M.S. THESIS

Auturi: An AUTomatic and Unified Framework for Searching Parallelization Configurations in Deep Reinforcement Learning

딥강화학습의 데이터 수집부 병렬화 최적화를 위한 프레임워크 개발

February 2023

Graduate School of Engineering
Seoul National University
Department of Computer Science and Engineering
JEONG CHAE HYUN

M.S. THESIS

Auturi: An AUTomatic and Unified Framework for Searching Parallelization Configurations in Deep Reinforcement Learning

딥강화학습의 데이터 수집부 병렬화 최적화를 위한 프레임워크 개발

February 2023

Graduate School of Engineering
Seoul National University
Department of Computer Science and Engineering
JEONG CHAE HYUN

Auturi: An AUTomatic and Unified Framework for Searching Parallelization Configurations in Deep Reinforcement Learning

딥강화학습의 데이터 수집부 병렬화 최적화를 위한 프레임워크 개발

지도교수 전 병 곤

이 논문을 공학석사 학위논문으로 제출함

2022년 11월

서울대학교 대학원

컴퓨터공학부

정 채 현

정채현의 공학석사 학위 논문을 인준함

2023년 1월

위 원 장:	염 헌 영	(인)
부위원장:	전 병 곤	(인)
위　　원:	엄 현 상	(인)

Abstract

Deep reinforcement learning (DRL) has effectively been used in a wide range of challenging tasks. Despite its growing popularity, RL practitioners frequently experience excessively long training time.

One of the major bottlenecks for this inefficiency in training is that RL must collect training dataset by itself during training iterations. To solve the bottleneck, many researchers proposed various strategies, parallelizing each component of DRL.

However, the best parallelization technique varies significantly, depending on the different tasks and given hardware circumstances. Each strategy shows difference in terms of synchronization and data copy overhead due to its distinctive structure, and the effect of such overhead differs by task. Thus, choosing the best strategy is a heavy burden for users.

In this paper, we propose Auturi, a system that automatically generates the optimal configuration based on an efficient and unified code base to run hybrid parallelization. Auturi takes an online exploration approach testing each strategy one by one. Our evaluation shows that Auturi chooses an optimal configuration to maximize the speed of the experience collection loop.

Keyword: Deep Reinforcment Learning, Parallel System

Student Number: 2021-22902

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Deep reinforcement learning (DRL) has evolved successfully in a wide range of challenging tasks, such as data management controls, video games, robotics, and other domains. Despite its growing interest, RL practitioners often suffer from its excessively long training time. AlphagoZero [18], for example, reported consuming 40 days to train, while OpenAI Five [13] trained for ten months to defeat the Dota 2 world champions.

One of the major reasons for this RL training inefficiency is known to be training dataset collection during training iterations. [4, 5, 21] While supervised learning typically trains from a fixed dataset that has already been prepared, RL algorithms generate training data instance by instance by utilizing their neural networks. Even worse, on-policy algorithms, which discard all previously generated data after parameter updates, suffer more from a long training data collection phase.

Experience tuples, corresponding to a training dataset in RL, are generated by interactions between two components: environment and policy. The interaction employs a repetitive structure where each component feeds the output of another component, processes it, and then the processed output is used as the other's input again. A naive implementation allows only one component to be active at any given time, causing a significant bottleneck during the experience collection phase.

To accelerate the experience collection loop, many researchers explored various parallelization strategies by creating multiple replicas of the environments and policies. Those approaches can be classified into three categories according to which components can be parallelized. Environment parellelism [16, 21] overlaps only environment components along the time axis, while policy parellelism [11, 14] parallelizes as well as policy components. Actor parallelism [7, 10] replicates the loop itself while dividing the number of tuples to generate.

However, choosing the best strategy is challenging; the optimal parallelism strategies vary widely for different tasks and given hardware circumstances. Furthermore, adjusting a single knob within the same strategy leads to a significant throughput difference. The hardware specification also limits the number of components that can be added.

To the best of our knowledge, there is no unified code base that executes three kinds of parallelism seamlessly. Current DRL frameworks [10, 19] support only a subset of parallelization strategies and assume the configuration to be static: the configuration is chosen before training commences and does not change until the end of training.

In this paper, we introduce Auturi, AUTomatic and Unified framework searching for the optimal configuration for parallelizing the experience collection loop in deep ReInforcment learning. We first propose seven configuration knobs that describe the parallelization strategies. Auturi takes an online exploration approach with configuration described via those knobs. Modular and hierarchical design of Auturi system makes it easy to replace Auturi's submodules with other existing implementations and to adopt other existing frameworks.

Our evaluation shows that Auturi automatically chooses an optimal configuration to maximize the throughput of the experience collection phase. With widely-used deep reinforcement learning workloads on top of [16], we demonstrate that the configuration found by Auturi accelerates the collection loop by x2.4 and ultimately reduces the total training time up to 1.51 times.

Our contributions are as follows:

- Classification and systemic analysis of existing collection loop parallelization approaches.
- Proposal for primitive knobs to be used to configure existing parallelization strategies strategies.
- System that automatically generates optimal configuration.
- Unified and efficient code base to run hybrid parallelization.

Chapter 2

Background

In this section, we demonstrate the simplified scenario of Deep reinforcement learning. We also introduce the basic terminologies used throughout the paper, then showcase common workflow of DRL.

