TLeague: A Framework for Competitive Self-Play based Distributed Multi-Agent Reinforcement Learning

Peng Sun[a,*], Jiechao Xiong[a,*], Lei Han[a,*],
Xinghai Sun[a], Shuxing Li[b,1], Jiawei Xu[b,1], Meng Fang[a], Zhengyou Zhang[a]

[a] *Tencent Robotics X, Shenzhen, China*
[b] *Tsinghua University, Shenzhen, China*

Abstract

Competitive Self-Play (CSP) based Multi-Agent Reinforcement Learning (MARL) has shown phenomenal breakthroughs recently. Strong AIs are achieved for several benchmarks, including Dota 2, Glory of Kings, Quake III, StarCraft II, to name a few. Despite the success, the MARL training is extremely data thirsty, requiring typically billions of (if not trillions of) frames be seen from the environment during training in order for learning a high performance agent. This poses non-trivial difficulties for researchers or engineers and prevents the application of MARL to a broader range of real-world problems. To address this issue, in this manuscript we describe a framework, referred to as TLeague, that aims at large-scale training and implements several main-stream CSP-MARL algorithms. The training can be deployed in either a single machine or a cluster of hybrid machines (CPUs and GPUs), where the standard Kubernetes is supported in a cloud native manner. TLeague achieves a high throughput and a reasonable scale-up when performing distributed training. Thanks to the modular design, it is also easy to extend for solving other multi-agent problems or implementing and verifying MARL algorithms. We present experiments over StarCraft II, ViZDoom and Pommerman to show the efficiency and effectiveness of TLeague. The code is open-sourced and available at `https://github.com/tencent-ailab/tleague_projpage`

Keywords: Competitive Self-Play, Reinforcement Learning, Multi-Agent, Distributed, Game AI

1. Introduction

In the last decade, Deep Reinforcement Learning (DRL) was shown to be a powerful tool for sequential decision-making problems. Human or super-human level performance is achieved for several well-known benchmarks, including Atari [1], Go [2, 3], etc. In particular, the Competitive Self-Play (CSP) [4] based Multi-Agent Reinforcement Learning (MARL) has

*Equal contribution, correspondence to the first three authors.
Email addresses: `pengsun000@gmail.com` (Peng Sun), `jchxiong@gmail.com` (Jiechao Xiong), `leihan.cs@gmail.com` (Lei Han)
[1] Work was done during the internship with Tencent Robotics X.

obtained impressive success for some non-trivial problems, e.g., Dota 2 [5], Glory of Kings [6], Quake III [7], StarCraft II [8]. The CSP-MARL method is Game Theoretic justified, as it performs *Nash Equilibrium* (NE) finding by Fictitious Self-Play where the opponent mixture is stochastically approximated by opponent sampling and the *Best Response* is realized with (Single-Agent) Reinforcement Learning that serves as a proxy algorithm [9, 10, 11, 12]. It is thus appealing in that the method is general-purpose and can be potentially applied to any MARL problem (Note that in cooperative or cooperative-competitive hybrid games, we can also sample for team-mates as in [7]). However, such a method is extremely data thirsty. For example, in the aforementioned Dota 2 or StarCraft II, it usually requires billions of (if not trillions of) frames generated from the environment during training (the equivalent in-game time is hundreds or thousands of years), otherwise the agent could be still under-trained and unlikely to perform well. It thus poses prominent difficulties for RL researchers or practitioners when applying the method to their own interested problems.

To address this issue, we propose a framework that aims at large-scale CSP-MARL training. We adopt a modular design. An Actor-Learner-InferenceServer architecture [13, 14] is taken to tackle the data producing (i.e., generating trajectories by interacting the environment and the agents) and the data consuming (i.e., learning from the trajectories by gradient descent). There are also dedicated modules that manage an opponent pool and the concrete neural net parameters. The modules coordinate to do the CSP-MARL training in parallel, with each module having a high work-load and little idle time. This way, it is able to maximally leverage a cluster of hybrid machines (with both CPUs and GPUs) and to substantially accelerate the training. In our testing, it achieves a high throughput and a reasonable scale-up when using hundreds of GPUs and tens of thousands of CPU cores. The large-scale run supports standard Kubernetes, and the development with TLeague can be in a cloud-native manner.

We have shipped with TLeague several main-stream algorithms . The opponent sampling can be Population Based Training (PBT) [7], Agent-Exploiter [8], (Prioritized) Fictitious Self-Play [5, 8]. Several typical *Policy Gradient* methods, such as PPO [15] and V-trace [13], are supported. One can also build desired neural nets in various architectures, ranging from a simple list structure to a complicated *Directed Acyclic Graph* (DAG). The code is designed to be flexible and friendly to extend TLeague to other multi-agent problems (e.g., adding new environments, or RL algorithms, or opponent sampling algorithms, etc).

The rest of this manuscript is organized as follows. We briefly review the related work in Section 2. Then we describe CSP-MARL and explain the design of our code implementation in Section 3. Finally, we discuss several experiments over StarCraft 2, ViZDoom [16] and Pommerman [17] in Section 4 to show the efficiency and effectiveness of TLeague.

2. Related Work

Since the breakthroughs of DRL in Atari [1] and GO [2, 3], several distributed RL frameworks were discussed in the literature.

Gorila [18] decouples the Actor, Learner and Replay Memory to allow a scalable distributed training. However, it only targets for DQN and uses asynchronous gradient descent. Ape-X [19] extends the Gorila framework in that it supports a centralized Prioritized Replay

and using synchronous gradient descent. Ape-X also incorporates more Q value based RL algorithms. R2D2 [20] is a successor of Ape-X, where the same architecture is adopted for implementing a more state-of-the-art Q learning.

In another line of work, attempts are made to parallelize the policy gradient algorithm. A3C [21] collects the trajectories and updates neural net gradients both in an asynchronous way, and only CPUs are used. GA3C [22] explicitly uses a GPU for neural net learning, and collects the trajectories across each actor asynchronously. The method is efficient, but it can cause data lagging and henceforth hurt the performance of an on-policy RL. To alleviate the problem, batched A2C [23] forces a simultaneous environment stepping and collect the batch synchronously. This method works for both on-policy and off-policy. However, GA3C (or batched A2C) uses the same GPU for both forward-passing (collecting trajectories) and backward-passing (learning), which can be still a bottleneck for scalable training.

IMPALA [13] also uses GPUs and explicitly takes a decoupled Actor-Learner architecture, where a state-of-the-art off-policy algorithm called V-trace is implemented. SEED [14] improves over IMPALA by performing the forward-pass over a separate Inference Server, which further increases the training speed.

The reverb library [24] implements a dedicated distributed Replay Memory, which can be a building block for other RL framework. The acme library [25] builds on top of reverb, adopting the Actor-Learner-InferenceServer architecture and implementing a bunch of modern RL algorithms.

