to more than one generation of agents to be learned for each player, such as in AlphaStar and OpenAI Five. Based on current mainstream players evolution techniques, players evolution can be divided into two types:

1) Self-play based: Different players share the same policy networks, and the player updates the current generation of the policy by confronting its past versions.

2) Population-play based: Different players have different policy networks, or called populations, and a player updates its current generation of policy by confronting other players or/and its past versions.

Finally, based on the above key components for DDRL, the taxonomy of DDRL is shown in Fig. 3. In Sections 3.2–3.4, we will summarize and compare representative methods based on their main characteristics. In addition, the bottom level techniques such as communications schemes will be introduced.

3.2 Coordinator types

Based on the coordinator types, DDRL algorithms can be divided into asynchronous based and synchronous based algorithms. For an asynchronous based DDRL method, there are two cases: The updating of global

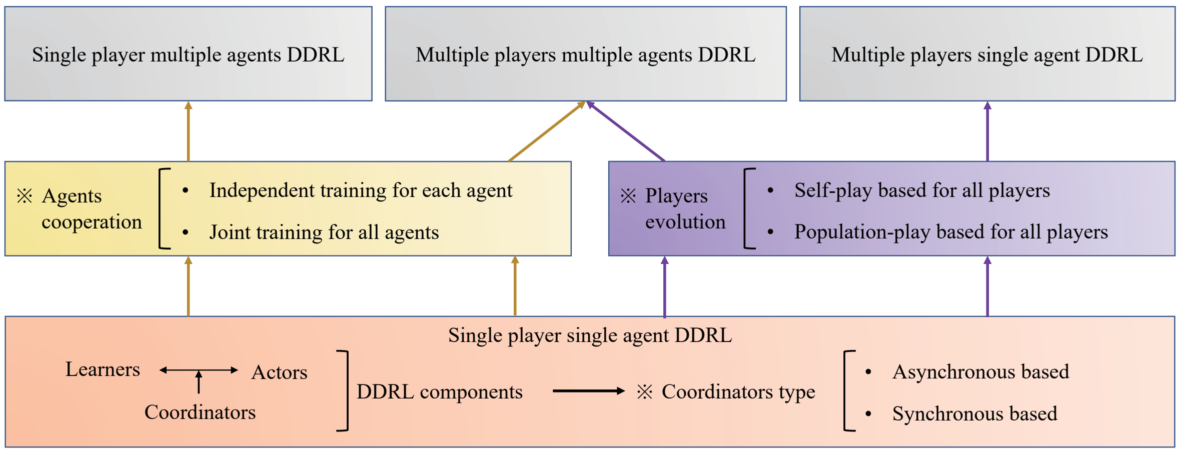

Single player multiple agents DDRL
Multiple players multiple agents DDRL
Multiple players single agent DDRL
※ Agents cooperation
Independent training for each agent
Joint training for all agents
※ Players evolution
Self-play based for all players
Population-play based for all players
Single player single agent DDRL
Learners
Actors
Coordinators
DDRL components
※ Coordinators type
Asynchronous based
Synchronous based

Fig. 3 The taxonomy of distributed deep reinforcement learning

policy parameters is asynchronous; the global policy parameters updating (by learners) and pulling (by actors) are asynchronous. For a synchronous based DDRL method, global policy parameters updating is synchronized, and pulling policy parameters (by actors) is synchronous.

3.2.1 Asynchronous based

Nair et al.[8] proposed probably the first massively distributed architecture for deep reinforcement learning, Gorila, which builds the basis of the succeeding DDRL algorithms. As shown in Fig. 4, a distributed deep Q-network (DQN) algorithm is implemented. There are multiple parallel actors to generate trajectories and send them to the Q-network and target Q-network of the learners. In addition, learners calculate gradients for parameter updating based on a central parameter server that can store a distributed neural network with multiple machines. The parameters updating (using gradients) is based on asynchronous stochastic gradient descent. Due to the implementation of DQN, neural network parameter updating of learners and trajectory collecting of actors are also asynchronously performed without waiting. In their paper, the implemented distributed DQN reduces the wall-time required to achieve compared or super results by an order of magnitude on most 49 games in Atari compared to non-distributed DQN.

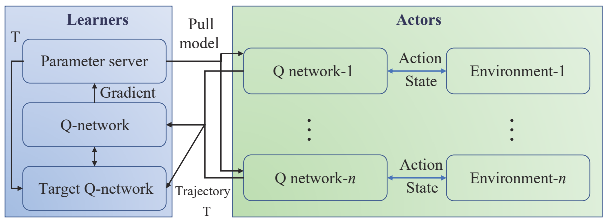


Fig. 4 Basic framework of Gorila

Similar to [8], Horgan et al.[33] introduced distributed prioritized experience replay, i.e., APE-X, to enhance Q-learning based distributed reinforcement learning. Specifically, prioritized experience replay is used to sample the most important trajectories, which are generated by all actors. Accordingly, a shared experience replay memory should be introduced to store all the generated trajectories. In the experiments, a fraction of the wall-clock training time is achieved on the arcade learning environment. To further enhance [33], Kapturowski et al.[35] proposed recurrent experience replay in distributed reinforcement learning, i.e., R2D2, by introducing RNN-based reinforcement learning agents. The authors investigate the effects of parameter lag and recurrent state staleness problems on the performance, obtaining the first agent to exceed human-level performance in 52 of the 57 Atari games with the designed training strategy.

Mnih et al.[32] proposed the asynchronous advantage actor-critic (A3C) framework, which can make full use of the multi-core CPU instead of the GPU, leading to cheap distribution of the reinforcement learning algorithm. As shown in Fig. 5, each actor calculates the gradient of the samples (mainly states, actions and rewards used for regular reinforcement learning algorithms), send them to the learners, and then update the global policy. The updating is asynchronous without synchronization among gradients from different actors. In addition, parameters (maybe not the latest version) are pulled by each actor to generate data with environments. Based on the multiple CPU threads on a single machine, the communication costs among machines no longer exist. In their paper, four specific reinforcement learning algorithms are established, i.e., asynchronous one-step Q-learning, asynchronous one-step Sarsa, asynchronous n-step Q-learning and asynchronous advantage actor-critic. Experiments show that half the time on a single multi-core CPU instead of a GPU is obtained on the Atari domain.

To make use of the GPU's computational power instead of just the multi-core CPU as in A3C, Babaeizadeh et al.[37] proposed asynchronous advantage actor-critic on a GPU, i.e., GA3C, which is a hybrid CPU/GPU version of A3C. As shown in Fig. 6, the learner consists of three parts: a predictor to dequeue prediction requests and obtain actions by the inference, a trainer to dequeue batches of trajectories for the agent model, and the agent model

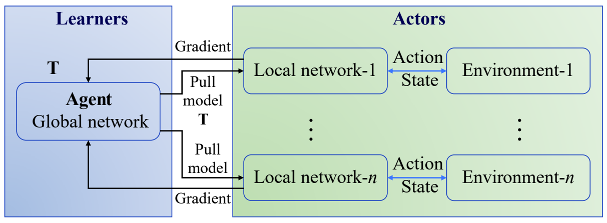


Fig. 5 Basic framework of A3C

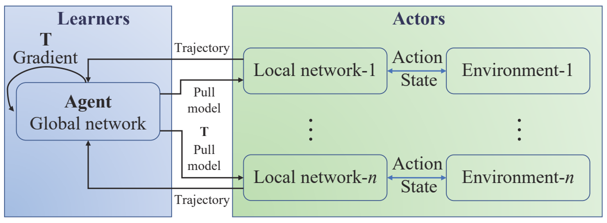


Fig. 7 Basic framework of IMPALA

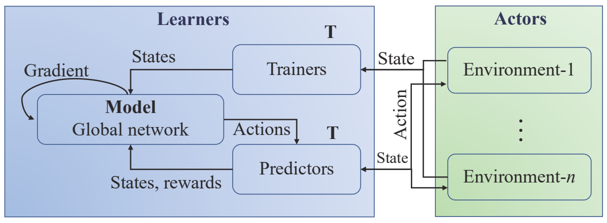


Fig. 6 Basic framework of GA3C

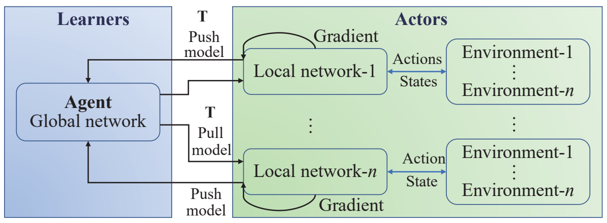


Fig. 8 Basic framework of APPO

to update the parameters with the trajectories. Note that the threads of the predictor and trainer are asynchronously executed. With the above multi-process multi-thread CPU for receiving actions and sending states, and several GPU threads for predicting actions and updating parameters, GA3C achieves a significant speed up compared to A3C.

Placing gradient calculation on the actor side will limit the data throughput of the whole DDRL system, i.e., trajectories collected per time unit, so Espeholt et al.[10] proposed importance weighted actor-learner architecture (IMPALA) to alleviate this problem. As shown in Fig. 7, parallel actors communicate with environments, collect trajectories and send them to the learners for parameter updating. Since gradient calculation is put on the learner side, which can be accelerated with GPUs, the framework is claimed to scale to thousands of machines without sacrificing data efficiency. Note that a synchronized parameter update is used when scaling to many machines, which is important to maintain data efficiency. Considering that the local policy used to generate trajectories is behind the global policy in the learners due to the asynchrony between learners and actors, a V-trace off-policy actor-critic algorithm is introduced to correct the harmful discrepancy. Experiments on DMLab-30 and Atari-57 show that IMPALA can achieve better performance with less data compared with previous agents.