2.1 Workflow of Reinforcement Learning

The ultimate goal of reinforcement learning is to obtain optimal *policy* - the component that yields the optimal *action* when given specific *observations*. In the example of CartPole game, which aims to hold the pole atop the cart as long as possible, the observation is the description of current state, such as cart's position and how tilted the pole is. Choosing whether to push the cart left or right becomes the action of the CartPole game.

For deep reinforcement learning where policy is constructed as a neural network, its training workflow is similar with that of modern deep learning. A general workflow of RL is composed of two phases: **experience collection**(Fig 2.(a)) that generates training dataset and **policy update** (Fig 2.(b)) that actually updates the parameter inside policy network, consuming data collected in previous experience collection phase.

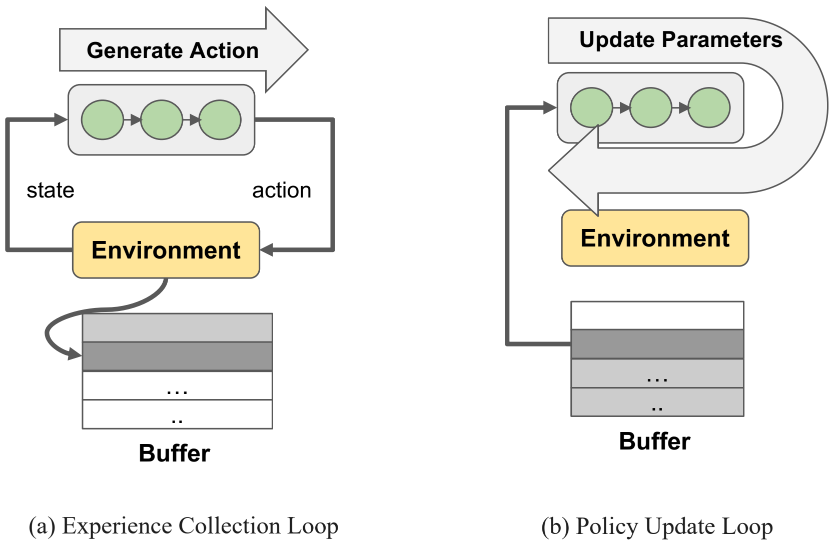

Figure 2.1: Two phases of General Deep Reinforcement Learning

The training dataset in DRL is called *experience tuple*, which is comprised of previous observation, action taken in previous step, current observation, and extra information.

Inside experience collection phase, *environment* and *policy* interacts with each other repetitively in order to generate experience tuple. Environment is usually a simulator mimicing real world, which feeds action and yields next observation. For example, environments in the CartPole game take boolean value indicating whether to push the cart left or right, then yields observation depicting current state such as coordination or velocity of a cart. On the contrary, policy takes observation from environment as an input then generates action by executing DNN, feeding into environment iteratively. The size of policy network can be varied from small MLP to large-scale LSTM [12].

The approaches how to update policy inside policy update loop differs by DRL algorithms. For example, PPO(Proximal Policy Optimization) [17] partitions stacked experience into mini-batches and incrementally update parameters of policy model, similar to mini-batch training in modern DL.

According to the policy version requirement in experience collection phase, RL algorithms can be further divided into *on-policy* and *off-policy*. On-policy algorithms (e.g., A2C, PPO [17]) require any experiences for updating the policy network to have been generated using the same policy, whereas off-policy algorithms (e.g., DDPG, SAC) does not. The absence of such requirement allows off-policy algorithms to re-use experience when forming training dataset. On the other hand, whenever policy network is modified by on-policy algorithms, all previously gathered experience tuples should be discarded and fresh experience should be generated from the scratch. Hence, the on-policy algorithm spends a greater proportion of their total training time in experience collection phase in than off-policy algorithms.

2.2 Bottleneck in DRL training

A common bottleneck unique in RL training is the interaction between the environments and the policy model. Unlike the typical setup in supervised learning using fixed dataset, RL algorithms generates data instance one by one inside collection loop.

Even worse, on-policy algorithms constraints forces two phases not be overlapped, exacerbating such bottleneck. Existing works [4, 21] reported that data collection loop takes up to 80% time of total training.

Chapter 3

Related Works

3.1 Existing Approaches

A common approach to accelerating experience collection loop is to manage multiple environment instances N rather than just a single one. 3.1(b)–(h) illustrates alternative ways for implementing $N = 4$ while 3.1(a) is a counterpart for $N = 1$.

Note that `env(x,y)` refers to the y-th simulation of x-th environment out of $N = 4$ and `policy` denotes action generation using input observations from environments. It is obvious that data dependencies in the sequence of `env(x,y)`, policy, and `env(x,y+1)` must be properly hold.

The naive approach exploiting 4 environments is demonstrated in 3.1(b). After executing environments in serial fashion, it generates action by batching N observations. Compared to 3.1(a) which exploits a single environment, it brings higher throughput since it reduces the number of policy call from N to 1.

However this serial method is impractical with large N as the time required for each iteration increases proportional to N. Instead, RL practitioners investigated various methods to parallelize the interactino between environments and policy component inside the collection loop. According to which components can be parallelize, those approaches can be classified into three categories: environment, actor and policy parallelism.

3.1.1 Environment Parallelism

Environment parallelism is a strategy to execute multiple environments in parallel. SubProcVecEnv [16], for example, executes each environment as a separate process from the master process, as shown in 3.1(c). EnvPool [21], on the other hand, parallelizes environments up to the number of available threads. EnvPool returns the batched observations to the policy as soon as the fastest K(user-defined knob) environments finish. 3.1(c) illustrates the example of EnvPool with 4 available threads and $K = 2$.

