SPREEZE: High-Throughput Parallel Reinforcement Learning Framework

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 methods, their final throughput and training effects have not yet reached the limit of hardware devices. In this paper, we propose Spreeze, a parallel framework for RL that efficiently utilizes hardware resources to approach the throughput limit. We asynchronously parallelize the experience sampling, network update, performance evaluation, and visualization operations, and adopt multiple efficient data transmission methods to transfer different types of data between processes. The framework can automatically adjust the parallelization hyperparameters according to the computing power of the hardware device to achieve efficient large-batch updates. Based on the characteristics of the "Actor-Critic" RL algorithm, our framework use dual GPUs to independently update the network of actors and critics to further improve throughput. The simulation results show that our framework can achieve up to 15,000Hz experience sampling and 370,000Hz network update frame rate with only a personal desktop computer, which is an order of magnitude higher than other mainstream parallel RL frameworks, making the training time efficiency significantly improved.

1 Introduction

The recent success of reinforcement learning (RL) is inseparable from the community's efforts to conduct faster and more efficient RL training. As RL is applied to solve increasingly challenging tasks, training time increases accordingly. 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 repeat experiments many times to verify the performance of their algorithms [Agarwal *et al.*, 2021]. Extremely long training time will slow down the progress of RL technique development, so it is more and more important to perform 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, an inappropriate RL framework will make it difficult to make full use of computing device resources efficiently. Therefore, the implementation of RL applications requires an efficient parallel framework for fast training.

Constrained by many factors, it is difficult to significantly increase the base clock frequency of CPU and GPU [Komoda *et al.*, 2013]. Therefore, the core of algorithm acceleration is to utilize multi-core CPU and GPU for parallelization [Abughalieh and Alawneh, 2019]. In supervised learning, the data is prepared in advance and the order of the data is not considered, so it is convenient to use GPU for parallelization acceleration. 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 [Mnih *et al.*, 2016], APE-X [Horgan *et al.*, 2018] and IMPALA [Espeholt *et al.*, 2018] have been proposed. In addition, there are also some more general algorithm frameworks such as RLlib [Liang *et al.*, 2018], Acme [Hoffman *et al.*, 2020] and rlpyt [Stooke and Abbeel, 2019] that can perform parallelization operations. These algorithm frameworks generally utilize the GPU parallelization function to process data at a high speed.

Although the existing frameworks have carried out some parallel processing, they have not considered the throughput of experience sampling and network update, and have not studied the parameter setting to make full use of computing resource hardware. Our work is to build a high-throughput RL framework that fully parallelizes experience sampling, network update, performance testing, and visualization functions as shown in Fig. 1, while taking full advantage of hardware devices such as GPU, CPU, memory, and hard disk. 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 mainly suitable for mainstream off-policy single-agent RL algorithms. At the same time, our framework also has the advantages of simplicity, light weight, easy editing, and universal use in various RL tasks based on gym environments. The main contributions of our work are as follows:

- We have built a multiprocess parallel RL framework that makes full use of the performance of GPU, CPU, memory, hard disk and other hardware devices. High throughput is achieved through multiple parallel sampling processes and unified large-batch network update.
- Our research reveals the influence and principle of various parallelization hyperparameter settings on training speed, and realizes the balance of experience sampling efficiency and network update efficiency according to the performance of hardware equipment.
- In view of the "Actor-Critic" RL algorithm feature, our framework designs a way to update the Actor network and Critic network in parallel with dual GPUs to further improve 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-speed 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.

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 physical 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 [Mnih *et al.*, 2016] 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 [Horgan *et al.*, 2018] stores the experience collected by multiple samplers in a unified experience pool for learners to update the network efficiently in large-batch. In IMPALA [Espeholt *et al.*, 2018], 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 [Wijmans *et al.*, 2020] is a parallel algorithm focusing on multi-GPU optimization.

RLlib [Liang *et al.*, 2018] is a well-known RL framework based on Ray [Moritz *et al.*, 2018] to achieve parallelization, which implements many RL algorithms including multi-agent RL and model-based RL. Acme [Hoffman *et al.*, 2020] 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 [Stooke and Abbeel, 2019] is a small and medium-sized deep RL framework based on PyTorch. Although it performs parallelization operations and also claims high throughput, specific throughput comparison experiments are lacking.

However, 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) [Stooke and Abbeel, 2018], 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 [Levine *et al.*, 2017], the use of a large batch size of up to 4096 in the last layer of the network can significantly improve the performance. [Hoffer *et al.*, 2017] 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 [Papini *et al.*, 2017] or for continuous actions [Han and Sung, 2017] have also been proposed. In our framework, setting a large batch size for large-batch training can effectively improve the training performance.

