Cleanba: A Reproducible, Efficient, and Scalable Distributed Reinforcement Learning Platform

Abstract

Cleanba is a platform for reproducible, efficient, and scalable reinforcement learning research. Our implementation of the PPO algorithm is inspired by DeepMind's Sebulba architecture and includes several significant modifications to enhance reproducibility, accessibility, and scalability. Cleanba is designed with simplicity in mind, with each implementation being a standalone, easy-to-read single file that is about 800 lines of code. Despite its simplicity, Cleanba delivers outstanding performance, achieving a median human-normalized score of 184.41% across 57 Atari games in just about 30 minutes per game; this is accomplished using 8 A100 GPUs, resulting in one of the shortest training wall times compared to past works. Cleanba is also highly scalable and can scale to hundreds of GPUs with linear scaling, obtaining 93% of ideal scaling efficiency. To ensure reproducibility, we proposed a simple technique that synchronizes the actor and learner, enabling Cleanba to produce consistent results across different hardware configurations. Cleanba is open-source and available at https://github.com/vwxyzjn/cleanba.

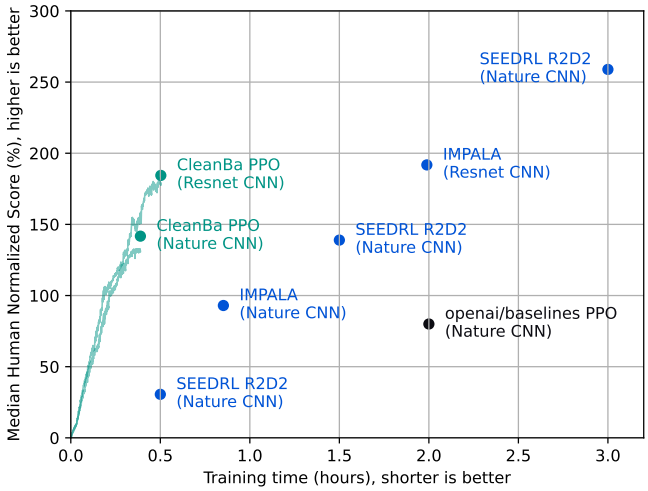

Figure 1: Atari-57 results. The median human-normalized scores and training times of past works. Data reproduced from [11] and Open RL Benchmark. It is worth noting that the learning curve for `Cleanba PPO (Nature CNN)` appears to be lower than the reported score of 141.73% in the figure's dot. This is due to subtle differences between the algorithms used to calculate the learning curve and the reported score, which we explain in detail in Appendix **??**.

Preprint. Under review.

Table 1: Performance of moolib compared to IMPALA [5].

	Architecture	Median HNS
IMPALA [5]	Nature CNN [17]	93.20%
`heiner/scalable_agent`'s IMPALA [5, 12]	Nature CNN [17]	47.64% (↓ 48.88%)
IMPALA [5]	Resnet CNN [5]	191.80%
Moolib's IMPALA [15]	Resnet CNN [5]	111.47% (↓ 41.88%)

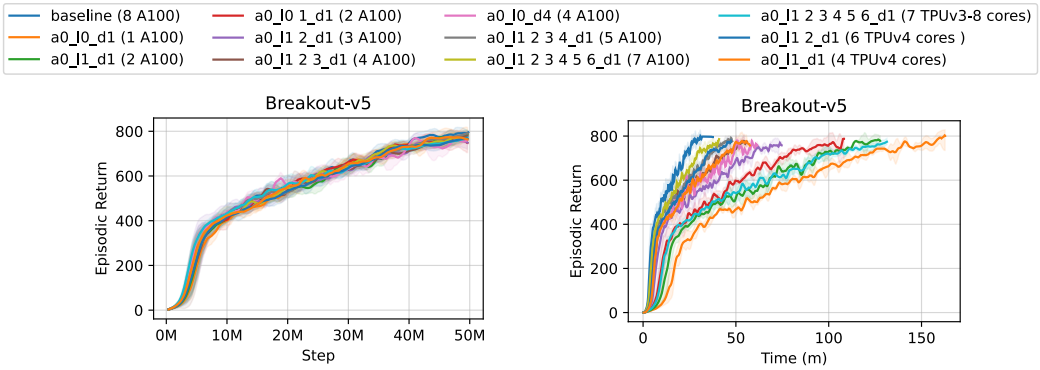

Figure 2: Reproducibility results for the `Cleanba PPO (Resnet CNN)` setting using different hardware configurations. The format `a0_l1 2 3 4_d1` means the actor runs on GPU 0 (`a0`), the learner runs on GPU 1 2 3 4 (`l1 2 3 4`), and the computation is replicated/distributed one time (`d1`). The blue baseline curve is taken from Figure 1, corresponding to `a0_l1 2 3_d2`.