The work mentioned above only addresses single agent RL and not covers MARL. In [7, 5, 8], the authors discuss a proprietary framework for CSP-MARL, and the code implementation is not publicly available.

The ray [26] library defines several *primitives* for parallel computing, on top of which an RL library called rllib [27] is built. However, the TLeague framework discussed in this manuscript takes a design that is much closer to RL algorithms and to the machines. No extra abstraction of parallel computing is introduced, allowing TLeague be easy to run in cloud native manner (i.e., using Kubernetes).

Our work is most similar to IMPALA and SEED in regards of how it decouples RL components, i.e., we also adopt the Actor-Learner-InferenceServer architecture. However, we make several notable extensions. 1) We support CSP-MARL, for which separate modules are designed to manage an opponent pool and maintain the neural net parameters. 2) We use Horovod [28] to do the synchronous gradient updating across multiple GPUs for the Learners.

There is other work [29, 30, 31] that devotes to providing various RL algorithms or environments, which is beyond our scope. However, we do borrow the code from the library openai/baselines [32] and deepmind/trfl [33] when implementing PPO [15] and V-trace [13] in TLeague.

3. Architecture

3.1. Mathematical Settings

We presume the readers have been familiar with Single-Agent RL (cf. to, e.g., [34]). Now we adopt the MARL settings from [35] and provide a brief description. A Multi-Agent *game*

is indicated by a tuple $< \mathcal{S}, \mathcal{P}, r^i, \gamma, \mathcal{O}^i, \mathcal{A}^i, \pi^i >$ where $i \in \{1, 2, ..., N\}$ is the index for the N agents. The *state* $s_t \in \mathcal{S}$ fully describes the game at time step t. The $o_t^i \in \mathcal{O}^i$ and $a_t^i \in \mathcal{A}^i$ are the *observation* and *action* for agent i, respectively. The dynamics of an *environment* is carved by a transition $\mathcal{P} : \mathcal{S} \times \mathcal{A}^1 \times \mathcal{A}^2 ... \times \mathcal{A}^N \mapsto \mathcal{S}$ which is represented by a *stationary* probability $P(s_{t+1}|s_t, a_t^1, a_t^2, ..., a_t^N)$ over the next state s_{t+1} when all the agents perform the actions $a_t^1, a_t^2, ..., a_t^N$ at current state s_t. Denote by $r_t^i : \mathcal{S} \times \mathcal{A}^1 \times \mathcal{A}^2 ... \times \mathcal{A}^N \mapsto \mathbb{R}$ the instantaneous *reward* $r_t^i(s_t, a_t^1, a_t^2, ..., a_t^N)$ received for agent i at time t. The i-th agent's *policy* $\pi^i : \mathcal{O}^i \mapsto \mathcal{A}^i$ gives the conditional probability $\pi^i(a_t^i|o_t^i)$ over the action a_t^i to take when observing o_t^i. The policy π_θ^i is usually represented by a parametrized *function approximator*, e.g., a neural network with parameters θ. When the function approximator is able to model sequential data (e.g., with an LSTM layer [36]), we can let the policy be conditioned on the entire observation history and rewrite it as $\pi^i(a_t^i|\{o_{t'}^i\}_{t'=0}^t)$. An MARL algorithm seeks to learn, for each agent, an optimal policy π^i that maximizes the expected *return* $\mathbb{E}_{\mathcal{P},\pi^1,...,\pi^N}[\sum_{t=0}^T \gamma^t r_t]$, where T is the *horizon* (i.e., the episode length in time for the game) and $\gamma \in (0, 1]$ is the *discount factor* that prevents the inflation for a too large or infinite horizon T. In particular, a *model-free* algorithm is able to learn π^i when $\mathcal{P}$ and r^i are unknown in their mathematical forms.

Note that the reward structure implies the game *mode*: is it competitive, cooperative, or hybrid? For example, suppose a two-agent zero-sum game where we have $r^1 + r^2 = 0$. The game is then competitive, as one agent benefits from the other agent's loss. We note that various real-world games can be modeled by such a competitive mode, ranging from Rock-Paper-Scissor to StarCraft 1vs1 full game.

How to perform a concrete MARL training? It is tempting to independently apply an (Single-Agent) RL for each agent. However, this is not mathematically justified, as in this way the agent i's dynamics is *non-stationary* when the other agents' policies are absorbed into agent i's environment. Such a treatment contradicts to the presumption of most model-free RLs that are derived for stationary dynamics. There exist cases that independent RL is reported to be effective for MARL [37], but in many other applications it leads to poor results [9]. In particular, it suffers from the policy-forgetting during training when the policy space is rich and contains circulation [9, 10]. An example is the game Rock-Paper-Scissor. A naive independent RL will circulate over pure-rock, pure-paper, pure-scissor, ... that the late policy (e.g., pure-scissor) forgets how to beat the early policy (e.g., pure-rock).

From the perspective of Game Theory, the "gradient field" of independent RL rotates over (but never converges to) an optimal point (i.e., the NE in parameter space). A remedy to this is the Fictitious Self-Play (FSP) algorithm that dates back to the 1950s [38]. Below we briefly explain FSP in a competitive two-agent case, where one is the learning agent and the other is an opponent. During FSP training, the learning agent plays against a mixture of the historical opponents, not just the current opponent as in independent RL. This way, it introduces a "centripetal force" pointing to NE in the gradient field (e.g., see Fig. 4 of [39]), avoiding policy forgetting and converging to NE.

In the case of a multi-step game (e.g., StarCraft II), it is non-trivial to implement FSP in a straightforward way, as one should maintain a mixture of historical policies conditioned at *every* state $s \in \mathcal{S}$ (Note in a one-step game like Rock-Paper-Scissor, there is only one possible

state and is usually omitted). There are studies [40, 41] where a neural network is adopted to model the conditional mixture term [2]. However, a more straightforward and convenient implementation is the opponent sampling based Monte Carlo method. Denote by θ the parameter of an agent's policy. Construct a pool $\mathcal{M} = \{\theta_1, \theta_2, ...\}$. On each episode beginning during training, an opponent, denoted by its parameter ϕ, is selected by sampling from the pool $\phi \sim Q(\mathcal{M})$. Various sampling distributions Q have been reported in the literature, including uniform [4], a probabilistic mixture of the current and the historic opponent [5], probabilistic Elo score matching [7], a function of win-rate [8], etc. The opponent sampling can be viewed as a stochastic approximation of the opponent policy mixture.

Once an opponent ϕ is selected, the parameter ϕ gets fixed and the learning agent tries to maximize the return by updating its own parameter θ. In the Game Theory community, this procedure is referred to as Best Response, which is actually an RL in view of Machine Learning [9]. Note the fixed ϕ leads to a stationary opponent policy π_ϕ, which is then absorbed into the environment and the dynamics remains stationary for the learning agent. To this extent, one can employ any favorite RL (e.g., PPO [15], V-trace [13], etc) as the "proxy algorithm". Morden RL is able to learn from *trajectory segment* defined as tuples of observation-reward-action in contiguous time steps:

$$\tau = (o_t, r_t, a_t, o_{t+1}, r_{t+1}, a_{t+1}, ..., o_{t+L}, r_{t+L}, a_{t+L}) \tag{1}$$

where L is the segment length and we've omitted the superscript for the learning agent. This permits a mini-batch style SGD for RL which is more compatible to the Deep Learning paradigm.

