CLEANBA: A REPRODUCIBLE AND EFFICIENT DISTRIBUTED REINFORCEMENT LEARNING PLATFORM

Shengyi Huang[‡] **Jiayi Weng**[*] **Rujikorn Charakorn**[♯] **Min Lin**[△]
Zhongwen Xu[◇] **Santiago Ontañón**[‡§]

[‡]Drexel University Hugging Face [§]Google [♯]VISTEC [△]Sea AI Lab [◇]Tencent AI Lab
costa.huang@outlook.com

ABSTRACT

Distributed Deep Reinforcement Learning (DRL) aims to leverage more computational resources to train autonomous agents with less training time. Despite recent progress in the field, reproducibility issues have not been sufficiently explored. This paper first shows that the typical actor-learner framework can have reproducibility issues even if hyperparameters are controlled. We then introduce Cleanba, a new open-source platform for distributed DRL that proposes a highly reproducible architecture. Cleanba implements highly optimized distributed variants of PPO (Schulman et al., 2017) and IMPALA (Espeholt et al., 2018). Our Atari experiments show that these variants can obtain equivalent or higher scores than strong IMPALA baselines in `moolib` and `torchbeast` and PPO baseline in CleanRL. However, Cleanba variants present 1) shorter training time and 2) more reproducible learning curves in different hardware settings. Cleanba's source code is available at `https://github.com/vwxyzjn/cleanba`

1 INTRODUCTION

Deep Reinforcement Learning (DRL) is a technique to train autonomous agents to perform tasks. In recent years, it has demonstrated remarkable success across various domains, including video games (Mnih et al., 2015), robotics control (Schulman et al., 2017), chip design (Mirhoseini et al., 2021), and large language model tuning (Ouyang et al., 2022). Distributed DRL (Espeholt et al., 2018; 2020) has also become a fast-growing field that leverages more computing resources to train agents. Despite recent progress, reproducibility issues in distributed DRL have not been sufficiently explored. This paper introduces Cleanba, a new platform for distributed DRL that addresses reproducibility issues under different hardware settings.

Reproducibility in DRL is a challenging issue. Not only are DRL algorithms brittle to hyperparameters and neural network architectures (Henderson et al., 2018), implementation details are often crucial for successfully applying DRL but frequently omitted from publications (Engstrom et al., 2020; Andrychowicz et al., 2021; Huang et al., 2022a). Reproducibility issues in distributed DRL are under-studied and arguably even more challenging. In particular, most high-profile distributed DRL works, such as Apex-DQN (Horgan et al., 2018), IMPALA (Espeholt et al., 2018), R2D2 (Kapturowski et al., 2019), and Podracer Sebulba (Hessel et al., 2021) are not (fully) open-source. Furthermore, earlier work pointed out that more actor threads not only improve training speed but cause reproducibility issues – different hardware settings could impact the data efficiency in a non-linear fashion (Mnih et al., 2016).

In this paper, we present a more principled approach to distributed DRL, in which different hardware settings could make training speed slower or faster but do not impact data efficiency, thus making scaling results more reproducible and predictable. We first analyze the typical actor-learner architecture in IMPALA (Espeholt et al., 2018) and show that its parallelism paradigm could introduce reproducibility issues due to the concurrent scheduling of different actor threads. We then propose a more reproducible distributed architecture by better aligning the parallelized actor and learner's compu-

[*]Currently at OpenAI.

tations. Based on this architecture, we introduce our Cleanba (meaning **Clean**RL-style (Huang et al., 2022b) Podracer Sebul**ba**) distributed DRL platform, which aims to be an easy-to-understand distributed DRL infrastructure like CleanRL, but also be scalable as Podracer Sebulba. Cleanba implements a distributed variant of PPO (Schulman et al., 2017) and IMPALA (Espeholt et al., 2018) with JAX (Bradbury et al., 2018) and EnvPool (Weng et al., 2022). Next, we evaluate Cleanba's variants against strong IMPALA baselines in `moolib` (Mella et al., 2022) and torchbeast (Küttler et al., 2019) and PPO baseline in CleanRL (Huang et al., 2022b) on 57 Atari games (Bellemare et al., 2013). Here are the key results of Cleanba:

1. **Strong performance**: Cleanba's IMPALA and PPO achieve about 165% median human normalized score (HNS) in Atari with sticky actions, matching `monobeast` IMPALA's 165% median HNS and outperforming `moolib` IMPALA's 140% median HNS.

2. **Short training time**: Under the 1 GPU 10 CPU setting, Cleanba's IMPALA is **6.8x faster** than `monobeast`'s IMPALA and **1.2x faster** than `moolib`'s IMPALA. Under a max specification setting, Cleanba's IMPALA (8 GPU and 40 CPU) is **5x faster** than `monobeast`'s IMPALA (1 GPU and 80 CPU) and **2x faster** than `moolib`'s IMPALA (8 GPU and 80 CPU).

3. **Highly reproducible**: Cleanba shows predictable and reproducible learning curves across 1 and 8 GPU settings given the same set of hyperparameters, whereas `moolib`'s learning curves can be considerably different, even if hyperparameters are controlled to be the same.

To facilitate more transparency and reproducibility, we have made available our source code at `https://github.com/vwxyzjn/cleanba`.

2 BACKGROUND

Distributed DRL Systems Utilizing more computational power has been an attractive topic for researchers. Earlier DRL methods like DQN (Mnih et al., 2015) were synchronous and typically used a single simulation environment, which made them slow and inefficient in using hardware resources. A3C (Mnih et al., 2016) spawns multiple actor threads; each interacts with its own copy of the environment and asynchronously accumulates gradient. To make distributed DRL more scalable, IMPALA decouples the actors and the learners (Espeholt et al., 2018; 2020). The actors produce training data asynchronously, while the learners produce new agent parameters, which are transferred asynchronously to the actor. Actor-learner systems can achieve higher throughput and shorter training wall time than A3C. Additional distributed actor-learner systems include GA3C (Babaeizadeh et al., 2017), IMPALA (Espeholt et al., 2018), Apex-DQN (Horgan et al., 2018), R2D2 (Kapturowski et al., 2019), and Podracer Sebulba (Hessel et al., 2021).

Reproducibility Issues with Different Hardware Settings Empirical evidence suggests that increasing the number of actor threads can enhance the training speed in distributed DRL (Mnih et al. (2016, Fig. 4)). However, this augmentation is not without its complications. It also impacts data efficiency and final Atari scores (Mnih et al. (2016, Fig. 3)), and these effects could manifest in a non-linear manner. While the authors found the side effects of value-based asynchronous methods to be positive and improve data efficiency, the side effects of contemporary distributed DRL systems, such as IMPALA, Apex-DQN, and R2D2, across various hardware configurations, have not been sufficiently explored.

Open-source Distributed DRL Infrastructure While many distributed DRL algorithms are not open-source, there have been many notable distributed DRL replications in the open-source software (OSS) community. These efforts include SEED RL (Espeholt et al., 2020), `rlplyt` (Stooke & Abbeel, 2018), Decentralized Distributed PPO (Wijmans et al., 2020), Sample Factory (Petrenko et al., 2020), HTS-RL (Liu et al., 2020), `torchbeast` (Küttler et al., 2019), and `moolib` (Mella et al., 2022). Many of them have shown high throughput and good empirical performance in select domains. Nevertheless, most of them either do not have evaluations on 57 Atari games or have various hardware restrictions, leading to reproducibility concerns. `moolib` is the only OSS infras-

IMPALA Actor-Learner Architecture

```
1  batch_size = 32
2  agent = Agent()
3  data_Q = queue()
4
5  def actor():
6    while True:
7      data = rollout(agent.param, 1)
8
9
10     data_Q.put(data)
11 def learner():
12   for _ in range(1, ITER):
13     data = data_Q.get_many(batch_size)
14     agent.learn(data)
15     broadcast_to_actors(agent.param)
16 for _ in range(num_actors):
17   thread(actor).start()
18 thread(learner).start()
```

Cleanba's architecture

```
batch_size = 32
agent = Agent()
data_Q = queue(max_size=1)
param_Q = queue(max_size=1)
def actor():
  for i in range(1, ITER):
    if i != 2:
      params = param_Q.get()
    data = rollout(params, batch_size)
    data_Q.put(data)
def learner():
  for _ in range(1, ITER):
    data = data_Q.get()
    agent.learn(data)
    param_Q.put(agent.param)
param_Q.put(agent.param)
thread(actor).start()
thread(learner).start()
```

Figure 1: The pseudocode for IMPALA architecture (left) and Cleanba's architecture (right). Colors are used to highlight the code differences between the two architectures. The `rollout(params, num_envs)` function collects rollout data on `num_envs` independent environments for `num_steps` steps.

tructure that has both evaluations on 57 Atari games in the standard 200M frames setting and can scale beyond a single GPU setting[1].

3 Reproducibility Issues in IMPALA

This section shows that IMPALA (Espeholt et al., 2018) has non-determinism by nature, which arises from the concurrent scheduling of different actor threads. This non-determinism could further cause subtle reproducibility issues.