1 Introduction

Deep Reinforcement Learning (DRL) is a paradigm to train autonomous agents to perform tasks, and in recent years, it has demonstrated remarkable success across various domains, including video games [18], robotics control [25], chip design [16], and large language model tuning [19]. Concurrent with the development of DRL is the rise of distributed DRL [5, 4], a fast-growing field that leverages more computing resources to train agents.

Despite recent accomplishments, reproducibility in distributed DRL remains a challenge. Most prominently, no third party has successfully reproduced IMPALA-level [5] performance in Atari using an open-source codebase. In our investigation, we compiled the results of popular open-source distributed RL implementation in Table 1. Interestingly, we found `heiner/scalable_agent` [12], a slightly-modified fork of the official IMPALA source code released in [5], to perform almost 48.88% worse than the reported IMPALA score. We also found Moolib to perform 41.88% worse than the reported IMPALA score of the corresponding setting. This clearly highlights a reproducibility issue in distributed DRL. As a result, the open-source community does not have a codebase that can reach 191.80% median HNS, which is a crucial challenge this work attempts to address.

A prominent issue with reproducibility in distributed DRL lies in the scarcity of third-party replications of reported results. For example, the IMPALA algorithm reported a median human-normalized score of 191% in Atari games, achieved after approximately two hours of training per game [5, 11, 4] (Appendix **??**). However, to the best of our knowledge, no third party has successfully reproduced this performance level using an open-source codebase. This reproducibility issue may due to various factors, such as insufficient experiment details, hardware-dependent design choices, and complex codebases. These challenges highlight the pressing need for improved standardization, documentation, and open-source codebases to bolster the reproducibility of distributed DRL research.

This work introduces Cleanba, a distributed Proximal Policy Optimization (PPO) [25] implementation inspired by the DeepMind's Sebulba architecture [8] with several important changes to enhance reproducibility, accessibility, and scalability. Here are the main highlights of Cleanba:

1. **Strong performance.** Cleanba achieves a median human-normalized score of 184.41% across 57 Atari games in approximately 30 minutes per game, using 8 A100 GPUs. This is a four-fold speedup in wall time compared to the IMPALA algorithm [5], as shown in Figure 1.
2. **Great reproducibility/accessibility.** Cleanba includes a simple mechanism to constrain the relative progress of the actor and learner, making it highly reproducible. Figure 2 demonstrates that near identical learning curves can be obtained with varying numbers of GPUs, where more GPUs only improve training speed.
3. **High scalability.** Cleanba supports the `jax.distributed` package, which enables computation to be scaled across high-performance clusters (HPC) and facilitates large batch-size training. Our experiments show that Cleanba can linearly scale to 128 A100 GPUs with 93% of ideal scaling efficiency, learning with a batch size of approximately 200K and running at approximately 1.6M frames per second in Atari, as demonstrated in Figure 5.
4. **Easy to understand.** Cleanba adopts the single-file implementation paradigm of CleanRL [10], with each implementation being standalone and easy to understand and modify (approximately 800 lines of code demonstrating end-to-end training).
5. **Transparent results.** To facilitate more transparency and reproducibility, we have made available our source code at https://github.com/vwxyzjn/cleanba, trained models at https://huggingface.co/cleanrl, and tracked experiment at https://wandb.ai/openrlbenchmark/cleanba.

