Spreeze: High-Throughput Parallel Reinforcement Learning Framework

Jing Hou, Guang Chen*, *Member, IEEE*, Ruiqi Zhang, Zhijun Li, *Fellow, IEEE*, Shangding Gu, and Changjun Jiang

Abstract—The promotion of large-scale applications of reinforcement learning (RL) requires efficient training calculations. Although the existing parallel RL frameworks cover a variety of RL algorithms and parallelization techniques, their overly heavy communication frameworks make the final throughput and training effects not yet reach the limit of the hardware on a single desktop. In this paper, we propose Spreeze, a lightweight parallel framework for RL that efficiently utilizes a single desktop hardware resource to approach the throughput limit. We asynchronously parallelize the experience sampling, network update, performance evaluation, and visualization operations, and employ multiple efficient data transmission techniques to transfer various types of data between processes. The framework can automatically adjust the parallelization hyperparameters based on the computing ability of the hardware device in order to perform efficient large-batch updates. Based on the characteristics of the "Actor-Critic" RL algorithm, our framework uses dual GPUs to independently update the network of actors and critics in order to further improve throughput. Simulation results show that our framework can achieve up to 15,000Hz experience sampling and 370,000Hz network update frame rate using only a personal desktop computer, which is an order of magnitude higher than other mainstream parallel RL frameworks, resulting in 73% reduction of training time. Our work on fully utilizing the hardware resources of a single desktop computer is fundamental to enabling efficient large-scale distributed reinforcement learning training.

Index Terms—Reinforcement learning, framework, asynchrony, shared memory, model parallelism.

1 Introduction

The recent success of reinforcement learning (RL) is inseparable from the community's efforts to make RL training increasingly faster and more efficient [1], [2]. In construct to supervised learning, RL trained in simulated environments necessitates constant time-consuming environment exploration during agent updates to gather experience. As RL is applied to solve increasingly challenging tasks, training time increases proportionally. What's more, due to the randomness of the interactive generation and collection of experience in RL, RL training is not as stable as supervised learning training. Researchers are required to conduct repeated experiments many times to validate the performance of their algorithms [3]. For complex and challenging tasks, extremely lengthy training time will retard the progress of RL techniques development, so it is becoming increasingly virtual to conduct faster RL training. In order to perform faster training, some expensive server computers with a large number of CPU and GPU cores and large memory are adopted. However, a flawed RL framework will make it difficult to make full use of computing device resources efficiently. Consequently, the implementation of RL applications requires an effective parallel framework for fast training.

Constrained by many factors, it is challenging to significantly increase the base clock frequency of CPU and GPU [4]. Therefore, the core of algorithm acceleration is to utilize multi-core CPU and GPU for parallelization [5]. Supervised learning lends itself well to GPU parallelization acceleration because the data is prepared in advance and the data order is not considered. However, RL algorithms need to constantly interact with the environment to obtain new data, so they are more complex to be parallelized compared to supervised learning algorithms.

The RL community has made a lot of efforts to achieve faster and more efficient RL training. Some parallel algorithms such as A3C [6], APE-X [7] and IMPALA [8] have been proposed. In addition, there are also some more general algorithm frameworks such as RLlib [1], Acme [2] and rlpyt [9] that can perform parallelization operations. These algorithm frameworks generally utilize the GPU parallelization function to process data at a high speed. However, some frameworks do not fully parallelize various computing operations, and there are still cases of sequential execution. The transmission of data between computing units such as CPU and GPU is also the bottleneck of computing throughput. For the multi-GPU case, there are few special optimizations made for the characteristics of reinforcement learning algorithms. In addition, many frameworks require users to manually set parallelization hyperparameters, which is not conducive to the convenient deployment of frameworks to different computing devices.

Our objective is to construct a high-throughput RL framework that fully parallelizes experience sampling, network update, and performance testing functions as depicted in Fig. 1. Our framework focuses on achieving as efficient RL training as possible on a desktop, combined with the idea of large-batch training, to maximize the use of the capabilities of hardware devices such as GPU, CPU, memory, and hard disk. Performing operations such as shared memory on a single desktop can make data transmission faster, and

• *G. Chen is the corresponding author.
E-mail address: guangchen@tongji.edu.cn*

efficient parallel design can meet most RL training tasks. Unlike some other frameworks that sacrifice the calculation efficiency in order to adapt as many algorithms as possible, Our framework focuses on efficiency and is primarily suitable for mainstream off-policy single-agent RL algorithms, resulting in a dramatic 73% reduction in average training time. Moreover, our framework also has the advantages of simplicity, light weight, easy editing, and universal applicability in a variety of RL tasks based on gym environments. The main contributions of our work are as follows:

- We have developed a multiprocess parallel RL framework that maximizes the performance of GPU, CPU, memory, hard disk and other hardware components. High throughput is achieved through multiple parallel sampling processes and the unified large-batch network update.
- Our research reveals the impact and underlying principle of various parallelization hyperparameter settings on training speed, and achieves the balance between experience sampling efficiency and network update efficiency based on the performance of hardware equipment.
- In consideration of the "Actor-Critic" RL algorithm feature, our framework designs a method to update the Actor network and Critic network concurrently with dual GPUs in order to further enhance the throughput.

We introduce the work related to high-speed RL training in the following section 2. In section 3, our RL framework and methods are introduced in detail. section 4 conducts experiments to verify the performance of our framework and reveal some principles of high-throughput parallel RL calculations. Then section 5 summarizes the whole paper.

2 Related works

The RL community has been committed to improving the speed of RL training and has proposed a variety of RL acceleration methods. Although methods such as the Zero series researches [10], [11] utilize a large number of GPUs for computation, where GPUs are mainly used for parallel Monte Carlo tree search and evaluation of the large value-network, the methods are not universally applicable to standard reinforcement learning algorithms.

2.1 Parallel Reinforcement Learning

At present, the training process of most RL research is completed in simulation, and then the trained network can be transferred to the real world for application. In simulation, experience sampling is relatively fast and cheap, unlike realworld training that pays much attention to sample efficiency. Therefore, multiple sampling programs can be established for parallel sampling to obtain a better exploration effect.