A natural question arises: *what happens when the learner produces a new policy while the actor is in the middle of producing a trajectory?* It turns out multiple policy versions could contribute to the actor's rollout data in line 7 of the IMPALA architecture Figure 1. Typically, the faster the policy updates, the more frequently the policies are transferred. However, this impacts the rollout data construction in a non-trivial way. From a reproducibility point of view, it is important to realize the frequency at which the policies are updated is a source of non-determinism.

However, non-determinism can be desirable in parallel programming because they make programs faster without making outputs significantly different. For example, some of NVIDIA's CuDNN operations are inherently non-determinisitic[2]. What is more important is to investigate if this non-determinism could cause reproducibility issues in terms of learning curves. To this end, we manufacture a specific experiment that magnifies this non-determinism in `monobeast`'s IMPALA. For the control group, we

1. decreased the number of trajectories in the batch from 32 to 8 to reduce training time, thus making the actor's policy updates more frequent;
2. used 80 actor threads and increased `monobeast`'s default unroll length from 20 to 240 to increase the chance of observing the actor's policy updates in the middle of a trajectory.

[1]While SEED RL also has evaluations on 57 Atari games and scale beyond 1 GPU, SEED RL trained the agents for 40 billion frames 40 hours per game.

[2]`https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html#reproducibility`

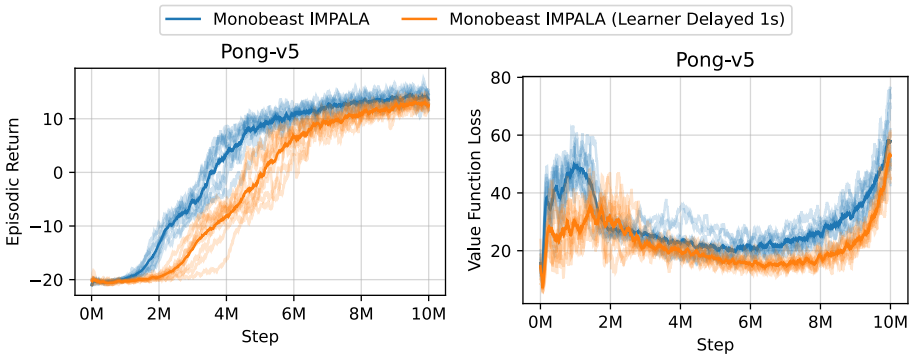

Figure 2: **IMPALA's reproducibility issue under different "speed" settings** — The y-axes show the episodic return and value function loss of two sets of `monobeast` experiments that use the *exact same hyperparameters*, but the orange set of experiments has its learner update manually delayed for 1 second to simulate slower learner updates. Note the learning curves across 10 random seeds are non-trivially different, implicating hyperparameters in IMPALA alone cannot always ensure good reproducibility.

For the experimental group, we used the above setting but *manually slowed down the policy broadcasting* by sleeping the learner for 1 second after the policy updates in order to simulate a case where the learner is significantly slower (such as when running the learner on CPU).

We found that in the control group, the actors, on average, changed their policy versions **12-13 times** in the middle of the 240-length trajectory. In the experimental group, because of the manual slowdown in broadcasting the learner's policy, the actors, on average, changed the policy **one time**. We note that the results vary on different hardware settings as well. For example, the control group changed their policy versions, on average, eight times when using 40 actor threads. We noted that in `moolib`, the actor's policy could also change mid-rollout. See Appendix G.

Figure 2 demonstrates the empirical effect of the experiments. Note that the learning and loss curves looked notably different across ten random seeds, even though the control and experimental group have the *exact same hyperparameters*. This experiment shows that IMPALA algorithmically could be susceptible to reproducibility issues across different hardware settings. While Figure 2 only shows the experimental results on one environment, the primary purpose of it is to show that this issue exists and is barely predictable. Furthermore, this type of issue can be much more subtle and difficult to diagnose at a much larger scale, so it is important that we investigate them.

4 Towards Reproducible Distributed DRL

Despite these reproducibility issues, the actor-learner architecture is useful because it allows us to parallelize the computations of the actors and learners. In this work, we address the reproducibility issues mentioned above by 1) decoupling hyperparameters and hardware settings and 2) proposing a synchronization mechanism that makes distributed DRL reproducible.

4.1 Decoupling hyperparameters and hardware settings

As mentioned in the previous section, different numbers of actor threads could make policy updates more or less frequent in the middle of a trajectory generation. This is unpredictable and need not be the case. A different number of actors also creates a different number of simulation environments and thus should be recognized as a hyperparameter setting.

To make a more clarified setting, we advocate decoupling the number of actor threads into two separate hyperparameters: 1) the number of environments, and 2) the number of CPUs. In this case, we can use a different number of CPUs to simulate a given number of environments. This decoupled interface is readily provided by EnvPool (Weng et al., 2022), which we use in our proposed architecture.

Table 1: The Synchronous and Cleanba's architecture. Under the Synchronous architecture, the actor and learner's computations are sequential and *not* parallelizable – the learner always learns from the rollout data of the latest policy $\pi_i \xrightarrow[\mathcal{D}_{\pi_i}]{} \pi_{i+1}$ (e.g., $\pi_2 \xrightarrow[\mathcal{D}_{\pi_2}]{} \pi_3$). Under Cleanba's architecture, we can parallelize the actor and learner's computation at the cost of introducing stale data – starting from iteration 3 the learner always learns from the rollout data obtained from the second latest policy $\pi_i \xrightarrow[\mathcal{D}_{\pi_{i-1}}]{} \pi_{i+1}$ (e.g., $\pi_2 \xrightarrow[\mathcal{D}_{\pi_1}]{} \pi_3$)

Iteration	1	2	3
Synchronous Arch.	$\pi_1 \to \mathcal{D}_{\pi_1}$ $\pi_1 \xrightarrow[\mathcal{D}_{\pi_1}]{} \pi_2$	$\pi_2 \to \mathcal{D}_{\pi_2}$ $\pi_2 \xrightarrow[\mathcal{D}_{\pi_2}]{} \pi_3$	$\pi_3 \to \mathcal{D}_{\pi_3}$ $\pi_3 \xrightarrow[\mathcal{D}_{\pi_3}]{} \pi_4$
Cleanba's Arch., Actor	$\pi_1 \to \mathcal{D}_{\pi_1}$	$\pi_1 \to \mathcal{D}_{\pi_1}$	$\pi_2 \to \mathcal{D}_{\pi_2}$
Cleanba's Arch., Learner		$\pi_1 \xrightarrow[\mathcal{D}_{\pi_1}]{} \pi_2$	$\pi_2 \xrightarrow[\mathcal{D}_{\pi_1}]{} \pi_3$

4.2 Deterministic Rollout Data Composition

To address the non-determinism in rollout data composition, we propose our *Cleanba's architecture*, which retains the benefit of parallelizing actor-learner computations but can produce deterministic rollout data composition. At its core, Cleanba's architecture is a simple mechanism for synchronizing the actor and learner, ensuring the learner performs gradient updates with rollout data of **second latest policy**.

Let us use the notation $\pi_i \to \mathcal{D}_{\pi_i}$ to denote that policy of version i is used to obtain rollout data $\mathcal{D}_{\pi_i}$; $\pi_i \xrightarrow[\mathcal{D}_{\pi_i}]{} \pi_{i+1}$ denotes policy of version i is trained with rollout data $\mathcal{D}_{\pi_i}$ to obtain a new policy π_{i+1}. Figure 1 is the pseudocode of the architecture and Table 1 illustrates how policies get updated. Under the Synchronous Architecture, the actor and learner's computations are sequential: it first perform rollout $\pi_1 \to \mathcal{D}_{\pi_1}$, during which the learner stays idle. Given the rollout data, the learner then performs gradient updates $\pi_1 \xrightarrow[\mathcal{D}_{\pi_1}]{} \pi_2$, during which the actor stays idle. More generally, the learner always learns from the rollout data of the latest policy $\pi_i \xrightarrow[\mathcal{D}_{\pi_i}]{} \pi_{i+1}$.

To parallelize actor and learner's computation, Cleanba's architecture needs to necessarily introduce stale data like IMPALA (Espeholt et al., 2018). In the second iteration of Cleanba's architecture in Figure 1, we skip the `param_Q.get()` call, so $\pi_1 \to \mathcal{D}_{\pi_1}$ happens concurrently with $\pi_1 \xrightarrow[\mathcal{D}_{\pi_1}]{} \pi_2$. Because `Queue.get` is blocking when the queue is empty and `Queue.put` is blocking when the queue is full (we set the maximum size to be 1), we make sure the actor process does not perform more rollouts and learner process does not perform more gradient updates. Starting iteration $i > 3$, the learner then learns from the rollout data of the second latest policy $\pi_i \xrightarrow[\mathcal{D}_{\pi_{i-1}}]{} \pi_{i+1}$. As a result, Cleanba's architecture can parallelize the actor and learner's computation at the cost of stale data.