2 Background

Let us consider the RL problem in a *Markov Decision Process (MDP)* [21], 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:

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

Assuming we parameterize the policy as π_θ, where θ is the policy parameters, the policy gradient algorithm would obtain the gradient of the expected discounted return w.r.t. θ [22, 27], where $\mathbf{Pr}$ denotes the probability of a random variable:

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \mathbb{E}_\tau \left[G(\tau) \right] = \nabla_\theta \mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \gamma^t r_t \right] = \mathbb{E}_\tau \left[\nabla_\theta \log \mathbf{Pr}(\tau) G(\tau) \right] \tag{2}$$
$$= \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta \left(a_t \mid s_t \right) \gamma^t G_t \right] \tag{3}$$

PPO [25] is a popular algorithm that proposes a clipped policy gradient objective to help avoid unstable updates [25, 23]:

$$J^{\text{CLIP}}(\pi_\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \min \left(\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} A^{\hat{\text{adv}}}_\pi(s_t, a_t), \text{clip} \left(\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta'}(a_t \mid s_t)}, 1 - \epsilon, 1 + \epsilon \right) A^{\hat{\text{adv}}}_\pi(s_t, a_t) \right) \right] \tag{4}$$

where $A^{\hat{\text{adv}}}_\pi$ is an advantage estimator called Generalized Advantage Estimator [24], ϵ is PPO's clipped coefficient, and θ_{old} is the policy parameters before the update. During the optimization phase, the agent also learns the value function and maximizes the policy's entropy, therefore optimizing the following joint objective:

$$J^{\text{JOINT}}(\theta) = J^{\text{CLIP}}(\pi_\theta) - c_1 J^{\text{VF}}(\theta) + c_2 S[\pi_\theta], \tag{5}$$

Algorithm 2 PPO's pseudocode, where GAE calculate the generalized advantage estimation[24], $clipped\_MSE$ does a clipped mean square error (see "Value Function Loss Clipping" in [9]), and S is the entropy function.

Initialize vectorized environment E containing N parallel environments
Initialize policy parameters θ_π
Initialize value function parameters θ_v
Initialize Adam optimizer O for θ_π and θ_v
Initialize next observation $s_{next} = E.reset()$
Initialize next done flag $d_{next} = [0, 0, ..., 0]$ # length N

for i = 0,1,2,..., I **do**
    (optional) **Anneal** learning rate α linearly to 0 with i
    Set $\mathcal{D} = (s, a, \log \pi(a|s), r, d, v)$ as tuple of 2D arrays

    # Rollout Phase:
    for t = 0,1,2,..., M **do**
        Cache $o_t = s_{next}$ and $d_t = d_{next}$
        Get $a_t \sim \pi(\cdot|s_t)$ and $v_t = v(s_t)$
        Step simulator: $s_{next}, r_t, d_{next} = E.step(a_t)$
        Let $\mathcal{D}.s[t] = s_t, \mathcal{D}.d[t] = d_t, \mathcal{D}.v[t] = v_t, \mathcal{D}.a[t] = a_t,$
            $\mathcal{D}.\log \pi(a|s)[t] = \log \pi(a_t|s_t), \mathcal{D}.r[t] = r_t$

    # Learning Phase:
    Estimate / Bootstrap next value $v_{next} = v(s_{next})$
    Let advantage $\hat{A_\pi^{\text{adv}}} = GAE(\mathcal{D}.r, \mathcal{D}.v, \mathcal{D}.d, v_{next}, d_{next}, \lambda)$
    Let $TD(\lambda)$ return $R = \hat{A_\pi^{\text{adv}}} + \mathcal{D}.v$
    Prepare the batch $\mathcal{B} = \mathcal{D}, \hat{A_\pi^{\text{adv}}}, R$ and flatten $\mathcal{B}$
    for $epoch$ = 0,1,2,..., K **do**
        for mini-batch $\mathcal{M}$ of size m in $\mathcal{B}$ **do**
            Normalize advantage $\mathcal{M}.\hat{A_\pi^{\text{adv}}} = \frac{\mathcal{M}.\hat{A_\pi^{\text{adv}}} - mean(\mathcal{M}.\hat{A_\pi^{\text{adv}}})}{std(\mathcal{M}.\hat{A_\pi^{\text{adv}}}) + 10^{-8}}$
            Let ratio $r = e^{\log \pi(\mathcal{M}.a|\mathcal{M}.s) - \mathcal{M}.\log \pi(a|s)}$
            Let $L^\pi = \min\left(r\mathcal{M}.\hat{A_\pi^{\text{adv}}}, \text{clip}\left(r, 1-\epsilon, 1+\epsilon\right)\mathcal{M}.\hat{A_\pi^{\text{adv}}}\right)$
            Let L^V = $clipped\_MSE(\mathcal{M}.R, v(\mathcal{M}.s))$
            Let $L^S = S[\pi(\mathcal{M}.s)]$
            Back-propagate 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 the optimizer O to initiate gradient descent

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[1]. For more detail on PPO's implementation, see [9]. Given this pseudocode, the following list unifies the nomenclature/terminology of PPO's key hyperparameters.