By using a synchronized sampling strategy for actors instead of the independent sampling of IMPALA, Stooke and Abbeel[38] proposed a novel accelerated method, which consists of two main parts, i.e., synchronized sampling and synchronous/asynchronous multi-GPU optimization. As shown in Fig. 8, individual observations of some environments are gathered into a batch for inference, which largely reduces the inference times compared with approaches that generate trajectories for each environment independently. However, such synchronized sampling may suffer from slowdown when different environments in different processes have large execution differences, which is alleviated by tricks such as allocating available CPU cores used for the environments evenly. For the learners, an efficient asynchronous updating is performed by using lock to prevent other reading or writing requests, and dividing the parameters into disjoint chunks to be updated separately. The implemented asynchronous version of PPO, i.e., APPO, learns successful policies in Atari games in mere minutes.

With the above synchronized sampling in [38], inference times will be largely reduced, but the communication burden between learners and actors will be a big problem when the networks are huge. Espeholt et al.[9] proposed scalable, efficient, deep-RL (SEEDRL), which features centralized inference and an optimized communication layer called gRPC. As shown in Fig. 9, the communication between learners and actors is mere states and actions, which largely reduce the communication burden. Furthermore, a streaming gRPC is used where the communication from actor to learner is kept open with metadata sent only once, which has minimal latency for the connection. The authors implemented policy gradients and Q-learning based algorithms and tested them on the Atari-57, DeepMind lab and Google research football environments, and a 40% to 80% cost reduction was obtained, showing great improvements.

In summary, Gorila builds the basis of most DDRL algorithms with four key components, i.e., parallel actors, parallel learners, a distributed neural network and a distributed store. By considering prioritized and recurrent experience replays for policy enhancement, APE-X and R2D2 are developed, respectively. To make full use of multi-core CPU, an A3C method is designed and deployed in a machine, which is further improved by GA3C to put parameters updating in the most suitable device, i.e., GPU. Increasing the model size will largely limit the

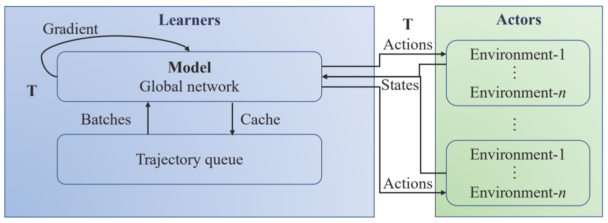


Fig. 9 Basic framework of SEEDRL

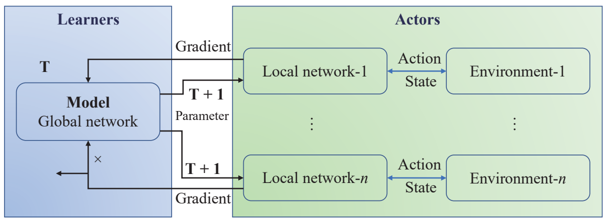


Fig. 11 Basic framework of DPPO

data throughput when putting the gradient calculation on the actor side as in A3C, so IMPALA puts the gradient calculation in the learner side, and uses V-trace to remedy the policy lag with learners and actors distributed in multiple machines. Compared to IMPALA, which uses each inference for each environment, APPO reduces the inference times with synchronized sampling. With the above synchronized sampling, SEEDRL further reduces the communication burden between learners and actors by just exchanging states and actions with an efficient streaming gRPC.

3.2.2 Synchronous based

As an alternative to A3C[32], Clemente et al.[39] found that a synchronous version, i.e., advantage actor-critic (A2C), can better use GPU resources, which should perform well with more actors. In the implementation of A2C, e.g., PAAC, a coordinator is utilized to wait for all gradients of the actors before optimizing the global network. As shown in Fig. 10, learners update the policy parameters before all the trajectories are collected, i.e., the job of actors is done, and when the learners are updating the policy, the trajectory sampling is stopped. As a result, all actors are coordinated to obtain the same global network to interact with environments in the following steps.

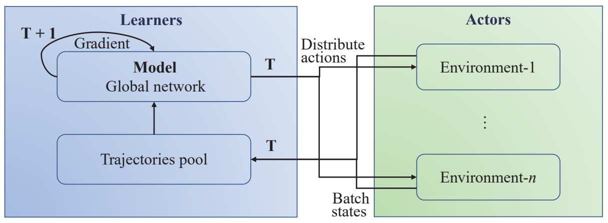


Fig. 10 Basic framework of PAAC

As an alternative to the A2C algorithm in handling continuous action space, the PPO algorithm[24] shows great potential due to its trust region constraint. Heess et al.[34] proposed large scale reinforcement learning with distributed PPO, i.e., DPPO, which has both synchronous and asynchronous versions and shows better performance with the synchronous update. As shown in Fig. 11, the implementation of DPPO is similar to A3C but with synchronization when updating the policy neural network. However, synchronization will limit the throughput of the entire system due to the different rhythms of the actors. The authors use a threshold for the number of actors whose gradients must be available for the learners, which makes the algorithm scale to a large number of actors.

Different from the DPPO algorithm, where a parameter server is applied for distributed neural network updating, Wijmans et al.[40] further proposed a decentralized DPPO framework, i.e., DDPPO, which exhibits near-linear scaling to the GPUs. As shown in Fig. 12, a learner and an actor are bundled together as a unit to perform trajectory collection and gradient calculation. Then, gradients from all the units are gathered together through some reduce operations, e.g., ring allreduce, to update the neural networks, which ensures that the parameters are the same for all the units. Note that to alleviate the synchronization overhead when performing trajectory collection in parallel units, similar strategies such as in DPPO are used to discard certain percentages of trajectories in several units. For implementation, the ring allreduce and trajectory recording operations can be achieved through public tools such as APIs in PyTorch. Experiments show a speedup of 107x on 128 GPUs over a serial implementation.

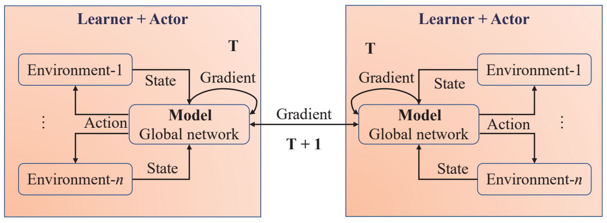


Fig. 12 Basic framework of DDPPO

In summary, PAAC and DPPO are similar with synchronous updating. However, DPPO introduces more tricks such as using a part of the data instead of waiting all the data to be ready, which will improve the throughput of the whole system. DDPPO is a different framework, where an actor and a learner bundled together serve as a unit, and a ring allreduce operation is used to synchronously update all the network parameters among the above units. This is different from the parameter server framework.

3.2.3 Discussion

Single machine or multiple machines. In the beginning of developing DDRL algorithms, researchers make previous non-distributed DRL methods distributed using one machine. For example, the parallel of several typical actor-critic algorithms is designed to use the multi-process of CPUs, e.g., A3C[32], and PAAC[39]. Recently, researchers have aimed to improve the data throughput of the whole DDRL system, e.g., IMPALA[10], and SEEDRL[9], which serves as a basic infrastructure for training complex games AI such as AlphaStar and OpenAI Five. These systems can usually make use of multiple machines. However, early DDRL algorithms designed for a single machine can also be deployed in multiple machines when communications between machines are solved, which is relatively simple by using open sourced tools.

Exchanging trajectories or gradients. Learners and actors serve as basic components for DDRL algorithms, and the data communicated between them can be trajectories or gradients based on whether to put the gradient calculation on the actor or learner side. For example, actors of A3C[32] are in charge of trajectory collection and gradient calculation, and the gradients are then sent to learners for policy updates, which make simple operations such as sum operations. Since gradient calculation is time-consuming, especially when the policy model is large, the calculating load between the learners and actors will be unbalanced. Accordingly, an increasing number of DDRL algorithms have put gradient calculation on the learner side by using more suitable devices, i.e., GPUs. For example, in higher data throughput DDRL frameworks such as IMPALA[10], learners are in charge of gradient calculation, and actors are in charge of trajectory collection.

Synchronized or independent inference. When actors are collecting trajectories by interacting with the environment, actions should be inferred. Basically, when performing a step on an environment, there should be one inference. Previous DDRL methods usually maintain an environment for an actor, where action inference is performed independently from other actors and environments. With the increasing number of environments to collect trajectories, it is resource consuming, especially when only CPUs are used on the actor side. By putting the inference on the GPU side, the resources are also largely wasted because the batch size of the inference is one. To cope with the above problems, many DDRL frameworks use an actor to manage several environments and perform synchronized action inference. For example, APPO[38] and SEEDRL[9] introduce synchronization to collect states and distribute actions obtained by environments and an actor, respectively.

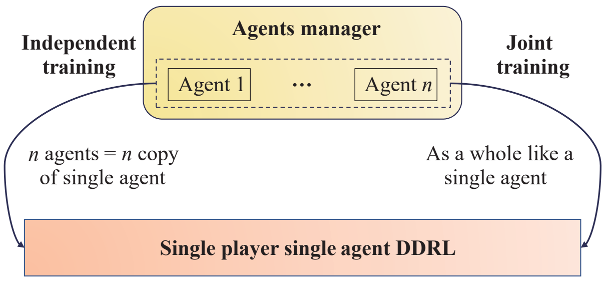
Asynchronous or synchronous DDRL. Both synchronous based and asynchronous based DDRL algorithms have advantages and disadvantages. For asynchronous DDRL algorithms, the global policy usually does not need to wait all the trajectories or gradients, and data collection conventionally does not need to wait the latest policy parameters. Accordingly, the data throughput of the whole DDRL system will be large. However, there exists a lag between the global policy and behavior policy, and such a lag is usually a problem for on-policy based reinforcement learning algorithms. DDRL frameworks such as IMPALA[10] introduces V-trace, and GA3C[37] brings in small term ε to alleviate the problem, but those kinds of methods will be unstable when the lags are large. For synchronous DDRL algorithms, synchronization among trajectories or gradients is required before updating the policy. Accordingly, waiting time is wasted for actors or learners when one side is working. However, synchronization makes the training stable, and it is easier to be implemented such as DPPO[34] and DDPPO[40].