A3C [6] is a distributed and asynchronous training method widely used at early stages. In its asynchronous distributed process, it not only performs experience sampling operations, but also performs network update operations. However, the experience provided by a single sampler is limited, and it is difficult to perform effective large-batch training to make full use of the GPU, which leads to low network updates and experience sampling throughput. Ape-X [7] stores the experience collected by multiple samplers in a unified experience pool for learners to update the network efficiently in large-batch. In IMPALA [8], the actors are also only responsible for experience sampling, but directly transmit experience to the learner to update the network without using the experience buffer. In addition, DD-PPO [12] is a parallel algorithm focusing on multi-GPU optimization. The seed rl method [13] proposed by Google is based on the traditional V-trace and Q-learning algorithms, and is not suitable for the current mainstream executor-reviewer algorithms such as SAC [14] and DDPG [15].

RLlib [1] is a well-known RL framework based on Ray [16] to achieve parallelization, which implements many RL algorithms including multi-agent RL and model-based RL. Acme [2] is a simplified novel RL framework proposed by DeepMind, which focuses on parallelization for high-speed training and supports multiple RL simulation environments. In addition, rlpyt [9] is a small and medium-sized deep RL framework based on PyTorch. Although it performs parallelization operations and also claims high throughput, quantitative throughput experiments are lacking.

Overall, although the existing RL frameworks have designed a variety of parallel experience sampling and learning operations, they do not pay special attention to their data throughput, which is actually not very high. Compared with the existing work, our framework is more thoroughly parallelized. Not only is the experience sampling parallelized, but also the network update, performance testing, and visualization functions are separated into dedicated processes to make full use of computer hardware equipment.

2.2 Large-Batch Training

Large-batch training is also an effective way to improve training effect and speed. In the accelerated methods of deep reinforcement learning (DRL) [17], it has been found that training with a batch size much larger than the standard can obtain relatively good training results. Similarly, in the shallow updates study of DRL [18], the use of a large batch size of up to 4096 in the last layer of the network can significantly improve the performance. [19] study the use of large batch size for machine learning and find that it can effectively improve the generalization ability of the network, thereby improving the training effect. In addition, some adaptive batch size methods for safety policies [20] or for continuous actions [21] have also been proposed. In our framework, setting a large batch size for large-batch training can effectively improve the training performance.

2.3 Data Parallelism and Model Parallelism

Data parallelism and model parallelism are two commonly used techniques in distributed machine learning systems. Data parallelism involves distributing the training data across multiple devices and training identical copies of the model on each device [22], [23]. Model parallelism involves dividing the model into smaller sub-models and running each sub-model on a different device. Research works have

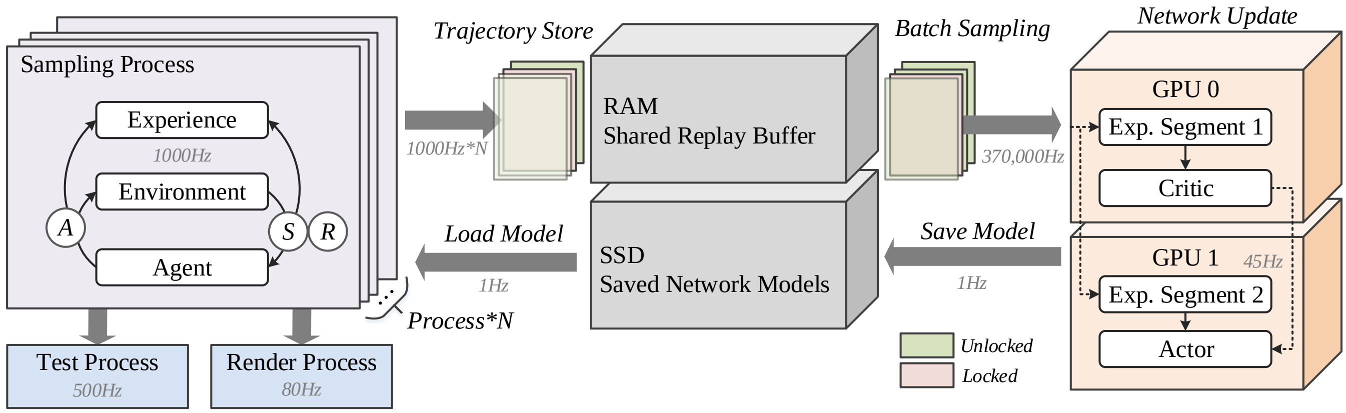


Fig. 1. **Overview of our Spreeze architecture.** The experience sampling process on the CPU and the network update process on the GPU transfer experience through shared random-access memory (RAM) and synchronize the network through solid-state drive (SSD). The shown throughput takes the pybullet Walker-2d task as an example.

explored various optimization techniques to improve the efficiency and scalability of these techniques, including gradient compression, network topology optimization, and dynamic load balancing [24]. For RL with a small amount of observed feature data, data parallelism will frequently transfer network gradients and parameters between different hardware devices, resulting in additional communication overhead and inefficient training. Consequently, we design an expandable model parallel method based on the characteristics of the actor-critic network structure of RL.

3 High-throughput Framework Design

3.1 Parallel Framework Overview

Our proposed Spreeze framework is a multi-process RL framework, which asynchronously parallelizes various time-consuming aspects of RL training, thereby maximizing the utilization of GPU, CPU, memory, hard disk and other hardware devices. As shown in Fig. 1, our framework includes multiple experience sampling processes, a network update process, a test process, and a visualization process.

3.2 Asynchronous Actor

The asynchronous actor method is a key component of our high-throughput parallel RL framework. This method enables multiple agents to learn and interact with an environment simultaneously, allowing for more efficient and effective learning. This approach improves both sample efficiency and learning speed by leveraging parallel computation resources. Specifically, the actors include experience sampling processes, test processes, and visualization processes.

3.2.1 Experience Sample

Our framework incorporates asynchronous experience sampling as a core component of training. The process involves generating rich experience data by allowing multiple agents to interact with the environment simultaneously. The parallelization of experience sampling is achieved using multi-process technology, where a large number of empirical sampling processes are generated concurrently. The maximum number of processes generated is dependent on the number of cores of a given CPU.

During training, N sampling processes are continuously running and interacting with the environment to generate experience trajectories. The forward propagation strategy network generates actions based on the collected data from the multiple sampling processes. This results in a more efficient and effective learning strategy, as the network updates are based on rich and diverse data.