1. `world_size` is the number of instances of training processes; typically this is 1 (e.g., you have a single GPU).
2. `local_num_envs` is the number of N 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.
3. `num_steps` is the number of M steps in which the agent samples a batch of N actions and receives a batch of N next observations, rewards, and done flags from the simulator (see

[1] https://github.com/openai/baselines

line 13), where the done flags signal if the episodes are terminated or truncated. `num_steps` has many names, such as the "sampling horizon" [26] and "unroll length" [6].

4. `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 24).
    - `batch_size` = `world_size` $*$ `local_batch_size` is the aggregated batch size across all training instances.
5. `update_epochs` is the number of K epochs that the agent goes through the training data in $\mathcal{B}$ (see line 25).
6. `num_minibatches` is the number of mini-batches that PPO splits $\mathcal{B}$ into (see line 26).
7. `local_minibatch_size` is $m =$ `local_batch_size` / `num_minibatches`, the size of each mini-batch $\mathcal{M}$ (see line 26).
    - `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 13-18), 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 21-34).

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 34. **Option 2** is especially desirable when the users want to leverage more computational resources, such as GPUs.

Interestingly, 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.

2.1 Actor-learner Architecture

Note that the PPO in Algorithm 1 is synchronous, where the rollout phase needs to occur before the learning phase. This could leave the agent susceptible to several bottlenecks. For example, parallel environment steps can be a bottleneck [28] (line 15). Furthermore, if the network or the training data is large, the learning phase could be a bottleneck (line 21-34).

A popular architecture to alleviate these two bottlenecks is to decouple the actors and the learners [5, 11, 4]. The actors are responsible for collecting the policy's interactions with the simulators and putting the interactions through a queue. The learners will get data from the queue and learn from the interactions to produce new policies. By running the actors and the learners concurrently, they do not block each other and consequently improve the system's throughput compared to synchronous architecture such as Algorithm 1.

Nevertheless, DRL is brittle and can be sensitive to hyperparameters and implementation details [7, 3, 9]. While the actor-learner architecture is intuitive and appears highly effective, several subtle issues can arise during its implementation. For example, it is common to implement some form of synchronization mechanism between the actors and the learners, but this implementation detail is rarely discussed in the literature. If no such mechanism exists and if the actors could accumulate data faster than the learner could consume, the size of the shared interaction data queue would simply gradually grow until it exhausts the memory. In `torchbeast` [12], by manually slowing down learning, we observed that its data queue has an implicit maximum size of 112 trajectories (Appendix **??**).

3 Cleanba

Modeled after DeepMind's Sebulba Podracer architecture [8], Cleanba (meaning CleanRL's Sebulba) includes a few different design choices to make distributed RL more reproducible and transparent to use. Cleanba is also highly efficient. It uses JAX [2] and EnvPool [28], both of which are designed to be efficient. To improve the actor's throughput, we employed the double buffer approach [26] – inference actions while the environment executes through EnvPool's Async API. To improve the learner's throughput, we allow using multiple learner devices via `pmap`. To improve the system's scalability, we use `jax.distibuted` to distribute to more nodes.

3.1 Reproducible Concurrent Architecture

At its core, Cleanba proposes a simple mechanism for synchronizing the actor and learner, ensuring the actor's policy version is **always exactly one step** behind the learner's policy version. Below is the pseudocode of the architecture, where the `rollout` function corresponds to the rollout phase (line 13-18) and `agent.learn` corresponds to the learning phase (line 21-34) in Algorithm 1.

```
def actor():
    for iteration in range(1, num_iterations):
        if iteration != 2:
            params = param_queue.get()
        data = rollout(params)
        rollout_queue.put(data)
def learner():
    for iteration in range(1, num_iterations):
        data = rollout_queue.get()
        agent.learn(data)
        param_queue.put(agent.param)
rollout_queue = Queue.queue(len=1)
param_queue = Queue.queue(len=1)
agent = Agent()
param_queue.put(agent.param)
threading.thread(actor).start()
threading.thread(learner).start()
```