Cleanba's architecture above has several benefits. First, it is easy to reason and reproduce. As highlighted in Table 1, we can ascertain the specific policy used for collecting the rollout data, so if we had delayed learner updates like in Section 3 for iteration i, iteration $i + 1$ would not start until the previous iteration is finished, therefore circumventing IMPALA's reproducibility issue. This knowledge about which policy generates the rollout data enhances the transparency and reproducibility of distributed RL and can help us scale up while maintaining good reproducibility principles. Second, Cleanba's architecture is easy to debug for throughput. For diagnosing throughput, we can evaluate the time taken for `rollout_Q.get()` and `param_Q.get()`. If, on average, `rollout_Q.get()` consumes less time than `param_Q.get()`, it becomes evident that learning is the bottleneck, and vice versa.

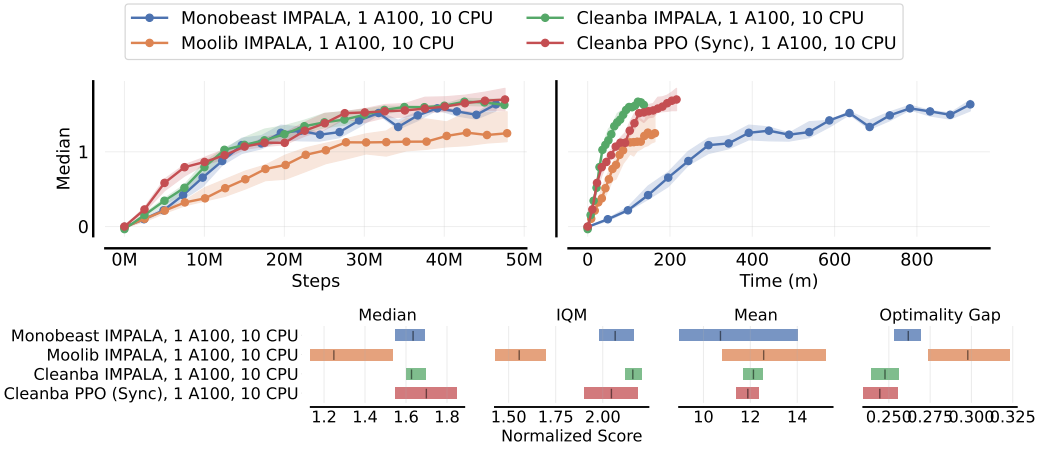

Figure 3: Base experiments. Top figure: the median human-normalized scores of Cleanba variants compared with `moolib` and `monobeast`. Bottom figure: the aggregate human normalized score metrics with 95% stratified bootstrap CIs. Higher is better for Median, IQM, and Mean; lower is better for Optimality Gap.

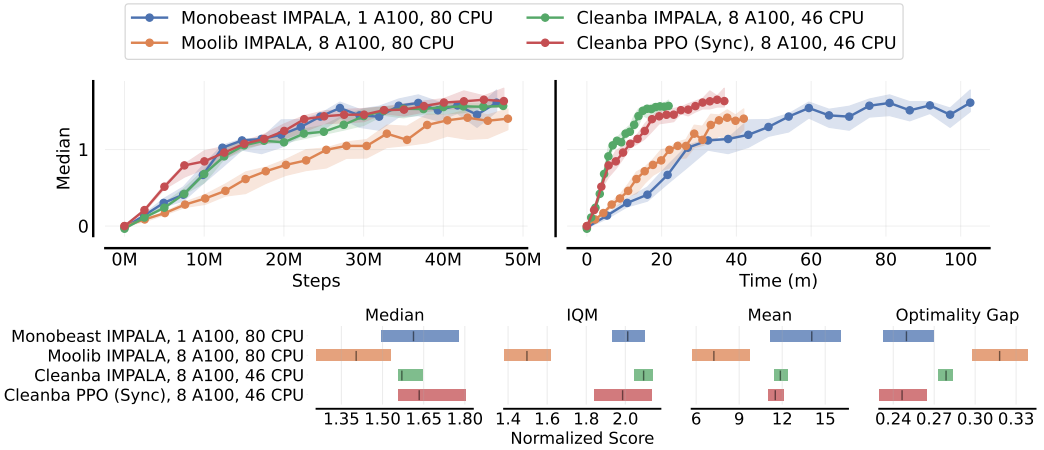

Figure 4: Workstation experiments. Top figure: the median human-normalized scores of Cleanba variants compared with `moolib`. Bottom figure: the aggregate human normalized score metrics with 95% stratified bootstrap CIs.

Based on Cleanba's architecture, this work introduces Cleanba as a reproducible distributed DRL platform. Cleanba is inspired by CleanRL (Huang et al., 2022b) and DeepMind's Sebulba Podracer architecture (Hessel et al., 2021). Its implementation uses JAX (Bradbury et al., 2018) and EnvPool (Weng et al., 2022), both of which are designed to be efficient. To improve the learner's throughput, we allow the use of multiple learner devices via `pmap`. To improve the system's scalability, we enable running multiple processes on a single node or multiple nodes via `jax.distibuted`.

5 Experiments

We perform experiments on Atari games (Bellemare et al., 2013). All experiments used 84×84 images with greyscale, an action repeat of 4, 4 stacked frames, and a maximum of 108,000 frames per episode. We followed the recommended Atari evaluation protocol by Machado et al. (2018), which used sticky action with a probability of 25%, no loss of life signal, and the full action space.

To make a more direct and fair comparison, we used the same AWS p4d.24xlarge instances[3] and the same Atari environment simulation setups via EnvPool and compared only the following codebase settings:

1. **Monobeast IMPALA**: the reference IMPALA implementations in `monobeast`[4];
2. **Moolib IMPALA**: the reference IMPALA implementations in `Moolib`;
3. **CleanRL PPO (Sync)**: the reference PPO implementations in CleanRL(Huang et al., 2022b);
4. **Cleanba PPO** and **Cleanba IMPALA**: our PPO and IMPALA implementation under the Cleanba Architecture;
5. **Cleanba PPO (Sync)** and **Cleanba IMPALA (Sync)** our PPO and IMPALA implementation under the Synchronous Architecture (Table 1), which can be configured by commenting out line 7 of the Cleanba's architecture in Figure 1.

Within the p4d.24xlarge instance, we also compared two hardware settings:

1. **Base experiments** uses 10 CPU and 1 A100 setting as a base comparison;
2. **Workstation experiments** uses 46 CPU and 8 A100s for Cleanba experiments, 80 CPU and 8 A100s for `moolib` experiments[5], and 80 CPU and 1 A100 for `monobeast` experiments.

Throughout all experiments, the agents used IMPALA's Resnet architecture (Espeholt et al., 2018), ran for 200M frames with three random seeds. The hyperparameters and the learning curves can be found in Appendix B. We evaluate the experiment results based on median HNS learning curves, interquartile mean (IQM) learning curves, and 95% stratified bootstrap confidence intervals for the mean, median, IQM, and optimality gap (the amount by which the algorithm fails to meet a minimum normalized score of 1) (Agarwal et al., 2021).

5.1 Comparison with `moolib` and `monobeast`'s IMPALA

Under the base experiments (Figure 3), Cleanba's IMPALA obtains a similar level of median HNS as `monobeast`'s IMPALA and a higher level of median HNS as `moolib`'s IMPALA. However, Cleanba's IMPALA is **6.8x faster** than `monobeast`'s IMPALA, mostly because Cleanba actors run on GPUs, whereas `monobeast`'s actors run on CPUs. Also, Cleanba's IMPALA is **1.2x faster** than `moolib`'s IMPALA, but the speedup difference is challenging to explain due to multiple confounding factors – Cleanba's variants benefit from JAX's just-in-time compilation, whereas `moolib` benefits from asynchronous operations (e.g., on gradient computation and environment steps). Cleanba's PPO (Sync) also obtains a high median HNS but takes longer training time, likely due to the longer training step time spent on reusing rollout data 4 times.

Under the workstation experiments (Figure 4), Cleanba's PPO (Sync) and IMPALA obtain a similar level of median HNS as `monobeast`'s IMPALA and a higher level of median HNS as `moolib`'s IMPALA. However, Cleanba's PPO (Sync) and IMPALA are both faster than `monobeast`'s and `moolib` IMPALA. Most prominently, Cleanba's IMPALA is **5x faster** than `monobeast`'s IMPALA and **2x faster** than `moolib`'s IMPALA.

Additionally, we examine the individual learning curves in Figure 5 and found that Cleanba's variants also produce more consistent learning curves. In comparison, in two hardware settings, `moolib`'s learning curves can be much more unpredictable.

[3]For some experiments, we used p4de.24xlarge instances but only GPU memory is different, which does not affect training speed.

[4]We wanted to test out IMPALA's official source code released in `deepmind/scalable_agent`, but it was built with `tensorflow 1.x` which does not support the A100 GPU tested in this paper.

[5]We used more CPUs for `moolib` experiments because 10 CPU per GPU seems to be the default scaling parameter for `moolib`. Also, for the `moolib` experiment, we conducted two sets of 3 random seeds. We reported the results with higher IQM and lower median. See Appendix C.