Every once in a while, the pool is updated by $\mathcal{M} \leftarrow \mathcal{M} \cup \{\theta\}$. This way, the learning agent still plays against a mixture of historical opponents stochastically. The initial size of the pool is one that $\mathcal{M} = \{\theta_1\}$, where the "seed" policy parameter θ_1 can be either randomly initialized or the one learned from *Imitation Learning*.

Finally, we note that FSP is easy to extend to multiple opponents ($>= 2$). For example, do the sampling $\phi \sim Q$ for each of the opponents, respectively, on each episode beginning as in [7].

3.2. Design

To implement the CSP-MARL algorithm described in Section 3.1 and allow it to be scalable, we adopt a modular design for our distributed training framework. Fig. 1 gives an overview. In the following, we describe each of the modules and explain how they correspond to CSP-MARL.

Actor. The Actor module produces the trajectory for the learning agent. It embeds two secondary modules, Env (environment) and Agt (agent). We require Env be OpenAI gym [49] compatible for the Multi-Agent case, that is, it should implement the two methods:

[2] In a more general setting, it is possible to perform a no-regret learning for NE finding [42, 43, 44, 45, 46, 47]. The corresponding discussion is beyond the scope of this manuscript.

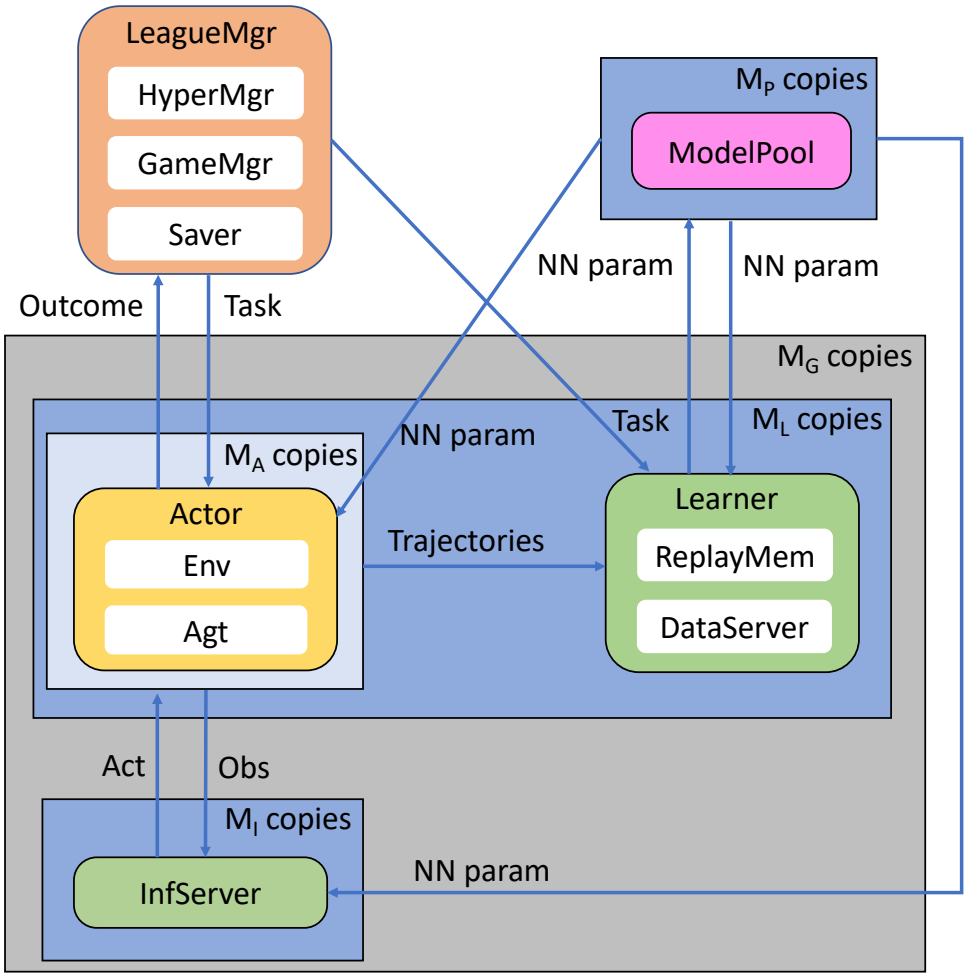


Figure 1: Diagram of the framework. The rounded rectangle denotes a primary module or a secondary module (if any), e.g., Actor is a primary module that embeds the secondary modules Env and Agt. Borrowing notations of [48], we use a rectangle with a number on top-right to denote how many copies/replicas there are for a module. In this convention, we can read that there are M_p ModelPools, $M_A \times M_L \times M_G$ Actors, etc. The method-call (or message-passing) is represented by arrows. The contact of the arrow indicates "how the messages are packed". For example, the "Trajectories" arrow starts from the "M_A copies" rectangle and ends at the "Learner" rounded rectangle, which indicates that the M_A Actors altogether send trajectories to a single one Learner. See the text for detailed explanations.

```
l_obs = env.reset() # episode beginning
l_obs, l_rwd, done, info = env.step(l_act) # in-episode stepping
```

where the `l_obs` represents a list of the observations from all the N agents $\{o_0^i\}_{i=1}^N$ ($t=0$, episode beginning) or $\{o_{t+1}\}_{i=1}^N$ ($t>0$, in-episode stepping). Likewise, `l_rwd` means $\{r_t^i\}_{i=1}^N$, `l_act` means $\{a_t^i\}$, `done` is a boolean variable indicating whether it is an episode ending, `info` is a Python dictionary that holds the extra information you want to pass (For example, in StarCraft II environment we use `info['outcome']` to indicate if an agent wins/losses/ties the match of this episode). The Agt carries a function approximator (usually a neural net) as the policy π^i and takes the action $a_t^i \sim \pi^i$, where the neural net forward pass can be done either in a local machine or be delegated to a (remote) InfServer. At each episode beginning, the Actor requests a *task* from the LeagueMgr to know what the current (itself) learning policy π_θ and what the opponent policy π_ϕ. At each episode ending, it also reports the game outcome to LeagueMgr. During the Env-Agt interaction loop, the trajectories of the learning agent are sent to Learner for the Neural Net training therein. Periodically, the Actor pulls up-to-date policy parameters θ and ϕ from ModelPool.