Environment parallelism is a widely used technique since it is easy to substitute existing code with this implementation. Because environment parallelism does not take into account policy components at all, its implementation is typically used as a wrapper providing an interface to handle multiple environments as a single one. However, as the policy should wait until every environment completes, this approach is ineffective for tasks with high variance in environment working time.

3.1.2 Actor Parallelism

Actor parallelism divides the number of experiences tuples by replicating collection loop itself as an abstraction known as an *actor*. This is a common strategy used by popular distributed RL frameworks such as [10, 7, 6]. In most cases, actors are uniform and includes only one policy. For an instance, when there are K actors, each actor handles N/K environments and contributes to collecting $1/K$ portion of total experience tuples. Users can configure the number of actors. 3.1(e) and 3.1(f) depict actor parallelism with four and two actors, respectively. Since actors do not need to interact with each other, this strategy loosens synchronization and reduce the volume

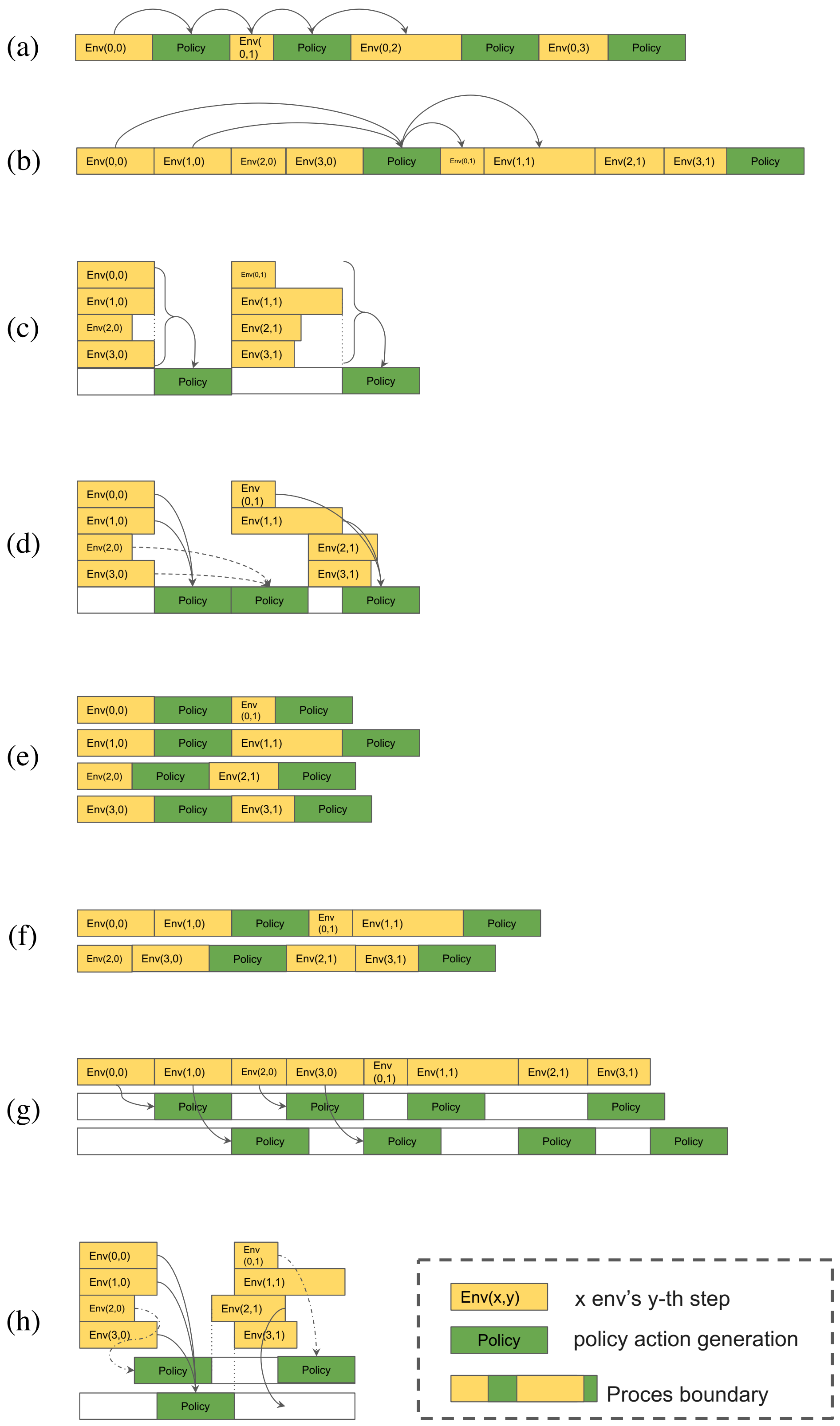
(a)
Env(0,0) Policy Env(0,1) Policy Env(0,2) Policy Env(0,3) Policy
(b)
Env(0,0) Env(1,0) Env(2,0) Env(3,0) Policy Env(0,1) Env(1,1) Env(2,1) Env(3,1) Policy
(c)
Env(0,0) Env(1,0) Env(2,0) Env(3,0) Policy
Env(0,1) Env(1,1) Env(2,1) Env(3,1) Policy
(d)
Env(0,0) Env(1,0) Env(2,0) Env(3,0) Policy Policy
Env (0,1) Env(1,1) Env(2,1) Env(3,1) Policy
(e)
Env(0,0) Policy Env (0,1) Policy
Env(1,0) Policy Env(1,1) Policy
Env(2,0) Policy Env(2,1) Policy
Env(3,0) Policy Env(3,1) Policy
(f)
Env(0,0) Env(1,0) Policy Env (0,1) Env(1,1) Policy
Env(2,0) Env(3,0) Policy Env(2,1) Env(3,1) Policy
(g)
Env(0,0) Env(1,0) Env(2,0) Env(3,0) Env (0,1) Env(1,1) Env(2,1) Env(3,1)
Policy Policy Policy Policy
Policy Policy Policy Policy
(h)
Env(0,0) Env(1,0) Env(2,0) Env(3,0) Policy Policy
Env (0,1) Env(1,1) Env(2,1) Env(3,1) Policy
Env(x,y) x env's y-th step
Policy policy action generation
Proces boundary