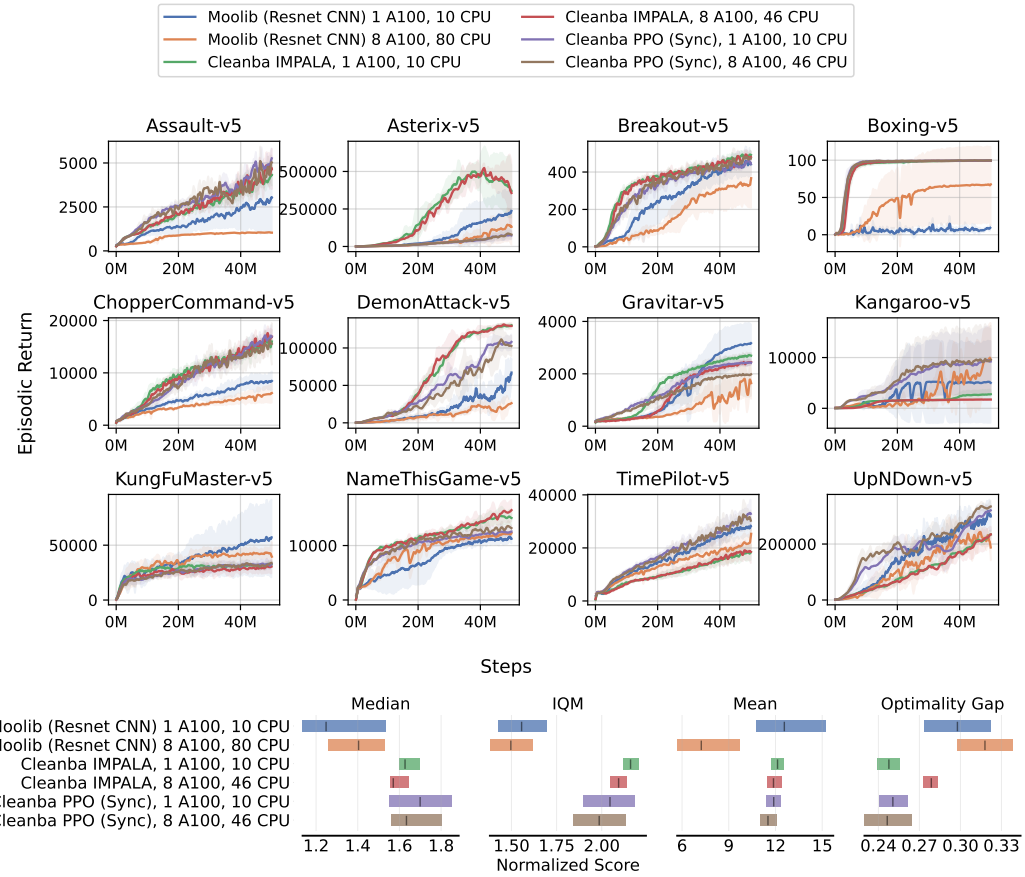

Figure 5: Reproducible learning curves – the Cleanba variants show more predictable learning curves in different hardware settings. In comparison, `moolib`'s IMPALA's learning curves under the 1 A100, 10 CPU setting (blue curve) and 8 A100, 80 CPU setting (orange curve) are meaningfully different, even if they use the same hyperparameters.

5.2 Discussion about `monobeast`'s IMPALA

Note that the `monobeast` experiments are interesting in several ways. First, it produces a higher median HNS than `moolib`'s IMPALA, which is the opposite of what was shown in Mella et al. (2022). This is probably because Mella et al. (2022) used "comparable environment settings" instead of the same environment settings used in our experiments. Interestingly, we found different Atari wrapper implementations can have a non-trivial impact on the agent's performance (Appendix D); for this reason, we use the same Atari wrapper implementation in the experiments presented in this section. Second, the `monobeast` experiments appear robust in two different hardware settings in practice, despite the reproducibility issues we showed in Section 3. While `monobeast` obtained high scores, it is significantly slower in the 1 A100 and 10 CPU settings due to poor GPU utilization. Its codebase also does not support multi-GPU settings and should scale less efficiently with larger networks because actor threads only run on CPUs when compared to `moolib` and Cleanba's variants.

5.3 Synchronous Architecture vs Cleanba Architecture

Figure 6 compares the PPO and IMPALA variants between Synchronous and Cleanba architecture and CleanRL's PPO, which uses the Synchronous architecture by design. We found using Cleanba architecture actually hurts Cleanba PPO's data efficiency. This is an interesting trade-off because the speed benefit of parallelizing actor and learner processes in Cleanba PPO is offset by the lower

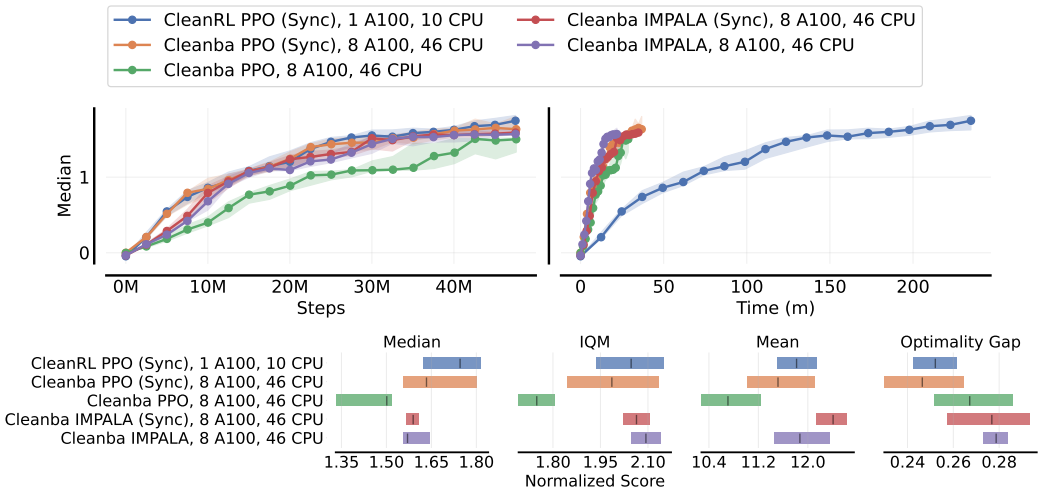

Figure 6: Comparing Cleanba's variants using Cleanba and Synchronous architecture. For PPO, Cleanba's Architecture (orange curve) runs faster but has lower data efficiency than Synchronous architecture (blue curve). For IMPALA, there is no discernible difference between Synchronous Architecture (red curve) and Cleanba's Architecture (brown curve). This means Cleanba's IMPALA can benefit from the speed-up of parallelizing actor-learner computation without paying a price for data efficiency under our hyperparameter settings, unlike Cleanba's PPO.

data efficiency. Among many possible causes, the main factor might be that PPO does 16 gradient updates (4 mini-batches and 4 update epochs) per rollout, whereas IMPALA in our setting only does 4 gradient updates. In comparison, we noticed Cleanba's IMPALA did not suffer from lower data efficiency compared to Cleanba IMPALA (Sync) architecture, meaning IMPALA can actually benefit from parallelizing actor and learner computations.

6 Limitation

There are several limitations to this work. First, our experiments could not completely control various other confounding settings in the reference codebase, such as optimizer settings and machine learning framework (e.g., PyTorch, JAX). For example, Cleanba's PPO and IMPALA use different learning rates indicated in their respective literature, making it difficult to compare PPO and IMPALA directly. We attempted to make a direct comparison by running Cleanba PPO with Cleanba IMPALA's setting and found it made PPO's data efficiency significantly worse – this could suggest the IMPALA's setting is well-tuned for IMPALA but brittle to PPO (Appendix E). Second, our finding that parallelizing actor and learner computation hurts PPO's data efficiency is specific to the PPO's default Atari hyperparameter setting, and it could perhaps be tuned in ways in which opposite findings can be drawn. That said, the main purpose of this work is not hyperparameter tuning. Rather, it is creating a codebase that replicates prior results and makes training reproducible, efficient, and scalable across more powerful hardware.

7 Conclusion

This paper presents Cleanba, a new distributed deep reinforcement learning platform. Our analysis shows that Cleanba's more principled architecture can circumvent reproducibility issues in IMPALA's architecture. Our Atari experiments demonstrate that Cleanba's PPO and IMPALA accurately replicate prior work but have faster training time and are highly reproducible across different hardware settings. We believe that Cleanba will be a valuable platform for the research community to conduct future distributed RL research.

REPRODUCIBILITY STATEMENT

Ensuring Cleanba's results are reproducible is a central theme in our paper. To this end, we have taken several measures to improve reproducibility:

1. **Open-source repository**: we made source code available at `https://github.com/vwxyzjn/cleanba`. The dependencies of the experiments are pinned, and our repository contains detailed instructions on replicating all Cleanba experiments presented in this paper.
2. **Reproducible architecture**: as demonstrated in Section 4, Cleanba introduces a more principled approach to understanding distributed DRL and gives clear expectations on where the rollout data comes from, making it easier to reason about the reproducibility of distributed DRL.
3. **Experiments on different hardware**: as demonstrated in Section 5, we also conducted experiments showing Cleanba's PPO and IMPALA variants can obtain near-identical data efficiency on different hardware, further demonstrating that this work is highly reproducible.