Others. Usually, multiple actors can be implemented with no data exchange, because their jobs, i.e., trajectory collection, can be independent. As for learners, most methods only maintain one learner, which will be enough due to limited model size and especially the limited trajectory batch size. However, large batch size is claimed to be important for complex games[6], and accordingly multiple learners become necessary. In the multiple learners case, usually a synchronization should be performed before updating the global policy network. Generally, a sum operation can handle the synchronization, but it is time consuming when the learners are distributed in multiple machines. An optimal choice is proposed in [40], where the ring allreduce operation can nicely deal with the synchronization problem, and an implementation of [40] is easy by using a toolbox such as Horovod[11]. On the other hand, when the model size is large and a GPU cannot load the whole model, a parameter-server framework[8, 33] based learner can be a choice, which may be combined with the ring allreduce operation to handle the large model size and large batch size challenge.

Brief summary. Finally, when a DDRL algorithm is needed, how to select a proper or efficient method largely relies on the available computing resources, the policy model size and the environment size. If there is only one machine with multiple CPU cores and GPUs, no extra communication is required except for the data exchange between the CPU and GPUs. However, if there are multiple machines, data exchange should be considered, which may be the bottleneck of the whole system. When the policy model is large, the exchange of the model between machines is time consuming, so methods such as SEEDRL[9] are proper due to only states and actions being exchanged. However, if the policy model is small, frequently exchanging trajectories will be time consuming, and methods such as DDPPO[40] will be a choice. When the environment size is large, massive CPU resources will be used to start-up environments, and a few GPUs will be competent for policy updating. Accordingly, DDRL methods such as IMPALA[10] will be suitable because a high data throughput can be obtained. Finally, there may be no best DDRL methods for any learning environment, and researchers can choose one that best suits their tasks.

3.3 Agents cooperation types

When confronting single agent reinforcement learning, previous DDRL algorithms can be easily used. However, when there are multiple agents, distributed multi-agent reinforcement learning algorithms are required to train multiple agents simultaneously. Accordingly, previous DDRL algorithms need to be modified to handle the multiple agents case. Based on current training paradigms for multi-agent reinforcement learning, agents cooperation types can be classified into two categories, i.e., independent training and joint training, as shown in Fig. 13. Usually, an agents manager is added to control all the agents in a game. Independent training trains each agent by considering other learning agents as part of the environment, and joint training trains all the agents as a whole by using typical multi-agent reinforcement learning algorithms.


Agents manager
Independent training
Joint training
Agent 1
···
Agent n
n agents = n copy of single agent
As a whole like a single agent
Single player single agent DDRL
Fig. 13 Basic framework of agents training

3.3.1 Independent training

Independent training makes n agents train as n independent training, and accordingly previous DDRL algorithms can be used with only a few modifications. The agents manager is mainly used to bring other agents' information, e.g., actions, into the current DDRL training of an agent because the dynamics of an environment should be driven by all agents. Considering the requirement of agents cooperation, independent training makes a greater contribution to promoting cooperation among independent agents.

Jaderberg et al.[41] proposed For the Win (FTW) agents for the Quake III Arena in capture the flag (CTF) mode, where several agents cooperate to fight another camp. To train scalable agents that can cooperate with any other agents even for unseen agents, the authors train agents independently, where a population of independent agents are trained concurrently, with each participating in thousands of parallel matches. To handle thousands of parallel environments, an IMPALA[10] based framework is used[1]. For the cooperation problem, the authors design rewards based on several marks between the agents cooperating to promote the emergence of cooperation. More specifically, all the agents share the same final global reward, i.e., win or lose. In addition, intermediate rewards are learned based on several events that consider teammates' actions, such as teammates capturing the flag and teammates picking up the flag.

[1] Mainly based on their codes released.

Berner et al.[6] proposed OpenAI Five for Dota2, where five heroes cooperate to fight another cooperated five heroes. In their AI, each hero is modeled as an agent and trained independently. To address large parallel environments for generating a batch size of more than a million time steps, a SEEDRL[9] framework is used. Unlike [41], which uses different policy networks for different agents, OpenAI Five uses the same policy for different agents, which may promote the emergence of cooperation. The action differences lie in the feature design, where different agents in Dota2 share almost the same features but with specific features such as hero ID. Finally, similar to [41], who designed rewards to promote cooperation, the authors use a weighted sum of individual and team rewards, which are given by following experience of human players, e.g., gaining resources and killing enemies.

Ye et al.[36] proposed JueWu[2] for Honor of Kings, which is a similar game compared to Dota2 but played on mobile devices instead of computer devices. As in [6], a SEEDRL[9] framework is adopted. In addition, the authors also use the same policy for different agents as in [6]. The policy network is different, where five value heads are used due to a deeper consideration of the game characteristics. The key difference between [6] is the training paradigm used to scale to a large number of heroes, which is not the main scope of this paper, and readers can refer to the original paper for more details.

Zha et al.[42] proposed DouZero for DouDiZhu, where a landlord agent and two peasant agents are confronting for a win. Three agents using three policy networks are trained independently, as in [41]. A Gorila[8] based DDRL algorithm is used to train the three agents with a single server. Cooperation between the peasants agents emerges with increasing training epochs.

Baker et al.[43] proposed multi-agent autocurricula for game hide-and-seek to study emergent tool use. As in [6], a SEEDRL[9] framework is used, and the same policy for different agents is used for training. In addition, the authors test using distinct policies for different agents, showing similar results but reduced sample efficiency.

In summary, previous DDRL algorithms or frameworks can be easily used for independent training with some modifications: making the learners maintain one or multiple policy networks for different agents, and driving the environment dynamics with the actions of all agents. The communication burden will increase because an actor of an agent must receive the actions of the other agents distributed in different machines. This situation will be worse if different agents use different parameters, such as DouZero and FTW, compared to agents sharing the same parameters, such as OpenAI Five and JueWu.

3.3.2 Joint training

Joint training trains all agents as a whole using typic-

[2] A recognized name.

al multi-agent reinforcement learning algorithms like a single agent. The difference is the trajectories collected, which have all the agents' data instead of just an agent. The agents manager can be designed to handle multi-agent issues, such as communication, and coordination, to further accelerate training. However, current multi-agent DDRL algorithms mostly consider a simple method, i.e., actor parallelization to collect enough trajectories. Accordingly, most previous DDRL algorithms can be easily implemented.

The implementation of QMIX[44], a popular Q value factorization-based multi-agent reinforcement learning algorithm, is implemented using multi-processing to interact with the environment[30]. Another example is RLlib[13], a part of the open source Ray project[12], which makes abstractions for DDRL and implements several jointly trained multi-agent reinforcement learning algorithms, e.g., QMIX and PPO with a centralized critic. Generally, previous joint training is similar to single agent training in the field of DDRL, but consideration of parallelized training for issues such as communication and coordination among agents may further accelerate the training speed.

In summary, joint training in the field of DDRL is rarely investigated, and only actor parallelization is widely used. In these methods, since multiple agents training is similar to the training of an agent, the implementation of actor parallelization may not require modifications of previous DDRL algorithms or frameworks.

3.3.3 Discussion

For independent training, even though different agents are trained independently, different methods take into account problems such as feature engineering and reward reshaping to promote cooperation. Since different agents are trained by making other agents part of the environment, conventional DDRL algorithms can be used without many modifications. From successful agents such as OpenAI Five and JueWu, we can see that SEEDRL or its revised versions are a good choice. Joint training is far from satisfactory because there is considerable room to improve parallelism among agents by properly considering the multi-agent issues such as communication when designing actors and learners.

3.4 Players evolution types

In most cases, we have no opponents to drive the capacity growth for a player[3]. To handle such a problem, the player usually fights against itself to increase its ability. For example, AlphaGo[1] uses DDRL and self-play for superhuman AI learning. Based on current learning paradigms for players evolution, current methods can be classified into two categories, i.e., self-play and population-play, as shown in Fig. 14. To maintain the players for evolution, a players manager is required for the DDRL algorithms for one or multiple players. Self-play maintains a player and its past versions, whereas, population-play maintains several distinct players and their past versions.

[3] Here player means a side for a game, which may control one agent such as Go or multiple agents such as Dota2.

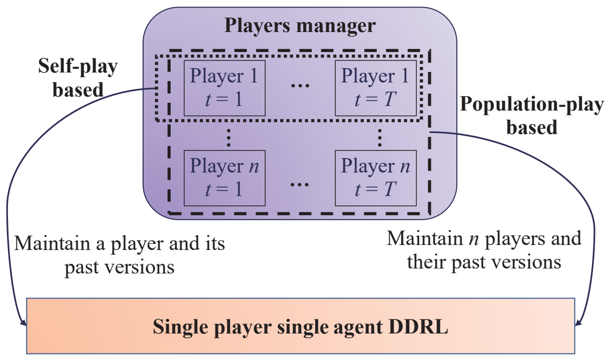


Fig. 14 Basic framework of player iteration

3.4.1 Self-play based evolution

Self-play has become a popular tool since the success of the AlphaGo series[1, 2, 45], which trains a player by fighting against itself. In an iteration or called generation of the player, the current version is trained based on previous DDRL algorithms by using one or some of its previous versions as opponents. The players manager decides which previous versions are used as opponents.

In JueWu[36], developed for the Honor of Kings, a naive self-play is used for two players (each controlling five agents) using the same policy. An SEEDRL[9] DDRL algorithm is used, and the self-play is used in the fixed lineup and random lineup stages for handling a large hero pool size. Players trained for hide-and-seek[43] are similar to JueWu[36], where a SEEDRL[9] DDRL algorithm and a naive self-play are used to prove the multi-agent auto-curricula. Another similar example is Suphx[5] proposed for Mahjong, which uses self-play for a player to confront the other three players (use the same policy). For the DDRL algorithm, an IMPALA[10] framework is applied for training each generation.