3 High-throughput Framework Design

Our proposed Spreeze framework is a multi-process RL framework that makes full use of the performance 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. Multiple sampling processes can make the algorithm have rich experience data for the network update process to optimize the strategy.

In addition, we also propose the test process from the network update process to further improve the speed and test accuracy. During the training process, the test process will generate an episode return curve to get a dense and accurate reward curve. Another process will generate a display of the process of interaction between the algorithm and the environment in order to show the strategy learned by the algorithm. Since the frame rate of the visualization process is much lower than that of the test process, the two are not integrated into one process. The test and visualization processes are both 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.

3.1 Network Update

The experience collected by multiple experience sampling processes will be transmitted to one network update process

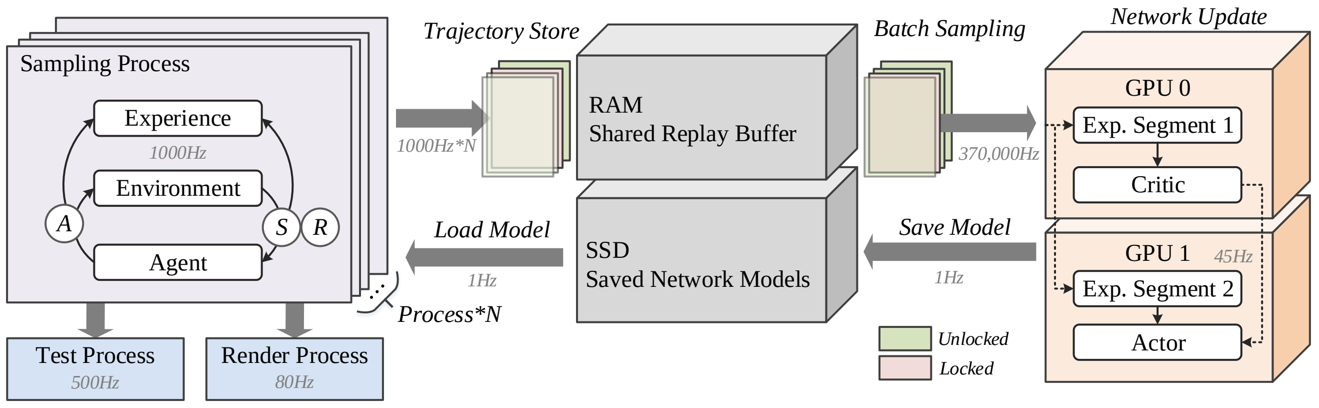


Figure 1: **Overview of our Spreeze architecture.** The experience sampling processes on the CPUs and the network update process on the GPUs transfer experience through shared random-access memory (RAM) and synchronize the network through solid-state drive (SSD). The shown throughput takes the PyBullet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

for parallelization by GPU. In RL, a robust strategy is required, so a large batch size is set for parallelization, which can not only achieve high training efficiency, but also keep the training curve stable. However, the batch size setting is restricted by the GPU memory capacity on the one hand, and the computing power of the GPU on the other hand. When the GPU computing power limit is reached, too large batch size will cause the network update frequency to decrease.

In addition to using the large-batch training for parallelization, our framework also uses multiple GPUs to perform network updates in parallel. Since most RL algorithms use the “Actor-Critic” dual-network architecture, and common personal desktops support the installation of two GPUs, an architecture in which actor and critic are distributed on two GPUs for independent update is designed. The parallel method of the multi-GPU model we design is different from the data parallel method commonly used in supervised learning. Because the network model of RL tasks is usually not as complex as in computer vision, the gradient calculation is not so time-consuming, and it is not cost-effective to distribute it to multiple GPUs for calculation [Pal *et al.*, 2019].

We take the SAC [Haarnoja *et al.*, 2018] as an example to specifically introduce how the framework parallelizes the multi-GPU network computing graph model. The specific calculation diagram is supplemented in Fig. 1 in the appendix. The core of multi-GPU calculation graph parallelization is that each GPU is responsible for part of the calculation and minimizes the amount of data communication between GPUs. So in our framework, GPU0 is mainly responsible for updating the policy network and GPU1 is mainly responsible for updating the value network. The input experience data will be distributed to two GPUs as required by the network model. The experience data is usually a five-tuple, including state s, action a, next state $s2$, reward r, and done flag d. The reward experience r and done experience d are only used to calculate the value network loss, so they are allocated to GPU1, and other experiences will be allocated to GPU0. Since the network architecture draws on the idea of 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, and GPU0 is responsible for calculating the loss of the policy network and performing gradient descent updates.