In sum, we have tried to make our work as transparent and reproducible as possible. By leveraging the source code, details provided in the main paper, and appendix, researchers should be well-equipped to reproduce or extend upon our findings.

REFERENCES

Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C Courville, and Marc Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in Neural Information Processing Systems*, 34, 2021.

Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Leonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What matters for on-policy deep actor-critic methods? a large-scale study. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=nIAxjsniDzg`.

Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. Reinforcement learning through asynchronous advantage actor-critic on a GPU. In *International Conference on Learning Representations*, 2017. URL `https://openreview.net/forum?id=r1VGvBcxl`.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47: 253–279, 2013.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, et al. Jax: composable transformations of python+ numpy programs. 2018.

Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation matters in deep rl: A case study on ppo and trpo. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=r1etN1rtPB`.

Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. In Jennifer G. Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1406–1415. PMLR, 2018. URL `http://proceedings.mlr.press/v80/espeholt18a.html`.

Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. Seed rl: Scalable and efficient deep-rl with accelerated central inference. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rkgvXlrKwH.

C Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax–a differentiable physics engine for large scale rigid body simulation. *arXiv preprint arXiv:2106.13281*, 2021.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Matteo Hessel, Manuel Kroiss, Aidan Clark, Iurii Kemaev, John Quan, Thomas Keck, Fabio Viola, and Hado van Hasselt. Podracer architectures for scalable reinforcement learning. *arXiv preprint arXiv:2104.06272*, 2021.

Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL https://openreview.net/forum?id=H1Dy---0Z.

Shengyi Huang, Rousslan Fernand Julien Dossa, Antonin Raffin, Anssi Kanervisto, and Weixun Wang. The 37 implementation details of proximal policy optimization. In *ICLR Blog Track*, 2022a. URL https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.

Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022b. URL http://jmlr.org/papers/v23/21-1342.html.

Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*, 2019.

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

Iou-Jen Liu, Raymond A. Yeh, and Alexander G. Schwing. High-throughput synchronous deep RL. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/c6447300d99fdbf4f3f7966295b8b5be-Abstract.html.

Marlos C Machado, Marc G Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

Vegard Mella, Eric Hambro, Danielle Rothermel, and Heinrich Küttler. moolib: A Platform for Distributed RL. 2022. URL https://github.com/facebookresearch/moolib.

Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria-Florina Balcan and Kilian Q. Weinberger (eds.), *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pp. 1928–1937. JMLR.org, 2016. URL http://proceedings.mlr.press/v48/mniha16.html.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022.

Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav S. Sukhatme, and Vladlen Koltun. Sample factory: Egocentric 3d control from pixels at 100000 FPS with asynchronous reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pp. 7652–7662. PMLR, 2020. URL http://proceedings.mlr.press/v119/petrenko20a.html.

Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. Trust region policy optimization. In Francis R. Bach and David M. Blei (eds.), *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pp. 1889–1897. JMLR.org, 2015. URL http://proceedings.mlr.press/v37/schulman15.html.

John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In Yoshua Bengio and Yann LeCun (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1506.02438.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *ArXiv preprint*, abs/1707.06347, 2017. URL https://arxiv.org/abs/1707.06347.

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

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. URL https://openreview.net/forum?id=BubxnHpuMbG.

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 *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=H1gX8C4YPr.

Algorithm 1 Proximal Policy Optimization

Initialize environment E containing `local_num_envs` parallel sub-environments
Initialize policy parameters θ_π, value parameters θ_v, optimizer O
Initialize observation s_{next}, done flag d_{next}
for i = 0,1,2,..., I **do**
  Set $\mathcal{D} = (s, a, \log \pi(a|s), r, d, v)$ as tuple of 2D arrays
  for t = 0,1,2,..., `num_steps` **do** ▷ Rollout Phase
    Cache $o_t = s_{next}$ and $d_t = d_{next}$
    Get $a_t \sim \pi(\cdot|s_t; \theta_\pi)$ and $v_t = v(s_t; \theta_v)$
    Step simulator: $s_{next}, r_t, d_{next} = E.step(a_t)$
    Store $s_t, d_t, v_t, a_t, \log \pi(a_t|s_t; \theta_\pi), r_t$ in $\mathcal{D}$
  Estimate next value $v_{next} = v(s_{next})$ ▷ Learning Phase
  Compute advantage $\hat{A}^{\text{adv}}_\pi$ and return R using $\mathcal{D}$ and v_{next}
  Prepare the batch $\mathcal{B} = \mathcal{D}$, $\hat{A}^{\text{adv}}_\pi$, R and flatten $\mathcal{B}$
  for *epoch* = 0,1,2,..., `update_epochs` **do**
    for mini-batch $\mathcal{M}$ of size m in $\mathcal{B}$ **do**
      Normalize advantage $\mathcal{M}.\hat{A}^{\text{adv}}_\pi$
      Compute policy loss L^π, value loss L^V, and entropy loss L^S using $\mathcal{M}$
      Back-propagate joint loss $L = -L^\pi + c_1 L^V - c_2 L^S$
      Clip maximum gradient norm of θ_π and θ_v to 0.5
      Step optimizer O w.r.t. θ_π and θ_v

A PRELIMINARIES

Let us consider the RL problem in a *Markov Decision Process (MDP)* (Puterman, 2014), where $\mathcal{S}$ is the state space and $\mathcal{A}$ is the action space. The agent performs some actions to the environment, and the environment transitions to another state according to its *dynamics* $P(s' \mid s, a) : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$. The environment also provides a scalar reward according to the reward function $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and the agent attempts to maximize the expected discounted return following a policy π:

$$J(\pi) = \mathbb{E}_\tau \left[G(\tau) \right] \tag{1}$$
$$\text{where } \tau \text{ is the trajectory } (s_0, a_0, r_0, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T)$$
$$\text{and } s_0 \sim \rho_0, s_t \sim P(\cdot|s_{t-1}, a_{t-1}), a_t \sim \pi_\theta(\cdot|s_t), r_t = r\left(s_t, a_t\right)$$

PPO (Schulman et al., 2017) is a popular algorithm that proposes a clipped policy gradient objective to help avoid unstable updates (Schulman et al., 2017; 2015):

$$J^{\text{CLIP}}(\pi_\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \min \left(r_t(\theta) \hat{A}^{\text{adv}}_\pi(s_t, a_t), \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \hat{A}^{\text{adv}}_\pi(s_t, a_t) \right) \right] \tag{2}$$

where $\pi_{\theta_{\text{old}}}$ is the policy parameter before the update, $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$, $\hat{A}^{\text{adv}}_\pi$ is an advantage estimator called Generalized Advantage Estimator (Schulman et al., 2016), and ϵ is PPO's clipped coefficient. During the optimization phase, the agent also learns the value function and maximizes the policy's entropy, therefore optimizing the following joint objective:

$$J^{\text{JOINT}}(\theta) = J^{\text{CLIP}}(\pi_\theta) - c_1 J^{\text{VF}}(\theta) + c_2 S[\pi_\theta], \tag{3}$$

where c_1, c_2 are coefficients, S is an entropy bonus, and J^{VF} is the squared error loss for the value function associated with π_θ. Algorithm 1 shows the pseudocode of PPO that more accurately reflects how PPO is implemented in the original codebase[6]. For more detail on PPO's implementation, see (Huang et al., 2022a). Given this pseudocode, the following list unifies the nomenclature/terminology of PPO's key hyperparameters.

- `world_size` is the number of instances of training processes; typically this is 1 (e.g., you have a single GPU).

[6] `https://github.com/openai/baselines`

- `local_num_envs` is the number of parallel environments PPO interacts within an instance of the training process (see line 1). `num_envs = world_size * local_num_envs` is the total number of environments across all training instances.
- `num_steps` is the number of steps in which the agent samples a batch of `local_num_envs` actions and receives a batch of `local_num_envs` next observations, rewards, and done flags from the simulator (see line 6), where the done flags signal if the episodes are terminated or truncated. `num_steps` has many names, such as the "sampling horizon" (Stooke & Abbeel, 2018) and "unroll length" (Freeman et al., 2021).
- `local_batch_size` is the batch size calculated as `local_num_envs * num_steps` within an instance of the training process (`local_batch_size` is the size of the $\mathcal{B}$ in line 13).
- `batch_size` = `world_size * local_batch_size` is the aggregated batch size across all training instances.
- `update_epochs` is the number of update epochs that the agent goes through the training data in $\mathcal{B}$ (see line 14).
- `num_minibatches` is the number of mini-batches that PPO splits $\mathcal{B}$ into (see line 15).
- `local_minibatch_size` is $m =$ `local_batch_size` / `num_minibatches`, the size of each mini-batch $\mathcal{M}$ (see line 15). `minibatch_size = world_size * local_minibatch_size` is the aggregated batch size across all training instances.