Figure 3.1: Parallelization strategies in previous works

of data communication compared to the environment parallelism.

3.1.3 Policy Parallelism

Policy parallelism parallelizes policy networks, whereas environment parallelism parallelizes environment components. Multiple replicas of the policy network can compute actions concurrently, as shown in 3.1(g), allowing environment and policy to overlap. This strategy improves resource utility in situations where the environment and policy use different accelerators.

HTS-RL [11], a concrete example using both environment and policy parallelism, executes each environment and policy as a separate process. The policy process polls the state of environments and computes actions whenever there are one or more available observations. 3.1(h) illustrates HTS-RL strategy with four environments and two policies. As the first step, the first policy process processes the observation from `Env(2,0)` while the second policy process consumes the results of `Env(0,0)`, `Env(1,0)`, `Env(3,0)`.

3.2 Parallelization Strategy Comparison

While many researchers proposed different parallelization strategies to accelerate environment-policy interaction in the experience collection loop, there is no single dominant strategy. Rather, the optimal structure varies depending on the task and the available hardware.

For example, actor parallelism shows the best throughput for tasks with large observation data sizes. It is optimal to take an approach that minimizes data communication between policy and environment for such a condition, which fits actor parallelism.

Furthermore, due to the improved accessibility of the environment, strategies employing policy parallelism are best suited for tasks with high environment step variance. While a policy has accessibility to the environments of the same actor in actor parallelism, entire environments are visible to policies in policy parallelism.

As we have seen so far, parallelization strategy has advantages and disadvantages in terms of data copy overhead or synchronization overhead. The processing time of each component or the size of exchanging data is distinct for each task and affects it differently for each configuration. The hardware specification also limits the number of components that can be added. Thus, selecting the optimal configuration for a given task is an imposing burden on users.

Chapter 4

Search Space

In this section, we introduce configuration knobs used as Auturi's primitives.

We assume that two requirements are specified by users: the number of total experience tuples and the range of number of environments to use. We ensure the lower limit of the number of sequential experience tuples generated by a single environment by receiving the maximum number of environments as user input for final prediction quality.

Auturi exploits actor parallelism by creating **`num_actors`** actors. Note that **`num_actors`** is the only globally defined knob, and all other following knobs are defined by actor. An actor generates **`num_experience`** tuples during a single collection loop. **`num_policy`** policies placed on **`policy_device`** (CPU or GPU) and **`num_env`** environments are managed inside an actor. **`env_parallel_degree`** indicates the degree of parallelism among environments within an actor. Thus **`num_envs/env_parallel_degree`** environments are sequentially executed in an individual process.

Finally, **`policy_batch_size`** refers the size of batched observations that policy consumes at once. This knob controls the granularity of the interaction between the environment and policy. When the total number of tuples to be collected is fixed, it is obvious that a smaller **`policy_batch_size`** value incurs more frequent data exchange.

With our newly defined knobs, tab:rel-works describes the parallelization strategies mentioned in section:rel-works.

Table 4.1: Expression of related works by Auturi configuration

	Parallelization Strategies	# actors	# environment per actor	# policies per actor	Environment parallelism degree per actor	Policy batch size
DummyVecEnv [16]	-	1	N	1	1	N
SubProcVecEnv [16]	E	1	N	1	N	N
EnvPool [21]	E	1	N	1	user knob (available threads)	user knob
RLLib [10], Acme [7]	A, E	user knob, M	N/M	1	Mostly 1	N/M
HTS-RL [11]	E, P	1	N	user knob	N	>=1
Sample Factory [14]	E, P	1	N	user knob	1	N/2

Chapter 5

System Design

In this section, we explain how we design Auturi system and user API.

5.1 System Overview

At a high-level, Auturi takes an online exploration approach with interaction between two main components, *Tuner* and *Executor* as depicted in fig:system-overview.

Tuner aims to find best configuration as soon as possible. It provides Executor with the next configuration to use while proceeding its own search algorithm or now, Tuner is implemented with a naive grid search algorithm, finding the best configuration by testing one by one.

Executor runs the collection loop with the configuration specified by Tuner until it observes stable throughput and passes it on to Tuner. Executor spawns one or multiple *actors* as separate process. Each actor writes its products(experience tuples) to *Rollout Buffer* which resides in shared memory, after running a single collection loop. Each actor has *VectorPolicy* and *VectorEnvironment* which manage multiple policy and environment worker processes, respectively. These two components communicate via shared memory, exchanging state and action data with each other.