In OpenAI Five[6] designed for Dota2, a more complex self-play is used for two players (each controls five agents) using the same policy. In each generation, instead of fighting against the current generation, such as naive self-play, the player trains the current policy against itself for 80% of games, and against its previous versions for 20% of games. Specifically, a dynamic sampling system is designed to select the past versions based on their dynamically generated quality score, which claims to alleviate cyclic strategies problem. As for the basic DDRL algorithm, a SEEDRL[9] framework is used for all the generations of players training.

In summary, the players manager maintains the past generations of the player, which serve as the opponent to drive the dynamics of the environment. Accordingly, conventional DDRL algorithms or frameworks should be modified like independent training in agents cooperation. For naive self-play, the current generation is used to train itself, whereas, revised self-play usually maintains an evaluation matrix to record the ability of each generation, based on which, a sampling strategy is used to select the opponent.

3.4.2 Population-play based evolution

Population-play can be seen as an advanced self-play, where more than one player and their past generations should be maintained for players evolution. It can be used for several cases: The policy used is different for different players (e.g., a landlord and two peasant players in DouZero); some auxiliary players are introduced for the target player to overcome game-theoretic challenges (e.g., main exploiter and league exploiter players in AlphaStar); parallel players are used with consistent roles to support concurrent training and to alleviate instability of self-play (e.g., populations in FTW).

In DouZero[42] designed for DouDiZhu, a landlord and two peasant players are trained simultaneously, where their current generations fight against each other to collect trajectories and to train the players. The basic DDRL algorithm is Gorila[8], which runs on a single machine, based on which all three players are trained asynchronously.

In AlphaStar[7] developed for StarCraft, the players manager maintains three main players for three different races, i.e., Protoss, Terran, and Zerg. In addition, for each race, several auxiliary players are designed, i.e., one main exploiter player and two league exploiter players. Those auxiliary players help the main player determine weaknesses and help all the players find systemic weaknesses. The authors claim that using such a population addresses the complexity and game-theoretic challenges of the StarCraft. For the DDRL algorithm, SEEDRL[9] is utilized to support large system training. Commander[46] is similar to [7], and more exploiter players are used.

In FTW[41] designed for Capture the Flag (CTF), the players manager maintains a population of players who cooperate and confront each other to learn scalable bots. The positions of all the players are the same, and a population-based training method is designed to adjust players with worse performance to improve the ability of all the players. For the basic DDRL algorithm, the IMPALA[10] method is used to have large data throughput to train tens of players.

In summary, the players manager maintains all the players and their past generations. Accordingly, a more complex evaluation matrix should be used to record the performances against past generations of the other players and the player itself. Compared to DouZero, the players in AlphaStar have different opponent selection strategies, which will largely increase the communication burden between learners and actors and the computational complexity of players manager in DDRL. For FTW, a population of players cooperate and compete against each other, which also brings in the difficulty of evaluating the performance of each player in the players manager.

3.4.3 Discussion

Self-play has a long history in multi-agent settings (different players naturally cause multiple agents), where early work explored it in genetic algorithms[47]. It has become very popular with the combination of deep reinforcement learning leading to the success of the AlphaGo series[1, 2], which is widely used for various AI systems, such as Libratus[48], DeepStack[49] and OpenAI Five[6], to reach human-level performance. In [50], the authors give a brief survey of self-play in reinforcement learning, where its major themes, criteria and techniques are introduced. Even though some important concepts, such as convergence, Pareto efficiency, and Nash equilibrium, are discussed, current self-play in DDRL largely depends on heuristic design based on the characteristics of the environment.

On the other hand, population-play can be seen as advanced self-play, which maintains more players to achieve ability improvement. Since there is more than one player to learn, the evolution becomes harder because different players need to be properly improved to ensure the improvement of the whole system. Accordingly, several challenges are raised when using population-play, such as the block for asymmetric games[27, 51]. Despite this, current works use population-play to accelerate training, overcome game-theoretic challenges, or just handle the problem that requires distinct players.

Compared with self-play, population-play is more flexible, and can handle diverse situations. However, self-play is easy to implement, and has proven its potential in complex games such as Dota2. Overall, there is no conclusion regarding whether self-play or population-play is better in games, and researchers can select self-play or population-play DDRL based on their specific requests.

4 Typical DDRL toolboxes

DDRL is important for complex environments using reinforcement learning as solvers, and several useful toolboxes have been released to help researchers reduce development costs. In this section, we analyze several typical toolboxes, hoping to give a clue when researchers are making a selection among them.

4.1 Typical toolboxes

Ray[12] is a distributed framework consisting of two main parts, i.e., a system layer to implement tasks scheduling and data management, and an application layer to provide high-level APIs for various applications. Using these APIs, researchers can easily implement a DDRL method without considering the node/machine connections and scheduling different calculations. For example, using @ray.remote as a decorator of a function, a remote function can be obtained, and the results or futures can be calculated with @ray.get and @ray.wait. Furthermore, an RLLib[13] package is introduced on top of Ray to support reinforcement learning such as A3C, APEX and IMPALA. In addition, several built-in multi-agent DDRL algorithms are provided, such as QMIX[44] and MADDPG[52]. Users can use and revise these DDRL algorithms with the above APIs.

Acme[53] is designed to enable distributed reinforcement learning to promote the development of novel RL agents and their applications. It involves many separate (parallel) acting, learning and diagnostic and helper processes, which are key building blocks for a DDRL system. Using these templates and specifically designed APIs, such as make_distributed_experiment, users can focus on reinforcement learning algorithm design instead of parallelization design. Furthermore, one of the main contributions is the in-memory storage system, called Reverb, which is a high-throughput data system that is suitable for experience replay based reinforcement learning algorithms. With the aim of supporting agents at various scales of execution, many mainstream DDRL algorithms have been implemented, i.e., online reinforcement learning algorithms such as Deep Q-networks[28], R2D2[35] and IMPALA[10], offline reinforcement learning such as behavior cloning and TD3[54], imitation learning such as adversarial imitation learning[55] and soft Q imitation learning[56].

Tianshou[57] is a highly modularized Python library that uses PyTorch for DDRL. Its main characteristic is the design of building blocks that support more than 20 classic reinforcement learning algorithms with distributed versions through a unified python interface. Specifically, building blocks for DDRL are provided, which can be used for fast prototyping. Since Tianshou focuses on small-to-medium-scale applications of DDRL with only parallel sampling, it is a lightweight platform that is research-friendly. It is claimed that Tianshou is easy to install, and users can apply Pip or Conda to accomplish installation on platforms covering Windows, macOS and Linux.

TorchBeast[58] is another DDRL toolbox that is based on PyTorch to support fast, asynchronous and parallel training of reinforcement learning agents. The authors provide two versions, i.e., pure-Python MonoBeast and multi-machine high-performance PolyBeast with several parts being implemented with C++. Users only require Python and PyTorch to implement DDRL algorithms. For example, using the threading function of Python to start threads of actors, where trajectories can be queued for learners. In the toolbox, IMPALA is supported and tested with the classic Atari suite.

MALib[59] is a scalable and efficient computing framework for population-based multi-agent reinforcement learning algorithms. Using a centralized task dispatching model, it supports self-generated tasks and heterogeneous policy combinations. In addition, by abstracting DDRL algorithms using actor-evaluator-learner, a higher parallelism for learning and sampling is achieved. The authors also claimed to have efficient code reuse and flexible deployments due to the higher-level abstractions of multi-agent reinforcement learning. In the released code, several popular reinforcement learning environments, such as Google research football and SMAC, are supported and typical population based algorithms, such as policy space response oracle (PSRO)[60] and pipeline-PSRO[61], are implemented. With these examples (highly abstracted), users may replace the environments and reinforcement learning algorithms for population based DDRL.

SEED[9] is a scalable and efficient deep reinforcement learning toolbox, as described in Section 3.2.1. Generally, it is verified on the tensor processing unit (TPU) device, which is a special chip customized by Google for machine learning. Typical DDRL algorithms are implemented, e.g., IMPALA[10] and R2D2[35], which are tested on four classical environments, i.e., Atari, DeepMind lab, Google research football and Mujoco. Distributed training is supported using the cloud machine learning engine of Google, and users can follow highly abstracted examples to implement their own reinforcement learning algorithms.

4.2 Discussion

Before comparing different kinds of toolboxes, we want to claim that there are no best DDRL toolboxes for any requirements, but the most suitable one depends on specific goals.

Tianshou and TorchBeast are lightweight platforms that support several typical DDRL algorithms. Users may easily modify the released codes by referring to the prototypes or examples using the PyTorch deep learning library. The user-friendly features make these toolboxes popular. However, even though those toolboxes are highly modularized, the scalability to a large number of machines for performing large learner parallel and actor parallel is not tested, and bottlenecks may appear with an increasing number of machines.

Ray, Acme and SEED are relatively large toolboxes that can theoretically support any DDRL algorithm with certain modifications. Using their open projects, users can utilize multiple machines to implement high data throughput DDRL algorithms. Moreover, multiple agents training and multiple players evolution can be achieved, such as for AlphaStar. However, modifications and debugging are not easy due to code nesting for brief abstractions and function calls. For example, when adding a new function (e.g., network parameter disturbance after exchanging with other players), the modifications may involve all abstractions.

MALib is similar to Ray, Acme and SEED, which is a specially designed DDRL toolbox for population-based multi-agent reinforcement learning. With their APIs, users may implement population based multi-agent reinforcement learning algorithms such as fictitious self-play[62] and PSRO. Similar to the previous toolboxes, the modifications are not easy due to deep code nesting. Although experiments for a large number of machines are not tested, this toolbox is fully functional (APIs provided) for various requirements of DDRL algorithms from single player single agent DDRL to multiple players multiple agents DDRL.