3.2 Variable Transmission

The variable transmission between processes is the key to affecting the efficiency of the framework. In the parallel RL framework, there are two main types of data that need to be transmitted: sampled experience and neural network weights. For the experience data, the amount of data is relatively large, so high-speed transmission is required. Since the speed of experience sampling is much lower than the rate of network update, multiple experience sampling processes are established to provide data for one network update process. That is to say, the transmission of experience data brings relatively large data reception pressure to the network update process. If conventional methods such as Queue or Pipe are used to transfer data between processes, the data needs to be moved in memory, and the large amount of experience data needs to take up a lot of receiving process time for data dump. Even if a large queue size can be used to transfer data in large quantities and efficiently, when the number of sampling processes is large, it will still take up about 20% of the time of the network update process for data reception. Moreover, a large queue size will cause the data used for training in the network update process to lag and affect the training effect.

In order to achieve efficient experience transfer and high-speed training, we use shared memory technique [Ahn *et al.*, 2018] to transfer experience data. The shared memory method can directly update the experience pool of the network update without occupying the time of the network update process. At the same time, this method also enables the experience obtained from sampling to be used for network update as soon as possible, without waiting for the large queue to be fully collected. When accessing the experience in the shared memory, each process needs to lock the experience being accessed as shown in Fig. 1 to avoid data confusion. The experience transfer using the Queue method is only about 0.2 Hz and 20% of the update process time is wasted,

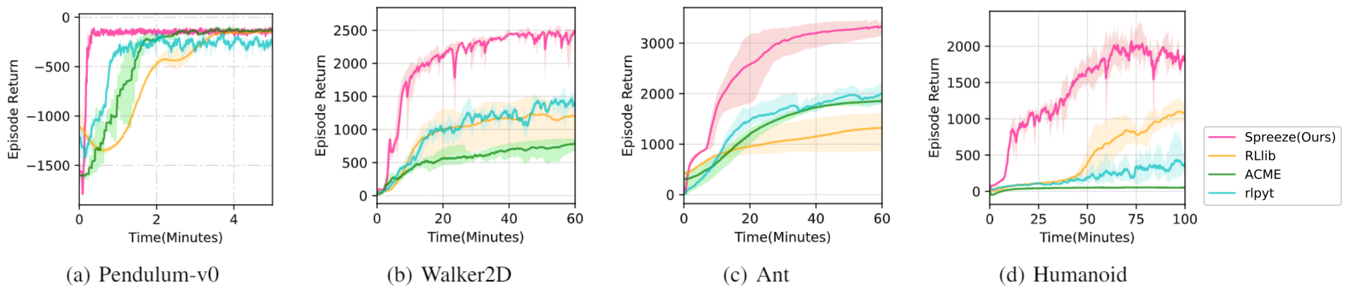


Figure 2: **Performance comparison of different frameworks in different environments.** We have performed a detailed hyperparameter search for each framework to ensure that the best performance of each framework can be basically achieved. Each curve represents the average of three random seed experiments.

while the shared experience can be increased to 10 Hz and there is no update process time wasted. The specific experiment is shown in section 4.2.

In our framework, the network weight is a Tensor variable [Abadi *et al.*, 2016], which is difficult to transmit using the Queue or shared memory operations. However, due to the low transmission frequency requirements of the network weight, and its inherent need to save checkpoint points from time to time, it can be transmitted through the solid state drive (SSD) by means of saving and reading by each process.

3.3 Hyperparameter Adaptation

For the parallel framework, the setting of parameters will greatly affect the degree of parallelization. Most of the existing parallel frameworks require users to manually adjust parallelization parameters in order to perform proper parallelization on different hardware devices. Due to the reliance on manual experience, poor manual hyperparameter settings may not achieve good parallelization training results.

In our parallel framework, we design a parallelization parameter adaptation function based on hardware performance. The key hyperparameters that affect parallelization performance mainly include batch size and the number of sampling processes. Batch size mainly affects the GPU. When the GPU occupancy rate is low, the increase of batch size does not affect the network update frequency, but increases the number of experience frames included in the network update (hereinafter referred to as the network update frame rate). When the GPU occupancy rate is close to saturation, the increase of batch size is hard to further increase the network update frame rate, but it makes the network update frequency decrease. Therefore, the principle of batch size adjustment is to make the GPU occupancy rate just reach saturation under the premise that GPU memory allows, so as to maximize the network update rate and frame rate at the same time.