To make understanding more concrete, let us consider an example of Atari training. Typically, PPO uses a single training instance (i.e., `world_size` = 1), `local_num_envs` = `num_envs` = 8, and `num_steps` = 128. In the rollout phase (line 6-10), the agent collects a batch of $8 * 128 = 1024$ data points in $\mathcal{D}$. Then, suppose `num_minibatches` = 4, $\mathcal{D}$ is evenly split to 4 mini-batches of size $m = 1024/4 = 256$. Next, if $K = 4$, the agent would perform K * `num_minibatches` = 16 gradient updates in the learning phase (line 11-20).

We consider two options to scale to larger training data. **Option 1** is to increment `local_num_envs` – the agent interacts with more environments, and as a result, the training data is larger. The second option is to increment `world_size` – have two or more copies of Algorithm 1 running in parallel and average the gradient of the copies in line 20. **Option 2** is especially desirable when the users want to leverage more computational resources, such as GPUs.

Note that both options can be equivalent *in terms of hyperparameters*. For example, when setting `world_size` = 2, the agent effectively interacts with two distinct sets of `local_num_envs` environments, making its `num_envs` doubled. To make option 1 achieve the same hyperparameters, we just need to double its `local_num_envs`. Below is a table summarizing the resulting hyperparameters of both options.

Hyperparameter	**Option 1**: Increment `local_num_envs`	**Option 2**: Increment `world_size`
`world_size`	1	2
`local_num_envs`	120	60
`num_envs`	**120**	**120**
`num_steps`	128	128
`local_batch_size`	15360	7680
`batch_size`	**15360**	**15360**
`num_minibatches`	4	4
`local_minibatch_size`	3840	1920
`minibatch_size`	**3840**	**3840**

Importantly, we can get the same hyperparameter configuration for PPO by adjusting `local_num_envs` and `world_size` accordingly. That is, we can obtain the same `num_envs`, `batch_size`, and `minibatch_size` core hyperparameters.

B Detailed experiment settings

For the experiments, the PPO and IMPALA's hyperparameters can be found in Table 2. The Vtrace implementation can be found in `rlax`[7].

Table 2: PPO hyperparameters.

Parameter Names	Parameter Values
N_{total} Total Time Steps	50,000,000
α Learning Rate	0.00025 Linearly Decreased to 0
N_{envs} Number of Environments	128
N_{steps} Number of Steps per Environment	128
γ (Discount Factor)	0.99
λ (for GAE)	0.95
N_{mb} Number of Mini-batches	4
K (Number of PPO Update Iteration Per Epoch)	4
ε (PPO's Clipping Coefficient)	0.1
c_1 (Value Function Coefficient)	0.5
c_2 (Entropy Coefficient)	0.01
ω (Gradient Norm Threshold)	0.5
Value Function Loss Clipping	False
Optimizer Setting	Adam optimizer with $\epsilon = 0.00001$

Table 3: IMPALA hyperparameters.

Parameter Names	Parameter Values
N_{total} Total Time Steps	50,000,000
α Learning Rate	0.0006 Linearly Decreased to 0
N_{envs} Number of Environments	128
N_{steps} Number of Steps per Environment	20
γ (Discount Factor)	0.99
λ (mixing parameter)	1.0
N_{mb} Number of Mini-batches	4
ρ (Clip Threshold for Importance Ratios)	1.0
ρ_{pg} (Clip Threshold for Policy Gradient Importance Ratios)	1.0
c_1 (Value Function Coefficient)	0.5
c_2 (Entropy Coefficient)	0.01
ω (Gradient Norm Threshold)	40.0
Optimizer Setting	RMSprop optimizer with $\epsilon = 0.01$, decay $= 0.99$

C `moolib` Experiments

By default, `moolib` uses 256 environments, 10 actor CPUs, and a single GPU. We followed the recommended scaling instructions to add 8 training GPU-powered peers, which in total used 2048 environments, 80 actor CPUs, and 8 GPUs. While the training time was reduced to about 27 minutes, sample efficiency dropped, and it obtained a catastrophic 28.51% median HNS after 200M frames. We suspected the drop was due to the 2048 environments used, so we set the total number of environments back to 256. Furthermore, we did not restrict `moolib` to use 50 CPUs because we worried it might change the learning behaviors due to the issues mentioned in Section 3, so we kept the default scaling to 80 CPUs. For comparison with `moolib`, `monobeast` experiments also use 80 CPUs.

[7] https://github.com/deepmind/rlax/blob/b53c6510c8b2cad6b106b6166e22aba61a77ee2f/rlax/_src/vtrace.py#L162-L193

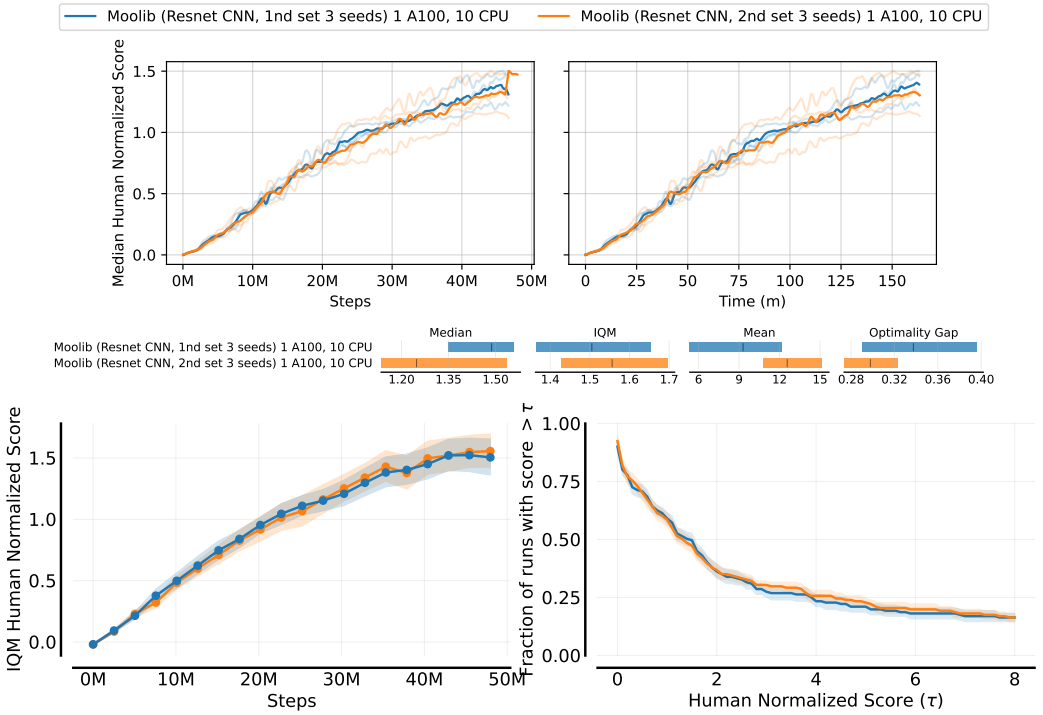

Figure 7: Top figure: the median human-normalized scores of the two sets of `moolib` experiments. Middle figure: the IQM human-normalized scores and performance profile (Agarwal et al., 2021). Bottom figure: the average runtime in minutes and aggregate human normalized score metrics with 95% stratified bootstrap CIs.

We conducted two sets of `moolib` experiments and reported the set with a lower median and higher IQM, as shown in Figure 7 for legacy reasons. During our debugging, we found the Asteroids experiments in the first set of `moolib` experiments to obtain high scores, but we ran Asteroids specifically for ten random seeds and found lower scores; this suggests the Asteroids experiments in the first set were likely due to lucky random seeds, so we re-run the `moolib` experiments.

D THE EFFECT OF DIFFERENT WRAPPERS ON MOOLIB'S PERFORMANCE

The Atari wrappers can be important to the agent's performance. As a preliminary study, we used `moolib`'s default Atari wrappers[8] implemented with `gym.AtariPreprocessing` to run experiments and compare the results with the ones presented in the main text of the paper. As shown in Figure 8, Atari wrappers matter – `moolib`'s default `AtariPreprocessing` wrappers result in lower median and mean HNS, although IQM is roughly the same. To make a fair comparison, the experiments presented in the main text all use the same EnvPool Atari wrappers.

E DIRECT PPO AND IMPALA COMPARISON

To make a direct (but not fair) comparison between PPO and IMPALA, we ran Cleanba PPO using IMPALA's settings and the results can be found at Figure 9.