In summary, current DDRL toolboxes provide a good support for DDRL algorithms, and several typical testing environments are embedded for performance validation. However, those DDRL toolboxes are either too lightweight for multiple players and multiple agents or too heavy for secondary development, and all toolboxes are not specially designed or tested for complex games, which we think is important because it may require flexible functions for environments and the agents trained. For example, the agents may asynchronously cooperate[63], and the environments for different players are asymmetric. In Section 5, we will design a user-friendly toolbox that focuses on multiple players and multiple agents of DDRL training on complex games.

5 A multi-player multi-agent reinforcement learning toolbox

In this section, we open a multi-player multi-agent reinforcement learning toolbox, M2RL, to support populations of players (with each possibly controlling several agents) for complex games, e.g., Wargame[51]. Note that this project is ongoing, so the main purpose is a preliminary introduction, and we will continue to improve this project. The hyperlink of the project is http://turingai.ia.ac.cn/ai_center/show/14.

5.1 Overall framework

The overall framework is shown in Fig. 15. Each player, consisting of one or multiple agents, has three key components: learner, actor and experience buffer. The multiple concurrently executed actors produce data for learners, which use the current player and other players as opponents based on the choice of players manager. The experience buffer is used to store trajectories of the player to support asynchronous or synchronous training. The learner for each player is used to update parameters of the player and send parameters to the actors. Apart from the above basic factors, players manager maintains self-play and population-play, which has two key parts: evaluating players and choosing opponent players.

More specific details of M2RL are shown in Fig. 16. To make M2RL easy to use for complex games, we design each part in a relatively flexible manner. For players manager, payoff data are updated once new generations of different players are added, based on which, opponents for each player can be sampled to train its next generation. A sampling function can be modified with any

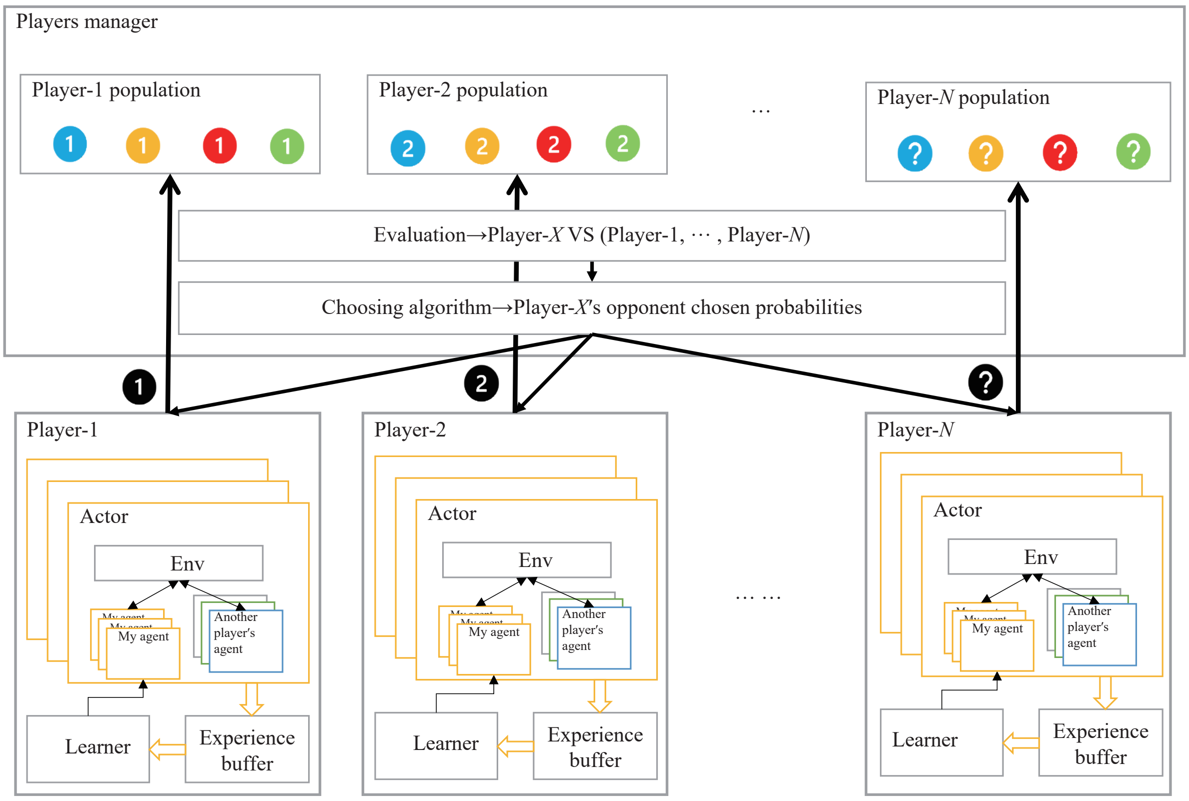


Fig. 15 Basic framework of the proposed multi-player multi-agent reinforcement learning toolbox M2RL

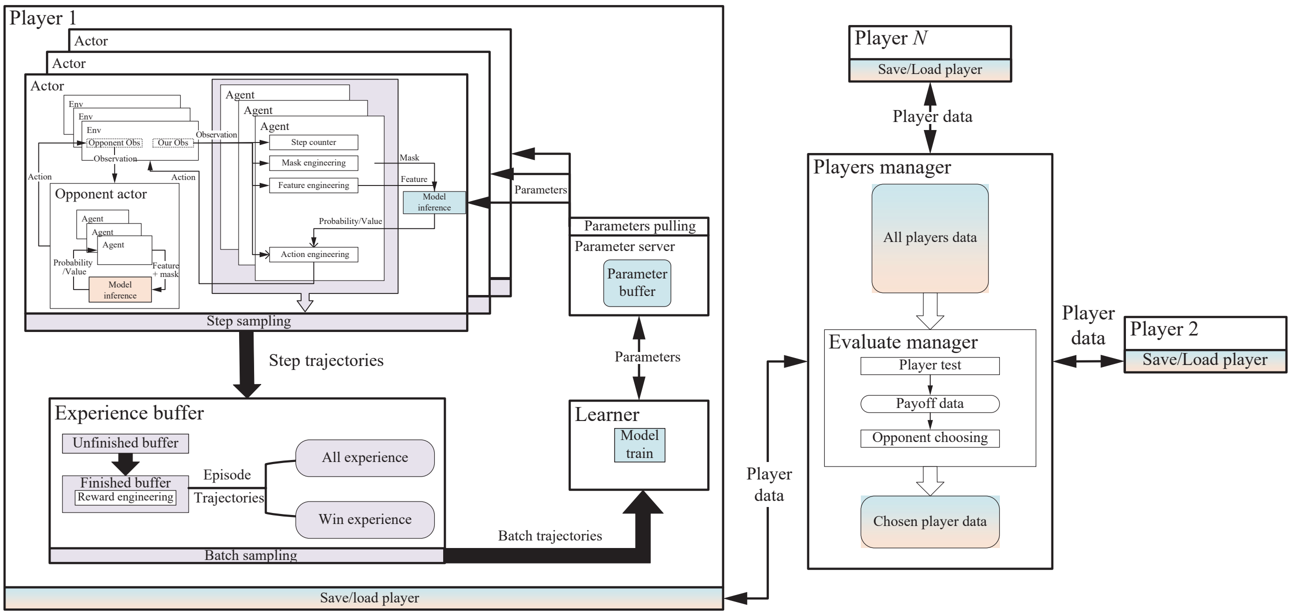


Fig. 16 Specific details of the proposed multi-player multi-agent reinforcement learning toolbox M2RL

form to implement any kind of self-play or population-play strategy. For each player, the actors are in charge of generating data with two kinds of inferences after receiving the observations, i.e., the current player and the opponent players from the players manager. The inference for opponent players is implemented with remote functions by loading the players. To support complex observations and actions, several engineering functions are provided to encode features and explain the actions for the policy network and the environment, respectively. For the experience buffer, we provide an extra buffer to revise the original buffer, which will be helpful for complex reinforcement learning algorithms, e.g., asynchronous multi-agent cooperation[63]. Finally, a learner is used to update the current player with a parameter server to store the distributed parameters.

In summary, the main characteristics of M2RL are as follows:

1) The players manager evaluates all saved players (including their past versions) using their confrontation results, based on which, various opponent selection methods can be implemented to promote players' evolution, e.g., revised self-play in OpenAI Five[6] and prioritized fictitious self-play in AlphaStar[7]. In addition, different solutions after training can be obtained such as Nash equilibrium and evolutionarily stable strategies[64].

2) Each player maintains its own learner, actor and experience buffer, making distinct players training possible, e.g., red and blue players in Wargame[51]. Considering that the game is complex with different observation and action spaces compared to the OpenAI gym, feature engineering and mask engineering are used in the framework. In addition, the experience buffer is revised to change an unfinished buffer to a finished buffer, which is very useful for asynchronous multi-agent cooperation[63].

3) There is little code nesting but mainly personalized functions or classes as interfaces to be modified, and the underlying codes are based on user-friendly remote functions from Ray, which are easy to deploy, revise and use. More specifically, we can make full use of computing resources by segmenting a GPU to several parts and assigning each part to different tasks, which is important for complex games under limited computing resources.

5.2 A case

Wargame is a very complex game similar to Dota2 and StarCraft, and it is not conquered like the breakthrough of OpenAI Five and AlphaStar[27]. Accordingly, we believe it will be a good testing environment to verify the usefulness of M2RL. In a Wargame map[4], the red player controls several fighting units to confront the blue player who also controls several units. The game is asymmetric because players have distinct strategy spaces, and usually the blue player has more forces, while the red player has a vision advantage. Please refer to [51] for more details of the Wargame.

