# Optimizing Communication in Deep Reinforcement Learning with *XingTian*

Lichen Pan[§]
Peking University
Beijing, China
plch368@pku.edu.cn

Jun Qian[§,‡]
Noah's Ark Lab, Huawei
Beijing, China
jack.qian@huawei.com

Wei Xia
Noah's Ark Lab, Huawei
Beijing, China

Hangyu Mao
Noah's Ark Lab, Huawei
Beijing, China

Jun Yao
Noah's Ark Lab, Huawei
Beijing, China

Pengze Li
Peking University
Beijing, China

Zhen Xiao[‡]
Peking University
Beijing, China
xiaozhen@pku.edu.cn

## ABSTRACT

Deep Reinforcement Learning (DRL) achieves great success in various domains. Communication in today's DRL algorithms takes non-negligible time compared to the computation. However, prior DRL frameworks usually focus on computation management while paying little attention to communication optimization, and fail to utilize the opportunity of the communication-computation overlap that hides the communication from the critical path of DRL algorithms. Consequently, communication can take more time than the computation in prior DRL frameworks. In this paper, we present *XingTian*, a novel DRL framework that co-designs the management of communication and computation in DRL algorithms. *XingTian* organizes the computation in DRL algorithms in a decentralized way and provides an asynchronous communication channel. *XingTian* makes the communication execute asynchronously and aggressively and takes advantage of the communication-computation overlapping opportunity from DRL algorithms. Experimental results show that *XingTian* improves data transmission efficiency and can transmit at least twice as much data per second as the state-of-the-art DRL framework RLLib. DRL algorithms based on *XingTian* achieve up to 70.71% more throughput than RLLib-based ones with better or similar convergent performance. *XingTian* maintains high communication efficiency under different scale deployments and the *XingTian*-based DRL algorithm achieves 91.12% higher throughput than the RLLib-based one when deployed in four machines. *XingTian* is open-sourced and publicly available at https://github.com/huawei-noah/xingtian.

## CCS CONCEPTS

• **Computer systems organization** → *Special purpose systems*; • **Computing methodologies** → *Artificial intelligence.*

## KEYWORDS

Deep Reinforcement Learning, Asynchronous Communication, Decentralized Computation, Communication-Computation Overlap

**ACM Reference Format:**
Lichen Pan[§], Jun Qian[§,‡], Wei Xia, Hangyu Mao, Jun Yao, Pengze Li, and Zhen Xiao[‡]. 2022. Optimizing Communication in Deep Reinforcement Learning with *XingTian*. In *23rd ACM/IFIP International Middleware Conference (Middleware '22), November 7–11, 2022, Quebec, QC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3528535.3565249

## 1 INTRODUCTION

Reinforcement learning (RL) [55] is an artificial intelligence method that focuses on training intelligent *agents* to take a sequence of *actions* to maximize the *return*, i.e., cumulative *rewards* they receive from interactive *environments*. Deep Reinforcement Learning (DRL), which combines the concepts of RL and deep learning (DL), has become one of the most active areas of artificial intelligence. Most of today's DRL algorithms focus on solving the optimal *policies* for agents with the help of deep neural networks (DNNs). DRL has achieved great success in various domains, including gaming [19, 50, 52], robotics [2, 33], system optimization [35, 36, 60], protein structure prediction [24], and other imaginative fields.

Today's widely used DRL algorithms rely on massive direct interaction with the environment to collect data used to train the DNNs, since the internal state transition dynamics of the environments cannot be precisely expressed in advance in most cases. In DRL algorithms, the collected training data are usually called *rollouts*, which comprise a series of *rollout steps*. A rollout step is a tuple of the *observation* of the environment, the action

[§]Equal Contribution. [‡]Corresponding authors.


*Middleware '22, November 7–11, 2022, Quebec, QC, Canada*

ACM ISBN 978-1-4503-9340-9/22/11…$15.00
https://doi.org/10.1145/3528535.3565249

**Table 1: Time to Transmit Rollouts and to Train.**

| DRL Algorithms | Rollout Size (KB) | Trans. Time in RLLib[30, 31] (ms) | Trans. Time in Launchpad[61] w/ Reverb[8] (ms) | Training Time (ms) |
|---|---|---|---|---|
| PPO[49] | 138,585.32 | 367.81 | 95,765.88 | 1,297.53 |
| DQN[38] | 1,912.96 | 54.13 | 811.47 | 8.00 |
| IMPALA[13] | 13,855.20 | 301.34 | 12,567.10 | 32.07 |

taken by the agent, the reward and the next observation after the action is applied, as well as other useful information. Rollout steps from the continuous agent–environment interaction are usually grouped by subsequences referred to as *episodes*.

Parallel sampling is a widely adopted approach that improves the convergent performance of DRL algorithms by diversifying the collectively-encountered state spaces of the environment [13, 37], and reduces the turnaround execution time. To apply parallel sampling, the agent in DRL algorithms is usually split into multiple *explorers* and a *learner* for interacting with the environment and optimizing DNN parameters, respectively. During execution, the parallel explorers send rollouts to the learner, and the learner sends updated DNN parameters back to the explorers. The explorers and the learner usually run in different processes for better parallelism. Consequently, most of today's DRL algorithms rely on the orchestration of two primary heterogeneous computational workloads in separate processes, namely the interaction with the environment and the training of the DNNs, between which high-frequency communication is required.

We observe that the communication latency is non-negligible compared to computation time in today's DRL algorithms. There is an optimization opportunity of overlapping communication and computation to hide the communication from the critical path of DRL algorithms. However, few prior DRL frameworks consider optimizing communication in DRL algorithms or taking advantage of the communication-computation overlapping opportunity. Prior DRL frameworks [7, 14, 20, 30, 31, 39, 46, 58] usually organize the computational components of DRL algorithms into task graphs, and use the centralized control logic to specify the components' execution order. In prior DRL frameworks, the communication does not start until the receiving component is allowed to run by the centralized control logic and then asks for data, even though the data may have been ready for a long time. In such a programming model, the communication between components has no chance to execute simultaneously with the computation. Consequently, complex operations associated with data transmission across processes or machines get in the way of the critical computations for DRL algorithms, i.e., the interaction with the environment and updating DNN parameters, which results in compromised performance. Table 1 illustrates that the communication can take more time than the computation in prior DRL frameworks.

In this paper, we present *XingTian*, a novel DRL framework that co-designs the management of communication and computation in DRL algorithms. *XingTian* gets rid of the centralized control logic and makes the explorers and the learner execute in a decentralized way. Based on the decentralized organization, *XingTian* provides an asynchronous communication channel between the explorers and the learner to push data to the desired destination as soon as the data are generated. The communication can thus be initiated by the sender once the data are ready without caring about the recipient. This way, *XingTian* makes the communication execute asynchronously and aggressively and utilizes the optimization opportunity of the communication-computation overlap from DRL algorithms. We also make sure that *XingTian* does not introduce the risk of unnecessary data transmission nor significant extra memory overheads compared to prior DRL frameworks.

We implement *XingTian* and make it expose researcher-friendly interfaces to construct DRL algorithms. We provide a DRL algorithm zoo based on *XingTian* and algorithms from the DRL algorithm zoo are applied in production projects of Huawei. Population-based training (PBT) is a widely-used algorithm to search for the optimal combination of hyperparameters for DRL algorithms. We can naturally extend *XingTian* to support PBT.

We evaluate the data transmission efficiency of *XingTian* and other DRL frameworks with a dummy DRL algorithm that keeps DRL algorithms' communication mode. Results show that *XingTian* is able to transmit at least twice as much data per second as RLLib [30, 31], and can transmit at least one order of magnitude more data per second than Acme [20] with Launchpad [61] and Reverb [8]. The improvement of the data transmission efficiency of *XingTian* is mostly attributed to its asynchronous aggressive communication model. The throughput of DRL algorithms is defined as the number of rollout steps consumed for DNN training per second, and reflects the speed at which DRL algorithms consume rollouts and optimize policies. Experimental results show that *XingTian*-based DRL algorithms achieve higher throughput than those implemented in the state-of-the-art DRL framework RLLib by up to 70.71% with better or similar convergent performance due to the utilization of the communication-computation overlapping chance. Moreover, *XingTian* maintains high communication efficiency under different scale deployments and the *XingTian*-based DRL algorithm achieves 91.12% higher throughput than the RLLib-based one when deployed in four machines.

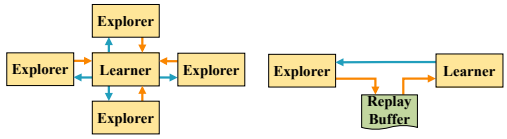
This paper makes the following contributions.

- We analyze the communication characteristics of DRL algorithms and figure out the optimization opportunity of the communication-computation overlap from DRL algorithms.
- We propose the design principles of co-designing communication and computation for DRL frameworks to take advantage of the opportunity of overlapping communication and computation and present the design of *XingTian*.
- We implement and evaluate *XingTian*. Results indicate that *XingTian* has better data transmission efficiency and *XingTian*-based DRL algorithms achieve higher throughput than those based on the state-of-the-art DRL framework RLLib with better or similar convergent performance. Moreover, *XingTian* maintains high communication efficiency under different scale deployments.