Let us use π_i to denote the policy of version i, $\mathcal{D}_i$ the rollout data of version i. In the second iteration, we skipped the `param_queue.get()` call, so $\pi_1 \rightarrow \mathcal{D}_2$ happens concurrently with $\mathcal{D}_1 \rightarrow \pi_2$. 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 made sure the actor's policy version is exactly 1 version preceding the learner's policy version. The following table demonstrates this process:

Iteration	1	2	3
Actor	$\pi_1 \to \mathcal{D}_1$	$\pi_1 \to \mathcal{D}_2$	$\pi_2 \to \mathcal{D}_3$
Learner		$\mathcal{D}_1 \to \pi_2$	$\mathcal{D}_2 \to \pi_3$

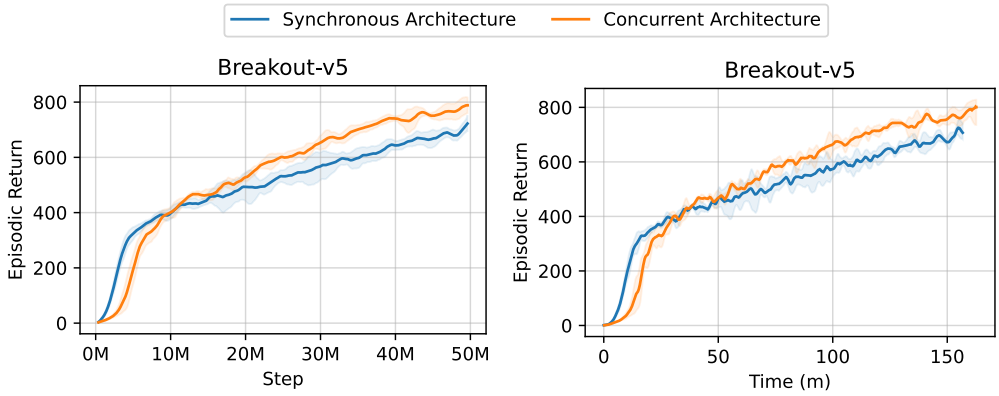

Figure 3: Ablation study comparing the concurrent architecture with the synchronous architecture used in Algorithm 1. Note that the concurrent architecture is slightly slower due to minor overheads.

The concurrent training loop above has the following benefits:

1. **Easy to reason & reproducible:** As demonstrated in the table, we know which policy is used for collecting the rollout data. We also know precisely that the size of the data is `local_batch_size`. Having a clear understanding of which policy is used to generate the rollout data improves the transparency and reproducibility of distributed RL.
2. **Easy to debug throughput:** To diagnose throughput, we can measure the time it takes for `rollout_queue.get()` `param_queue.get()`. If `rollout_queue.get()` on average takes a shorter time than `param_queue.get()`, it is clear that the learning is the bottleneck and vice versa.

It is crucial to recognize the fundamental algorithmic differences between the concurrent and synchronous architectures. Specifically, the synchronous architecture consistently collects rollout data for the most recent policy. To illustrate, its training process can be represented as follows:

$\pi_1 \to \mathcal{D}_1, \mathcal{D}_1 \to \pi_2, \pi_2 \to \mathcal{D}_2, \mathcal{D}_2 \to \pi_3, \pi_3 \to \mathcal{D}_3$.

In contrast, the concurrent architecture collects rollout data for the second most recent policy, so it is likely to have different performance characteristics.

We conducted an ablation study to compare the concurrent and synchronous architectures, as shown in Figure 3. We observed that the synchronous architecture is more sample-efficient during the initial 10M steps. This intuitively makes sense, as the agent continually learns from data generated by its most recent policy.

However, we observed that the synchronous architecture performs worse than the concurrent architecture between 10M and 50M steps, which is surprising. This suggests that the concurrent architecture effectively functions as a *different hyperparameter setting*.

3.2 Multi-buffered Sampling

To enhance the actor's throughput, we employ a method known as Multi-buffered Sampling. This technique enables us to sample actions for a subset of environments while the remaining environments continue stepping.

We accomplish this by utilizing an asynchronous environment stepping interface, such as the Async API provided by EnvPool [28]. This interface is a generalization of the double-buffered sampling approach found in Sample Factory [20].