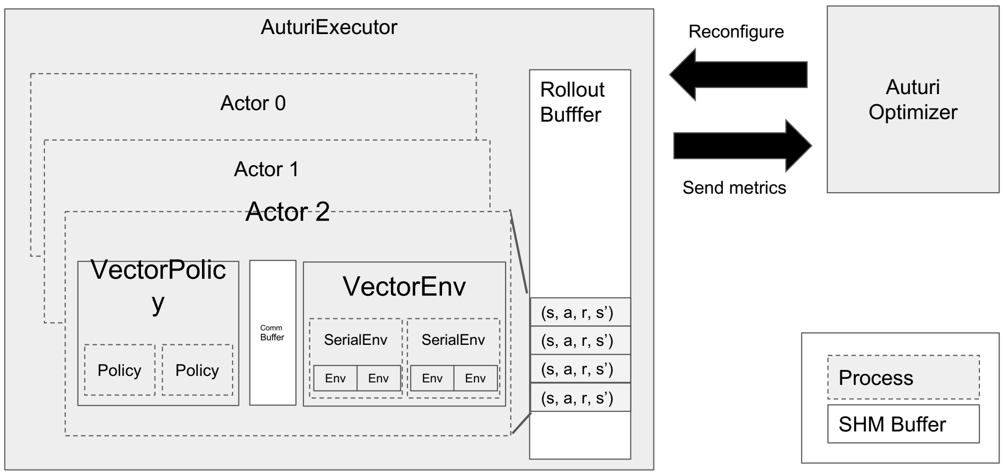

Figure 5.1: Overall architecture of Auturi System

The overall philosophy of Auturi design is as follows.

- **Pluggability** It should be easy to replace Auturi's submodules in other existing implementations.
- **Usability** Users do not have to rewrite code in other languages such as C++ or XLA. Also, Auturi is able to be ported to any other RL framework with minimal code modifications.
- **Efficiency** Auturi should provide an efficient code base that can execute the same configuration at least not slower than other implementations.

5.2 Components Hierarchy

In order to support hybrid parallelization mentioned in chapter:search-space, Auturi Executor is implemented with hierarchical and modular design to fully express tuner-given configuration knobs.

Executor is in charge of handling `num_actors` actors. VectorEnvironment and VectorPolicy inside an actor manages `num_envs` environment and `num_policies` policy components respectively. VectorEnvironment spawns `environment parallelism degree` processes which is the minimum parallel unit of environment steps. The parent-child relationship in Executor-Actor, VectorPolicy-Policy, VectorEnv-SerialEnv is controlled with primitives provided by Python multiprocessing library.

Thanks to this modular design, it is easy to plug-in external libraries Auturi's submodules. For instance, VectorEnvironment module can be entirely replaced by existing environment wrapper implementations [1, 21, 5] (purple section in system-hierarchy) Users can take advantage of faster simulators written in other languages such as C++ [21] or XLA [5]) without any additional code changes.

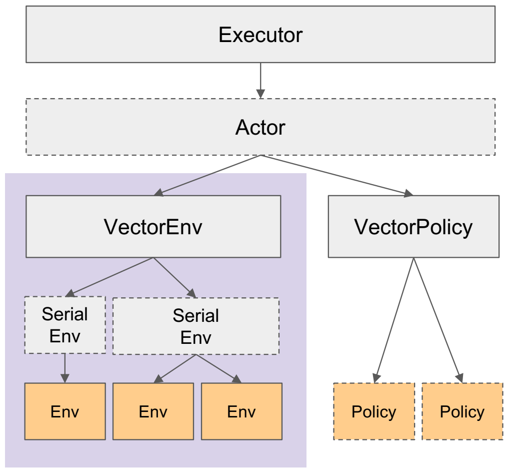

Figure 5.2: Auturi's hierarchical component design. Purple section is replaceable to external multiple environment wrapper implementations. Users should implement yellow components according to given interface.

[HTML] Class	Interface	Description
`Environment`	step(action)	Simulate with given action and yield state
	aggregate()	Return locally stored experience tuples
`Policy`	compute_actions(state)	Inference policy network with given state
	load_model(device)	Load internal policy network to given device

Table 5.1: User exposed APIs to plug Auturi on other frameworks.

5.3 User API

Users can easily adopt Auturi system atop many popular DRL frameworks [10, 7, 16] by simply rewriting environment and policy components according to the interface in table:api. The example code snippet following table:api is represented at code:api-example.

`step(action)` and `compute actions(state)` defines environment-policy interaction interface. The input and output data formats of these functions should be a single numpy array. For example, `CustomEnv` wraps the general `gym.Env` instance, returning only `obs` while storing other byproducts such as rewards, in `local_buffer` dictionary when `step(action)` is called. This is necessary since the data shape for environment-policy communication should be known in advance to be allocated in shared memory.

At the end of the collection loop, the locally stored experience tuples are aggregated to be written to Executor's Rollout Buffer by calling `aggregate()`.

```
class CustomEnv(AuturiEnv):
    def __init__(self, gym_env):
        self.env = gym_env
        self.local_buffer = dict()

    def step(self, action):
        obs, reward, done, info = self.env.step(action)
        self.local_buffer["obs"].append(obs)
        self.local_buffer["reward"].append(reward)
        return obs

  def aggregate(self):
    return self.local_buffer

class CustomPolicy(AuturiPolicy):
    def compute_actions(self, state):
        actions = self.policy(state)
        return actions.cpu().to_numpy()

    def load_model(policy, device):
        self.policy = policy.to(device)
```

Listing 5.1: API usage example label

Chapter 6

Implementation

The interaction of the environment and policy is a unique workload that exchanges small amounts of data in short time intervals. In most cases, the size of observation and action data is under 1MB and the environment steps and action computations are millisecond-level. A naive implementation of the interaction would impose high synchronization and data copy overhead. In this section, we investigate the other implementations [16, 10]. By comparing those approaches, we explain how we implemented Auturi system in detail.