Increasing the number of sampling processes can increase the rate of experience sampling, but if too many processes make the CPU all occupied, it will also reduce the efficiency of the network update process. Therefore, the adaptive rule for the number of sampling processes is to maintain a moderate CPU occupancy rate under the condition of memory allowance. In our framework, by adjusting the batch size and the number of experience sample processes, the GPU occupancy rate is maintained at approximately 85% and the GPU occupancy rate is maintained at approximately 75%.

4 Performance Evaluation

In this section, numerical studies are conducted to evaluate the performance of our high-throughput RL framework.

4.1 Experiment Setup

In our experiments, we choose the PyBullet [Coumans and Bai, 2016] robot control simulation platform, which is widely used as a RL robot control benchmark. 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 [Brockman *et al.*, 2016] for experiments. The main hardware platform used in the experiment includes a 12-core AMD 5900X CPU, NVIDIA GTX 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

Parallel Framework Performance Comparison

First, we conduct performance comparison experiments between different frameworks. We choose the widely used RLlib [Liang *et al.*, 2018], Acme [Hoffman *et al.*, 2020] and rlpyt [Stooke and Abbeel, 2019] as the comparison frameworks. In RLlib, it implements the parallel algorithm APE-X, and we use its APE-DDPG algorithm for our continuous motion control experiments. RLlib also has parallelization for the SAC algorithm, but it does not realize the unified experience replay like ours, so the overall training efficiency is low. 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. 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 [Schulman *et al.*, 2017], D4PG [Barth Maron *et al.*, 2018],

Table 1: **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.

and SAC [Haarnoja *et al.*, 2018] respectively. Since the experience sampling is cheap and fast in the simulation, the parallelization framework mainly focuses on time-efficiency rather than sample-efficiency [Hoffman *et al.*, 2020]. As shown in Fig. 2, the horizontal axis is the training time and the vertical axis is the test episode return. In addition, because not all frameworks support multi-GPU training, we use only one GPU when comparing different frameworks. The results in Fig. 2 show that our framework significantly is 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 effect, our training time is only about 30% of other frameworks.

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 1 in the appendix. 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.

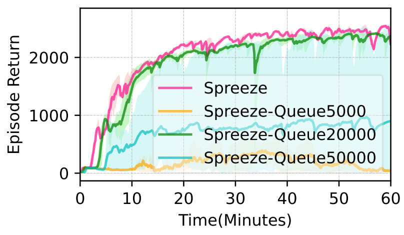
Next, we will also conduct a comparative analysis of hardware usage and throughput among different frameworks. Various algorithms and parameters under the RLlib, Acme, rlpyt and Spreeze frameworks have been tested, fully demonstrating the performance of each framework. The results are shown in Table 1. 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.

Ablation Experiments

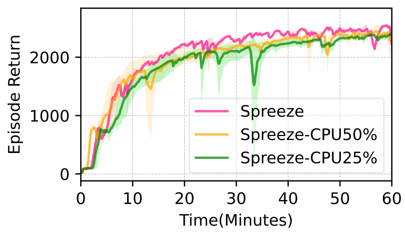
Next, we conduct ablation experiments from two aspects of parallelization technology and hardware equipment. In the experiments, we use the dual GPU independent update "Actor-Critic" network technique described in section 3.3, and our experiment performs three random seed trainings for each setting in the Walker2D 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. 3(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 1 in the appendix.

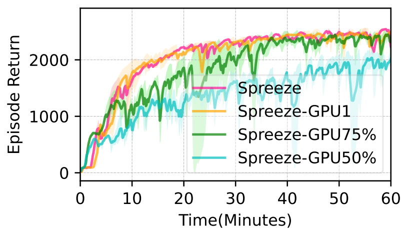
On the other hand, we conduct hardware device ablation experiments by restricting the framework's use of CPU and GPU devices. Fig. 3(b) shows the ablation experiments of the Spreeze framework using different amounts of CPU resources, namely 100% CPU, 50% CPU and 25% CPU. As the use of CPU hardware resources is restricted, the training



(a) Experience transition method.

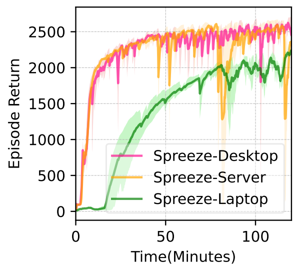


(b) CPU hardware device.

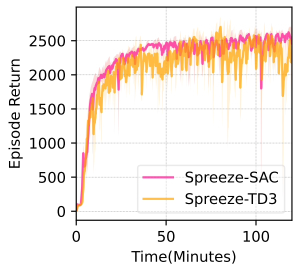


(c) GPU hardware device.