Each experience sampling process can fully utilize the computing resources of one CPU core, allowing for maximum use of available resources. However, since the CPU also performs other tasks, such as test verification, visualization, or data transfer, not all experience sampling processes can run continuously. To ensure optimal performance, the running number N of experience sampling processes is dynamically adjusted based on the CPU load during training.

The adjustment strategy is implemented to ensure that there is no wastage of resources and that the available computing power is effectively allocated to each process. This results in a more efficient and faster training process, allowing for the generation of high-quality RL models. For the specific adjustment strategy, see Section 3.5. In conclusion, the asynchronous experience sample actor method is a crucial component of our high-throughput framework. It allows for the generation of rich experience data by enabling multiple agents to interact with the environment simultaneously. The process is optimized through network updates based on collected data from the multiple sampling processes.

3.2.2 Validation and Visualization

In addition to the asynchronous experience sampling process, our high-throughput parallel RL framework also incorporates the asynchronous validation and visualization sample actor method to further enhance the training process. The test process generates an episode return curve to obtain a dense and accurate reward curve, while the visualization process displays the process of interaction between the algorithm and the environment to showcase the learned strategy.

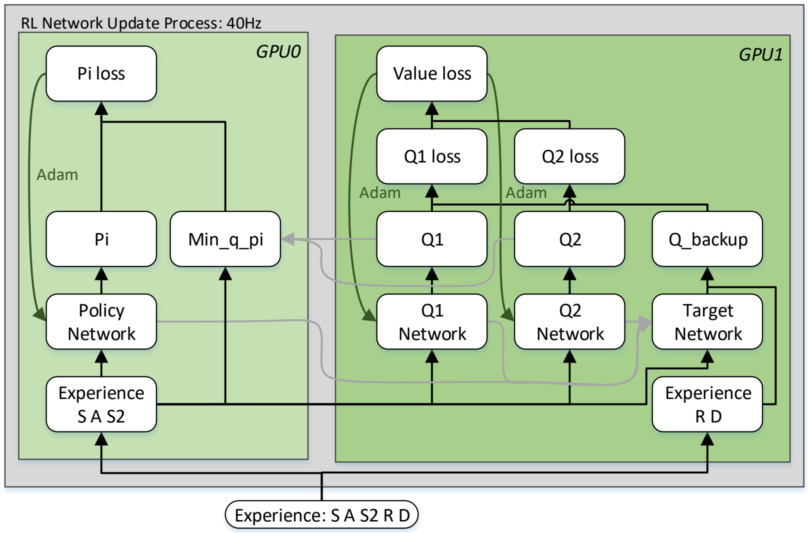


Fig. 2. **Network update architecture.** Two GPUs respectively update the actor policy network and the critic value network with as little data transmission as possible. The shown throughput takes the PyBullet Walker-2d task as an example.

These processes are variants of the experience sampling process, but they do not randomize the actions output by the policy network and do not transmit experience to the update process. As the frame rate of the visualization process is much lower than that of the test process, the two are not integrated into one process. This approach results in a more efficient and effective training process, leading to the generation of high-quality RL models.

3.3 Network Update

3.3.1 *Large-Batch Training*

Large-batch training is a method of reinforcement learning that utilizes parallelization to accelerate the algorithm. Due to hardware limitations, it is challenging to increase the frequency, so large-batch parallelization is the core of algorithm acceleration. In our framework, the experience collected from multiple experience sampling processes is transmitted to a single network update process for parallelization by GPU. To ensure robustness in RL training, a large batch size is set for parallelization. A large batch size can achieve high training efficiency and keep the training curve stable. However, the batch size is limited by the GPU's memory capacity and computing power.

To address this limitation, our framework adaptively selects the largest possible batch size based on the GPU's performance to achieve high-throughput network updates. The hyperparameter determination strategy is described in Section 3.3. By selecting the optimal batch size, we can balance the computational efficiency and stability of the training process, resulting in a faster and more stable RL training process.

3.3.2 *Actor-Critic Model Parallelism*

In addition to large-batch training, the parallelization of multiple GPUs for network updates is another method used in our framework for RL training. Since most RL algorithms use the "Actor-Critic" dual-network architecture, which consists of an actor network responsible for selecting actions and a critic network responsible for evaluating the quality of actions, an architecture is designed in which actor and critic are distributed on two GPUs for independent updates. This approach is different from the data parallel method used in supervised learning because the network model of RL tasks is usually less complex, and it is not cost-effective to distribute gradient calculations to multiple GPUs for calculation.

To specifically illustrate how the framework parallelizes the multi-GPU network computing graph model, we take the SAC [14] algorithm as an example. The specific calculation diagram is shown in Fig. 2. In this method, each GPU is responsible for a part of the calculation and minimizes the amount of data communication between GPUs. GPU0 is mainly responsible for updating the policy network, while GPU1 is mainly responsible for updating the value network. Experience data, including state s, action a, next state $s2$, reward r, and done flag d, is distributed to two GPUs as required by the network model. The reward and done experiences are allocated to GPU1 because they are only used to calculate the value network loss, while other experiences are allocated to GPU0.

Since the network architecture draws on the double-Q algorithm to prevent overestimation, there are two value networks Q1 and Q2. They are updated by GPU1 together with the target network, while GPU0 is responsible for calculating the loss of the policy network and performing gradient descent updates. The parallelization of the multi-GPU model reduces the time required for network updates and accelerates the training process.

In summary, the Actor-Critic Model Parallelism method for reinforcement learning in our framework distributes the actor and critic networks on two GPUs for independent updates, minimizing the amount of data communication between GPUs. This approach reduces the time required for network updates and accelerates the training process, making it a valuable addition to the large-batch training method for parallelization.

3.4 Variable Transmission

3.4.1 Transmission Framework

Our high-throughput parallel RL training framework relies on efficient variable transmission between processes to achieve optimal performance. This framework involves two main types of data transmission: sampled experience and neural network weights. Since the experience sampling rate is much slower than the network update rate, multiple experience sampling processes are established to provide data for a single network update process. However, this leads to relatively large data reception pressure on the network update process, which necessitates high-speed transmission for the experience data.