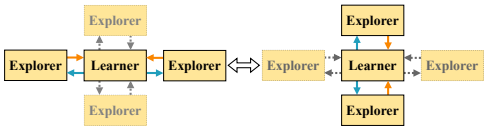
## 2 BACKGROUND AND MOTIVATION

In this section, we first briefly sort out the development of DRL algorithms to figure out their communication characteristics and the optimization opportunity, and then discuss the shortcomings of existing DRL frameworks in communication management[1].

[1]The more detailed review of related work is in Section 6.




(a) PPO (actor-critic-based, on-policy) (b) DQN (value-based, off-policy)




(c) IMPALA (actor-critic-based, off-policy)

**Figure 1: Execution of Three Representative DRL Algorithms.**

## 2.1 Communication Characteristics of DRL Algorithms

DRL algorithms deal with sequential decision-making problems that can be formalized as a Markov decision process (MDP) [55]. An MDP is defined by a four-tuple $(\mathcal{S}, \mathcal{A}, T, R)$, where $\mathcal{S}$ is the state space of the environment; $\mathcal{A}$ is the action space of the agent; $T$ is the transition function and $T(s, a, s')$ is the probability that the action $a$ in state $s$ will lead to next state $s'$; $R$ is the reward function and $R(s, a, s')$ gives the immediate reward $r$ after the transition from state $s$ to $s'$ when the agent takes action $a$. Policy $\pi$ is a mapping from states to the probabilities of selecting each possible action. If the agent is following policy $\pi$ at time $t$, $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. Ideally, if the transition function $T$ and reward function $R$ are known, we can find the optimal policy $\pi^*$ that maximizes the total return through dynamic programming methods such as policy iteration and value iteration [4, 5, 55]. However, this is an almost impossible condition in practice, and most DRL algorithms today rely on massive direct interaction with the environment by trial and error to sample state transitions.

Model-based DRL algorithms try to learn a model of the transition dynamics of a particular environment and work well for problems relying on lookahead search [25, 48, 50–52]. In contrast, model-free DRL algorithms treat the model as a black box and have the agent interact with the environment to observe state transitions and rewards, which are more widely used as most problems are challenging to produce precise models [34]. There are three main families of model-free DRL algorithms: *policy-based* algorithms [32, 56, 59] maintain and optimize a policy explicitly; *value-based* algorithms [19, 38, 45] learn the value function or action-value function (Q function) and extract the target policy accordingly; and *actor-critic-based* [27, 49] algorithms combine the former two types of algorithms where the policy-based actor learns the optimal policy directly, and the value-based critic evaluates the actions taken by the actor.

DNNs play different roles in different kinds of DRL algorithms, including building the environment's state transition model, expressing the policy directly, or evaluating the expected total return of the current state of the environment. The rollouts used to optimize parameters of those DNNs are collected on-the-fly. *On-policy* DRL algorithms evaluate and optimize the same policy that is used to interact with the environment, which means rollouts generated with a specific version of DNN parameters can only be used to optimize the DNN parameters of the same version. *Off-policy* DRL algorithms can evaluate and optimize a policy different from the one used for rollout generation, which means the DNN parameters can be optimized utilizing older rollouts.

Parallel sampling is a widely adopted approach that improves the convergent performance of DRL algorithms by diversifying the collectively-encountered state spaces of the environment, and reduces the turnaround execution time. To apply parallel sampling in value-based DRL algorithms, multiple explorers interact with the environment simultaneously and put the rollout steps in a replay buffer [47], while the learner samples experiences from the replay buffer asynchronously, trains the DNN, and broadcasts weights to explorers [21, 26, 40]. Classical policy-based and actor-critic-based DRL algorithms are usually on-policy. Some solutions make all the explorers and the learner work synchronously in a lock-step manner [10, 17, 18], others tolerate a certain degree of policy lag and instability due to asynchronization for higher throughput [3, 37]. Importance Weighted Actor-Learner Architecture (IMPALA) [13] and subsequent algorithms [12, 29, 42] utilize V-trace, which brings the off-policy feature to actor-critic-based DRL algorithms.

We further illustrate the communication characteristics for DRL algorithms of different types with three popular representative algorithms in Fig. 1: Proximal Policy Optimization [49] (PPO, actor-critic-based & on-policy), Deep Q-Learning [38] (DQN, value-based & off-policy), and IMPALA [13] (actor-critic-based & off-policy).

*PPO.* Fig. 1(a) shows the execution of PPO. Due to PPO's on-policy constraint, the learner and the explorers run synchronously: the learner waits to collect rollouts from all the explorers for training, and all the explorers then wait for DNN parameters of the latest version from the learner before they can interact with the environment.

*DQN.* As shown in 1(b), DQN maintains a replay buffer for experience replay. During execution, the explorer keeps putting rollout steps into the replay buffer. After the number of rollout steps in the replay buffer exceeds the configured threshold, each time the explorer inserts a certain number of rollout steps, the learner performs a training session with a batch of rollout steps sampled from the replay buffer. The learner sends out DNN parameters every a few training sessions, and the explorer then applies the latest version of the DNN parameters before it continues generating subsequent rollout steps. In DQN, the explorer and the learner can run asynchronously, and communication is quite busy among the explorer, the learner, and the replay buffer.

*IMPALA.* As shown in Fig. 1(c), explorers and the learner in IMPALA communicate with each other directly. The learner can start training when it only collects rollouts from a portion of explorers and then sends updated DNN parameters exactly to the explorers it gets rollouts from. Because IMPALA is off-policy, the learner can utilize rollouts generated with the relatively older version of DNN parameters from other explorers in later training. This way, the learner in IMPALA updates the DNN parameters more frequently. The communication between the asynchronous learner and explorers is more frequent, too.

From the brief overview above, we can conclude that most of today's DRL algorithms rely on the orchestration of two primary heterogeneous computational workloads, namely interaction with the environment and training of the DNNs, between which high-frequency communication is required. Besides, there is an optimization opportunity of the **communication-computation overlap** to hide the communication from the critical path of DRL algorithms. For example, there is no strong dependency between the learner and the explorers in off-policy DRL algorithms, and they can run asynchronously. More specifically, some explorers can produce and send out rollouts for the learner to use later while the learner is training. As a result, the learner can continue working without spending too much time waiting for rollout transmission. However, few existing DRL frameworks utilize such an optimization chance.

## 2.2 Shortcomings of Prior DRL Frameworks

Existing codebases providing reference implementations of DRL algorithms [9, 11, 15, 16, 28, 43, 44] serve as helpful tools to get started with DRL algorithms, which describe different DRL algorithms by separate overall control sequences. To improve flexibility and modularity, studies for DRL frameworks [7, 14, 20, 30, 31, 39, 46, 58] focus on abstracting out reusable computational components for DRL algorithms. Most existing DRL frameworks follow the programming model that organizes the computational components into a task graph and then specify their execution order with centralized control logic. In such a programming model, the communication is usually initiated by the receiving component and has no chance to execute simultaneously with the computation.

Several DRL frameworks [7, 14, 20] always insert a data management buffer between the explorers and the learner, and make them always communicate indirectly through the buffer. For example, Acme [20] is a single-thread DRL framework, and can be deployed in a distributed manner with the help of Reverb [8] and Launchpad [61]. Reverb implements the data buffer, and Launchpad is a distributed computing framework that builds the task graph. In such a DRL framework, the central control logic defines the execution sequence of the the explorers the learner, i.e., the explorers ask for the DNN parameters of the latest version, interact with the environment, and put rollouts into the data buffer, and then the learner retrieves the rollouts from the buffer before updating DNN parameters.

RLLib [30, 31] is a DRL framework based on Ray [39], which builds a task graph to organize computational components and uses a global control store, an in-memory distributed object store, and wrapped RPCs to manage communication. RLLib can make the explorers and the learner communicate with each other directly when the DRL algorithm itself does not need a replay buffer, and relies on the central logic to define the execution sequence of the explorers and the learner. Communication in RLLib cannot start until the learner or explorer is scheduled to run and asks for data.

Therefore, despite the opportunity of overlapping the communication and computation in DRL algorithms, communication and computation in existing DRL frameworks have to be performed serially. Consequently, operations related to data transmission such as memory copy, serialization & deserialization, compression & decompression, and NIC-bandwidth-bounded transfer across machines get in the way of the critical computations, i.e., interaction with the environment and updating DNN parameters. Table 1 illustrates the size of rollouts used for a training iteration from PPO, DQN, and IMPALA, the measured transmission time in RLLib and Launchpad (with a Reverb buffer), respectively, as well as the corresponding training time[2]. We can tell from Table 1 that the communication latency is no longer negligible compared to the computation time. Besides, the communication can spend more time than computation in existing DRL frameworks, which we believe is unacceptable.

## 2.3 Summary

In this section, we make it clear that today's DRL algorithms rely on the orchestration of two computational workloads with high-frequency communication requirement, and show that most existing DRL frameworks fail to utilize the communication-computation overlapping opportunity, which compromises the performance of DRL algorithms. As a result, it is quite important and remains an open research question to design a DRL framework that leverages the opportunity of overlapping communication and computation in DRL algorithms and stops independent communication and computation from blocking each other.