Learner. A learning agent can own M_L Learner modules. Each Learner receives trajectories from M_A Actors associated to it. Each Learner can bind to a GPU, and the M_L Learners synchronize parameter gradients using the library Horovod [28] which performs an efficient *allreduce* algorithm and can benefit from a fast inter-GPU connection via NCCL2 [50]. We allow there be up to M_G learning agents that train in parallel. In a full run, we have $M_G \times M_L$ Learners (henceforth that many GPUs) and $M_G \times M_A \times M_L$ Actors. Each Learner embeds exactly one DataServer and one ReplayMem (Replay Memory), performing a series of trajectory data pre-processing, e.g., receiving the trajectory segments sent from actors and storing the data in a Replay Memory, calculating the algorithm specific terms (say, the λ-return), GPU-prefetching for the mini-batch to be learned, etc. On each *learning period* beginning, the Learner receives a *task* from the LeagueMgr to know what the current policy θ it is training, and the learner task must be consistent with the actor task. During training, the Learner also periodically updates the policy parameter θ stored in the ModelPool. Note the M_L Learners are strictly synchronized, thereby only one Learner suffices to do the task requesting. We let the 0-th Learner (i.e., the *rank*-0 machine in MPI [51] semantics) do the job. At the end of a learning period, the current policy θ is frozen in the ModelPool. A learning period should be long enough to ensure the policy has been well trained.

InfServer. The InfServer (abbreviation of Inference Server) is optional. When enabled, an InfServer collects a batch of observations from different Actors and feeds them into the neural net to predict the actions which are then returned to each of the Actors, respectively. InfServer is usually deployed on GPU machines so that the batch forward-pass can be highly efficient. Overall, such a scheme can lead to a higher throughput than that a one-step forward-pass (batch size 1) be done locally on each Actor. In more sophisticated RL, we may want to penalize the KL divergence between current policy and a teacher policy, where we can also do the teacher policy forward-pass on an InfServer.

ModelPool. The ModelPool stores the concrete neural net parameters of the opponent pool $\mathcal{M}$. During the whole training lifecycle, ModelPool must respond to any parameter

requesting (read) or updating (write) instantaneously, from either a Learner or an Actor. We then use a load-balance technique for high concurrency and high performance, where up to M_M ModelPool replicas can be run simultaneously and a random one is picked to do the concrete response. We also keep the neural net parameters in-memory to allow a fast read/write operation.

LeagueMgr. The LeagueMgr (abbreviation of League Manager) sponsors the training and coordinates the other modules. A key secondary module is the GameMgr (Game Manager), which maintains a payoff matrix for all the models stored in the pool $\mathcal{M}$ and implements (in the derived class) various opponent sampling algorithms aforementioned in Section 3.1. The selected learning agent and/or opponent is wrapped as a task sent to Actor or Learner. Another secondary module is the HyperMgr (Hyper-parameter Manager), which maintains the hyperparameters associated with each model $\{\theta_i\} \in \mathcal{M}$. Here the hyperparameters can account for various algorithmic settings, e.g., the learning rate or discount factor for RL, the variance term of the Gaussian Elo matching probability for opponent sampling [7], the z-statistics as required by the AlphaStar policy neural net [8], etc. The HyperMgr can also perturb or vary the hyper-parameters, as required by some algorithm like PBT [7].

3.3. System-Level Design

For small-scale training, we can simply run all the aforementioned modules in a single machine. However, for contemporary large-scale RL that requires high throughput training, it is unrealistic to do the single machine. For example, one may need thousands of Actors to produce the trajectories in parallel and hundreds of GPU Learners to consume and learn from the data, which exceeds the capacity of a common machine. Keeping this in mind, we have designed the framework so that the modules can be deployed across multiple machines to support a scalable distributed training. We adopt a *Microservices* paradigm [52], that is, each module can be launched as an OS *Process*, exposing its APIs to other modules and behaving like a *Service*. The modules/processes talk to each other via *RPC (Remote Procedure Call)*, which maps the code-level class interaction to system-level inter-process communication. We define our private inter-process message (i.e., the API protocol) in native Python3 language and rely on the library ZeroMQ [53] for RPC. [3]

3.4. Large-scale Run and Kubernetes

When performing large-scale training, it is challenging to do it over *bare metal* machines due to several reasons:

- It is difficult and error-prone to initiate and maintain the basic setups for each machine, e.g., a list of the IPs/hostnames, password-less SSH, package dependencies, etc.
- It is tedious to manually start/stop each module on the desired machine.
- The error-tolerance mechanism is absent when running the modules. For example, we hope the Actor can auto restart when it crashes due to some low-level error that is out of our control (e.g., the occasional core-dump of the Env binary).

[3] Other scheme is possible, e.g, using protobuf [54] and gRPC [55]

- It is error-prone when updating and keeping consistent the code over all machines during the development period.

To address these issues, we embrace the cloud-native philosophy [52]. We rely on Kubernetes (abbreviated as k8s) [56] to manage our large-scale distributed run [4] as follows.

Depending on the role it plays in the framework, each TLeague module is made as a proper k8s *resource* [56]. Specifically, LeagueMgr, ModelPool, Learner and InfServer is made as k8s *Service*, respectively, exposing the APIs via an endpoint idiom in the format of hostname:ip-port, (e.g., `tcp://signature-league-mgr:9003`). Actor is made as k8s *Deployment* or *ReplicaSet*, which allows us to scale-up/-down the number of Actors to adjust the trajectory producing speed during the whole training lifecycle. It also automatically restarts the Actor in case it encounters an unrecoverable error thanks to the k8s imperative semantic. The concrete TLeague module carrier is a k8s *pod*, which can be placed and co-located in desired type of machine using k8s *nodeSelector*. We prepare everything of a distributed training in a yaml file, including both the RL algorithm settings (e.g., learning rate, discount factor, etc.) and the k8s settings. Then we submit it to a k8s cluster using the kubectl command-line. We also employ jinja2 (a template library [57]) to generate the yaml in a configurable and concise way. Suppose, for example, a training specification has been written in a file named `foobar.yml.jinja2`, then we can start or stop the training by simply running something like the bash commands:

```
# start
python render_template.py foobar.yml.jinja2 | kubectl apply -f -
# stop
python render_template.py foobar.yml.jinja2 | kubectl delete -f -
```

We note that an alternative solution is the kubeflow [58], where we can write a dedicated "operator" for each TLeague module. However, we found the yml-jinja2-kubectl way suffices and is convenient enough in all our experiments, and will not discuss the kubeflow solution in this manuscript.

In this way, our specification of a distributed training becomes more compute-resource centric. For example, a yml.jinja2 file can be as descriptive as plain English like "Alright, I want 56 Learners and 8 InfServers, each Learner corresponds to 16 actors. Each Learner requires 1 GPU, each InfServer requires 1 GPU and each Actor requires 4 CPU cores. Every 7 Learners and 1 InfServer must be co-located in the same GPU machine. Each ModelPool must be placed in the machine with a high-speed Network Adaptor and >200G memory."