On the other hand, the network weight is a Tensor variable [25] in our framework that poses a challenge for transmission using traditional Queue or shared memory operations. Nevertheless, due to its infrequent transmission requirements and the need to save checkpoint points periodically, the network weight can be transmitted using solid-state drive (SSD) storage. By saving and reading the variable in each process, we can ensure that the network weight is transmitted accurately and efficiently, without causing any bottlenecks in the framework. Our variable transmission framework helps to optimize the use of computing resources and improve the training speed and accuracy of our RL models.

3.4.2 Shared Memory

To achieve efficient experience transfer and high-speed training in parallel reinforcement learning, conventional methods such as Queue or Pipe operations used for data transfer between processes can be problematic. They consume a lot of time for data dump, causing delays in the training process, and a large queue size can cause data lag, affecting the training performance. To overcome these issues, we use shared memory technique [26] to transfer experience data.

The shared memory method enables direct updating of the experience pool for network updates without consuming the time of the network update process. This approach allows experience obtained from sampling to be used for network updates as soon as possible, without waiting for the queue to be fully collected. Locking mechanisms are used to prevent data confusion when accessing the experience in shared memory.

Compared to the Queue method, the shared memory technique offers a significant improvement in experience transfer frequency, achieving 10 Hz without wasting update process time. In contrast, the Queue method only achieves a transfer frequency of 0.2 Hz, wasting about 20% of the update process time. Our experiments in section 5 demonstrate the efficacy of this approach.

3.5 Hyperparameter Adaptation

3.5.1 Hyperparameter Impact

Hyperparameters play a crucial role in high-throughput hyperparameter adaptation parallel RL training as they significantly impact the degree of parallelization. Setting hyperparameters manually can be challenging and time-consuming, and may not achieve optimal training results. Therefore, in our parallel framework, we have designed a parameter adaptation function that adjusts the hyperparameters based on hardware performance to achieve optimal parallelization results automatically.

Two key hyperparameters that affect parallelization performance are batch size and the number of sampling processes. Batch size mainly affects the GPU, as it determines the number of experience frames included in the network update. When the GPU occupancy rate is low, increasing the batch size does not affect the network update frequency but increases the number of experience frames included in the network update. On the other hand, when the GPU occupancy rate is close to saturation, increasing the batch size becomes difficult to further increase the network update frame rate, but it makes the network update frequency decrease. The number of sampling processes, on the other hand, impacts the CPU performance. Increasing the number of sampling processes can enhance the rate of experience sampling, but if too many processes occupy the CPU, it can reduce the efficiency of the network update process.

In general, the high -parameter adaptation function based on hardware performance requires our parallel framework to optimize the key super parameters, so as to obtain better parallelism and performance to achieve parallel RL training adapted to high-throughput supers tipping.

3.5.2 Adaptation Strategy

Our hyperparameter adaptation strategy is designed to automatically adjust key hyperparameters, including batch size and the number of sampling processes, based on hardware performance. The goal is to optimize the calculation occupancy and maximize both the network update and empirical sampling throughput. To achieve this, we use an adaptive rule for the batch size, which aims to make the GPU occupancy rate reach saturation while staying within the limits of available memory. This maximizes the network update rate and frame rate simultaneously. Similarly, we use an adaptive rule for the number of sampling processes to maintain a moderate CPU occupancy rate while staying within the memory allowance. This helps to increase the rate of experience sampling without affecting the efficiency of the network update process.

We enumerate a set of predetermined values for the batch size and the number of sampling processes and adaptively determine these hyperparameters to maximize both CPU and GPU usage. By doing so, we can ensure that both the GPU occupancy rate and the CPU usage rate are maintained at a relatively high ratio, as shown in Table 2, leading to efficient parallelization of the RL training process. According to the experiments, the GPU occupancy rate is maintained at approximately 85% and the GPU occupancy rate is maintained at approximately 75%. Overall, this approach helps to overcome the limitations of manual hyperparameter tuning and improves the performance of parallel RL training.

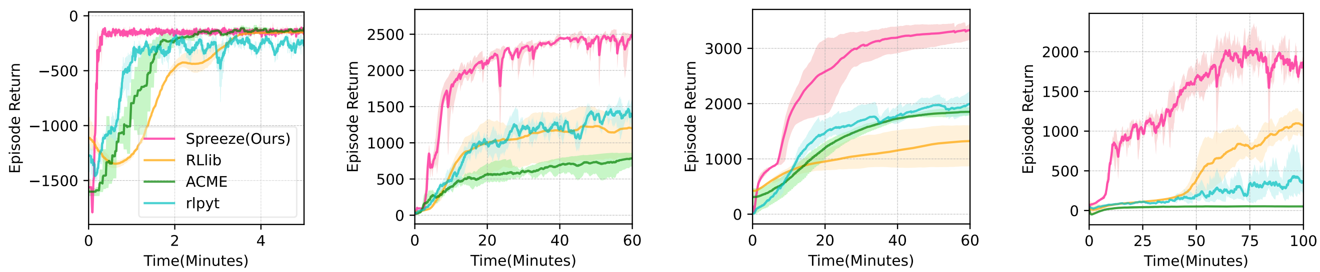
4 Performance Evaluation

In this section, numerical studies are conducted to evaluate the performance of our high-throughput RL framework. We compare the throughput, training effect, and hardware

TABLE 1
Comparison of time to solve between different frameworks

Env\Framework	Spreeze(Ours)	RLlib	ACME	rlpyt	Time Save
Pendulum	**16.3** ± **1.4**	209.4 ± 1.1	173.6 ± 22.5	158.3 ± 20.9	89.7%
HalfCheetah	**215.3** ± **49.8**	4337.2 ± 1124.5	700.2 ± 126.2	1754.2 ± 96.9	69.3%
Walker	**220.9** ± **5.5**	1884.6 ± 1798.1	3485.1 ± 677.1	748.4 ± 78.2	70.5%
Ant	**414.6** ± **104.8**	997.8 ± 630.3	614.2 ± 100.8	582.4 ± 78.8	28.8%
Humanoid	**668.2** ± **79.3**	3861.2 ± 819.2	-	8215.4 ± 1930.0	82.7%
HumanoidRunFlag	**904.1** ± **66.4**	19461.3 ± 2019.0	-	-	95.4%
average	-				72.7%

* The unit of time results in the table is "seconds". Bold numbers indicate time results for the least time-consuming frame in each scene. "Time Save" refers to the percentage of time our framework saves compared to the best of the three other frameworks.