[8] `https://github.com/facebookresearch/moolib/blob/main/examples/atari/atari_preprocessing.py`

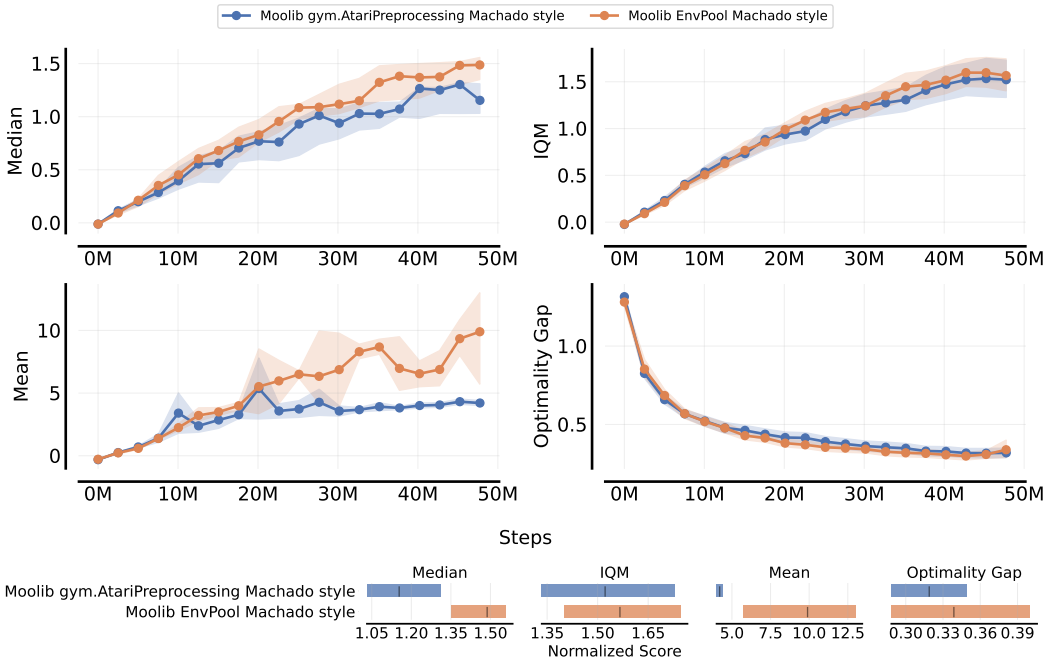

Figure 8: Atari wrappers matter. When using `gym.AtariPreprocessing` wrappers with a comparable setting to our EnvPool setup, we found `moolib`to have lower median and mean HNS, although IQM is roughly the same.

F Large Batch Size Training

Cleanba can also scale to the hundreds of GPUs in multi-host and multi-process environments by leveraging the `jax.distributed` package, allowing us to explore training with even larger batch sizes. Here we use an earlier version of the codebase to conduct experiments with 16, 32, 64, and 128 A100 GPUs. For convenience, we also adjust a few settings: 1) turn off the learning rate annealing, 2) run for 100M steps instead of the standard 50M steps, and 3) keep doubling the `num_envs`, `batch_size`, and `minibatch_size` with a larger number of GPUs.

Due to hardware scheduling constraints, we only ran the experiments for 1 random seed. The results are shown in Figure 10. We make the following observations:

- **Linear scaling w/ 93% of ideal scaling efficiency.** As we increased the number of GPUs to 16, 32, 64, 128, we observed a linear scaling in steps per second (SPS) in Cleanba achieving 93% of the ideal scaling efficiency. This is likely empowered by the fast connectivity offered by NVIDIA GPUDirect RDMA (remote direct memory access) in Stability AI's HPC. When using 128 GPUs, the agent has an SPS of 403253, translating to over *1.6M FPS* in Breakout.
- **Small batch sizes train more efficiently.** As we increase batch sizes, particularly in the first 40M steps, the sample efficiency tends to decline. This outcome is unsurprising, given that the initial policy is random and Breakout initially has limited explorable game states. In this case, the data in the batch is going to have less diverse data, which makes the large batch size less valuable.
- **Large batch sizes train more quickly.** Like (McCandlish et al., 2018), we find increasing the batch size does make the agent reach some given scores faster. This suggests that we could always increase the batch size to obtain shorter training times if sample efficiency is not a concern.

While we observed limited benefits of scaling Cleanba to use 128 GPUs, the objective of the scaling experiments is to show we can scale to large batch sizes. Given a more challenging task, the training

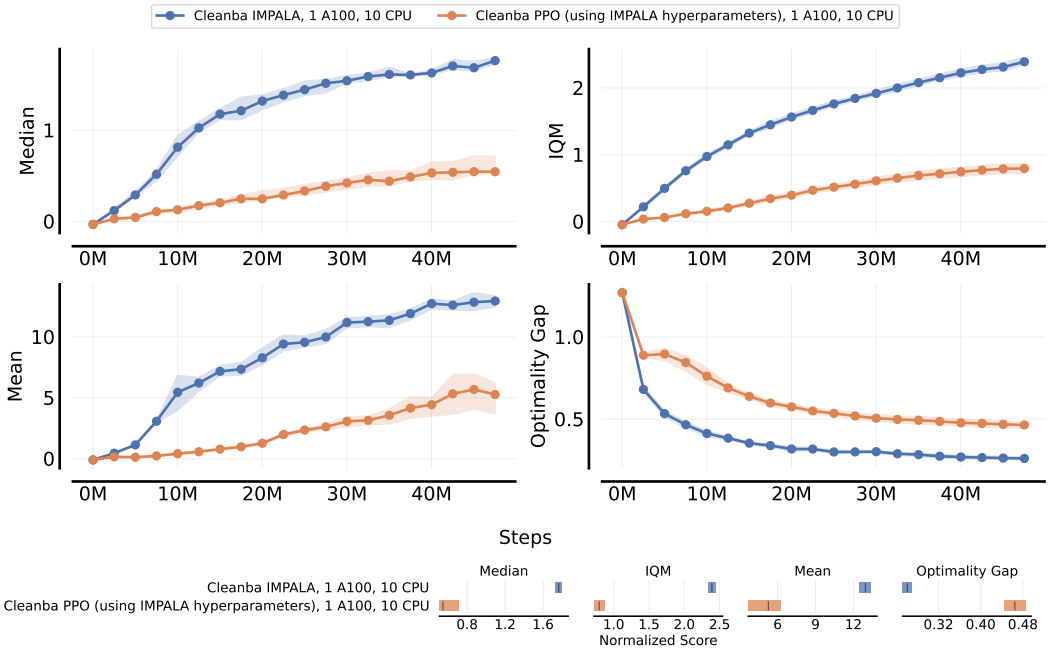

Figure 9: A direct PPO and IMPALA comparison. Running Cleanba PPO using Cleanba IMPALA's setting. Note that this is not a fair comparison because Cleanba IMPALA's setting is likely well-tuned IMPALA setting but not well-tuned PPO setting

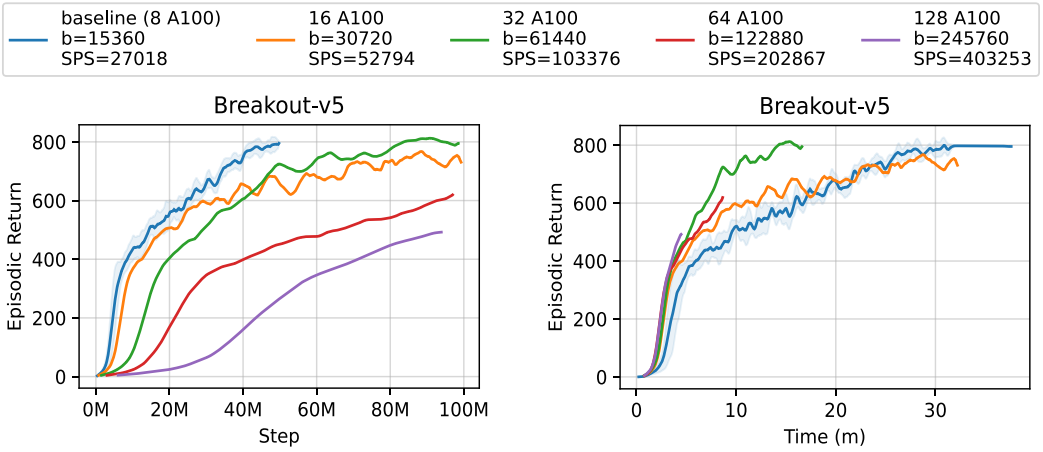

Figure 10: Cleanba's results from large batch size training. `b=15360` denotes `batch_size=15360`.

data is likely going to be more diverse and have a higher *gradient noise scale* (McCandlish et al., 2018), which would help the agent utilize large batch sizes more efficiently, resulting in a reduced decline in sample efficiency.