TLeague can be run over any standard k8s cluster. For example, most experiments in Section 4 were run over Tencent Cloud [59], where a k8s cluster is created and maintained via TKE (Tencent Kubernetes Engine [60]) that provides an off-the-shelf k8s solution. The *node* machines (most of them are Tencent Cloud CVM [61]) can be added/removed by several clicks through a web console (or by programming with Tencent Cloud APIs if you like). Tencent Cloud supports *Cluster Autoscaler* for those node machines. In addition with a

[4] We also provide examples of how to write Shell scripts for running in a single machine (or several machines) to cover the use-case of small-scale training.

flexible on-demand pricing strategy, it allows us to achieve a very high peak computing ability with limited (monetary) budget, which is, we argue, the best solution to running a full-scale experiment that requires tremendous computing resources (e.g., hundreds of GPUs and tens of thousands of CPU cores) for only a few days or weeks. Indeed, in most of the time during a development cycle, we should run small-scale experiments for idea verification or prototyping.

In our development, we also benefit from the Tencent Cloud ecosystem that provides abundant facilities. We employ the CFS [62] (Tencent's extension to the standard NFS) and make it a k8s *PVC (Persistent Volume Claim)* to do the directory sharing across pods. We take advantage of the *DevOps* concept to accelerate our developing iteration. Specifically, we use a CI (Continuous Integration) tool to build the *Docker Image* and push it to our private *Docker Registry*, which can be triggered, for example, by a simple *git commit* pushing. The full development workflow is summarized in Fig. 2.

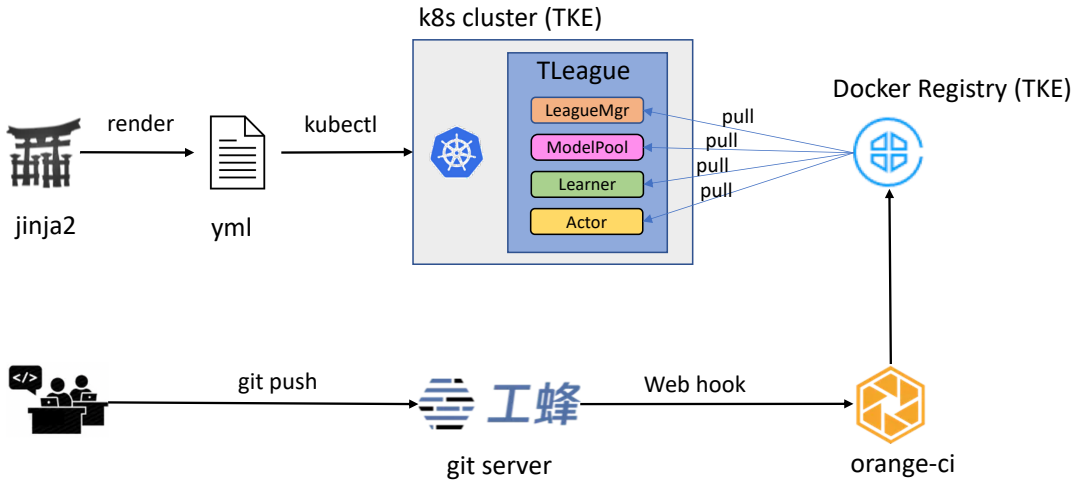


Figure 2: Workflow of our TLeague developing and training. See the text for detailed explanations. Although some tools are currently Tencent internal, they have public available counterparts. The CI tool Orange-CI can be replaced with Travis-CI. Also, the private docker registry is usually provided by any other Cloud vendor (or just use the public dockerhub).

3.5. Code Structure

We've split the framework described in Section 3.2 into several repositories in regards of the functionality, satisfying a principle of high cohesion and low coupling. Each repository is made as an independent Python package that can be pip installed. In the following we provide more details for each repository.

Arena. We've prepared the OpenAI gym compatible environments in a toolbox called *Arena* [63]. Moreover, the *observation space* and *action space* is implemented as *Arena Interface*, which allows us to write the code only once for either training (binds to an environment

like a *gym env wrapper*) or testing (binds to an agent), see [63] for detailed explanations of this mechanism. When the logic of a specific environment is too heavy, we recommend placing the corresponding code in a separate repository. For example, we've made a repository called **TImitate** dedicated to SC2 full game, which involves feature engineering, replay file parsing, z-statistic extraction, etc.

TPolicies. A Neural Network library tailored for Reinforcement Learning and Imitation Learning. It uses Tensorflow 1.x APIs and is in the style of the library *tf-slim* and *tf.contrib.layers*. With TPolicies one can build policy net or value net in various architectures, ranging from a simple one in list structure (e.g., a ConvNet plus LSTM for Atari [1, 21] or ViZDoom [16]) to a complicated one of general Directed Acyclic Graph (e.g., the net for SC2 full game [8], containing layers/blocks of ResNet, Transformer, Pointer Net, Gated Linear Unit, Auto-regressive Action Heads, etc.). TPolicies also provides RL related Tensorflow *ops*, e.g., for building policy gradient loss, for computing λ-return. Such RL related code is borrowed and adopted from the library openai/baselines [32] and deepmind/trfl [33].

TLeague. Most modules and the main functionality described in Section 3.2 are implemented in this repository, which depends on (must `import` in Python) Arena and TPolicies.

3.6. Extension

Thanks to the modular design, it's convenient to extend the framework for your own interested applications. In the following we provide guidelines for some typical use-cases.

Adding New Env. Prepare the Env code in `arena.env`, and write a thin wrapper in `tleague.envs`. To customize the observation space and action space, write a corresponding Arena Interface [63]. Refer to the following Python modules for how to add the environment named pong-2p [63]:

```
arena.env.Pong2pEnv # game logic
tleague.envs.pong # a thin wrapper to make it visible to TLeague
```

Adding New RL Algorithm. Derive from `tleague.actors.BaseActor` for how to produce the trajectories (data producing), and derive from `tleague.learners.BaseLearner` for how to learn from the trajectories (data consuming). Also, derive from `tleague.utils.DataStructure` to specify the trajectory data layout (e.g., a time step should contain an observation, a reward, a discount factor, etc.), serving as a contract between Actor and Learner. Write the policy gradient related or value related loss in `tpolicies.losses`. For example, refer to the following Python modules for how to implement the V-trace [13] algorithm:

```
tleague.actors.VtraceActor # Trajectories generating for V-trace
tleague.learners.VtraceLearner # Trajectories learning for V-trace
tleague.utils.VtraceData # data layout for V-trace
tpolicies.losses.vtrace_loss. # V-trace related loss implementation
tpolicies.net_zoo.mnet_v6d6.mnet_v6d6.mnet_v6d6_loss # use V-trace loss when
    building the net
```

Adding New Opponent Sampling Algorithm. Derive from the class `tleague.game_mgr.GamgeMgr` and implement the required methods such `get_player()`, `add_player()`. Refer to the following Python module as a minimal example which implements a uniform sampling from the historical opponents.