RLlib [10] implemented environment-policy interaction via Ray backend [9]. In RLLib, each environment is handled as a remote Ray process and each `env.step()` call is an individual ray object managed by centralized master.

With Ray's clean RPC API to control child processes (Ray actors), RLLib is free from implementing additional control logic to send command and receive output from children. However, as [20] pointed out, the master process gets burdened with recording metadata updates and internal scheduling as the number of remote calls increases. Thus, it is inefficient to use Ray for tasks controlling many components and requiring to collect many experience tuple.

SubprocVecEnv [16] is a popularly used implementation based on environment

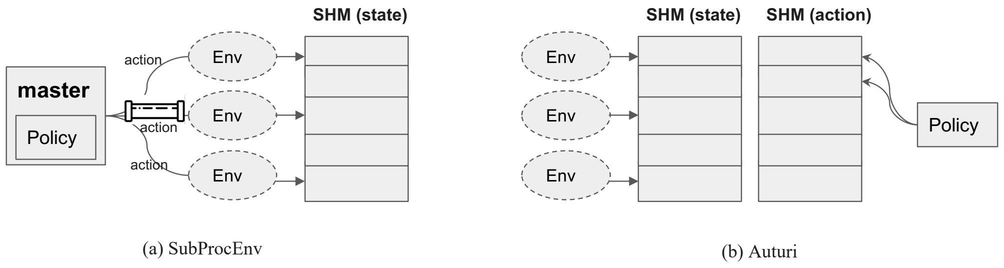

Figure 6.1: Environment-Policy interaction implementation in SubprocVecEnv and Auturi

parallelism. Since SubprocVecEnv does not rely on external RPC backend, it created its own control logic to communicate with its children processes. SubprocVecEnv sends actions via pipe and reads observations and other byproduct from shared memory, as illustrated in Fig 6.1(a). This implementation lets SubprocVecEnv be free from metadata management, leading to a performance gain compared to RLLib. When the number of environments increases, however, serially executing action sending phase becomes bottleneck.

Auturi improved the scalability by creating additional shared memory for actions atop SubprocVecEnv. While the policy writes actions to the action buffer, each environment process polls its own section in the action buffer, waiting for the next action to complete (Fig 6.1(b)).

Chapter 7

Evaluation

7.1 Evaluation Setup

7.1.1 Environment

We evaluated our system on a machine equipped with two Intel Xeon CPU E5-2695 @ 2.10 GHz processors, 256GB DRAM, and a NVIDIA Titan Xp GPU. We used PyTorch 1.10.0, Ubuntu 18.04, and CUDA 11.1 for all of our experiments

7.1.2 Workloads

We evaluated Auturi's performance for PyBullet [3], Atari [2] and Google Football [8] tasks. We followed [15] to set the training hyperparameters, policy network configuration and the post-processing of raw environment output for Pybullet and Atari. In case of Football, we matched those configurations to implementations from [11]. All E2E experiments were done based on [16] framework.

7.2 End-to-end Training

7.1 shows e2e training time using Auturi. The training iteration on [16] is normalized to 1. Since Auturi does not target for policy update phase, using Auturi does not reduce policy network update time. Auturi does not show breakthrough gain for Pong-v4 and academy_3_vs_1_with_keeper tasks, as the optimal configuration falls in environment parallelism category. For MinitaurBulletEnv-v0 task, Auturi accelerates the collection loop by x2.4 and ultimately reduces the total training time up to 1.51 times.

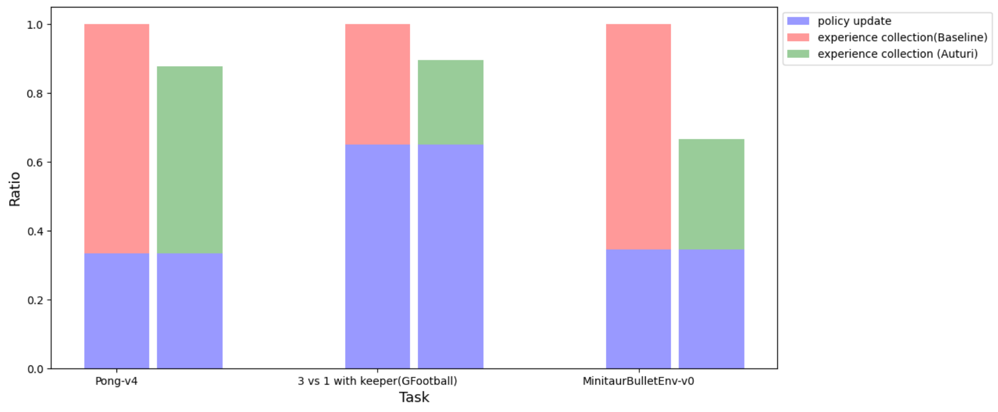

Figure 7.1: E2E training time comparison with Stable-baseline3 and Auturi.