In EnvPool's async mode, we define an `async_batch_size` to represent the number of environments included in a batch for the agent to sample actions, while the other environments step.

Below is an example of using EnvPool's Async API. When the agent calls `env.recv()`, it receives a batch of three observations (`obs`), rewards (`rew`), done flags (`done`), and an information dictionary (`info`). The `info["env_id"]` contains three environment IDs, indicating the source of the three observations. The agent then samples actions and sends them to EnvPool, along with their corresponding environment IDs. During the action sampling process, the remaining nine environments step independently without being affected by Python's Global Interpreter Lock (GIL).

```
import numpy as np
import envpool
num_envs = 9
async_batch_size = 3
env = envpool.make("Pong-v5", env_type="gym", num_envs=num_envs,
            batch_size=async_batch_size)
env.async_reset()
while True:
    obs, rew, done, info = env.recv()
    env_id = info["env_id"]
    action = np.random.randint(async_batch_size, size=len(env_id))
    env.send(action, env_id)
```

To understand the effect of multi-buffered sampling, we perform an ablation study on the `async_batch_size` in Figure 4. We found the asynchronous settings to be faster than the synchronous setting, but the speed-up is not like the 2x speed improvement shown in the EnvPool's paper [28]. This is likely because model sampling also plays a role. While EnvPool's async mode is faster, the model sampling is slower because the batch size for action sampling is lower, resulting in a slower throughput. As a result, deciding the proper EnvPool's batch size requires some empirical testing and is specific to the particular environment.

To evaluate the impact of multi-buffered sampling, we conducted an ablation study focused on the `async_batch_size`, as illustrated in Figure 4. Our findings revealed that asynchronous settings are faster than synchronous setting. This may appear surprising because [28] reported a nearly 2x speed enhancement reported, but the discrepancy is likely attributable to the role of model sampling.

Although EnvPool's asynchronous mode offers increased environment stepping speed, the model sampling process is slower due to the reduced batch size for action sampling. Consequently, determining the optimal EnvPool's batch size requires empirical testing and is dependent on the specific environment in question.

4 Empirical Evaluation

We test out Cleanba's PPO on 57 Atari games [1], using the environment interface provided by EnvPool [28]. To compare our work with past works, we used the same preprocessing techniques used in IMPALA [5]. Most notably, the preprocessing techniques do not use sticky action [13] and use a terminal signal for loss of life, 30 NOOP actions at the start of the episode, and the game-specific minimal action sets. We run the experiments for 200M frames with three random seeds.

For the neural network architecture, we perform two sets of experiments. One uses the Resnet CNN used in IMPALA (deep) [5], and the other uses the Nature CNN used in IMPALA (shallow) [5]. In both experiments, we did not use LSTM.

The median human-normalized score can be found in Figure 1, and the learning curves for each individual can be found in Figure **??** and **??**. A summary of the hardware usage can be found in Table 2.

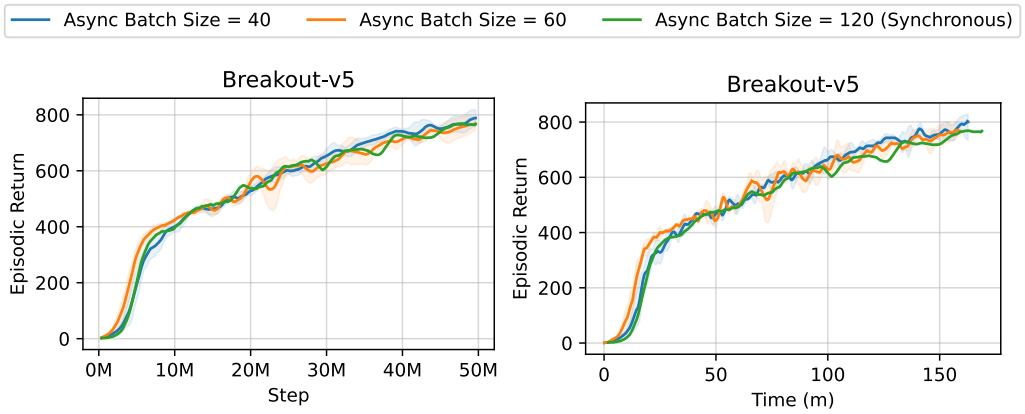

Figure 4: Ablation study comparing the different EnvPool's batch size. The `num_envs` is set to 120, so an `async_batch_size=120` would make EnvPool completely synchronous.