G TORCHBEAST LOGS

```
$ python -m torchbeast.monobeast_study \
--num_actors 80 \
```

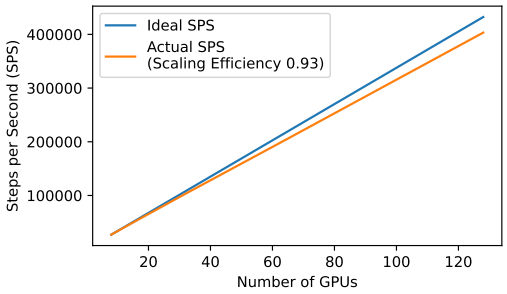

Figure 11: Cleanba's SPS scaling results from large batch size training.

```
--total_steps 10000000 \
--learning_rate 0.0006 \
--epsilon 0.01 \
--entropy_cost 0.01 \
--batch_size 8 \
--unroll_length 240 \
--num_threads 1 \
--env Pong-v5
actor_index 32 initial policy_version 8 policy_version after rollout 20
actor_index 13 initial policy_version 8 policy_version after rollout 20
actor_index 57 initial policy_version 8 policy_version after rollout 20
actor_index 12 initial policy_version 8 policy_version after rollout 21
actor_index 51 initial policy_version 8 policy_version after rollout 21
actor_index 2 initial policy_version 8 policy_version after rollout 21
actor_index 56 initial policy_version 8 policy_version after rollout 21
actor_index 38 initial policy_version 9 policy_version after rollout 21
actor_index 37 initial policy_version 9 policy_version after rollout 22
actor_index 59 initial policy_version 9 policy_version after rollout 22
actor_index 9 initial policy_version 9 policy_version after rollout 22
actor_index 69 initial policy_version 9 policy_version after rollout 22
actor_index 35 initial policy_version 9 policy_version after rollout 22
actor_index 66 initial policy_version 9 policy_version after rollout 22
actor_index 10 initial policy_version 9 policy_version after rollout 22
actor_index 55 initial policy_version 10 policy_version after rollout 22
actor_index 53 initial policy_version 10 policy_version after rollout 22
actor_index 46 initial policy_version 10 policy_version after rollout 22
actor_index 54 initial policy_version 10 policy_version after rollout 23
actor_index 50 initial policy_version 10 policy_version after rollout 23
actor_index 8 initial policy_version 10 policy_version after rollout 23
actor_index 64 initial policy_version 10 policy_version after rollout 23
actor_index 77 initial policy_version 10 policy_version after rollout 23
actor_index 3 initial policy_version 11 policy_version after rollout 23
actor_index 7 initial policy_version 11 policy_version after rollout 23
actor_index 28 initial policy_version 11 policy_version after rollout 23
actor_index 49 initial policy_version 11 policy_version after rollout 23
actor_index 16 initial policy_version 11 policy_version after rollout 23
actor_index 24 initial policy_version 11 policy_version after rollout 23
actor_index 11 initial policy_version 11 policy_version after rollout 23
actor_index 14 initial policy_version 11 policy_version after rollout 23
actor_index 43 initial policy_version 13 policy_version after rollout 26
actor_index 58 initial policy_version 13 policy_version after rollout 26
actor_index 23 initial policy_version 13 policy_version after rollout 26
actor_index 29 initial policy_version 13 policy_version after rollout 26
actor_index 68 initial policy_version 13 policy_version after rollout 26
actor_index 75 initial policy_version 14 policy_version after rollout 26
actor_index 48 initial policy_version 14 policy_version after rollout 27
actor_index 67 initial policy_version 14 policy_version after rollout 27
actor_index 5 initial policy_version 14 policy_version after rollout 27
actor_index 18 initial policy_version 14 policy_version after rollout 27
actor_index 41 initial policy_version 15 policy_version after rollout 27
```

```
actor_index 78 initial policy_version 14 policy_version after rollout 27
actor_index 15 initial policy_version 15 policy_version after rollout 27
actor_index 34 initial policy_version 15 policy_version after rollout 27
actor_index 45 initial policy_version 15 policy_version after rollout 28
actor_index 22 initial policy_version 15 policy_version after rollout 28
actor_index 4 initial policy_version 16 policy_version after rollout 28
actor_index 6 initial policy_version 16 policy_version after rollout 28
actor_index 20 initial policy_version 16 policy_version after rollout 28
actor_index 39 initial policy_version 16 policy_version after rollout 28
actor_index 33 initial policy_version 16 policy_version after rollout 29
actor_index 74 initial policy_version 16 policy_version after rollout 29
actor_index 60 initial policy_version 16 policy_version after rollout 29
actor_index 42 initial policy_version 17 policy_version after rollout 29
actor_index 72 initial policy_version 17 policy_version after rollout 30
actor_index 25 initial policy_version 17 policy_version after rollout 30
actor_index 31 initial policy_version 17 policy_version after rollout 30
actor_index 19 initial policy_version 17 policy_version after rollout 30
actor_index 1 initial policy_version 18 policy_version after rollout 31
actor_index 79 initial policy_version 18 policy_version after rollout 31
actor_index 65 initial policy_version 18 policy_version after rollout 31
actor_index 73 initial policy_version 18 policy_version after rollout 31
actor_index 36 initial policy_version 18 policy_version after rollout 31
actor_index 21 initial policy_version 18 policy_version after rollout 31
actor_index 0 initial policy_version 18 policy_version after rollout 31
actor_index 30 initial policy_version 18 policy_version after rollout 31
actor_index 44 initial policy_version 18 policy_version after rollout 31
actor_index 63 initial policy_version 19 policy_version after rollout 31
actor_index 76 initial policy_version 19 policy_version after rollout 32
actor_index 47 initial policy_version 19 policy_version after rollout 32
actor_index 52 initial policy_version 19 policy_version after rollout 32
actor_index 26 initial policy_version 19 policy_version after rollout 32
actor_index 71 initial policy_version 19 policy_version after rollout 32
actor_index 70 initial policy_version 19 policy_version after rollout 32
actor_index 17 initial policy_version 20 policy_version after rollout 32
actor_index 62 initial policy_version 20 policy_version after rollout 33
actor_index 40 initial policy_version 20 policy_version after rollout 33
actor_index 27 initial policy_version 20 policy_version after rollout 33
actor_index 13 initial policy_version 20 policy_version after rollout 33
actor_index 57 initial policy_version 20 policy_version after rollout 33
actor_index 32 initial policy_version 20 policy_version after rollout 33
actor_index 51 initial policy_version 21 policy_version after rollout 33
actor_index 61 initial policy_version 20 policy_version after rollout 33
actor_index 2 initial policy_version 21 policy_version after rollout 33
actor_index 56 initial policy_version 21 policy_version after rollout 34
actor_index 12 initial policy_version 21 policy_version after rollout 34


$ python -m torchbeast.monobeast_study \
--num_actors 80 \
--total_steps 10000000 \
--learning_rate 0.0006 \
--epsilon 0.01 \
--entropy_cost 0.01 \
--batch_size 8 \
--unroll_length 240 \
--num_threads 1 \
--env Pong-v5  \
--learner_delay_seconds 1.0

actor_index 72 initial policy_version 9 policy_version after rollout 10
actor_index 22 initial policy_version 9 policy_version after rollout 10
actor_index 37 initial policy_version 9 policy_version after rollout 10
actor_index 41 initial policy_version 9 policy_version after rollout 10
actor_index 16 initial policy_version 9 policy_version after rollout 10
actor_index 61 initial policy_version 10 policy_version after rollout 11
actor_index 18 initial policy_version 10 policy_version after rollout 11
actor_index 13 initial policy_version 10 policy_version after rollout 11
```

```
actor_index 56 initial policy_version 10 policy_version after rollout 11
actor_index 28 initial policy_version 10 policy_version after rollout 11
actor_index 4 initial policy_version 10 policy_version after rollout 11
actor_index 7 initial policy_version 10 policy_version after rollout 11
actor_index 65 initial policy_version 10 policy_version after rollout 11
actor_index 12 initial policy_version 11 policy_version after rollout 12
actor_index 14 initial policy_version 11 policy_version after rollout 12
actor_index 5 initial policy_version 11 policy_version after rollout 12
actor_index 3 initial policy_version 11 policy_version after rollout 12
actor_index 35 initial policy_version 11 policy_version after rollout 12
actor_index 51 initial policy_version 11 policy_version after rollout 12
actor_index 0 initial policy_version 11 policy_version after rollout 12
actor_index 6 initial policy_version 11 policy_version after rollout 12
actor_index 60 initial policy_version 12 policy_version after rollout 13
actor_index 77 initial policy_version 12 policy_version after rollout 13
actor_index 48 initial policy_version 12 policy_version after rollout 13

$ python -m torchbeast.monobeast_study \
--num_actors 40 \
--total_steps 10000000 \
--learning_rate 0.0006 \
--epsilon 0.01 \
--entropy_cost 0.01 \
--batch_size 8 \
--unroll_length 240 \
--num_threads 1 \
--env Pong-v5

actor_index 34 initial policy_version 12 policy_version after rollout 18
actor_index 25 initial policy_version 13 policy_version after rollout 18
actor_index 4 initial policy_version 13 policy_version after rollout 18
actor_index 5 initial policy_version 13 policy_version after rollout 18
actor_index 14 initial policy_version 13 policy_version after rollout 18
actor_index 16 initial policy_version 13 policy_version after rollout 18
actor_index 12 initial policy_version 13 policy_version after rollout 18
actor_index 39 initial policy_version 13 policy_version after rollout 18
actor_index 30 initial policy_version 13 policy_version after rollout 18
actor_index 18 initial policy_version 13 policy_version after rollout 18
actor_index 13 initial policy_version 13 policy_version after rollout 18
actor_index 23 initial policy_version 13 policy_version after rollout 19
actor_index 35 initial policy_version 13 policy_version after rollout 19
actor_index 3 initial policy_version 14 policy_version after rollout 19
actor_index 17 initial policy_version 14 policy_version after rollout 19
actor_index 9 initial policy_version 14 policy_version after rollout 19
actor_index 6 initial policy_version 14 policy_version after rollout 19
```