(a) Pendulum-v0 (b) Walker2D (c) Ant (d) Humanoid

Fig. 3. **Performance comparison of different frameworks in different environments.** We have performed a detailed hyperparameter search for each framework to ensure that the best performance of each framework can be basically achieved. Each curve represents the average of five random seed experiments.

utilization of our framework to those of other prevalent RL parallel frameworks, and reveal the fundamental principles for achieving high-throughput parallel training.

4.1 Experiment Setup

Our framework is compatible with any standard OpenAI Gym [27] environment, which is a mainstream RL training environment. In our experiments, we choose the PyBullet [28] robot control simulation platform, which is widely used as a RL robot control benchmark and is based on Gym. We chose three robot models with different difficulty levels, from easy to difficult, Walker2D, Ant, and Humanoid. In addition, in order to test the training efficiency of each RL framework in a relatively simple environment, we also select the Pendulum-v0 environment based on OpenAI gym [27] for experiments. The main hardware platform used in the experiment includes a 12-core AMD 5900X CPU, NVIDIA 1060 GPU, and 32G RAM, which are common personal desktop configurations. Our framework will automatically adjust parameters such as batch size and the number of processes based on hardware performance.

4.2 Results

4.2.1 Parallel Framework Performance Comparison

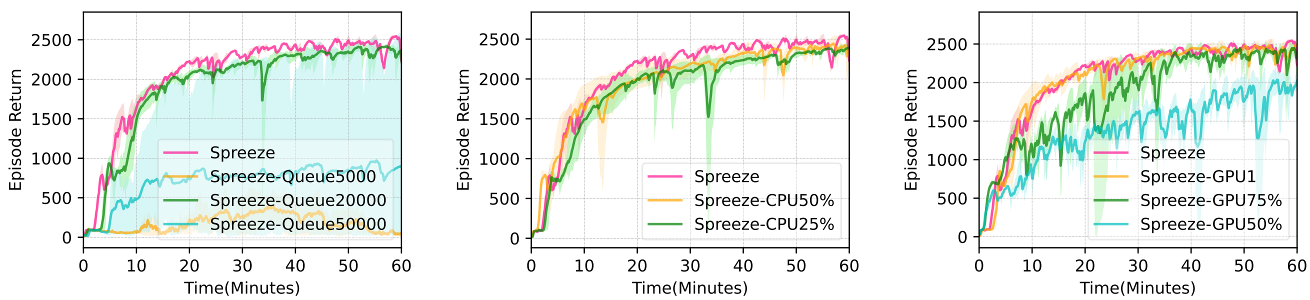
First, we conduct performance comparison experiments between different frameworks. We choose the widely used RLlib [1], Acme [2] and rlpyt [9] as the comparison frameworks. RLlib implements distributed training based on the Ray parallel AI framework [16]. The rlpyt framework uses the PyTorch library to unify the three types of algorithms: deep Q-learning, policy gradients, and Q-value policy gradients. Acme, on the other hand, focuses on enabling scalable distributed RL training at different execution scales.

Due to the different ways of parallelization for each framework, the optimal RL algorithm and optimal parameters for each framework to complete the experimental task are different. At first we expected to compare the same mainstream RL algorithm such as SAC in different frameworks. But in fact, although RLlib parallelizes the SAC algorithm, it does not implement unified experience playback like ours, so the overall training efficiency is low, far inferior to the APEX algorithm. In RLlib, the parallel algorithm APE-X implemented has relatively strong performance, and we use its APE-DDPG variant algorithm for our continuous motion control experiments. Therefore, the following experiments will perform hyperparameter searches for different frameworks to achieve the optimal performance of each framework and make a fair comparison. We choose the most suitable algorithms for PyBullet tasks in the RLlib, Acme, and rlpyt frameworks, which are PPO [29], D4PG [30], and SAC [14] respectively. Since the experience sampling is cheap and fast in the simulation, the parallelization framework mainly focuses on time-efficiency rather than sample-efficiency [2]. In addition, because not all frameworks support multi-GPU training, we use only one GPU when comparing different frameworks. As shown in Fig. 3, the horizontal axis is the training time and the vertical axis is the test episode return. Following the setting of [1], we sort out the time required for each framework to solve different tasks as shown in Tab. 1. The target returns of Pendulum, HalfCheetah, Walker, Ant, Humanoid, HumanoidRunFlag are -200, 800, 850, 850, 1800, 100, respec-

TABLE 2
Comparison of hardware usage and throughput between different frameworks

Framework\Index	CPU Usage ↑	Sampling Frame Rate (Hz) ↑	GPU Usage ↑	Network Update Frame Rate (Hz) ↑	Network Update Frequency (Hz) ↑
Spreeze(Ours)	**75%**	**15342**	**82%**	**3.7E+5**	45.2
Spreeze-BS128(Ours)	**75%**	**15564**	60%	4.2E+4	**330.3**
RLlib-APEX-BS128	64%	4132	32%	3.3E+4	257.6
RLlib-APEX-BS4096	63%	4513	30%	3.6E+4	8.8
RLlib-PPO-CPU-BS128	25%∼100%	2244	0%	2244	17.5
RLlib-PPO-CPU-BS8192	25%∼100%	2204	0%	2204	0.3
RLlib-PPO-GPU-BS128	15%∼100%	1268	42%	1268	9.9
ACME-BS256	11.7%	420	18.7%	4.9E+3	0.593
ACME-BS8192	10.1%	590	11.2%	1.1E+4	39.8
rlpyt-BS128	52%	11080	45%	1.3E+4	103.6
rlpyt-BS512	60%	14898	35%	4.5E+4	88.7

* "↑" indicates that the higher the performance index, the better. The bold values represent the best performance in each index. "BS" represents the batch size.



(a) Experience Transition Method (b) CPU Hardware Device (c) GPU Hardware Device

Fig. 4. **Ablation Experiment.** (a) Performance comparison between standard shared memory experience transfer and queue experience transfer with different queue size. (b) Ablation experiment limited to use only 50% and 25% CPU hardware resources. (c) Ablation experiment limited to use only a single or 75% or 50% GPU hardware resources. Five random seed trainings are performed under each experimental condition.

tively. The results show that our framework is significantly better than mainstream RL parallel frameworks such as RLlib, Acme and rlpyt in terms of training efficiency on tasks of various difficulties. To achieve the same control performance, our training time is reduced by about 73% compared to other frameworks.