We can naturally model Wargame as a two players multiple agents problem, where each fighting unit is regarded as an agent. To train two AI bots for the red and blue players, we use several widely adopted settings, such as shared PPO policy for each agent, dual-clip for the PPO and prioritized fictitious self-play in OpenAI Five[6], JueWu[36] and AlphaStar[7]. Each player trains its bot using approximately 200 000 games, and uses 9 500 games for the players manager to evaluate each generation of the player. The computing resources used here are as follows: 2×Intel(R) Xeon(R) Gold 6 240R CPU @ 2.40GHz,

[4] http://wargame.ia.ac.cn/main, ID = 2010431153

4× NVIDIA GeForce RTX 2 080 Ti, and 500 GB memory. With the above resources, the training lasts for five days, and we finally obtain 20 generations for each player. To evaluate the performance of these bots, we use the built-in demo agent as the baseline and bring in three professional-level AI bots designed by teams who have studied Wargame for several years, represented as knowledge_1, knowledge_2 and knowledge_3. It should be noted that those professional AI bots do not participate in training. Similar to the evaluation for AlphaGo[1] and AlphaStar[7], we use Elo as metrics. The detailed codes are released and the results are shown in Fig. 17.

In our experiments, we make no comparison with existing toolboxes because they are not specifically designed for complex games such as Wargame, which require large modifications. For example, the most similar toolbox MALib is claimed to lack enough optimization for GPUs, and the released codes may lack important components, such as experience storage and model call and storage for self-play and population-play. Other similar toolboxes, such as Acme and SEED, are designed for a single agent, and the released codes may lack essential components such as multi-agent population evaluation and evolution. From Fig. 17, it can be seen that with increasing players evolution, the learned policy for each player becomes stronger. This is also verified from Fig. 18 after showing the asymmetric replicator dynamic[64], i.e., the 20th and 19th generations being chosen with increasing evolution for the red and blue players, respectively. Overall, the results show the ability of the proposed M2RL to some extent. Since this project is an ongoing item, the main purpose of this part is an introduction, and we will continue to improve the toolbox in the future, e.g., more testing and comparison on various complex environments.

6 Challenges and opportunities

Many DDRL algorithms and toolboxes have been proposed, which have largely promoted the study of reinforcement learning and its applications. We believe that current methods still suffer from several challenges, which may be future directions. First, current methods rarely consider accelerating complex reinforcement learning algorithms, such as those studying exploration, communication and generalization problems. Second, current approaches mainly use a ring allreduce or parameter server for learners, which seldom handle large model size and batch size situations simultaneously. Third, self-play or population-play are important methods for multiple play-

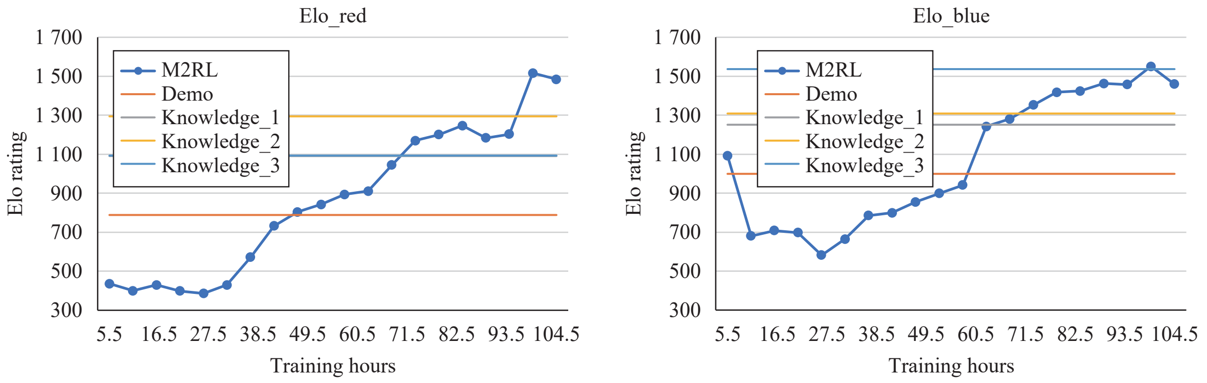


Fig. 17 Elo results of the trained AI bots (red and blue players) based on M2RL. Knowledge_1, knowledge_2 and knowledge_3 are three professional level AI bots. Demo is an AI with a strategy to select the highest priority action when it is possible.

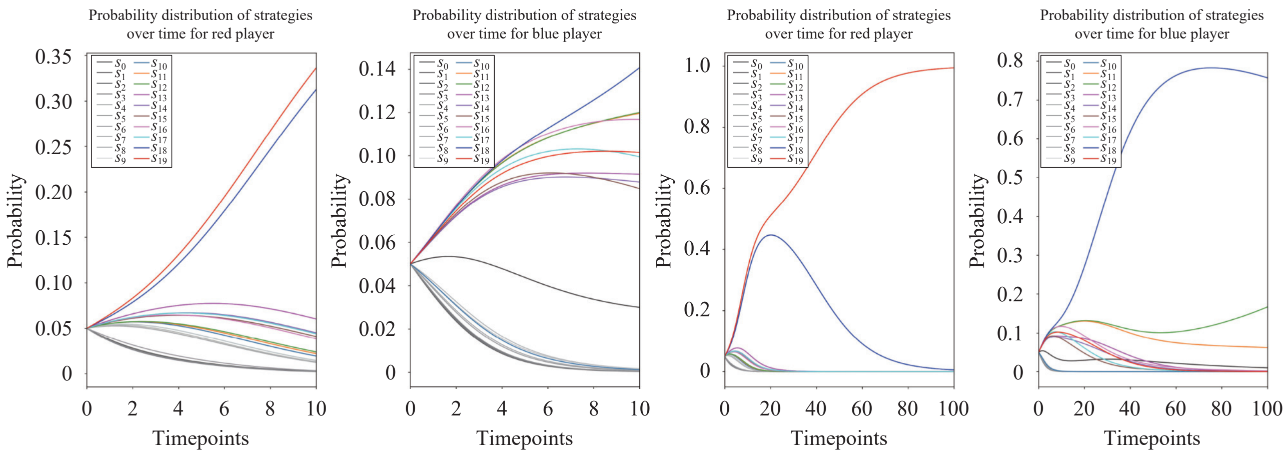


Fig. 18 Asymmetric replicator dynamics for the red and blue players with time =10 and time=100

ers and multiple agents training, which are also flexible without strict restrictions, but deeper study is deficient. Fourth, several famous DDRL toolboxes have been developed, but none of them have been verified with large scale training, e.g., tens of machines for complex games.

DDRL with advanced reinforcement learning algorithms. The research and application of reinforcement learning show explosive growth since the success of AlphaGo. New topics have emerged, such as hierarchical deep reinforcement learning, model-based reinforcement learning, multi-agent reinforcement learning, off-line reinforcement learning, and meta reinforcement learning[16, 18], but DDRL methods have rarely considered those new research areas. Distributed implementation is kind of engineering but not naive. For example, when considering information communication for a multi-agent reinforcement learning algorithm, agents manager should reasonably parallelize agent communication calculation to improve data throughput. Accordingly, how to accelerate advanced reinforcement learning algorithms with distributed implementation is an important direction.

DDRL with large model size and batch size. With the success of foundation models in the field of computer vision and natural language processing, large models in reinforcement learning will be a direction[27]. This requires DDRL methods to handle large model size and batch size situations simultaneously. Currently, the learners in DDRL are based on techniques such as ring allreduce or parameter server, with each having its advantages. For example, a parameter server can store large model in different GPUs, and ring allreduce can quickly exchange gradients between different GPUs. However, none of them are applied for large model sizes and batch sizes in reinforcement learning. Accordingly, how to combine these techniques to fit DDRL algorithms for efficient training is a future direction.

Self-play and population-play based DDRL methods. Self-play and population-play are mainstream evolution methods for reinforcement learning agents, which are widely used in current professional human-level AI systems, e.g., OpenAI Five[6] and AlphaStar[7]. Generally, self-play and population-play have no strict restrictions on the players, which means a player can fight against any past versions for the same player or different players. Currently, the widely used heuristic designs make exploring the best configuration difficult, which also makes designing templates for a toolbox a tricky problem. In the future, self-play and population-play based DDRL methods are worthy of further study, e.g., adaptively determining the best configuration.

Toolboxes construction and validation. Several famous scientific research institutions such as DeepMind, OpenAI, and UC Berkeley, have released toolboxes to support DDRL methods. Most of them use gym to test the performance, such as data throughput, and linearity. However, the environments in gym are relatively small compared with the environments in real world applications. On the other hand, most of the testing use one or two nodes/machines with limited numbers of CPU and GPU devices, making the testing insufficient to discover bottlenecks in the toolboxes. Accordingly, even though most current DDRL toolboxes are highly modularized, the scalability to a large number of machines for performing large learner parallel and actor parallel for complex environments has not been fully tested. Future bottlenecks of the toolboxes may be discovered with large testing.

7 Conclusions

In this paper, we surveyed representative distributed deep reinforcement learning methods. By summarizing key components to form a distributed deep reinforcement learning system, single player single agent distributed deep reinforcement learning methods are compared based on different types of coordinators. Furthermore, by bringing in agents cooperation and players evolution, multiple players multiple agents distributed deep reinforcement learning approaches are presented in detail. To support DDRL implementation, some popular distributed deep reinforcement learning toolboxes are introduced and discussed, based on which, a new multiple players and multiple agents learning toolbox is developed, hoping to assist learning for complex games. Finally, we discuss the challenges and opportunities of this exciting field. Through this paper, we hope to provide a reference for researchers and engineers when they are exploring novel reinforcement learning algorithms and solving practical reinforcement learning problems and conclude challenges and opportunities for future study.