`tleague.game_mgr.SelfPlayGameMgr`

4. Experiment

We did experiments over three games: StarCraft II, ViZDoom and Pommerman (Fig. 3).

Figure 3: The games on which we did experiments: StarCraft II (left), ViZDoom (Middle) and Pommerman (Right).

4.1. SC2 full game

We investigate StarCraft II zerg-vs-zerg full game and perform a CSP-MARL training with TLeague. The technical details and the results are reported in a separate study [64].

4.2. ViZDoom

ViZDoom [16] is an AI research platform based on the FPS (First Person Shooter) game Doom. We adopt the CIG 2016 competition track 1 protocol [65], where 8 AI players join in a maze and play against each other. After a period of 10 minutes (in-game time), the players are ranked by the FRAG, which is defined as kills minus suicides (due to own rocket splash).

In our experiment, the observation is an RGB image, which is the first-person-view raw screen pixels as what a human player sees. The action is discrete in the size of 6, representing "turn-left", "move-forward", "fire", etc. We employ a neural network consisting of 2 blocks of convolution layer followed by max pooling layer and an LSTM block. We perform a two-stage training. In the first stage, the agent is trained to navigate in the maze, for which we use reward shaping to encourage the agent to explore the map with the "fire" action disabled. In the second stage, the agent is trained by CSP-MARL for the full match, explained as follows. On each episode beginning, we sample from the pool $\mathcal{M}$ for the rest 7 agents. We simply adopt a uniform sampling over the most recent 50 models. The proxy RL algorithm used in our experiment is PPO [15], where we've controlled the trajectory producing and consuming speed to ensure the on-policy.

The final trained agent (called *MyPlayer* hereafter) is tested in several ways. In Table 1 we give the results for playing against 7 builtin bots (called *bots* for short hereafter).

We also add a Single-Agent RL based AI named *F1* [66], which was the champion for CIG 2016 track 1. We test the following settings: "1 MyPlayer + 1 F1 + 6 Builtin Bots", "2 MyPlayer + 2 F1 + 4 Builtin Bots", "4 MyPlayer + 4 F1", and give the results in Table 2. The testing code and the "F1" policy net is adopted from [67]. All testing is done in a 12-core

Table 1: A testing of 5 matches for "1 MyPlayer, 7 bots". FRAG is reported. In all the 5 matches, MyPlayer ranks 1.

	1	2	3	4	5	Average
MyPlayer	26	24	31	27	30	27.6

Table 2: A testing of 5 matches for three settings. "1 MyPlayer, 1 F1, 6 bots" (top part), "2 MyPlayer, 2 F1, 4 bots" (middle part), "4 MyPlayer, 4 F1" (bottom part). The best FRAG is reported. For example, in the last row 29 is the best score of the 4 F1s in the 5th match.

	1	2	3	4	5	Average
MyPlayer	36	32	37	31	34	34.0
F1	28	33	32	22	32	29.0
MyPlayer	38	31	37	28	35	33.8
F1	30	26	26	31	34	29.4
MyPlayer	38	34	33	33	31	33.8
F1	33	28	26	24	29	28.0

CPU desktop machine. To ensure fairness, we use the *synchronous* mode for MyPlayer, F1 and the dedicated host so that the ViZDoom game core waits until it receives the actions from all players when performing in-game stepping.

Note that it is a self-play from scratch, and the agent has never seen builtin bot or F1 during training. As can shown in the Tables, MyPlayer gets higher score (the FRAG) than both the builtin bot and F1.

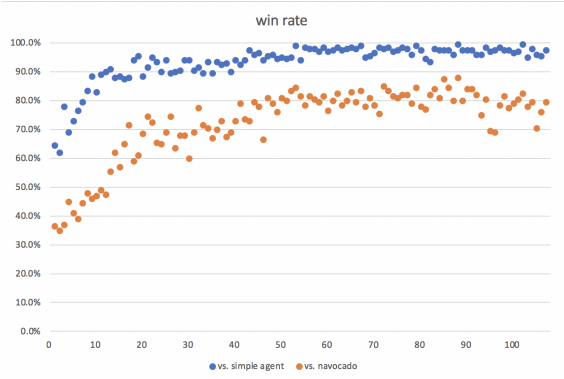
4.3. Pommerman

Pommerman [17] is a variant of the famous game Bomberman and is used as a benchmark for multi-agent learning. Typically, there are 4 agents that each can move and place bomb on an 11×11 board. At each step, an agent can take one of the 6 actions: {Idle, Move Up, Move Down, Move Left, Move Right, Place a Bomb}. A power-up item might appear when a wooden wall is destroyed by a bomb. Pommerman supports three modes: Free-for-All (FFA), Team and Team-Radio. In the FFA mode, the board is fully observable and each agent's goal is to be the last survivor. While in Team and Team-Radio mode, two agents cooperate as a team and fight against another 2-agent team. Each agent can only see a 9×9 board in its neighborhood. In Team mode agents are not allowed to communicate, while in Team-Radio mode a limited-bandwidth radio between teammates is provided. The team wins if it eliminates the 2 agents in the opponent team, and it gets a tie if the game is not finished within a maximal length of 800 steps.

In our Pommerman experiments, we adopt the Team mode, which is also used as the NeurIPS 2018 Competition environment. We use a decentralized policy to control the two agents. Each agent only uses its own observation, including the fogged board and self's attributes indicating the ammo number, blast strength, can-kick power-up and alive. The attributes are expanded as constant-value images that are concatenated with the board channels, yielding the feature maps as the observation. We employ a neural network consisting

of 5 convolutional blocks, followed by a gather op to collect the feature vector from the pixels where the agents reside, then passed through an LSTM block. To encourage the cooperation between the two teammate agents, we build a centralized value network that takes as inputs the two agents' LSTM embeddings. The output value is then used as the critic for PPO [15]. The two-agent team is viewed as a single agent (by performing the forward-pass twice for the only one neural net), and is trained from scratch with CSP-MARL. The opponent sampling is a mixture of 35% pure self-play and 65% PFSP, which is much like how the *Main Agent* samples as described in [8].

We test our agents by playing against Simple Agent (a rule-based builtin AI of Pommerman) and Navocado (announced as the best learning-based agent in NeurIPS 2018 Competition). The win-rates curves for the training iteration is shown in Fig. 4. As can be seen, the trained agent outperforms both Simple Agent and Navocado.