Table 2: The experiments' hardware specifications and the corresponding outcomes.

	Accelerators	CPUs	median HNS	runtime (minutes)
Cleanba PPO (Resnet CNN)	8 A100	50	**184.41%**	**30.16**
w/ Machado Atari wrappers [13]	8 A100	50	142.73%	roughly 30
Cleanba PPO (Nature CNN)	4 A100	50	**141.73%**	**23.32**
IMPALA (Resnet CNN) [5]	1 P100	unknown	**191.80%**	51.13
IMPALA (Nature CNN) [5]	1 P100	unknown	93.00%	119.31
SEED RL's R2D2 [4]	8 TPUv3	216	30.58%	30
SEED RL's R2D2 [4]	8 TPUv3	216	138.96%	90
SEED RL's R2D2 [4]	8 TPUv3	216	**258.90%**	180
`openai/baselines` PPO	1 RTX 3060 Ti	8	79.59%	134.37

4.1 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. We conduct experiments with 16, 32, 64, and 128 A100 GPUs. For convenience, we also adjust a few settings:

- Turn off the learning rate annealing.
- Run for 100M steps instead of the standard 50M steps shown in Figure 1.
- Keep doubling the `num_envs`, `batch_size`, and `minibatch_size` with larger number of GPUs.

Due to hardware scheduling constraints, we only ran the experiments for 1 random seed. The results are shown in Figure 5 and 6. 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.

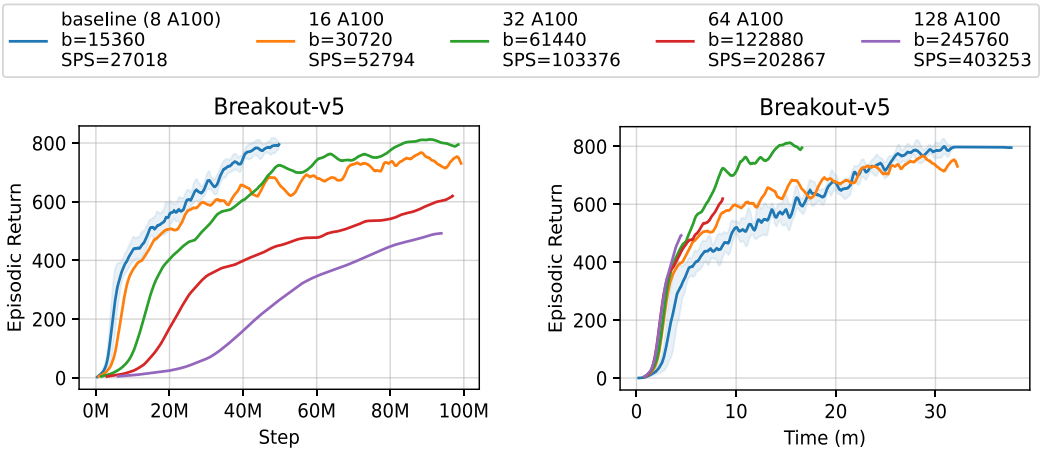

Figure 5: Cleanba's results from large batch size training. The blue baseline is taken directly from the experiments presented in Figure 1. The `b=15360` denotes the `batch_size=15360`. Note that the baseline experiments only took ∼31 minutes of training time[2].

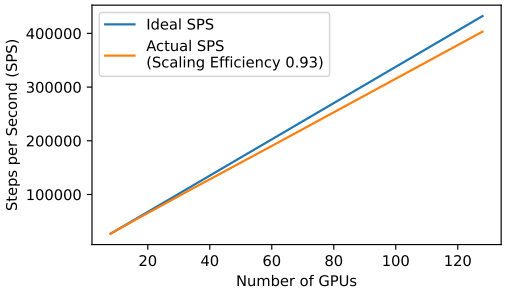

Figure 6: Cleanba's SPS scaling results from large batch size training.

- **Large batch sizes train more quickly.** Like [14], 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 data is likely going to be more diverse and have a higher *gradient noise scale* [14], which would help the agent utilize large batch sizes more efficiently, resulting in a reduced decline in sample efficiency.