# 3 DESIGN

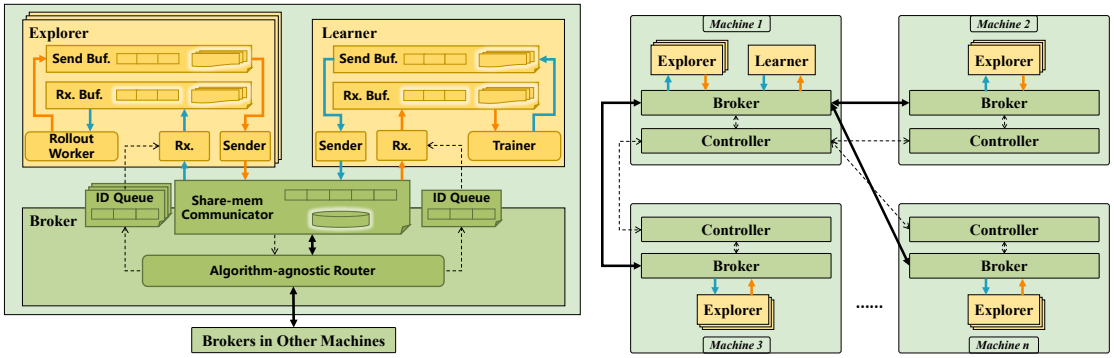
In this section, we first illustrate the design principles of a DRL framework that can overcome the limitations of prior work in managing communication. We then present the architecture of *XingTian*, a novel DRL framework that co-designs the management of communication and computation in DRL algorithms and takes advantage of the communication-computation overlapping opportunity.

## 3.1 Design Principles

As discussed in Section 2, although there are various kinds of DRL algorithms and parallel sampling is widely adopted, most DRL algorithms rely on the orchestration of two computational workloads with high-frequency communication requirements, namely the interaction with the environment and the training of the DNNs. Compared to traditional distributed computing jobs, the dependency between the computational workloads in DRL algorithms is relatively simpler and we argue that existing DRL frameworks overuse the abstraction of task graphs. Besides, having the communication initiated by the receiver makes the communication and computation block each other despite the overlapping opportunity. Making the receiver pull data avoids unnecessary data transmission. However, in most DRL algorithms, rollouts generated by explorers are always used by the learner, and the DNN parameters updated by the learner are always applied by explorers, so transmitting data more aggressively in DRL algorithms will not introduce the risk of unnecessary data transmission.

To overcome the limitations of existing DRL frameworks, we need to fully exploit the overlapping opportunity by enabling the communication and computation to cooperate with each other based on the characteristics of DRL algorithms.

[2]Setups for measurement are the same as those used in Section 5.




(a) Decentralized Computation and Asynchronous Communication. (b) A Distributed Deployment Demonstration of *XingTian*.

**Figure 2: Architecture of *XingTian*. In both sub-figures, dashed arrows denote the transmission of lightweight message headers or control commands, while the bold arrows denote data block transmission. More precisely, orange bold arrows denote the transmission of rollouts and blue bold arrows denote the transmission of DNN parameters.**

*Computation Management.* We get rid of the task graph and the centralized control logic, and make the explorers and the learner execute in a decentralized way instead. In the decentralized model, computational workloads are driven by the arrival of the data awaited, and can send out produced data immediately.

*Communication Management* We provide an asynchronous communication channel between the explorers and the learner to push data to the desired destinations aggressively as soon as they are generated. The communication can thus be initiated by the sender once the data are ready without caring about the recipient. We can utilize the optimization opportunity of overlapping communication from DRL algorithms and computation by making the communication start aggressively. Besides, the communication channel is algorithm agnostic and can be applied in various DRL algorithms. Consequently, we can encapsulate complicated operations associated with communication across processes or machines within the framework's communication channel, and leave only simple local buffer reads and writes for the computational workloads to deal with.

As shown in Section 5, not only in off-policy algorithms, but even in on-policy algorithms like PPO, the DRL framework that follows these design principles can exploit the optimization opportunity of the communication-computation overlap and achieve performance gain.

## 3.2 Architecture of *XingTian*

To realize these design principles, we design *XingTian* as illustrated in Fig. 2. Fig. 2(a) illustrates the decentralized computation and asynchronous communication of *XingTian*, and Fig. 2(b) demonstrates that *XingTian* is scalable and how *XingTian* is deployed across several machines.

### 3.2.1 *Decentralized Computation and Asynchronous Communication*

As illustrated in Fig. 2(a), *XingTian* organizes DRL algorithms with three kinds of processes. The *explorer* process manages interaction with the environment, the *learner* process manages DNN training, and the *broker* process manages communication between the explorer and the learner processes.

The *rollout worker* thread of the explorer process is the workhorse performing rollout generation. Typically several explorer processes execute simultaneously for parallel sampling. The generated rollout steps are packaged in messages, each comprising a message body and a message header that is generated by *XingTian*. The *send buffer* and *receive buffer* are designed for intra-process data transmission and staging. The message headers are put in the header queue of the buffer, while the message bodies are inserted into the data list of the buffer.

The learner process has an almost symmetrical structure to the explorer process. The *trainer* thread of the learner process is the workhorse for generating the DNN parameters of the latest version. The updated DNN parameters are encapsulated into the message body, and *XingTian* generates the corresponding message header. The learner process also maintains a send buffer and a receive buffer.

Inter-process communication relies on the *shared memory communicator* created by the broker process. The shared memory communicator holds message headers in the header queue and keeps message bodies inside the object store implemented via shared memory for zero-copy communication among processes. The *ID queue* is used to pass object IDs of message bodies in the object store with other metadata inside the message header across processes. The broker maintains a separate ID queue for each explorer/learner process.

As soon as a message is produced by the workhorse thread (rollout worker thread or trainer thread), it is stored in the send buffer maintained in the corresponding process. The *sender* thread that monitors the send buffer's header queue then finds the new message and immediately transmits the message to the shared memory communicator. The message header contains metadata such as the source and destination of the message. After the message body is inserted into the object store of the shared memory communicator, the object ID is also attached to the message header

by the sender thread. The *algorithm-agnostic router* monitors the header queue in the shared memory communicator. Once the router notices a new message comes, it parses the header to find out the destination of the message. The message carrying updated DNN parameters from the learner is usually broadcast to multiple explorers and thus the message header may contain multiple destinations, while the message of rollouts is always sent to the single learner. The algorithm-agnostic router then puts the message header that contains the message body's object ID into the ID queues associated with all the destination processes. The *receiver* thread that monitors its ID queue in each destination process then retrieves the message body from the object store and puts the whole message into the receive buffer immediately. At this point, messages that have completed transmission are just waiting to be consumed. When *XingTian* is deployed among multiple machines, the algorithm-agnostic router sends the message with one or more remote destinations out to the router in each target machine. After receiving a message from another machine, the algorithm-agnostic router inserts the message body into the object store, attaches the object ID to the message header, and puts the message header in all the target ID queues. The workhorse threads still consume messages from the local buffer and will not perceive any difference.

The workflow presented above illustrates the decentralized computation and asynchronous communication in *XingTian*. *XingTian* does not use central logic to control the execution flow of DRL algorithms. On the contrary, workhorse threads in different processes are driven by the arrival of the data awaited and execute autonomously by aggressively consuming received messages and producing new messages. The broker process plays a key role in creating and maintaining the asynchronous communication channel. The broker process is totally different from the data management buffer in any existing DRL frameworks. The broker process only focuses on pushing data to the desired destinations as soon as possible with the help of the shared memory communicator and the algorithm-agnostic router inside, and does not understand or process the data on behalf of DRL algorithms. The algorithm-agnostic router dispatches messages to the desired destinations by informing the corresponding receivers of object IDs, and the receiver copies the message body to the local buffer immediately. *XingTian* makes IO operations of the workhorse threads only related to the local buffers, and guarantees that data transmission across processes or machines is performed aggressively and efficiently by the asynchronous communication channel.

*Utilization of the Communication-Computation Overlapping Opportunity.* *XingTian* manages to utilize the optimization opportunity of overlapping communication and computation in DRL algorithms because the timing to initiate communication in *XingTian* is only decided by whether the data to transmit are prepared. *XingTian* decouples operations related to data transmission such as memory copy, serialization & deserialization, compression & decompression, and NIC-bandwidth-bounded transfer across machines from the execution of the computational workloads and makes them happen in parallel. With such a design, rollouts produced by rollout worker threads of explorer processes are put into the buffer within the address space of the trainer thread in the learner process as soon as possible, even when the trainer thread is busy updating DNN parameters with rollouts received earlier. The updated DNN parameters are also transmitted to the explorers immediately.

XingTian accelerates on-policy DRL algorithms because explorers still execute asynchronously although the learner and explorers run synchronously. In XingTian-based on-policy DRL algorithms, fast explorers' rollout transmission can overlap with slow explorers' environment-interacting computation. Once fast explorers generate rollouts they initiate rollout transmission immediately without waiting for pulling requests or being held back by slow explorers. However, in prior DRL frameworks, the transmission is not started until the central control logic ensures all the explorers have generated rollouts and allows the learner to ask for the data.

For DRL algorithms that require a replay buffer for experience replay, *XingTian* leaves the replay buffer maintenance inside the trainer thread in the learner process. As a result, the sampling from the replay buffer no longer involves cross-process communication, and can be deemed as the utilization of data already stored in the local buffer. Such a design decision is also the utilization of the optimization opportunity of the communication-computation overlap in essence.