4.2.2 Hardware Usage and Throughput Analysis

The core of improving the training efficiency of the RL framework is to make full use of computing hardware resources and increase data throughput, so we conduct experiments and analysis on the hardware usage and throughput. We take the Walker2D environment as an example to analyze the impact of different parameter settings in our framework on the above indicators, and the results are shown in Table 3. The index of hardware throughput includes CPU usage, sampling frame rate, GPU usage, network update frame rate, network update frequency, experience transfer cycle, experience transmission frame rate, and experience transmission loss. Among them, the network update frame rate refers to the multiplication of network update frequency and batch size. In order to facilitate the analysis of GPU occupancy, the experiments in this section still only use a single GPU.

Next, we will also conduct a comparative analysis of hardware usage and throughput among different frameworks. Various algorithms and parameters under the RLlib, Acme, rlpyt and Spreeze frameworks have been tested, fully demonstrating the performance of each framework. The results are shown in Table 2. It can be found that the throughput of our framework is significantly larger than that of other frameworks in both experience sampling and network update. In particular, the network update frame rate of our framework is one order of magnitude higher than that of other frameworks, which is considered the key to our framework to train RL agents faster. In other frameworks, the problem of low network update frame rate and low GPU utilization is common. After trying to increase the batch size for them, the network update frame rate can only be slightly increased, but this significantly reduces the network update frequency and worsens the training effect. Increasing the batch size in the rlpyt framework will greatly occupy memory, and 32G memory only supports increasing the batch size to 512. In addition, the CPU and GPU occupancy rates of our framework are also the highest among all the frameworks. However, our occupancy rates have not reached 100% because as shown in section 3.3 and Table 2, fully occupying the CPU or GPU will bring a series of adverse effects.

4.2.3 Ablation Experiments

Next, we conduct experiments from two aspects of parallelization technique ablation and hardware device limitations. In the experiments, we use the dual GPU independent update "Actor-Critic" network technique described in section 3.3, and our experiment performs five random seed

TABLE 3
The impact of hyperparameters in our framework on hardware usage and throughput

Framework\Index	CPU Usage ↑	Sampling Frame Rate (Hz) ↑	GPU Usage ↑	Network Update Frame Rate (Hz) ↑	Network Update Frequency (Hz) ↑	Experience Transfer Cycle (s) ↓	Experience Transmission Loss ↓
Spreeze	75%	15342	82%	3.7E+5	45.2	0	3%
Spreeze-BS32768	75%	16207	86%	**4.1E+5**	<u>12.4</u>	0	3%
Spreeze-BS128	75%	15564	<u>60%</u>	<u>4.2E+4</u>	**330.3**	0	3%
Spreeze-SP16	**90%**	**17122**	79%	<u>3.2E+5</u>	<u>39.1</u>	0	<u>10%</u>
Spreeze-SP2	<u>21%</u>	<u>3300</u>	84%	**3.9E+5**	**48.1**	0	**2%**
Spreeze-QS5000	<u>50%</u>	<u>8044</u>	81%	<u>2.2E+5</u>	<u>26.9</u>	<u>5.0</u>	<u>45%</u>
Spreeze-QS20000	<u>50%</u>	<u>8672</u>	83%	<u>3.2E+5</u>	<u>39.1</u>	<u>18.1</u>	<u>16%</u>
Spreeze-QS50000	<u>49%</u>	<u>8630</u>	82%	3.6E+5	44.5	<u>46.0</u>	<u>9%</u>

* "↑" indicates that the higher the performance index, the better, while "↓" indicates that the lower the index, the better. The boldface indicates performance improvement compared to the standard configuration, and the underline indicates performance deterioration. "BS" represents the batch size, "SP" represents the number of sample processes, and "QS" represents the queue size.

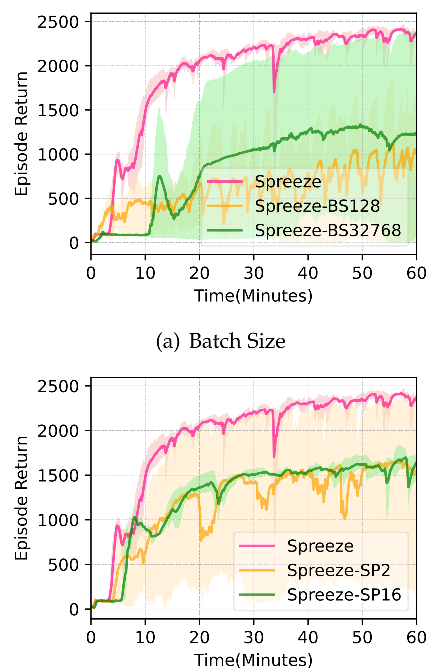
trainings for each setting in the Pybullet 2D humanoid robot control environment.

For parallelization technology, we test the final performance difference between using shared memory and using queues of different sizes to transfer experience. It can be seen from Fig. 4(a) that the training effect of the queue transmission method is sensitive to the queue size. Even if the queue size is fine-tuned, the queue method is still inferior to the shared memory method because the use of shared memory can greatly reduce the experience transmission cycle without occupying the network update process time. The quantitative results of the throughput using queues to transfer experience are shown in rows 6∼8 of Table 3.

On the other hand, we conduct hardware device limitation experiments by restricting the framework's use of CPU and GPU devices. Fig. 4(b) shows the experiments of the Spreeze framework using different amounts of CPU resources, namely 100% CPU, 50% CPU and 25% CPU. As the use of CPU hardware resources is restricted, the training effect has slightly decreased. This shows that it is meaningful for our framework to use a large number of experience sampling processes to sample experiences in parallel and at high speed. Similarly, Fig. 4(c) shows the results of the experiment that restricts the use of the GPU. By default, the framework uses two GPUs for model parallel network update. In Fig. 5(c), GPU1 indicates that only one graphics card is used for network training without model parallelization, which will cause the network update throughput to drop by about 10%, and the final training curve will be slightly affected. Furthermore, restricting the framework to only use 75% or 50% of a single GPU will further significantly deteriorate the training effect. Compared to restricting the use of the CPU, restricting the GPU will have a greater impact on training performance. This also shows that the focus of the future performance improvement direction of the parallel framework is to improve the efficiency of network updates under the conditions of large batch training.