5 Conclusion

In this paper, we have presented Cleanba, a distributed deep reinforcement learning platform that prioritizes reproducibility, efficiency, and scalability. Our Atari experiments demonstrate that Cleanba achieves performance comparable to that of IMPALA, but with four times less wall time. Furthermore, our reproducibility experiments show that Cleanba can be reliably reproduced across different hardware configurations, and our scalability experiments demonstrate that Cleanba can leverage hundreds of GPUs to train agents efficiently on HPC clusters. Despite these advanced features, the code required to implement Cleanba is only approximately 800 lines, making it easy to read, comprehend, and modify. We believe that Cleanba will be a valuable platform for the research

[2]The runs took extra ∼9 minutes to run evaluations and model uploading; the communication of the last training metric data point was delayed due to how Weights and Biases synchronize metric, which causes the "flat line" shown in the right figure in Figure 5.

community to conduct open-source RL research and make progress towards solving complex real-world problems.

Acknowledgement

We thank Stability AI for generously providing the GPU computational resources to this project. We also thank Google's TPU research cloud for providing the TPU computational resources.

References

[1] 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, 2013.

[2] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, et al. Jax: composable transformations of python+ numpy programs. 2018.

[3] L. Engstrom, A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry. Implementation matters in deep rl: A case study on ppo and trpo. In *International Conference on Learning Representations*, 2020.

[4] L. Espeholt, R. Marinier, P. Stanczyk, K. Wang, and M. Michalski. Seed rl: Scalable and efficient deep-rl with accelerated central inference. In *International Conference on Learning Representations*, 2020.

[5] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. In J. G. Dy and A. Krause, editors, *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*, pages 1406–1415. PMLR, 2018.

[6] C. D. Freeman, E. Frey, A. Raichuk, S. Girgin, I. Mordatch, and O. Bachem. Brax–a differentiable physics engine for large scale rigid body simulation. *arXiv preprint arXiv:2106.13281*, 2021.

[7] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[8] M. Hessel, M. Kroiss, A. Clark, I. Kemaev, J. Quan, T. Keck, F. Viola, and H. van Hasselt. Podracer architectures for scalable reinforcement learning. *arXiv preprint arXiv:2104.06272*, 2021.

[9] S. Huang, R. F. J. Dossa, A. Raffin, A. Kanervisto, and W. Wang. The 37 implementation details of proximal policy optimization. In *ICLR Blog Track*, 2022.

[10] S. Huang, R. F. J. Dossa, C. Ye, J. Braga, D. Chakraborty, K. Mehta, and J. G. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022.

[11] S. Kapturowski, G. Ostrovski, J. Quan, R. Munos, and W. Dabney. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*, 2019.

[12] H. Küttler, N. Nardelli, T. Lavril, M. Selvatici, V. Sivakumar, T. Rocktäschel, and E. Grefenstette. Torchbeast: A pytorch platform for distributed rl. *arXiv preprint arXiv:1910.03552*, 2019.

[13] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

[14] S. McCandlish, J. Kaplan, D. Amodei, and O. D. Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

[15] V. Mella, E. Hambro, D. Rothermel, and H. Küttler. moolib: A Platform for Distributed RL. 2022.

[16] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *ArXiv preprint*, abs/1312.5602, 2013.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[19] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022.

[20] A. Petrenko, Z. Huang, T. Kumar, G. Sukhatme, and V. 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.

[21] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[22] J. Schulman. *Optimizing Expectations: From Deep Reinforcement Learning to Stochastic Computation Graphs*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2016.

[23] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz. Trust region policy optimization. In F. R. Bach and D. M. Blei, editors, *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*, pages 1889–1897. JMLR.org, 2015.

[24] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. In Y. Bengio and Y. LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[25] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *ArXiv preprint*, abs/1707.06347, 2017.

[26] A. Stooke and P. Abbeel. Accelerated methods for deep reinforcement learning. *arXiv preprint arXiv:1803.02811*, 2018.

[27] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[28] J. Weng, M. Lin, S. Huang, B. Liu, D. Makoviichuk, V. Makoviychuk, Z. Liu, Y. Song, T. Luo, Y. Jiang, Z. Xu, and S. YAN. Envpool: A highly parallel reinforcement learning environment execution engine. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022.