By exploiting the optimization opportunity of overlapping the communication and computation, *XingTian* does not introduce the risk of unnecessary data transmission as discussed in Section 3.1. Besides, *XingTian* does not bring in significant extra memory overheads compared to prior DRL frameworks. The queues are always filled in with lightweight metadata of message headers. The shared memory object store is used for zero-copy communication across processes. The send and receive buffers in the explorer & learner processes hold data directly generated or consumed by the workhorse threads.

#### 3.2.2 *Distributed Deployment of* XingTian

Fig. 2(b) shows a distributed deployment demonstration of *XingTian*. Section 3.2.1 introduces *XingTian*'s architecture from the view of execution and omits the *controller* process. The controller is algorithm agnostic and responsible for managing the life cycle of all the above-mentioned processes, establishing the global fabrics, and dispatching control commands. *XingTian* is scalable and its configuration file contains information such as IPs of the machines to deploy *XingTian*, the machine to start the learner, and the number of explorers in each machine. After launched in one machine, *XingTian* starts a controller process locally and logs into other machines automatically to start a controller process in each machine. *XingTian* connects controllers with a fully connected fabric to dispatch control commands. We call the controller in the machine where *XingTian* is launched the *center controller*. Upon initialization, *XingTian* broadcasts control commands to each controller via the center controller to create brokers, the learner, explorers, and the communication channel inside each machine. In addition, *XingTian* creates another fabric among brokers in different machines to transmit data between machines. The machine in which the learner is located is always the center for data transmission. Fig. 2(b) presents active connections of the fabrics when the learner is located in the same machine as the center controller.

The center controller also collects and visualizes statistics from explorers and the learner. Workhorse threads put statistic

messages to the local send buffers periodically, and the algorithm-agnostic router sends the messages with statistics to the center controller directly or through the algorithm-agnostic router of the broker process in the center controller's machine. When the center controller realizes that the training goal has been achieved, e.g., the learner has consumed enough rollout steps or the explorers have received the target return, the center controller broadcasts commands to shutdown all processes and release resources.

# 4 IMPLEMENTATION AND EXTENSION OF *XINGTIAN*

## 4.1 Implementation Details

We implement *XingTian* in python. The message headers in *XingTian* are organized as python `dicts`. The message header queue in send/receive buffer is based on `queue.Queue`, and the queues used across the boundary of processes (i.e., the header queue in the shared memory communicator and the ID queues) are implemented via `multiprocessing.Queue`. Both `Queue` provide a `get` method that blocks when empty. Therefore, the threads that monitor these queues can make messages flow aggressively through the asynchronous communication channel in an event-driven manner, i.e., once a new message header is put into the queue, the blocking `Queue.get` invoked in the monitoring thread returns, and the monitoring thread starts the corresponding data transmission immediately.

The message bodies have to be serialized before they are inserted into the object store [57] and then deserialized when put into the receive buffer. Compression is often applied to reduce network traffic and memory usage. Meanwhile, compression and decompression also introduce non-negligible computational overheads. So we leave compression a configurable option in *XingTian*, and *XingTian* compresses message bodies larger than 1 MB by default. When compression is enabled, *XingTian* uses the LZ4 algorithm to compress message bodies when they are inserted into the object store of the shared memory communicator, and the message bodies are then decompressed when they are fetched to the receive buffers.

## 4.2 Construct DRL Algorithms with *XingTian*

As discussed in Section 2, DRL algorithms rely on orchestrating two computational workloads with high-frequency communication requirement, i.e., rollout generation and DNN training. *XingTian* handles issues about the orchestration and exposes interfaces friendly to researchers to implement specific computational logic.

Given that *XingTian* takes care of the computation organization and communication management, the remained problems related to implementing the computational logic of a DRL algorithm are as follows.

- With which environment is the DRL algorithm going to interact?
- What DNN structure should be adopted for the policy network / value network / Q network / environment model?
- How to organize the rollouts and to train the DNNs with the collected rollouts?
- How to use the DNNs to interact with the environment and collect rollouts?

*XingTian* exposes interfaces encapsulated in four classes so that researchers can implement computation logic to solve the above four problems. The classes include `Environment`, `Model`, `Algorithm`, and `Agent`.

• The `Environment` class is a wrapper for both widely-used testbed environments and self-defined ones and exposes standard gym-style [6] interfaces such as `init`, `reset`, and `step`, etc.

• The `Model` class holds the definition for DNNs and corresponding APIs. Researchers are free to define the DNNs with any deep learning framework, e.g., PyTorch [41], TensorFlow [1], and MindSpore [22], a new deep learning computing framework.

• The `Algorithm` class is used to specify how to update the DNNs with collected rollouts. Typically researchers need to implement the `train` and `prepare_data` interfaces. The `train` interface is used to define the computation logic for DNN training, and multi-GPU technologies can be applied here for acceleration. Researchers specify how to organize the received rollouts by implementing the `prepare_data` interface, and the maintenance of the replay buffer also happens here if necessary. For researchers' convenience, *XingTian* provides implementations of several kinds of replay buffers. Besides, `Algorithm` contains other implemented methods, e.g., the method for DNN inference, and the method to save the checkpoints of the DNNs periodically to restore DNN parameters after failure , which provides sufficient fault tolerance for DRL algorithms without significant overheads.

• The `Agent` class is used to specify how to interact with the environment for rollout generation. The `Agent` has a variable of the `Algorithm` type to maintain copies of the DNNs. Typically researchers need to implement the `infer_action` and `handle_env_feedback` interfaces. Researchers define how to generate actions given the observation of the environment in the `infer_action` interface, and specify how to sort out observations and rewards from the environment to suit different DRL algorithms in the `handle_env_feedback` interface.

The configuration file of *XingTian* is used to combine the implemented classes together for a specific DRL algorithm. *XingTian* instantiates them in the rollout worker thread and trainer thread accordingly upon initialization.

We further provide a DRL algorithm zoo based on *XingTian* that covers a wide range of DRL algorithms of different types, including model-based algorithms (e.g., MuZero [48]) and all three types of model-free algorithms (e.g., DQN [38], PPO [49], DDPG [32], IMPALA [13], etc.). The DRL algorithm zoo shows that *XingTian*'s design is suitable for a wide variety of DRL algorithms. Besides, algorithms from the DRL algorithm zoo are applied in production projects of Huawei.

## 4.3 Extending *XingTian*: Supporting Population-Based Training

A DRL algorithm usually has many configurable hyperparameters, which have a significant impact on the convergence of the DRL algorithms. PBT [23] is a widely-used algorithm to search for the optimal combination of hyperparameters. *XingTian* provides better usability for researchers by supporting PBT based on its architecture natively. We first briefly present how PBT works and then introduce how *XingTian* naturally supports PBT.

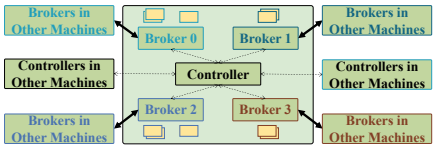



**Figure 3: A PBT Example with Four Populations in *XingTian*.**

**Table 2: Specification of the Experimental Platform.**

| | Specification |
|---|---|
| Server | FusionServer Pro 2288H V5 Rack Server |
| Processor | Intel(R) Xeon(R) Gold 6154 CPU @ 3.00 GHz (72-core, 24.75MB LLC) |
| DRAM | 1 TB @ 2666 MT/s |
| GPU | Tesla V100 32GB * 1 |
| Ethernet NIC | Intel Corporation Ethernet Connection X722 for 1GbE |

The PBT algorithm starts with multiple populations, each of which takes a different combination of hyperparameters. A center scheduler periodically evaluates each population's behavior based on specified metrics. For each evolution interval, the scheduler eliminates the population with the worst metrics, and calculates a new hyperparameter combination through mutation, crossover, and other strategies. Finally, the scheduler starts a new population to replace the eliminated one. The PBT algorithm ends after running for a certain number of generations or after the desired values of the metrics are reached. For DRL algorithms, PBT usually takes the average episode return as the metric. Besides, DNN weights of the best population in the last generation are applied to the new population so that it can catch up with others at the beginning.

Fig. 3 illustrates a PBT example with four populations in *XingTian*. During initialization, *XingTian* creates one controller and four brokers with different ranks in each machine. Then *XingTian* creates the fabric among brokers with the same rank. Brokers with different ranks do not communicate with each other. *XingTian* supports separated populations via isolated broker sets with the learner and explorers attached to each broker set. Learners from different populations can be deployed in different machines to fully use GPUs in each machine. The center controller in *XingTian* acts as the center scheduler of PBT. For each evolution interval, the center controller evaluates metrics from each population, decides which population to eliminate, and calculates the new hyperparameter combination. The center controller broadcasts commands to kill all processes of the eliminated population and starts the new population by starting a new broker set with the new learner and explorers. Researchers can deploy PBT based on *XingTian* by specifying the number of populations and the lists of alternative hyperparameters in the configuration file.