Acknowledgements

This work was supported by Open Fund/Postdoctoral Fund of the Laboratory of Cognition and Decision Intelligence for Complex Systems, Institute of Automation, Chinese Academy of Sciences, China (No. CASIA-KFKT-XDA27040809).

Declarations of conflict of interest

The authors declared that they have no conflicts of interest to this work.

Open Access

References

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, vol. 529, no. 7587, pp. 484–489, 2016. DOI: 10.1038/nature16961.

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. T. Chen, T. Lillicrap, F. Hui, L. Sifre, G. Van Den driessche, T. Graepel, D. Hassabis. Mastering the game of go without human knowledge. *Nature*, vol. 550, no. 7676, pp. 354–359, 2017. DOI: 10.1038/nature24270.

[3] Y. Yu. Towards sample efficient reinforcement learning. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, Stockholm, Sweden, pp. 5739–5743, 2018. DOI: 10.24963/ijcai.2018/820.

[4] X. P. Qiu, T. X. Sun, Y. G. Xu, Y. F. Shao, N. Dai, X. J. Huang. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences*, vol. 63, no. 10, pp. 1872–1897, 2020. DOI: 10.1007/s11431-020-1647-3.

[5] J. J. Li, S. Koyamada, Q. W. Ye, G. Q. Liu, C. Wang, R. H. Yang, L. Zhao, T. Qin, T. Y. Liu, H. W. Hon. Suphx: Mastering mahjong with deep reinforcement learning, [Online], Available: https://arxiv.org/abs/2003.13590, 2020.

[6] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. D. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, S. S. Zhang. Dota 2 with large scale deep reinforcement learning, [Online], Available: https://arxiv.org/abs/1912.06680, 2019.

[7] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Y. Wang, T. Pfaff, Y. H. Wu, R. Ring, D. Yogatama, D. Wünsch, K. Mckinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, D. Silver. Grandmaster level in StarCraft ii using multi-agent reinforcement learning. *Nature*, vol. 575, no. 7782, pp. 350–354, 2019. DOI: 10.1038/s41586-019-1724-z.

[8] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, D. Silver. Massively parallel methods for deep reinforcement learning, [Online], Available: https://arxiv.org/abs/1507.04296, 2015.

[9] L. Espeholt, R. Marinier, P. Stanczyk, K. Wang, M. Michalski. SEED RL: Scalable and efficient deep-RL with accelerated central inference. In *Proceedings of the 8th International Conference on Learning Representations*, Addis Ababa, Ethiopia, 2020.

[10] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, K. Kavukcuoglu. IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures. In *Proceedings of the 35th International Conference on Machine Learning*, Stockholm, Sweden, pp. 1407–1416, 2018.

[11] A. Sergeev, M. Del Balso. Horovod: Fast and easy distributed deep learning in TensorFlow, [Online], Available: https://arxiv.org/abs/1802.05799, 2018.

[12] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. H. Yang, W. Paul, M. I. Jordan, I. Stoica. Ray: A distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, Carlsbad, USA, pp. 561–577, 2018.

[13] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, I. Stoica. RLlib: Abstractions for distributed reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning*, Stockholm, Sweden, pp. 3053–3062, 2018.

[14] M. R. Samsami, H. Alimadad. Distributed deep reinforcement learning: An overview, [Online], Available: https://arxiv.org/abs/2011.11012, 2020.

[15] J. Czech. Distributed methods for reinforcement learning survey. *Reinforcement Learning Algorithms: Analysis and Applications*, B. Belousov, H. Abdulsamad, P. Klink, S. Parisi, J. Peters, Eds., Cham, Switzerland: Springer, pp. 151–161, 2021. DOI: 10.1007/978-3-030-41188-6_13.

[16] K. Arulkumaran, M. P. Deisenroth, M. Brundage, A. A. Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017. DOI: 10.1109/MSP.2017.2743240.

[17] T. M. Moerland, J. Broekens, C. M. Jonker. Model-based reinforcement learning: A survey, [Online], Available: https://arxiv.org/abs/2006.16712, 2020.

[18] S. Gronauer, K. Diepold. Multi-agent deep reinforcement learning: A survey. *Artificial Intelligence Review*, vol. 55, no. 2, pp. 895–943, 2022. DOI: 10.1007/s10462-021-09996-w.

[19] Y. D. Yang, J. Wang. An overview of multi-agent reinforcement learning from game theoretical perspective, [Online], Available: https://arxiv.org/abs/2011.00583, 2021.

[20] T. Ben-Num, T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys*, vol. 52, no. 4, Article number 65, 2020. DOI: 10.1145/3320060.

[21] W. Wen, C. Xu, F. Yan, C. P. Wu, Y. D. Wang, Y. R. Chen, H. Li. TernGrad: Ternary gradients to reduce communication in distributed deep learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, Long Beach, USA, pp. 1508–1518, 2017.

[22] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, A. Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neur-*

al Information Processing Systems, Lake Tahoe, USA, pp. 1223–1231, 2012.

[23] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. F. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Q. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Q. Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems, [Online], Available: https://arxiv.org/abs/1603.04467, 2016.

[24] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov. Proximal policy optimization algorithms, [Online], Available: https://arxiv.org/abs/1707.06347, 2022.

[25] J. Park, S. Samarakoon, A. Elgabli, J. Kim, M. Bennis, S. L. Kim, M. Debbah. Communication-efficient and distributed learning over wireless networks: Principles and applications. *In Proceedings of the IEEE*, vol. 109, no. 5, pp. 796–819, 2021. DOI: 10.1109/JPROC.2021.3055679.

[26] T. C. Chiu, Y. Y. Shih, A. C. Pang, C. S. Wang, W. Weng, C. T. Chou. Semisupervised distributed learning with non-IID data for AIoT service platform. *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 9266–9277, 2020. DOI: 10.1109/JIOT.2020.2995162.

[27] Q. Y. Yin, J. Yang, K. Q. Huang, M. J. Zhao, W. C. Ni, B. Liang, Y. Huang, S. Wu, L. Wang. AI in human-computer gaming: Techniques, challenges and opportunities. *Machine Intelligence Research*, vol. 20, no. 3, pp. 299–317, 2023. DOI: 10.1007/s11633-022-1384-6.

[28] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. DOI: 10.1038/nature14236.

[29] Y. Burda, H. Edwards, A. Storkey, O. Klimov. Exploration by random network distillation. In *Proceedings of the 7th International Conference on Learning Representations*, New Orleans, USA, 2019.

[30] M. Samvelyan, T. Rashid, C. S. de Witt, G. Farquhar, N. Nardelli, T. G. J. Rudner, C. M. Hung, P. H. S. Torr, J. N. Foerster, S. Whiteson. The starcraft multi-agent challenge. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, Montreal, Canada, pp. 2186–2188, 2019.

[31] M. Lanctot, E. Lockhart, J. B. Lespiau, V. Zambaldi, S. Upadhyay, J. Pérolat, S. Srinivasan, F. Timbers, K. Tuyls, S. Omidshafiei, D. Hennes, D. Morrill, P. Muller, T. Ewalds, R. Faulkner, J. Kramár, B. De Vylder, B. Saeta, J. Bradbury, D. Ding, S. Borgeaud, M. Lai, J. Schrittwieser, T. Anthony, E. Hughes, I. Danihelka, J. Ryan-Davis. OpenSpiel: A framework for reinforcement learning in games, [Online], Available: https://arxiv.org/abs/1908.09453, 2020.

[32] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning*, New York City, USA, pp. 1928–1937, 2016.

[33] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. van Hasselt, D. Silver. Distributed prioritized experience replay. In *Proceedings of the 6th International Conference on Learning Representations*, Vancouver, Canada, 2018.

[34] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Y. Wang, S. M. Ali Eslami, M. A. Riedmiller, D. Silver. Emergence of locomotion behaviours in rich environments, [Online], Available: https://arxiv.org/abs/1707.02286, 2017.

[35] S. Kapturowski, G. Ostrovski, J. Quan, R. Munos, W. Dabney. Recurrent experience replay in distributed reinforcement learning. In *Proceedings of the 7th International Conference on Learning Representations*, New Orleans, USA, 2019.

[36] D. H. Ye, G. B. Chen, W. Zhang, S. Chen, B. Yuan, B. Liu, J. Chen, Z. Liu, F. H. Qiu, H. S. Yu, Y. Y. T. Yin, B. Shi, L. Wang, T. F. Shi, Q. Fu, W. Yang, L. X. Huang, W. Liu. Towards playing full moba games with deep reinforcement learning. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, Vancouver, Canada, pp. 621–632, 2020.

[37] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons, J. Kautz. Reinforcement learning through asynchronous advantage actor-critic on a GPU. In *Proceedings of the 5th International Conference on Learning Representations*, Toulon, France, 2017.

[38] A. Stooke, P. Abbeel. Accelerated methods for deep reinforcement learning, [Online], Available: https://arxiv.org/abs/1803.02811, 2019.

[39] A. V. Clemente, H. N. Castejón, A. Chandra. Efficient parallel methods for deep reinforcement learning, [Online], Available: https://arxiv.org/abs/1705.04862, 2017.

[40] E. Wijmans, A. Kadian, A. Morcos, S. Lee, I. Essa, D. Parikh, M. Savva, D. Batra. DD-PPO: Learning near-perfect pointgoal navigators from 2.5 billion frames. In *Proceedings of the 8th International Conference on Learning Representations*, Addis Ababa, Ethiopia, 2020.

[41] M. Jaderberg, W. M. Czarnecki, I. Dunning, L. Marris, G. Lever, A. G. Castañeda, C. Beattie, N. C. Rabinowitz, A. S. Morcos, A. Ruderman, N. Sonnerat, T. Green, L. Deason, J. Z. Leibo, D. Silver, D. Hassabis, K. Kavukcuoglu, T. Graepel. Human-level performance in 3D multiplayer games with population-based reinforcement learning. *Science*, vol. 364, no. 6443, pp. 859–865, 2019. DOI: 10.1126/science.aau6249.