7.3 Scaling Overhead

In this section, we compare Auturi with other implementations by fixing specific configuration. To measure performance on environment-parallel configuration, we compared our code base with SubprocVecEnv and custom code implemented with Ray backend. For actor-parallel configuration, we chose RLlib as a baseline.

7.2 shows the scaling overhead of environment-policy interaction by increasing

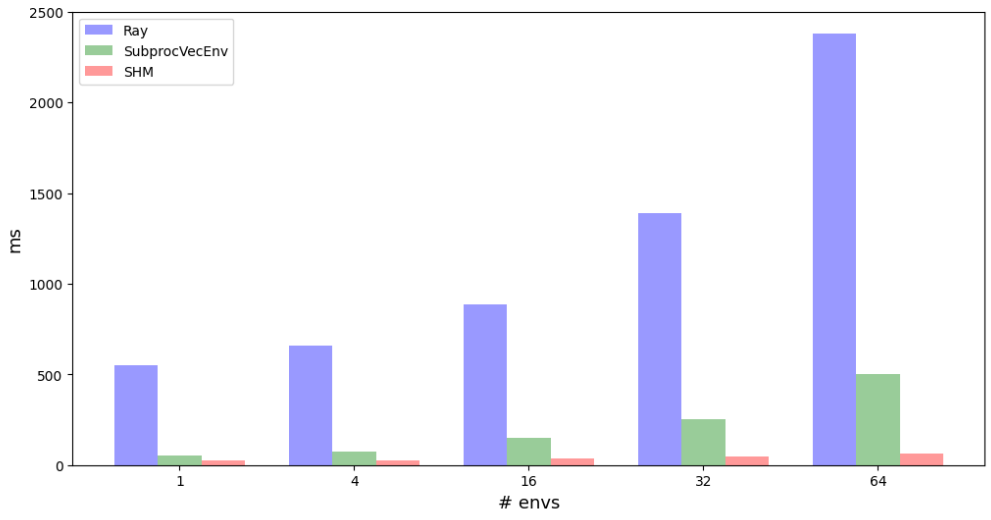

Figure 7.2: Scaling overhead of environment parallelism on HalfCheetah-v3 task.

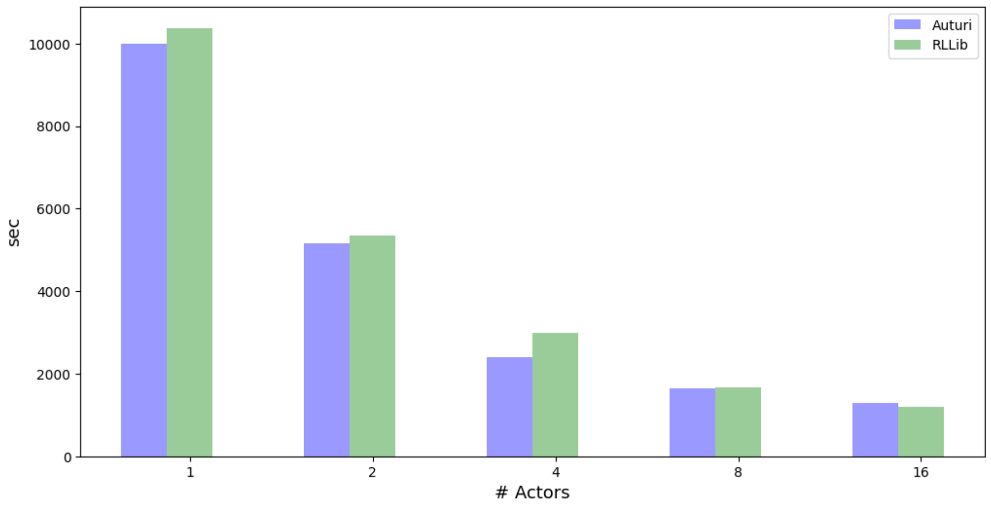

Figure 7.3: Scaling overhead of actor parallelism on Pong-v4 task.

number of environments, N. We fixed actions to constant values and configuration to environment parallelism configuration with parallelism degree equals to N. Ray backend shows large overhead even with a single environment. This results corresponds to the analysis in Section 6 that master process is burdened proportional to the number of remote calls. SubprocVecEnv and Auturi shows similar performance with small N. While serialized pipe communication of SubprocVecEnv hinders scalability from $N >= 16$, Auturi shows almost constant throughput even with $N = 64$. Since Auturi implemented entire control plane with shared memory, communication with children processes becomes $O(1)$.

7.3 demonstrates the scaling overhead of executing actor-parallel strategy on Auturi by increasing number of actors, K. Actor parallelism does not need inter-actor communication, and the processes are synchronized only two times: the beginning and the end of the collection loop. Such small communication burden leads almost linear scaling of both Auturi and RLlib shows compared to 7.2.

However, Auturi shows slight performance gain compared to RLlib. This is because Auturi actor writes rollout data to shared memory bypassing master process at the end of the collection loop.

Chapter 8

Conclusion

In this paper, we categorized the existing approaches to accelerate the collection loop and suggested primitive knobs to describe them. We configured existing strategies with combination of those knob, and pointed out that different tasks and hardware configurations require quite diverse parallelism methods. Based on those findings, we proposed Auturi. Auturi is the system that automatically generates the optimal configuration for the experience collection loop in the DRL based on an efficient and unified code base to run hybrid parallelization.