Fig. 5 shows the training curve for adjusting the hyperparameters of the batch size and the number of sampling processes. The performance of the adjusted hyperparameters is not as good as the performance with the 8192 batch size and 16 sample processes automatically determined by the framework.

The first row of experimental data in the Table 3 represents the result of the framework automatically determining the hyperparameters. The batch size is automatically determined to be about 8192, and the number of sampling processes is determined to be about 16. The 2∼5 rows in the table are the results obtained after manually changing one parameter. As shown in Fig. 5, the performance after manually changing the parameters is not as good as that of the default parameters automatically determined by the framework.

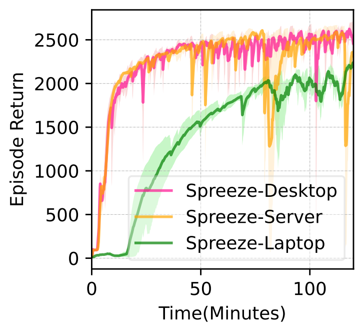


(a) Batch Size

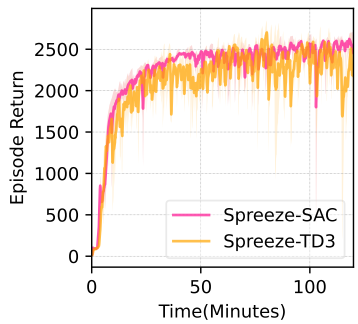
(b) Number of Sample Processes

Fig. 5. **The effect of hyperparameters on the final training performance of our framework.** Each curve represents the average of five random seed experiments.

In addition, it can be seen from Table 3 that the batch size parameter mainly affects network updates and GPU usage. Compared with the default setting of 8192, when the batch size is set too large to 32768, although the network update frame rate can be slightly increased, the network update frequency is significantly reduced, thereby deteriorating the



(a) Device robustness.



(b) Algorithm robustness.

Fig. 6. **Robustness experiment.** Five random seed trainings are performed under each experimental condition.

training effect. When the batch size is as small as 128, the network update frame rate will be reduced by an order of magnitude, and GPU utilization will also be significantly reduced. The number of sampling processes mainly affects experience collection and CPU usage. When it is too high, the sampling frame rate and CPU usage are increased, but the experience transmission loss is aggravated, which in turn takes up the network update time. When it is too low, the training effect is poor because the experience sampling frame rate is low and the collected experience is not rich enough.

And if the program uses queue transfer experience instead of memory sharing experience, the size of the queue will also be a key parameter that affects parallelization performance. A large queue size can reduce the time occupied by the experience transmission in the network update process and improve the experience transmission efficiency. But this will also make the experience transmission cycle longer, and the network update process will not get the data sampled with the latest strategy.

4.2.4 Robustness experiment

As a general RL framework, our Spreeze framework can be used on a variety of hardware devices and algorithms. We have conducted experiments to ensure that our framework can maintain the best possible training effects on a variety of hardware devices and algorithms. In addition to the desktop computer used in the previous experiments, a computing server with 40-core Intel 5128R CPU and Nvidia 2080Ti GPU, and a laptop with 4-core Intel 4710MQ CPU and Nvidia 950M GPU are used for the robustness experiment. The experiment is trained in the PyBullet walker2D environment for 120 minutes to evaluate the reward curve. As described in section 3.3, our framework automatically adjusts the parallelization hyperparameters for different hardware devices. On the server, the batch size and the number of sampling processes are set to 16384 and 16 respectively, while on the laptop, these two parameters are set to 2048 and 4. The result of the training curve is shown in Fig.6(a). The server's GPU is not much better than that of the desktop, so the server robustly maintains a training effect similar to that of the desktop. For the laptop, due to its poor GPU computing power, its training curve is significantly worse than that of the desktop computer and the server. In general, as much parallelization as possible has been achieved on each computing device. The training effect that is proportional to the computing power of the hardware device verifies that our framework has good robustness and gives full play to the capabilities of each device.

In addition to the SAC algorithm, our framework can also be easily extended to off-policy RL algorithms such as TD3 [31]. The experimental results of the robustness of different algorithms are shown in Fig.6(b), and each algorithm can be well parallelized. As a result, under the strong parallelization, the performance gap between the algorithms appears to be quite small.

5 Conclusion

In summary, this paper proposes a parallel RL framework that fully exploits hardware devices, and achieves high throughput via efficient data transmission, parameter adaptation, and network update, and reduces training time by an average of 73%. Our framework is compatible with the widely used OpenAI-gym system environment and significantly accelerates RL training on the most prevalent single desktop platform. Making full use of hardware devices on a single desktop is the basis for realizing a high-throughput distributed parallel training framework. Our framework is dedicated to advancing RL into an era of more extensive, practical, user-friendly, and efficient large-scale applications.

Acknowledgments

This work is funded by the Shanghai Municipal Science and Technology Major Project (No.2018SHZDZX01) of ZJ Lab and Shanghai Center for Brain Science and Brain-Inspired Technology, Shanghai Rising Star Program (No. 21QC1400900) and Tongji-Westwell Autonomous Vehicle Joint Lab Project.

References

[1] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica, "Rllib: Abstractions for distributed reinforcement learning," in *International Conference on Machine Learning*, pp. 3053–3062, PMLR, 2018.

[2] M. Hoffman, B. Shahriari, J. Aslanides, G. Barth-Maron, F. Behbahani, T. Norman, A. Abdolmaleki, A. Cassirer, F. Yang, K. Baumli, *et al.*, "Acme: A research framework for distributed reinforcement learning," *arXiv preprint arXiv:2006.00979*, 2020.

[3] R. Agarwal, M. Schwarzer, P. S. Castro, A. Courville, and M. G. Bellemare, "Deep reinforcement learning at the edge of the statistical precipice," *arXiv preprint arXiv:2108.13264*, 2021.