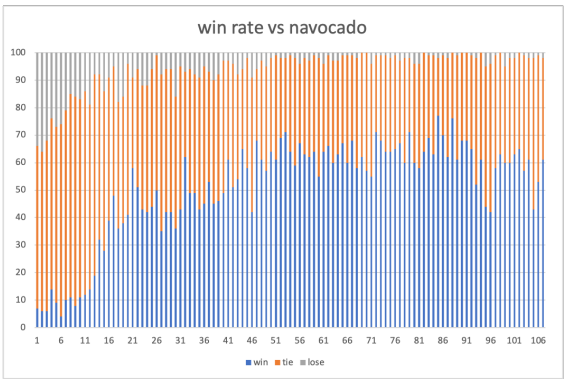


Figure 4: Win-rates curve for the training iteration. Each number is reported by taking 100 games. **Left**: Win-rates of our agent against Simple Agent, where a tie is counted as 0.5 win and 0.5 lose. **Right**: Number of "wins/losses/ties" for our agents against Navocado.

4.4. Throughput

To help readers evaluate how fast and how large-scale the training requires for a non-trivial environment, we provide some information in Table 3, where M_G denotes the number of the parallel learning agents (Section 3.2), "#CPU cores" and "#GPUs" denote how many CPU cores and GPUs are required per learning agent, respectively. The total number should be multiplied by M_G. For example, in TStarBot-X we use $96 \times 3 = 192$ GPUs in total. Unless noted otherwise, the GPU type is V100 where RDMA/RoCE is disabled and only TCP connection is used for Horovod allreduce. "rfps" denotes the receiving frames-per-second (number of frames sent from the Actors), and "cfps" denotes the consuming frames-per-second (number of frames learned on the Learners). In Table 3, rfps and cfps are also reported as per learning agent. The "in-game fps" means how many frames are rendered by the game core for an in-game second, for which we've accounted for the frame-skip. Take ViZDoom for example, there are 35 raw frames for one in-game second [68]. In our experiments we use a frame-skip = 2, henceforth the number 17.5 (= 35/2) in Table 3. One can infer the training speed-up or how long the in-game time has been spent by reading the rfps and in-game fps.

When rfps and cfps are almost equal (e.g., implementing a blocking queue for receiving data on the learner), the on-policyness for an RL algorithm will appear to be good. When cfps > rfps, the ratio cfps/rfps indicates on average how many times a frame is learned repeatedly.

Table 3: Throughput and other related information for several environments. See the texts for explanations.

Env	M_G	#CPU cores	#GPUs	rfps	cfps	in-game fps
Dota 2 1v1 [69]	1	60,000	256[a]	1.1M	2.8M	10
Dota 2 5v5 [69]	1	128,000	256[b]	493K	1.0M	7.5
StarCraft II (AlphaStar [8])[*]	12	4,200	256[c]	25K	50K	<4.4
Quake III [7]	30	$64 \times c^{\ddagger}$	N/A	N/A	N/A	N/A
StarCraft II (TStarBot-X [64])[†]	3	4,200	96	1.7K	4.2K	1.7
ViZDoom	1	1,152	32[d]	6.0K	8.2K	17.5
Pommerman	1	100	2	2.9K	20.0K	N/A

[*] All the three races: Terran, Protoss, Zerg

[†] Only one race: Zerg

[‡] The constant c denotes number of CPU cores assigned to a game core process, which is not reported in [7]

[a] K80

[b] P100

[c] The number 256 means 256 TPU-v3 cores

[d] M40

5. Conclusion and Future Work

We introduce an open source framework for competitive self-play based Multi-Agent Reinforcement Learning, referred to as TLeague. It is able to perform distributed training over a heterogeneous cluster (hybrid of GPU and CPU machines), achieving a high throughput and a reasonable scale-up. It can be run over any standard k8s cluster and be deployed on most Cloud Computing vendors (e.g., the Tencent Cloud). The code is well structured and is easy to extend for your own interested applications. Using common policy gradient and opponent sampling algorithms, we've shown that the agent trained with TLeague yields satisfactory results for several popular benchmark environments including StarCraft II (zvz full game), ViZDoom (CIG 2016 track 1) and Pommerman (NeurIPS 2018 2vs2 competition). It will be interesting to consider whether the TLeague large-scale training be applicable to even more complicated problems, e.g., the strategic video game CMANO [70], the trading system [71], the real-world robotics such as the FPV drone racing [72, 73] and the robot swarm [74, 75].

Acknowledgement

Thanks Qing Wang (drwang) for developing an early version of the framework. Thanks Zhuobin Zheng (jackzbzheng) and Jiaming Lu (loyavejmlu) for initiating the ViZDoom experiments during the internship with Tencent AI Lab. Thanks Yinyuting Yin (mailyyin), Tengfei Shi (francisshi), Bei Shi (beishi), Haobo Fu (haobofu) and Xipeng Wu (haroldwu) for helpful discussions on distributed training. Thanks Qingwei Guo (leoqwguo), Xinan

Jiang (xinanjiang), Feihu Zhou (hopezhou) for showing us the advanced usage of Tensorflow. Thanks Huayuan Xiao (howardxiao), Ling Cui (gracecui), Ang Li (marcriverli), Zhiguo Hong (zhiguohong), Jike Song (jikesong), Dekai Li (asinli), Guangyou Yu (garyyu), Hui Zou (joezou), Dandan Song (dandansong), Keyang Xie (keyangxie), Hongyu Zou (leonardzou), Shiyong Wang (warriorwang), Chong Zha (tillerzha), Xiaofei Wang (kendywang), Rennan Yao (liamyao) and many other colleagues from Tecent CSIG and TEG for preparing and maintaining the compute resources. Shuxing Li (meowli) and Jiawei Xu (ztjiaweixu) are funded by Tencent AI Lab RhinoBird Focused Research Program JR201986.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[2] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[3] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[4] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. *arXiv preprint arXiv:1710.03748*, 2017.

[5] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[6] Deheng Ye, Zhao Liu, Mingfei Sun, Bei Shi, Peilin Zhao, Hao Wu, Hongsheng Yu, Shaojie Yang, Xipeng Wu, Qingwei Guo, et al. Mastering complex control in moba games with deep reinforcement learning. In *AAAI*, pages 6672–6679, 2020.

[7] Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865, 2019.

[8] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[9] Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Julien Perolat, David Silver, Thore Graepel, et al. A unified game-theoretic approach to multiagent reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 4190–4203, 2017.

[10] David Balduzzi, Marta Garnelo, Yoram Bachrach, Wojciech M Czarnecki, Julien Perolat, Max Jaderberg, and Thore Graepel. Open-ended learning in symmetric zero-sum games. *arXiv preprint arXiv:1901.08106*, 2019.

[11] Zifan Li and Ambuj Tewari. Sampled fictitious play is hannan consistent. *Games and Economic Behavior*, 109:401–412, 2018.

[12] Brian Swenson, Soummya Kar, and Joao Xavier. Single sample fictitious play. *IEEE Transactions on Automatic Control*, 62(11):6026–6031, 2017.

[13] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

[14] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. Seed rl: Scalable and efficient deep-rl with accelerated central inference. *arXiv preprint arXiv:1910.06591*, 2019.

[15] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[16] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. Vizdoom: A doom-based ai research platform for visual reinforcement learning. *arXiv preprint arXiv:1605.02097*, 2016.