Though the evaluation in this paper is limited to only PPO algorithms, the synchronous on-policy algorithm is not the only RL algorithm that Auturi can apply. Since Auturi supports general structure of environment-policy interaction, asynchronous RL algorithms or multi-agent algorithms can be accelerated by Auturi. The evaluation of such diverse workloads is reserved for future work.

Finally, future research should be devoted to the development of search algorithms. Auturi currently exhaustively searches for the best configuration, adjusting knobs one by one. Although it always guarantees to find the global optima, adopting such an exhaustive mechanism for a task with a large search space is infeasible. By focusing on the repetitive property of how policies and environments exchange data alternately, it will be able to develop a faster search algorithm.

Bibliography

[1] Openai gym.

[2] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

[3] Benjamin Ellenberger. Pybullet gymperium. https://github.com/benelot/pybullet-gym, 2018–2019.

[4] James Gleeson, Srivatsan Krishnan, Moshe Gabel, Vijay Janapa Reddi, Eyal de Lara, and Gennady Pekhimenko. Rl-scope: Cross-stack profiling for deep reinforcement learning workloads, 2021.

[5] James Gleeson, Daniel Snider, Yvonne Yang, Moshe Gabel, Eyal de Lara, and Gennady Pekhimenko. Optimizing data collection in deep reinforcement learning, 2022.

[6] Danijar Hafner, James Davidson, and Vincent Vanhoucke. Tensorflow agents: Efficient batched reinforcement learning in tensorflow. *CoRR*, abs/1709.02878, 2017.

[7] Matthew W. Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Nikola Momchev, Danila Sinopalnikov, Piotr Stańczyk, Sabela Ramos, Anton

Raichuk, Damien Vincent, Léonard Hussenot, Robert Dadashi, Gabriel Dulac-Arnold, Manu Orsini, Alexis Jacq, Johan Ferret, Nino Vieillard, Seyed Kamyar Seyed Ghasemipour, Sertan Girgin, Olivier Pietquin, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Abe Friesen, Ruba Haroun, Alex Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A research framework for distributed reinforcement learning, 2022.

[8] Karol Kurach, Anton Raichuk, Piotr Stańczyk, Michał Zajac, Olivier Bachem, Lasse Espeholt, Carlos Riquelme, Damien Vincent, Marcin Michalski, Olivier Bousquet, and Sylvain Gelly. Google research football: A novel reinforcement learning environment, 2019.

[9] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning, 2017.

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

[11] Iou-Jen Liu, Raymond Yeh, and Alexander Schwing. High-throughput synchronous deep rl. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 17070–17080. Curran Associates, Inc., 2020.

[12] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, Jiwoo Pak, Andy Tong, Kavya Srinivasa, William Hang, Emre Tuncer,

Quoc V. Le, James Laudon, Richard Ho, Roger Carpenter, and Jeff Dean. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021.

[13] OpenAI. Openai five. https://blog.openai.com/openai-five/, 2018.

[14] Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav Sukhatme, and Vladlen Koltun. Sample factory: Egocentric 3d control from pixels at 100000 fps with asynchronous reinforcement learning. In *International Conference on Machine Learning*, pages 7652–7662. PMLR, 2020.

[15] Antonin Raffin. Rl baselines3 zoo. https://github.com/DLR-RM/rl-baselines3-zoo, 2020.

[16] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

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

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

[20] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for Fine-Grained tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 671–686. USENIX Association, April 2021.

[21] Jiayi Weng, Min Lin, Shengyi Huang, Bo Liu, Denys Makoviichuk, Viktor Makoviychuk, Zichen Liu, Yufan Song, Ting Luo, Yukun Jiang, Zhongwen Xu, and Shuicheng YAN. Envpool: A highly parallel reinforcement learning environment execution engine. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022.

Abstract

딥 강화학습(Deep Reinforcement Learning)은 로보틱스, 게임, 컴파일러 등 다양한 분야에서 도전적인 과제를 학습하는데 큰 성공을 거두어왔다. DRL의 인기가 높아졌음에도 불구하고, DRL은 종종 학습에 지나치게 긴 시간이 들었는데, 그 주요 병목 현상 중 하나는 RL이 훈련 과정 내에 자체적으로 훈련을 위한 데이터셋을 스스로 만들어야 한다는 점이다.

이러한 병목 지점을 해결하기 위해, 많은 연구자들은 DRL을 구성하는 요소인 환경(Environment), 정책 네트워크(Policy network) 등을 병렬화하는 등의 다양한 전략을 제시했다. 그러나 주어진 하드웨어 환경, 수행하고자 하는 과제에 따라 최적의 병렬화 전략이 달라진다. 병렬화 전략마다 고유한 구조로 인한 동기화 및 데이터 복사 오버헤드가 크게 차이가 나는데, 각 과제(task)마다 이에 미치는 영향이 다르기 때문이다.

본 논문에서는 자동으로 최적의 병렬화 전략을 찾아주는 시스템인 Auturi를 소개한다. 하이브리드 병렬화 전략을 효율적이고 유연하게 실행할 수 있는 코드를 기반으로, Auturi는 각 전략을 하나씩 테스트하는 온라인 탐색 방식을 취한다. 논문에서는 널리 쓰이는 DRL 벤치마크에서 실험함으로써, Auturi가 DRL 훈련시간을 효과적으로 줄일 수 있음을 보인다.

주요어: Deep Reinforcement Learning, Parallel System

학 번: 2021-22902