# 5 EVALUATION

In this section, we evaluate *XingTian* and illustrate that ❶ *XingTian* improves the data transmission efficiency in the communication mode of DRL algorithms with its asynchronous and aggressive communication model, ❷ DRL algorithms based on *XingTian* achieve higher throughput with better or similar convergent performance compared to those based on the state-of-the-art DRL framework due to leveraging the optimization opportunity of the communication-computation overlap, and ❸ *XingTian* maintains high communication efficiency under different scale deployments.

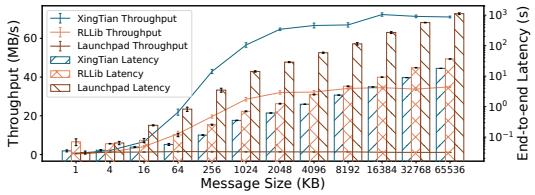
There are several DRL frameworks we can compare *XingTian* against, i.e., Intel Coach [7], Surreal [14], Acme [20] (with Reverb [8] and Launchpad [61]), and RLLib [30, 31]. All these frameworks are open-sourced. However, the repositories of Intel Coach and Surreal are not actively maintained. Besides, there is no published paper corresponding to Intel Coach, and there are only technical reports but not research papers introducing Acme as well as Reverb and Launchpad. As a result, we mainly compare *XingTian* with RLLib [30, 31], the state-of-the-art DRL framework with publications and an open-sourced codebase maintained actively. We also conduct several experiments with respect to Acme, Reverb, and Launchpad. We use RLLib of version 1.8.0., Acme of version 0.4.0, Reverb of version 0.7.0, and Launchpad of version 0.4.1.

We conduct all the experiments in this section in the experimental platform described in Table 2. Benchmarks we use for evaluation will be introduced in the following subsections.

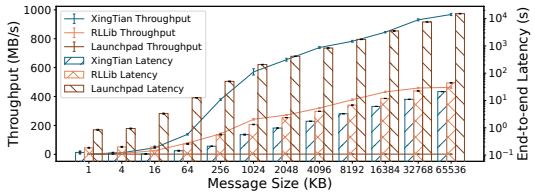
## 5.1 Data Transmission Efficiency

To evaluate the data transmission throughput of *XingTian* and other DRL frameworks, we design a dummy DRL algorithm, where the explorers only send out a certain number of messages of a configurable size, while the learner asynchronously receives the messages and reports the end-to-end latency and data transmission throughput after receiving all messages. Every explorer is configured to send 20 messages in total, and the learner receives messages asynchronously for 20 rounds. For example, if there are 16 explorers, in each round the learner receives 16 messages without caring about the senders of these messages and reports that this round is over. This way, we can evaluate the data transmission throughput expressed as the size of the messages received by the learner per second. We omit the data transmission in the other direction in real DRL algorithms, i.e., the learner does not broadcast anything to explorers. The reason is that the throughput of DRL algorithms measures how fast rollouts can be transmitted from the explorers to the learner and then be consumed to optimize the policy. We design the dummy DRL algorithm to keep the communication mode of DRL algorithms and to evaluate the maximum throughput of rollout transmission of DRL algorithms in the ideal situation where the explorers send out messages aggressively without waiting for DNN parameters or interacting with the environment, and the learner receives messages asynchronously in each iteration without computational workloads.

In *XingTian*, we implement this dummy DRL algorithm by implementing corresponding interfaces to make the rollout worker thread and trainer thread behave accordingly; in RLLib, we implement the dummy DRL algorithm with RLLib's low-level data streaming API. We also implement the dummy DRL algorithm in Launchpad by inserting a Reverb-based buffer between the dummy explorers and the dummy learner. The technical report of Launchpad suggests that a DRL algorithm can be implemented separately on Launchpad, and the explorers and the learner can communicate with each other directly. However, in most practices, DRL algorithms are implemented based on Acme that always inserts a data management buffer (usually implemented with Reverb)

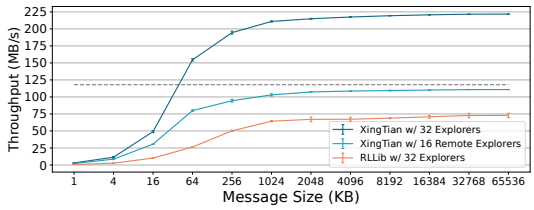



(a) One Explorer.

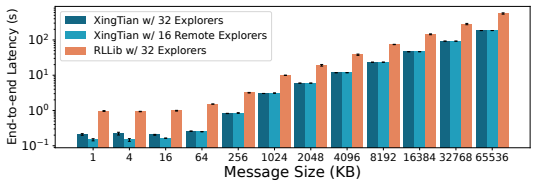



(b) 16 Explorers.

**Figure 4: Data Transmission Results in a Single Machine.**




(a) Throughput. The dashed line denotes the NIC bandwidth between machines.




(b) End-to-end Latency.

**Figure 5: Data Transmission Results in two Machines.**

between the explorers and the learner, and Launchpad is only used for distributed deployment. Therefore, we keep the structure with a data buffer in the dummy DRL algorithm to evaluate the data transmission efficiency of Acme with Launchpad and Reverb.

We evaluate the data transmission efficiency using messages that vary in size between 1KB and 64MB, which covers the range of rollout message sizes in typical DRL algorithms according to our experience. Results in this subsection are averaged from five runs, and error bars are attached to all the figures.

Fig. 4 illustrates the data transmission throughput and the end-to-end latency reported by the learner after receiving messages of 20 rounds from all explorers when deploying the dummy DRL algorithm in a single machine. Fig. 4(a) shows the results when the dummy DRL algorithm is configured with one explorer and Fig. 4(b) shows the results when there are 16 explorers. With one explorer, *XingTian* achieves data transmission throughput of 71.01 MB/s when the message is 64 MB, which is 103.32% higher than RLLib. With 16 explorers, such performance gains still hold and *XingTian* achieves 108% higher data transmission throughput than RLLib at 967.91 MB/s when the message is 64 MB. *XingTian* doubles the data transmission throughput in a single machine compared to RLLib because data transmission in *XingTian* starts immediately once the data are ready instead of waiting for the requirement originated from the recipient. Such a rapid transmitting pattern helps a lot even if there is only one explorer. Fig. 4 also illustrate the end-to-end latency and data transmission throughput of the dummy DRL algorithm implemented with Launchpad and Reverb. In both cases with one explorer and 16 explorers, the data transmission throughput of Launchpad and Reverb is always lower than 2 MB/s. It is noticeable that deploying more explorers does not improve the data transmission throughput of Launchpad and Reverb, and the data buffer based on Reverb is the bottleneck. We also evaluate the data transmission efficiency of the dummy DRL algorithm implemented separately with Launchpad (not illustrated in the figures). The solely Launchpad-based dummy DRL algorithm achieves data transmission throughput of no more than 10 MB/s and 100 MB /s, with one explorer and 16 explorers, respectively. Although the data transmission throughput of the solely Launchpad-based dummy DRL algorithm is better than when there is a Reverb-based data buffer, it is still not comparable to *XingTian*.

Fig. 5 presents data transmission results when deploying the dummy DRL algorithm in two machines. We deploy the dummy DRL algorithm based on *XingTian* across two machines by assigning exactly 16 explorers in each machine in the configuration file, and deploy the RLLib version with 32 explorers in two machines by setting the `placement_strategy` as `SPREAD`. Launchpad currently can only be deployed in a single machine. We further deploy the *XingTian*-based dummy DRL algorithm with the learner in one machine and 16 explorers in the other machine. The dashed line (118.04 MB/s) in Fig. 5(a) illustrates the NIC bandwidth between the machines measured by `iperf`. With 64MB messages, *XingTian* with 32 explorers achieves 221.73 MB/s data transmission throughput; *XingTian* with 16 remote explorers achieves 110.84 MB/s data transmission throughput; and RLLib with 32 explorers achieves 72.88 MB/s data transmission throughput. *XingTian* with 16 remote explorers achieves data transmission throughput near the NIC bandwidth. With 32 explorers spreading in two machines, the data transmission throughput of *XingTian* is 3.04 times that of RLLib. Besides, it is noticeable that when message size is larger than 64KB, the end-to-end latency of *XingTian* with 32 explorers is almost the same as that of *XingTian* with 16 remote explorers. As a result, the learner in the dummy DRL algorithm with 32 explorers observes twice as much data transmission throughput as the learner with 16 remote explorers observes. This indicates that data transmission across processes inside a machine in *XingTian* is almost shadowed by the data transmission across machines. Data transmission cannot start in RLLib until the dummy learner asks for data through RPC,

which sometimes gets across machines and introduces high latency. Consequently, the pulling communication model of RLLib delays data transmission between machines, while *XingTian* starts data transmission as soon as the data are ready and is able to fully utilize the NIC bandwidth.

We can tell from the results in this subsection that *XingTian*'s asynchronous and aggressive communication model dramatically improves the data transmission efficiency in the communication mode of DRL algorithms.