[4] T. Komoda, S. Hayashi, T. Nakada, S. Miwa, and H. Nakamura, “Power capping of cpu-gpu heterogeneous systems through coordinating dvfs and task mapping,” in *2013 IEEE 31st International Conference on computer design (ICCD)*, pp. 349–356, IEEE, 2013.
[5] K. M. Abughalieh and S. G. Alawneh, “A survey of parallel implementations for model predictive control,” *IEEE Access*, vol. 7, pp. 34348–34360, 2019.
[6] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” in *International conference on machine learning*, pp. 1928–1937, PMLR, 2016.
[7] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. Van Hasselt, and D. Silver, “Distributed prioritized experience replay,” *arXiv preprint arXiv:1803.00933*, 2018.
[8] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, *et al.*, “Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures,” in *International Conference on Machine Learning*, pp. 1407–1416, PMLR, 2018.
[9] A. Stooke and P. Abbeel, “rlpyt: A research code base for deep reinforcement learning in pytorch,” *arXiv preprint arXiv:1909.01500*, 2019.
[10] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, “Mastering the game of go without human knowledge,” *nature*, vol. 550, no. 7676, pp. 354–359, 2017.
[11] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, *et al.*, “Mastering atari, go, chess and shogi by planning with a learned model,” *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.
[12] E. Wijmans, A. Kadian, A. Morcos, S. Lee, I. Essa, D. Parikh, M. Savva, and D. Batra, “Dd-ppo: Learning near-perfect pointgoal navigators from 2.5 billion frames,” *arXiv preprint arXiv:1911.00357*, 2019.
[13] L. Espeholt, R. Marinier, P. Stanczyk, K. Wang, and M. Michalski, “Seed rl: Scalable and efficient deep-rl with accelerated central inference,” *arXiv preprint arXiv:1910.06591*, 2019.
[14] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor,” in *International conference on machine learning*, pp. 1861–1870, PMLR, 2018.
[15] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” *arXiv preprint arXiv:1509.02971*, 2015.
[16] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, *et al.*, “Ray: A distributed framework for emerging {AI} applications,” in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 561–577, 2018.
[17] A. Stooke and P. Abbeel, “Accelerated methods for deep reinforcement learning,” *arXiv preprint arXiv:1803.02811*, 2018.
[18] N. Levine, T. Zahavy, D. J. Mankowitz, A. Tamar, and S. Mannor, “Shallow updates for deep reinforcement learning,” *arXiv preprint arXiv:1705.07461*, 2017.
[19] E. Hoffer, I. Hubara, and D. Soudry, “Train longer, generalize better: closing the generalization gap in large batch training of neural networks,” *arXiv preprint arXiv:1705.08741*, 2017.
[20] M. Papini, M. Pirotta, and M. Restelli, “Adaptive batch size for safe policy gradients,” in *The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.
[21] S. Han and Y. Sung, “Amber: Adaptive multi-batch experience replay for continuous action control,” *arXiv preprint arXiv:1710.04423*, 2017.
[22] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, *et al.*, “Large scale distributed deep networks,” *Advances in neural information processing systems*, vol. 25, 2012.
[23] C. Ying, S. Kumar, D. Chen, T. Wang, and Y. Cheng, “Image classification at supercomputer scale,” *arXiv preprint arXiv:1811.06992*, 2018.
[24] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, *et al.*, “Gpipe: Efficient training of giant neural networks using pipeline parallelism,” *Advances in neural information processing systems*, vol. 32, 2019.
[25] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, “Tensorflow: A system for large-scale machine learning,” in *12th {USENIX} symposium on OSDI 16)*, 2016.
[26] S. Ahn, J. Kim, E. Lim, and S. Kang, “Soft memory box: A virtual shared memory framework for fast deep neural network training in distributed high performance computing,” *IEEE Access*, vol. 6, pp. 26493–26504, 2018.
[27] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” *arXiv:1606.01540*, 2016.
[28] E. Coumans and Y. Bai, “Pybullet, a python module for physics simulation for games, robotics and machine learning,” *URL http://pybullet. org*, 2016.
[29] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv preprint arXiv:1707.06347*, 2017.
[30] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. Tb, A. Muldal, N. Heess, and T. Lillicrap, “Distributed distributional deterministic policy gradients,” *arXiv preprint arXiv:1804.08617*, 2018.
[31] S. Fujimoto, H. Hoof, and D. Meger, “Addressing function approximation error in actor-critic methods,” in *ICML*, 2018.

Jing Hou received the B.E. degree in vehicle engineering from Hunan University, Changsha, China in 2019. He is currently working toward the Ph.D. degree in vehicle engineering at the Tongji University, Shanghai, China. His research interests include reinforcement learning and autonomous driving.

Guang Chen is a research professor at Tongji University. He is leading the Intelligent Perception System group at Tongji University. His research interests include computer vision, image processing and machine learning. He received his Ph.D degree in the Faculty of Informatics in Technical University of Munich, Germany.

Ruiqi Zhang is an undergraduate student of School of Automobile Studies of Tongji University. He is a long-term research of ISPC-Lab supervised by Prof. Guang Chen. In 2022, he joined the School of Computing of National University of Singapore as the remote research assistant. His research interest lies on reinforcement learning, robotics and computer vision.

Zhijun Li received the Ph.D. degree in mechatronics from Shanghai Jiao Tong University, Shanghai, China. Since 2017, he has been a Professor with University of Science and Technology, Hefei, China. His current research interests include wearable robotics, teleoperation systems, nonlinear control, and neural network optimization. Dr. Li has been the Co-Chair of the IEEE SMC TC B^2S) and the IEEE-RAS TC Neuro-Robotics Systems. He currently serves as Associate Editor of several IEEE transactions.

ShangDing Gu received the B.S. degree in transportation engineering from Hefei University of Technology, in 2017. He is currently pursuing the M.S. degree in traffic information engineering & control at Wuhan University of Technology.

Changjun Jiang received the Ph.D. degree from the Institute of Automation, Chinese Academy of Sciences, Beijing, China, in 1995. He was a PostDoctoral Researcher with the Institute of Computing Technology, Chinese Academy of Sciences, in 1997. He is currently a Professor with the Department of Computer Science and Engineering, Tongji University, Shanghai. His current research interests include concurrent theory, Petri net, formal verification of software, concurrency processing, and intelligent transportation systems. He is a Council Member of the China Automation Federation and the Artificial Intelligence Federation, the Vice-Director of the Professional Committee of Petri Net of the China Computer Federation, the Vice-Director of the Professional Committee of Management Systems of the China Automation Federation, and an Information Area Specialist of the Shanghai Municipal Government.