Figure 3: **Ablation experiment.** (a) Performance comparison between standard shared memory experience transfer and queue experience transfer with different queue sizes. (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. Each curve represents the average of three random seed experiments.



(a) Device robustness.



(b) Algorithm robustness.

Figure 4: **Robustness experiment.** Three random seed trainings are performed under each experimental condition.

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. 3(c) shows the results of an ablation 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.

Robustness Experiments

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 5128R CPU and 2070Super GPU, and a laptop with 4710MQ CPU and 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 20 respectively, while on the laptop, these two parameters are set to 2048 and 4. The result of the training curve is shown in Fig.4(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 [Fujimoto *et al.*, 2018]. The experimental results of the robustness of different algorithms are shown in Fig.4(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 makes full use of hardware devices, and achieves high throughput through efficient data transmission, parameter adaptation, and network update. Our framework is suitable for the most common personal desktop platforms and the widely used OpenAI-gym system environment. This framework is dedicated to pushing RL to a more extensive, convenient, people-friendly, and efficient era of large-scale applications. In the future, parallel reinforcement learning frameworks can be combined with parallel environments to obtain faster training speeds.

References

[Abadi *et al.*, 2016] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on OSDI 16)*, 2016.

[Abughalieh and Alawneh, 2019] Karam M Abughalieh and Shadi G Alawneh. A survey of parallel implementations for model predictive control. *IEEE Access*, 7:34348–34360, 2019.

[Agarwal *et al.*, 2021] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G Bellemare. Deep reinforcement learning at the edge of the statistical precipice. In *NeurIPS*, 2021.

[Ahn *et al.*, 2018] Shinyoung Ahn, Joongheon Kim, Eunji Lim, and Sungwon Kang. Soft memory box: A virtual shared memory framework for fast deep neural network training in distributed high performance computing. *IEEE Access*, 6:26493–26504, 2018.

[Barth Maron *et al.*, 2018] Gabriel Barth Maron, Matthew Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva Tb, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. Distributed distributional deterministic policy gradients. In *ICLR*, 2018.

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

[Coumans and Bai, 2016] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. 2016.

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

[Fujimoto *et al.*, 2018] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *ICML*, 2018.

[Haarnoja *et al.*, 2018] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*, 2018.

[Han and Sung, 2017] Seungyul Han and Youngchul Sung. Amber: Adaptive multi-batch experience replay for continuous action control. *arXiv:1710.04423*, 2017.

[Hoffer *et al.*, 2017] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: Closing the generalization gap in large batch training of neural networks. *arXiv:1705.08741*, 2017.

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

[Horgan *et al.*, 2018] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. In *ICLR*, 2018.

[Komoda *et al.*, 2013] Toshiya Komoda, Shingo Hayashi, Takashi Nakada, Shinobu Miwa, and Hiroshi Nakamura. Power capping of cpu-gpu heterogeneous systems through coordinating dvfs and task mapping. In *IEEE 31st ICCD*, 2013.

[Levine *et al.*, 2017] Nir Levine, Tom Zahavy, Daniel J Mankowitz, Aviv Tamar, and Shie Mannor. Shallow updates for deep reinforcement learning. In *NeurIPS*, 2017.

[Liang *et al.*, 2018] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *ICML*, 2018.

[Mnih *et al.*, 2016] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

[Moritz *et al.*, 2018] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging ai applications. In *13th USENIX Symposium on OSDI 18*, 2018.

[Pal *et al.*, 2019] Saptadeep Pal, Eiman Ebrahimi, Arslan Zulfiqar, Yaosheng Fu, Victor Zhang, Szymon Migacz, David Nellans, and Puneet Gupta. Optimizing multi-gpu parallelization strategies for deep learning training. *IEEE Micro*, 39(5):91–101, 2019.

[Papini *et al.*, 2017] Matteo Papini, Matteo Pirotta, and Marcello Restelli. Adaptive batch size for safe policy gradients. In *NeurIPS*, 2017.

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

[Stooke and Abbeel, 2018] Adam Stooke and Pieter Abbeel. Accelerated methods for deep reinforcement learning. *arXiv:1803.02811*, 2018.

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

[Wijmans *et al.*, 2020] Erik Wijmans, Abhishek Kadian, Ari Morcos, Stefan Lee, Irfan Essa, Devi Parikh, Manolis Savva, and Dhruv Batra. Dd-ppo: Learning near-perfect pointgoal navigators from 2.5 billion frames. In *ICLR*, 2020.