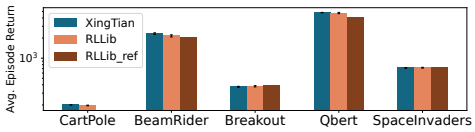
## 5.2 Performance of DRL Algorithms

We then evaluate the performance of three different kinds of DRL algorithms, i.e., IMPALA, DQN, and PPO. Acme is a single-thread DRL framework and relies on Launchpad to deploy distributed DRL algorithms. However, integral multi-process DRL algorithm implementations are not yet available in the open source repositories of Acme or Launchpad. For example, one of the collaborators from the Acme community say that they do not plan to add more detailed documentation for implementing IMPALA in Acme [3]. Besides, there are known and unsolved issues [4] that get in the way of implementing IMPALA utilizing multiple processes. So we conduct experiments with RLLib-based DRL algorithms and *XingTian*-based DRL algorithms in this subsection. Implementations of RLLib-based DRL algorithms are from the public RLLib repository.

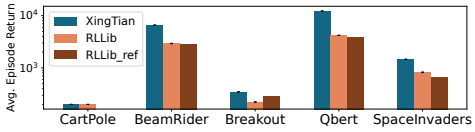
We deploy *XingTian*-based and RLLib-based DRL algorithms in the same experimental platform and with the same hyperparameters such as DNN structures, the learning rate, and the discount rate, etc. We deploy the DRL algorithms with five environments, including a gym environment CartPole and four Atari environments BeamRider, Breakout, Qbert, and SpaceInvaders [6]. We use the basic DQN algorithm with a single explorer, and we assign ten explorers and 32 explorers in PPO and IMPALA, respectively. In DQN, we allocate a replay buffer with a size of 1,000,000 rollout steps, and configure the learner to start training after 20,000 steps are collected. After training begins, every time the explorer inserts four new rollout steps into the replay buffer, the learner performs a training session with 32 steps sampled from the replay buffer. Explorers in PPO and IMPALA send out messages of 200 rollout steps when interacting with CartPole, and send out messages containing 500 rollout steps in Atari environments. The learner of PPO has to consume rollout steps from all explorers in each iteration, so the batchsize is 2,000 and 5,000 when interacting with CartPole and Atari environments, respectively. The learner of IMPALA updates the DNN parameters with rollout steps from one explorer in each iteration with a batchsize of 200 or 500.

### 5.2.1 *Convergence and Execution Time of DRL Algorithms*

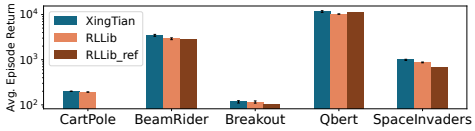
We measure the convergence of the DRL algorithms based on different frameworks with the average episode return received by the explorers after the learner trains the DNNs consuming a certain number of rollout steps (1M steps for CartPole and 10M steps for Atari environments). Fig. 6 illustrates the average episode returns in different DRL algorithms. We also put the public reference results of RLLib for Atari environments[5] in the figures that are denoted as "RLLib_ref". Results show that all three kinds of DRL algorithms based on *XingTian* attain better or similar convergent performance compared to the RLLib-based algorithms we deploy and the public RLLib reference results. *XingTian*-based IMPALA gets 1.80% and 6.80% more average episode return than RLLib-based IMPALA and RLLib reference results, respectively. *XingTian*-based DQN gets 89.90% and 121.93% more average episode return than RLLib-based DQN and RLLib reference results, respectively. *XingTian*-based PPO gets 11.24% and 22.32% more average episode return than RLLib-based PPO and RLLib reference results, respectively.




(a) IMPALA.




(b) DQN.




(c) PPO.

**Figure 6: Average Episode Return in Different DRL Algorithms.**

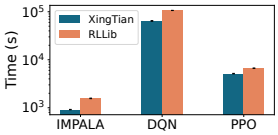


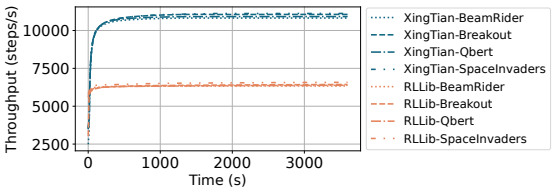

**Figure 7: Time to Complete 10M steps in Different DRL Algorithms with Atari Environments.**

Fig. 7 shows that DRL algorithms implemented in *XingTian* complete training consuming 10M rollout steps in Atari environments within less time than RLLib-based ones due to higher communication efficiency. *XingTian*-based IMPALA takes 41.54% less time than RLLib-based IMPALA to complete 10M steps. *XingTian*-based DQN takes 39.47% less time than RLLib-based DQN. *XingTian*-based PPO takes 22.92% less time than RLLib-based PPO.

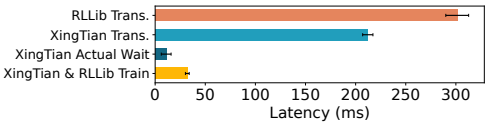
### 5.2.2 *Throughput of DRL Algorithms*

We further evaluate the throughput of the DRL algorithms in Atari environments, which is typically defined as rollout steps consumed by the learner per second. We illustrate the throughput

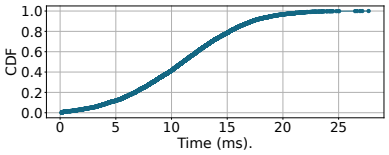
[3] https://github.com/deepmind/acme/issues/50
[4] https://github.com/deepmind/acme/issues/133
[5] https://github.com/ray-project/rl-experiments




(a) Throughput.




(b) Rollout Transmission Latency and Training Time.



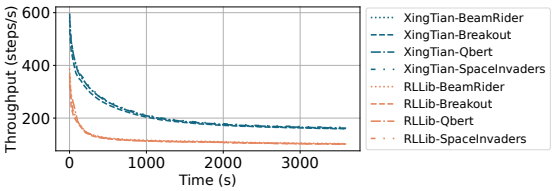

(c) CDF of Time to Wait for Rollouts before Training.

**Figure 8: Throughput and Transmission Time Analysis of IMPALA.**

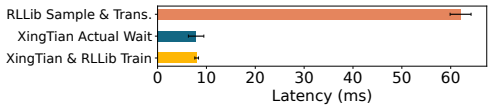
of each DRL algorithm within one hour of execution in the figures. We present the throughput of DRL algorithms based on different frameworks in different colors, and different line styles distinguish different environments.

*IMPALA*. Fig. 8(a) shows that *XingTian*-based IMPALA achieves 70.71% higher throughput than RLLib-based IMPALA on average. During execution, the explorers send messages containing rollouts of size 13,855.20 KB to the learner. Fig. 8(b) shows that in RLLib-based IMPALA, every time the learner wants to spend about 32 ms to update the DNN parameters, it has to wait for about 301 ms to ask the explorer to transmit rollouts. In *XingTian*, it takes about 212 ms to transmit a message with the same size as the rollouts. However, the results show that the learner in *XingTian* waits only about 11 ms for the rollouts before starting training on average. We further illustrate the cumulative distribution function (CDF) of the time to wait for rollouts before training in *XingTian*-based IMPALA in Fig. 8(c), which shows that the learner spends no more than 20 ms under 96.61% of the cases to wait for rollouts and waits for less than 5 ms under 11.85% of the cases. In XingTian-based IMPALA, the rollout transmission overlaps with **multi-round** training computation. Rollout transmission from an explorer targeting the learner starts aggressively and takes place simultaneously while the learner updates DNN parameters consuming rollouts from other explorers.

*DQN*. As illustrated in Fig. 9(a), the throughput of DQN is large at the beginning and then decreases as training starts, and *XingTian*-based DQN achieves 58.44% higher throughput than RLLib-based DQN on average. During execution, the learner samples 32 rollout




(a) Throughput.




(b) Rollout Sampling & Transmission Latency and Training Time.

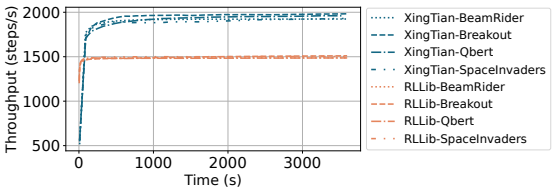
**Figure 9: Throughput and Sampling & Transmission Time Analysis of DQN.**

steps from the replay buffer for training each time. The size of a message containing 32 rollout steps is 1,912.96 KB, and training consuming these rollout steps takes about 8 ms. As shown in Fig. 9(b), sampling and transmitting 32 rollout steps from the replay buffer actor in another process in RLLib takes about 62 ms, and in *XingTian*, the learner only need to wait for the sampling latency of about 8 ms from the local replay buffer.

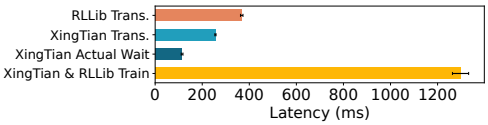
*PPO*. Fig. 10(a) shows that *XingTian*-based PPO achieves 30.91% higher throughput than RLLib-based PPO on average. Although the explorers and the learner of PPO execute in a synchronous manner, *XingTian* grabs the communication-computation overlapping opportunity and transmits rollouts from each explorer to the the learner aggressively. During execution, the learner has to collect rollouts of size 138,585.32 KB in total from all the ten explorers before starting training. Fig. 10(b) shows that in RLLib-based PPO, every time the learner wants to spend about 1,298 ms for DNN training, it has to wait for about 368 ms to ask all the explorers to transmit rollouts. In *XingTian*, it takes about 256 ms to transmit the messages with the same size as the rollouts from ten explorers. Results show that it actually takes the learner in *XingTian* about 114 ms to wait for the rollouts before the training starts.