[17] Cinjon Resnick, Wes Eldridge, David Ha, Denny Britz, Jakob Foerster, Julian Togelius, Kyunghyun Cho, and Joan Bruna. Pommerman: A multi-agent playground. *arXiv preprint arXiv:1809.07124*, 2018.

[18] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

[19] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

[20] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*, 2018.

[21] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[22] Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. Reinforcement learning through asynchronous advantage actor-critic on a gpu. *arXiv preprint arXiv:1611.06256*, 2016.

[23] Alfredo V Clemente, Humberto N Castejón, and Arjun Chandra. Efficient parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1705.04862*, 2017.

[24] Albin Cassirer, Gabriel Barth-Maron, Thibault Sottiaux, Manuel Kroiss, Eugene Brevdo. Reverb: An efficient data storage and transport system for ml research. `https://github.com/deepmind/reverb`, 2020. [Online; accessed 01-June-2020].

[25] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, et al. Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*, 2020.

[26] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018.

[27] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062, 2018.

[28] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[29] Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G. Bellemare. Dopamine: A Research Framework for Deep Reinforcement Learning. 2018.

[30] Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C. Lawrence Zitnick. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 2659–2669. Curran Associates, Inc., 2017.

[31] Yuhang Song, Andrzej Wojcicki, Thomas Lukasiewicz, Jianyi Wang, Abi Aryan, Zhenghua Xu, Mai Xu, Zihan Ding, and Lianlong Wu. Arena: A general evaluation platform and building toolkit for multi-agent intelligence. In *AAAI*, pages 7253–7260, 2020.

[32] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

[33] trfl. `https://github.com/deepmind/trfl`.

[34] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[35] Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pages 157–163. Elsevier, 1994.

[36] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[37] Nikos Vlassis. A concise introduction to multiagent systems and distributed artificial intelligence. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 1(1):1–71, 2007.

[38] Julia Robinson. An iterative method of solving a game. *Annals of mathematics*, pages 296–301, 1951.

[39] Sriram Srinivasan, Marc Lanctot, Vinicius Zambaldi, Julien Pérolat, Karl Tuyls, Rémi Munos, and Michael Bowling. Actor-critic policy optimization in partially observable multiagent environments. In *Advances in neural information processing systems*, pages 3422–3435, 2018.

[40] Johannes Heinrich, Marc Lanctot, and David Silver. Fictitious self-play in extensive-form games. In *International Conference on Machine Learning*, pages 805–813, 2015.

[41] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. *arXiv preprint arXiv:1603.01121*, 2016.

[42] Kevin Waugh and J Andrew Bagnell. A unified view of large-scale zero-sum equilibrium computation. *arXiv preprint arXiv:1411.5007*, 2014.

[43] Neil Burch. Time and space: Why imperfect information games are hard. 2018.

[44] Michael Bradley Johanson. Robust strategies and counter-strategies: Building a champion level computer poker player. 2007.

[45] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In *Advances in neural information processing systems*, pages 1729–1736, 2008.

[46] Noam Brown, Adam Lerer, Sam Gross, and Tuomas Sandholm. Deep counterfactual regret minimization. In *International conference on machine learning*, pages 793–802. PMLR, 2019.

[47] Hui Li, Kailiang Hu, Zhibang Ge, Tao Jiang, Yuan Qi, and Le Song. Double neural counterfactual regret minimization. *arXiv preprint arXiv:1812.10607*, 2018.

[48] Michael I Jordan et al. Graphical models. *Statistical science*, 19(1):140–155, 2004.

[49] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[50] Sylvain Jeaugey. Nccl 2.0. *GTC*, 2017.

[51] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, 1999.

[52] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *Ieee Software*, 33(3):42–52, 2016.

[53] Ømq - the guide. https://zguide.zeromq.org/. Accessed November 6, 2020.

[54] Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data. https://developers.google.com/protocol-buffers. Accessed November 6, 2020.

[55] A high-performance, open source universal rpc framework. https://grpc.io/. Accessed November 6, 2020.

[56] Marko Luka. *Kubernetes in action*. Manning Publications, 2017.

[57] Jinja2. https://www.fullstackpython.com/jinja2.html. Accessed November 6, 2020.

[58] kubeflow. https://www.kubeflow.org/. Accessed November 6, 2020.

[59] Tencent cloud. https://cloud.tencent.com/. Accessed November 6, 2020.

[60] Tencent kubernetes engine. https://intl.cloud.tencent.com/product/tke. Accessed November 6, 2020.

[61] Tencent cloud cvm. https://intl.cloud.tencent.com/product/cvm. Accessed November 6, 2020.

[62] Cloud file storage. https://intl.cloud.tencent.com/product/cfs. Accessed November 6, 2020.

[63] Qing Wang, Jiechao Xiong, Lei Han, Meng Fang, Xinghai Sun, Zhuobin Zheng, Peng Sun, and Zhengyou Zhang. Arena: a toolkit for multi-agent reinforcement learning. *arXiv preprint arXiv:1907.09467*, 2019.

[64] Lei Han, Jiechao Xiong, Peng Sun, Xinghai Sun, Meng Fang, Qingwei Guo, Qiaobo Chen, Tengfei Shi, and Zhengyou Zhang. TStarBot-X: An open-sourced and comprehensive study for efficient league training in starcraft ii full game. *arXiv preprint arXiv:todo*, 2020.

[65] Vizdoom cig 2016 competition. http://vizdoom.cs.put.edu.pl/competitions/vdaic-2016-cig. Accessed November 6, 2020.

[66] Yuxin Wu and Yuandong Tian. Training agent for first-person shooter game with actor-critic curriculum learning. In *International Conference on Learning Representations*, 2017.

[67] ViZDoom testing code. https://github.com/mihahauke/VDAIC2017. Accessed November 6, 2020.

[68] ViZDoom doc. https://github.com/mwydmuch/ViZDoom/blob/master/doc/Types.md#gamestate. Accessed November 6, 2020.

[69] Open ai five. https://blog.openai.com/openai-five/. Accessed August 30, 2018.

[70] Command: Modern air naval operations. https://en.wikipedia.org/wiki/Command:_Modern_Air_Naval_Operations. Accessed November 6, 2020.

[71] Wenhang Bao and Xiao-yang Liu. Multi-agent deep reinforcement learning for liquidation strategy analysis. *arXiv preprint arXiv:1906.11046*, 2019.

[72] Welcome to airsim. `https://github.com/microsoft/AirSim`. Accessed November 6, 2020.

[73] Drone racing. `https://en.wikipedia.org/wiki/Drone_racing`. Accessed November 6, 2020.

[74] Werfel, Justin, Petersen, Kirstin, Nagpal, and Radhika. Designing collective behavior in a termite-inspired robot construction team. *Science*, 2014.

[75] Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. Programmable self-assembly in a thousand-robot swarm. *ence*, 345(6198):795–9, 2014.