[42] D. C. Zha, J. R. Xie, W. Y. Ma, S. Zhang, X. R. Lian, X. Hu, J. Liu. DouZero: Mastering DouDizhu with self-play deep reinforcement learning. In *Proceedings of the 38th International Conference on Machine Learning*, pp. 12333–12344, 2021.

[43] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, I. Mordatch. Emergent tool use from multi-agent autocurricula. In *Proceedings of the 8th International Conference on Learning Representations*, Addis Ababa, Ethiopia, 2020.

[44] T. Rashid, M. Samvelyan, C. S. de Witt, G. Farquhar, J. N. Foerster, S. Whiteson. QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning*, Stockholm, Sweden, pp. 4295–4304, 2018.

[45] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018. DOI: 10.1126/science.aar6404.

[46] X. J. Wang, J. X. Song, P. H. Qi, P. Peng, Z. K. Tang, W. Zhang, W. M. Li, X. J. Pi, J. J. He, C. Gao, H. T. Long, Q. Yuan. SCC: An efficient deep reinforcement learning agent mastering the game of StarCraft II. In *Proceedings of the 38th International Conference on Machine Learning*, pp. 10905–10915, 2021.

[47] J. Paredis. Coevolutionary computation. *Artificial Life*, vol. 2, no. 4, pp. 355–375, 1995. DOI: 10.1162/artl.1995.2.4.355.

[48] N. Brown, T. Sandholm. Superhuman AI for heads-up no-limit poker: Libratus beats top professionals. *Science*, vol. 359, no. 6374, pp. 418–424, 2018. DOI: 10.1126/science.aao1733.

[49] M. Moravčík, M. Schmid, N. Burch, V. Lisý, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, M. Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, vol. 356, no. 6337, pp. 508–513, 2017. DOI: 10.1126/science.aam6960.

[50] A. DiGiovanni, E. C. Zell. Survey of self-play in reinforcement learning, [Online], Available: https://arxiv.org/abs/2107.02850, 2021.

[51] Q. Y. Yin, M. J. Zhao, W. C. Ni, J. G. Zhang, K. Q. Huang. Intelligent decision making technology and challenge of wargame. *Acta Automatica Sinica*, vol. 49, no. 5, pp. 9132–928, 2023. DOI: 10.16383/j.aas.c210547. (in Chinese)

[52] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, I. Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, Long Beach, USA, pp. 6382–6393, 2017.

[53] M. W. Hoffman, B. Shahriari, J. Aslanides, G. Barth-Maron, N. Momchev, D. Sinopalnikov, P. Stańczyk, S. Ramos, A. Raichuk, D. Vincent, L. Hussenot, R. Dadashi, G. Dulac-Arnold, M. Orsini, A. Jacq, J. Ferret, N. Vieillard, S. K. S. Ghasemipour, S. Girgin, O. Pietquin, F. Behbahani, T. Norman, A. Abdolmaleki, A. Cassirer, F. Yang, K. Baumli, S. Henderson, A. Friesen, R. Haroun, A. Novikov, S. G. Colmenarejo, S. Cabi, C. Gulcehre, T. Le Paine, S. Srinivasan, A. Cowie, Z. Y. Wang, B. Piot, N. de Freitas. Acme: A research framework for distributed reinforcement learning, [Online], Available: https://arxiv.org/abs/2006.00979, 2020.

[54] S. Fujimoto, H. Hoof, D. Meger. Addressing function approximation error in actor-critic methods. In *Proceedings of the 35th International Conference on Machine Learning*, Stockholm, Sweden, pp. 1587–1596, 2018.

[55] J. Ho, S. Ermon. Generative adversarial imitation learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, Barcelona, Spain, pp. 4572–4580, 2016.

[56] S. Reddy, A. D. Dragan, S. Levine. SQIL: Imitation learning via reinforcement learning with sparse rewards. In *Proceedings of the 8th International Conference on Learning Representations*, Addis Ababa, Ethiopia, 2019.

[57] J. Y. Weng, H. Y. Chen, D. Yan, K. C. You, A. Duburcq, M. H. Zhang, Y. Su, H. Su, J. Zhu. Tianshou: A highly modularized deep reinforcement learning library. *Journal of Machine Learning Research*, vol. 23, no. 267, pp. 1–6, 2022.

[58] H. Küttler, N. Nardelli, T. Lavril, M. Selvatici, V. Sivakumar, T. Rocktäschel, E. Grefenstette. Torchbeast: A pytorch platform for distributed RL, [Online], Available: https://arxiv.org/abs/1910.03552, 2019.

[59] M. Zhou, Z. Y. Wan, H. J. Wang, M. N. Wen, R. Z. Wu, Y. Wen, Y. D. Yang, W. N. Zhang, J. Wan. MALiB: A parallel framework for population-based multi-agent reinforcement learning, [Online], Available: https://arxiv.org/abs/2106.07551, 2021.

[60] P. Muller, S. Omidshafiei, M. Rowland, K. Tuyls, J. Pérolat, S. Q. Liu, D. Hennes, L. Marris, M. Lanctot, E. Hughes, Z. Wang, G. Lever, N. Heess, T. Graepel, R. Munos. A generalized training approach for multiagent learning. In *Proceedings of the 8th International Conference on Learning Representations*, Addis Ababa, Ethiopia, 2020.

[61] S. McAleer, J. Lanier, R. Fox, P. Baldi. Pipeline psro: A scalable approach for finding approximate nash equilibria in large games. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, Vancouver, Canada, pp. 20238–20248, 2020.

[62] J. Heinrich, M. Lanctot, D. Silver. Fictitious self-play in extensive-form games. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning*, Lille, France, pp. 805–813, 2015.

[63] H. T. Jia, Y. J. Hu, Y. F. Chen, C. X. Ren, T. J. Lv, C. J. Fan, C. J. Zhang. Fever basketball: A complex, flexible, and asynchronized sports game environment for multi-agent reinforcement learning, [Online], Available: https://arxiv.org/abs/2012.03204, 2020.

[64] E. Accinelli, E. J. S. Carrera. Evolutionarily stable strategies and replicator dynamics in asymmetric two-population games. *Dynamics, Games and Science I*, M. M. Peixoto, A. A. Pinto, D. A. Rand, Eds., Berlin, Germany: Springer, pp. 25–35, 2011. DOI: 10.1007/978-3-642-11456-4_3.

Qiyue Yin received the Ph.D. degree in pattern recognition and intelligence systems from the National Laboratory of Pattern Recognition (NLPR), Institute of Automation, Chinese Academy of Sciences (CASIA), China in 2017. He is currently an associate professor at CASIA, China.

His research interests include machine learning, pattern recognition and artificial intelligence on games.

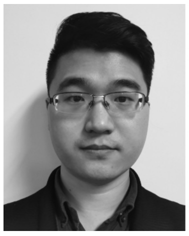
E-mail: qyyin@nlpr.ia.ac.cn (Corresponding author)

ORCID iD: 0000-0002-3442-6275

Tongtong Yu received the master's degree in computer science and technology from Beijing University of Technology, China in 2020. She is currently an engineer at Institute of Automation, Chinese Academy of Sciences (CASIA), China.

Her research interests include machine learning and artificial intelligence on games.

E-mail: tongtong.yu@ia.ac.cn



Shengqi Shen received the master's degree in control science and engineering from Beijing University of Chemical Technology, China in 2018. He is currently an engineer at Institute of Automation, Chinese Academy of Sciences (CASIA), China.

His research interests include machine learning, decision making in games.

E-mail: shengqi.shen@ia.ac.cn

Jun Yang received the Ph.D. degree in control science and engineering from Tsinghua University, China in 2011. He is currently an associate professor with the Department of Automation, Tsinghua University, China.

His research interests include multi-agent reinforcement learning and game theory.

E-mail: yangjun603@tsinghua.edu.cn

Meijing Zhao received the Ph.D. degree in pattern recognition and intelligence systems from Integrated Information System Research Center, Institute of Automation, Chinese Academy of Sciences (CASIA), China in 2016. She is currently an associate professor at CASIA, China.

Her research interests include semantic information processing, knowledge representation and reasoning.

E-mail: meijing.zhao@ia.ac.cn

Wancheng Ni received the Ph.D. degree in contemporary integrated manufacturing systems from Department of Automation, Tsinghua University, China in 2007. She is currently a professor at Institute of Automation, Chinese Academy of Sciences (CASIA), China.

Her research interests include information processing and knowledge discovery, group intelligent decision-making platform and evaluation.

E-mail: wancheng.ni@ia.ac.cn

Kaiqi Huang received the Ph.D. degree in communication and information processing from Southeast University, China in 2004. He is currently a professor at Institute of Automation, Chinese Academy of Sciences (CASIA), China.

His research interests include visual surveillance, image understanding, pattern recognition, human-computer gaming and biological based vision.

E-mail: kqhuang@nlpr.ia.ac.cn

Bin Liang received the Ph.D. degree in precision instruments and mechanology from Tsinghua University, China in 1994. He is currently a professor with the Department of Automation, Tsinghua University, China.

His research interests include artificial intelligence, anomaly detection, space robotics, and fault-tolerant control.

E-mail: bliang@tsinghua.edu.cn

Liang Wang received the Ph.D. degree in pattern recognition and intelligence systems from the National Laboratory of Pattern Recognition (NLPR), Institute of Automation, Chinese Academy of Sciences (CASIA), China in 2004. He is currently a professor at CASIA, China.

His research interests include computer vision, pattern recognition, machine learning, and data mining.

E-mail: wangliang@nlpr.ia.ac.cn