## 5.3 Scalability

There are two key considerations on scalability: (1) how many resources a framework can manage properly and (2) how many resources an DRL algorithm can use [62] and we mainly focus on the former. We evaluate the communication efficiency of *XingTian* compared to RLLib when deploying *XingTian*-based and RLLib-based IMPALA under different scale deployments with a different number of explorers. Experiments with no more than 64 explorers are conducted in a single machine; the experiments with 128 explorers and 256 explorers are deployed in two machines and four machines, respectively. We use the BeamRider environment for all the experiments, and explorers in all experiments send out messages containing 500 rollout steps. The learners in the experiments with 128 and 256 explorers are configured with a batchsize of 1,000




(a) Throughput.




(b) Rollout Transmission Latency and Training Time.

**Figure 10: Throughput and Transmission Time Analysis of PPO.**

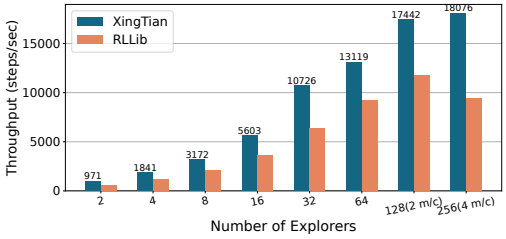



**Figure 11: Scalability Results.**

rollout steps, and the batchsize of learners with fewer explorers is 500 rollout steps.

As illustrated in Fig. 11, the throughput of *XingTian*-based IMPALA is always higher than RLLib-based IMPALA. Within up to 32 explorers, *XingTian* achieves an approximately linear acceleration ratio. The throughput growth slows down when the number of explorers continues to increase because the learner is gradually saturated and we leave it for future work to develop a DRL algorithm where 256 explorers do not saturate the learner. It is noticeable that when deployed in four machines with 256 explorers, RLLib-based IMPALA's throughput drops while *XingTian*-based IMPALA's throughput still gets improved and is 91.12% higher than the RLLib-based one. This is because communication across machines gets more and the pulling communication model in RLLib compromises performance, whereas messages in *XingTian* are pushed asynchronously and aggressively once they are produced.

Due to the decentralized organization and the asynchronous aggressive communication model, *XingTian* maintains high communication efficiency when the communication becomes more complicated in larger scale deployments. This way, *XingTian* enables researchers to accelerate DRL algorithms by optimizing the computational logic to exploit more resources without dealing with attendant complex orchestration issues.

## 6 RELATED WORK

Single-threaded one-off reference implementations of DRL algorithms such as OpenAI Baselines [11], Stable Baselines [44], Keras RL [43], and Dopamine[9] provide a convenient starting point for DRL researchers. Tensorforce [28] and Garage [15] exploit multiprocessing and multithreading. ReAgent [16] focuses on offline training of industrial DRL tasks where experiments do not run in simulators. Stooke et al. [53, 54] discuss parallelization and acceleration methods for existing DRL algorithms and share parallelized implementations of some algorithms.

Other studies for general-purpose DRL frameworks wrap up common building blocks of DRL algorithms and expose interfaces for researchers. *XingTian* is closely related to such frameworks. Fiber [62] extends the `multiprocessing` library from python to deploy DRL algorithms in multiple machines. *XingTian* provides high-level interfaces that are more friendly and efficient for researchers. Acme [20], Intel Coach [7], and Surreal [14] are similar in that they insert separate data buffers between the explorers and the learner. Acme relies on Reverb [8] to implement the data buffer and Launchpad [61] for distributed deployment, and all components communicate with each other by RPC. Intel Coach uses data stores such as NFS, S3 to transfer policy, and uses distributed memory libraries such as Redis, Kinesis to transfer rollouts. Surreal uses ZeroMQ between components. RLLib [30] is a DRL framework based on Ray [39], which builds a task graph to organize computational components. The later version RLLib Flow [31] provides stream APIs to develop DRL algorithms. RLgraph [46] extends the computational graph in TensorFlow for DRL algorithms. Tianshou [58] provides a DRL framework based on PyTorch. Both RLgraph and Tianshou rely on Ray to manage the explorers. These DRL frameworks use centralized control logic to organize the computation, pay little attention to optimizing communication, and fail to utilize the communication-computation overlapping opportunity.

## 7 CONCLUSION

We present *XingTian*, a novel DRL framework that co-designs the management of communication and computation in DRL algorithms and takes advantage of the communication-computation overlapping opportunity. *XingTian* organizes the computation in DRL algorithms in a decentralized way and provides an asynchronous communication channel. We implement *XingTian* and evaluate it. Experimental results show that *XingTian* has better data transmission efficiency and that *XingTian*-based DRL algorithms achieve up to 70.71% higher throughput than those implemented in the state-of-the-art DRL framework RLLib with better or similar convergent performance. Moreover, *XingTian* maintains high communication efficiency under different scale deployments.

## ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their thoughtful suggestions. Jun Qian and Zhen Xiao are the corresponding authors.

# REFERENCES


[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (Savannah, GA, USA, November 2-4) *(OSDI 2016)*. USENIX Association, USA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Józefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. 2020. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research* 39, 1 (2020), 3–20.

[3] Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. 2017. Reinforcement Learning through Asynchronous Advantage Actor-Critic on a GPU. In *Proceedings of the 5th International Conference on Learning Representations* (Toulon, France, April 24-26, 2017) *(ICLR 2017)*. OpenReview.net, USA. https://openreview.net/forum?id=r1VGvBcxl

[4] Richard Bellman. 1957. A Markovian decision process. *Journal of mathematics and mechanics* 6, 5 (1957), 679–684.

[5] Dimitri P Bertsekas. 2011. Dynamic programming and optimal control 3rd edition, volume II. *Belmont, MA: Athena Scientific* (2011).

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

[7] Itai Caspi, Gal Leibovich, Gal Novik, and Shadi Endrawis. 2017. Reinforcement Learning Coach. Retrieved May 13, 2022 from https://doi.org/10.5281/zenodo.1134899

[8] Albin Cassirer, Gabriel Barth-Maron, Eugene Brevdo, Sabela Ramos, Toby Boyd, Thibault Sottiaux, and Manuel Kroiss. 2021. Reverb: A Framework For Experience Replay. *arXiv preprint arXiv:2102.04736* (2021).

[9] Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G Bellemare. 2018. Dopamine: A research framework for deep reinforcement learning. *arXiv preprint arXiv:1812.06110* (2018).

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

[11] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. 2017. OpenAI Baselines. Retrieved May 15, 2022 from https://github.com/openai/baselines

[12] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. 2020. SEED RL: Scalable and Efficient Deep-RL with Accelerated Central Inference. In *Proceedings of the 8th International Conference on Learning Representations* (Addis Ababa, Ethiopia, April 26-30, 2020) *(ICLR 2020,)*. OpenReview.net, USA. https://openreview.net/forum?id=rkgvXlrKwH

[13] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. 2018. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. In *Proceedings of the 35th International Conference on Machine Learning* (Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018) *(ICML 2018, Vol. 80)*. PMLR, USA, 1406–1415. http://proceedings.mlr.press/v80/espeholt18a.html

[14] Linxi Fan, Yuke Zhu, Jiren Zhu, Zihua Liu, Orien Zeng, Anchit Gupta, Joan Creus-Costa, Silvio Savarese, and Li Fei-Fei. 2018. SURREAL: Open-Source Reinforcement Learning Framework and Robot Manipulation Benchmark. In *Proceedings of the 2nd Annual Conference on Robot Learning* (Zürich, Switzerland, 29-31 October 2018) *(CoRL 2018, Vol. 87)*. PMLR, USA, 767–782. http://proceedings.mlr.press/v87/fan18a.html

[15] The garage contributors. 2019. Garage: A toolkit for reproducible reinforcement learning research. Retrieved May 13, 2022 from https://github.com/rlworkgroup/garage

[16] Jason Gauci, Edoardo Conti, Yitao Liang, Kittipat Virochsiri, Yuchen He, Zachary Kaden, Vivek Narayanan, Xiaohui Ye, Zhengxing Chen, and Scott Fujimoto. 2018. Horizon: Facebook's open source applied reinforcement learning platform. *arXiv preprint arXiv:1811.00260* (2018).

[17] Danijar Hafner, James Davidson, and Vincent Vanhoucke. 2017. Tensorflow agents: Efficient batched reinforcement learning in tensorflow. *arXiv preprint arXiv:1709.02878* (2017).

[18] Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, SM Eslami, et al. 2017. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286* (2017).

[19] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. 2018. Rainbow: Combining Improvements in Deep Reinforcement Learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence* (New Orleans, Louisiana, USA, February 2-7, 2018) *(AAAI-18)*. AAAI Press, USA, 3215–3222. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17204

[20] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Alex Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Caglar Gulcehre, Tom Le Paine, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. 2020. Acme: A Research Framework for Distributed Reinforcement Learning. *arXiv preprint arXiv:2006.00979* (2020). https://arxiv.org/abs/2006.00979

[21] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. 2018. Distributed Prioritized Experience Replay. In *Proceedings of the 6th International Conference on Learning Representations* (Vancouver, BC, Canada, April 30 - May 3, 2018) *(ICLR 2018)*. OpenReview.net, USA. https://openreview.net/forum?id=H1Dy---0Z

[22] Huawei. 2020. MindSpore. Retrieved May 13, 2022 from https://www.mindspore.cn/

[23] 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. 2019. Human-level performance in 3D multiplayer games with population-based reinforcement learning. *Science* 364, 6443 (2019), 859–865.

[24] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. 2021. Highly accurate protein structure prediction with AlphaFold. *Nature* 596, 7873 (01 Aug 2021), 583–589.

[25] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H. Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Afroz Mohiuddin, Ryan Sepassi, George Tucker, and Henryk Michalewski. 2020. Model Based Reinforcement Learning for Atari. In *Proceedings of the 8th International Conference on Learning Representations* (Addis Ababa, Ethiopia, April 26-30, 2020) *(ICLR 2020)*. OpenReview.net, USA. https://openreview.net/forum?id=S1xCPJHtDB

[26] Steven Kapturowski, Georg Ostrovski, John Quan, Rémi Munos, and Will Dabney. 2019. Recurrent Experience Replay in Distributed Reinforcement Learning. In *Proceedings of the 7th International Conference on Learning Representations* (New Orleans, LA, USA, May 6-9, 2019) *(ICLR 2019)*. OpenReview.net, USA. https://openreview.net/forum?id=r1lyTjAqYX

[27] Vijay R. Konda and John N. Tsitsiklis. 1999. Actor-Critic Algorithms. In *Proceedings of the Advances in Neural Information Processing Systems* (Denver, Colorado, USA, November 29 - December 4, 1999) *(NIPS Conference 1999)*. The MIT Press, USA, 1008–1014. http://papers.nips.cc/paper/1786-actor-critic-algorithms

[28] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. 2017. Tensorforce: a TensorFlow library for applied reinforcement learning. Retrieved May 14, 2022 from https://github.com/tensorforce/tensorforce

[29] Heinrich Küttler, Nantas Nardelli, Thibaut Lavril, Marco Selvatici, Viswanath Sivakumar, Tim Rocktäschel, and Edward Grefenstette. 2019. Torchbeast: A pytorch platform for distributed rl. *arXiv preprint arXiv:1910.03552* (2019).

[30] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael I. Jordan, and Ion Stoica. 2018. RLlib: Abstractions for Distributed Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning* (Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018) *(ICML 2018, Vol. 80)*. PMLR, USA, 3059–3068. http://proceedings.mlr.press/v80/liang18b.html

[31] Eric Liang, Zhanghao Wu, Michael Luo, Sven Mika, Joseph E. Gonzalez, and Ion Stoica. 2021. RLlib Flow: Distributed Reinforcement Learning is a Dataflow Problem. In *Proceedings of the Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021* (Virtual Event, December 6-14, 2021) *(NeurIPS 2021)*. USA, 5506–5517. https://proceedings.neurips.cc/paper/2021/hash/2bce32ed409f5ebcee2a7b417ad9beed-Abstract.html

[32] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning. In *Proceedings of the 4th International Conference on Learning Representations* (San Juan, Puerto Rico, May 2-4, 2016) *(ICLR 2016)*. USA. http://arxiv.org/abs/1509.02971

[33] Pinxin Long, Tingxiang Fan, Xinyi Liao, Wenxi Liu, Hao Zhang, and Jia Pan. 2018. Towards Optimally Decentralized Multi-Robot Collision Avoidance via Deep Reinforcement Learning. In *Proceedings of the 2018 IEEE International Conference on Robotics and Automation* (Brisbane, Australia, May 21-25, 2018) *(ICRA 2018)*. IEEE, USA, 6252–6259. https://doi.org/10.1109/ICRA.2018.8461113

[34] Amjad Yousef Majid, Serge Saaybi, Tomas van Rietbergen, Vincent Francois-Lavet, R Venkatesha Prasad, and Chris Verhoeven. 2021. Deep Reinforcement Learning Versus Evolution Strategies: A Comparative Survey. *arXiv preprint arXiv:2110.01411* (2021).

[35] Hangyu Mao, Wulong Liu, Jianye Hao, Jun Luo, Dong Li, Zhengchao Zhang, Jun Wang, and Zhen Xiao. 2020. Neighborhood Cognition Consistent Multi-Agent Reinforcement Learning. In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence* (New York, NY, USA, February 7-12, 2020) *(AAAI 2020)*. AAAI Press, USA, 7219–7226. https://ojs.aaai.org/index.php/AAAI/article/view/6212

[36] Hangyu Mao, Zhengchao Zhang, Zhen Xiao, Zhibo Gong, and Yan Ni. 2020. Learning Agent Communication under Limited Bandwidth by Message Pruning. In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence* (New York, NY, USA, February 7-12, 2020) *(AAAI 2020)*. AAAI Press, USA, 5142–5149. https://ojs.aaai.org/index.php/AAAI/article/view/5957

[37] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of the 33nd International Conference on Machine Learning* (New York City, NY, USA, June 19-24, 2016) *(ICML 2016, Vol. 48)*. JMLR.org, USA, 1928–1937. http://proceedings.mlr.press/v48/mniha16.html

[38] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[39] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA, October 8-10, 2018) *(OSDI 2018)*. USENIX Association, USA, 561–577.

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

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019* (Vancouver, BC, Canada, December 8-14, 2019) *(NeurIPS 2019)*. USA, 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[42] Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav S. Sukhatme, and Vladlen Koltun. 2020. Sample Factory: Egocentric 3D Control from Pixels at 100000 FPS with Asynchronous Reinforcement Learning. In *Proceedings of the 37th International Conference on Machine Learning* (Virtual Event, 13-18 July 2020) *(ICML 2020, Vol. 119)*. PMLR, USA, 7652–7662. http://proceedings.mlr.press/v119/petrenko20a.html

[43] Matthias Plappert. 2016. keras-rl. Retrieved May 13, 2022 from https://github.com/keras-rl/keras-rl

[44] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. 2019. Stable Baselines3. Retrieved May 13, 2022 from https://github.com/DLR-RM/stable-baselines3

[45] Gavin A Rummery and Mahesan Niranjan. 1994. *On-line Q-learning using connectionist systems*. Vol. 37. Citeseer.

[46] Michael Schaarschmidt, Sven Mika, Kai Fricke, and Eiko Yoneki. 2019. RLgraph: Modular Computation Graphs for Deep Reinforcement Learning. In *Proceedings of Machine Learning and Systems 2019* (Stanford, CA, USA, March 31 - April 2, 2019) *(Mlsys 2019)*. mlsys.org, USA, 65–80. https://proceedings.mlsys.org/book/279.pdf

[47] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized Experience Replay. In *Proceedings of the 4th International Conference on Learning Representations* (San Juan, Puerto Rico, May 2-4, 2016) *(ICLR 2016)*. USA. http://arxiv.org/abs/1511.05952

[48] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. 2020. Mastering atari, go, chess and shogi by planning with a learned model. *Nature* 588, 7839 (2020), 604–609.

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

[50] 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. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

[51] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144.

[52] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *nature* 550, 7676 (2017), 354–359.

[53] Adam Stooke and Pieter Abbeel. 2018. Accelerated methods for deep reinforcement learning. *arXiv preprint arXiv:1803.02811* (2018).

[54] Adam Stooke and Pieter Abbeel. 2019. rlpyt: A research code base for deep reinforcement learning in pytorch. *arXiv preprint arXiv:1909.01500* (2019).

[55] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[56] Richard S. Sutton, David A. McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Proceedings of the Advances in Neural Information Processing Systems* (Denver, Colorado, USA, November 29 - December 4, 1999) *(NIPS Conference 1999)*. The MIT Press, USA, 1057–1063. http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation

[57] Apache Arrow Development Team. 2021. Apache Arrow. Retrieved May 13, 2022 from https://arrow.apache.org/

[58] Jiayi Weng, Huayu Chen, Dong Yan, Kaichao You, Alexis Duburcq, Minghao Zhang, Yi Su, Hang Su, and Jun Zhu. 2022. Tianshou: A Highly Modularized Deep Reinforcement Learning Library. *Journal of Machine Learning Research* 23, 267 (2022), 1–6. http://jmlr.org/papers/v23/21-1127.html

[59] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8, 3 (1992), 229–256.

[60] Mingzhe Xing, Hangyu Mao, and Zhen Xiao. 2022. Fast and Fine-grained Autoscaler for Streaming Jobs with Reinforcement Learning. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence* (Vienna, Austria, 23-29 July 2022) *(IJCAI 2022)*. ijcai.org, USA, 564–570. https://doi.org/10.24963/ijcai.2022/80

[61] Fan Yang, Gabriel Barth-Maron, Piotr Stańczyk, Matthew Hoffman, Siqi Liu, Manuel Kroiss, Aedan Pope, and Alban Rrustemi. 2021. Launchpad: A Programming Model for Distributed Machine Learning Research. *arXiv preprint arXiv:2106.04516* (2021).

[62] Jiale Zhi, Rui Wang, Jeff Clune, and Kenneth O Stanley. 2020. Fiber: A platform for efficient development and distributed training for reinforcement learning and population-based methods. *arXiv preprint arXiv:2003.11164* (2020).