A Usage of EnvPool

In this section, we include comprehensive examples of the Python user APIs for EnvPool usage, including both synchronous and asynchronous execution modes, and for both OpenAI `gym` and `dm_env` APIs.

A.1 Synchronous Execution, OpenAI `gym` APIs

```
import numpy as np
import envpool

# make gym env
env = envpool.make("Pong-v5", env_type="gym", num_envs=100)
obs = env.reset()  # with shape (100, 4, 84, 84)
act = np.zeros(100, dtype=int)
obs, rew, done, info = env.step(act, env_id=np.arange(100))
# env_id = info["env_id"]
```

A.2 Synchronous Execution, DeepMind `dm_env` APIs

```
import numpy as np
import envpool

# make dm_env
env = envpool.make("Pong-v5", env_type="dm", num_envs=100)
obs = env.reset().observation.obs  # with shape (100, 4, 84, 84)
act = np.zeros(100, dtype=int)
timestep = env.step(act, env_id=np.arange(100))
# timestep.observation.obs, timestep.observation.env_id,
# timestep.reward, timestep.discount, timestep.step_type
```

A.3 Asynchronous Execution

For maximizing the throughput of the environment execution, users may use the asynchronous execution mode. Both typical `step` API and more low-level APIs `recv`, `send` are provided.

```
import numpy as np
import envpool

# async by original API
env = envpool.make_dm("Pong-v5", num_envs=10, batch_size=9)
action_num = env.action_spec().num_values
timestep = env.reset()
env_id = timestep.observation.env_id
while True:
    action = np.random.randint(action_num, size=len(env_id))
    timestep = env.step(action, env_id)
    env_id = timestep.observation.env_id
```

```
# or use low-level API, faster than previous version
env = envpool.make_dm("Pong-v5", num_envs=10, batch_size=9)
action_num = env.action_spec().num_values
env.async_reset()  # this can only call once at the beginning
while True:
    timestep = env.recv()
    env_id = timestep.observation.env_id
    action = np.random.randint(action_num, size=len(env_id))
    env.send(action, env_id)
```

```
import numpy as np
import envpool

# make asynchronous with gym API
num_envs = 10
batch_size = 9
env = envpool.make("Pong-v5", env_type="gym", num_envs=num_envs,
                   batch_size=batch_size)
env.async_reset()
while True:
    obs, rew, done, info = env.recv()
    env_id = info["env_id"]
    action = np.random.randint(batch_size, size=len(env_id))
    env.send(action, env_id)
```

B CPU Specifications for Pure Environment Simulation

This section lists the detailed CPU specifications for the pure environment simulation experiments presented in the main paper.

The laptop has 12 Intel CPU cores, with Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz. And the workstation has 32 AMD CPU cores, with AMD Ryzen 9 5950X 16-Core Processor. Evaluating EnvPool on these two configurations can demonstrate its effectiveness with small-scale experiments.

An NVIDIA DGX-A100 has 256 CPU cores with AMD EPYC 7742 64-Core Processor and 8 NUMA nodes. Note that running multi-processing on each NUMA node not only makes the memory closer to the processor but also reduces the thread contention on the `ActionBufferQueue`.

C Speed Improvements on Single Environment

We present experiments with a single environment (i.e., $N = 1$) in Table 2, where EnvPool manages to reduce overhead compared to the Python counterpart and achieves considerable speedup.

D ActionBufferQueue and StateBufferQueue

D.1 ActionBufferQueue

`ActionBufferQueue` is the queue that caches the actions from the `send` function, waiting to be consumed by the `ThreadPool`. Many open-source general-purpose thread-safe event queues can be used for this purpose. In this work, we observe that in our case the total number of environments N, the `batch_size` M, and the number of threads are all pre-determined at the construction of EnvPool. The `ActionBufferQueue` can thus be tailored for our specific case for optimal performance.

Table 2: Single environment simulation speed on different hardware setups. The speed is in frames per second.

System	Method	Atari Pong-v5	MuJoCo Ant-v3	dm_control cheetah run
Laptop	Python	4,891	12,325	6,235
Laptop	EnvPool	7,887	15,641	11,636
Laptop	Speedup	1.61×	1.27×	1.87×
Workstation	Python	7,739	19,472	9,042
Workstation	EnvPool	12,623	25,725	16,691
Workstation	Speedup	1.63×	1.32×	1.85×
DGX-A100	Python	4,449	11,018	5,024
DGX-A100	EnvPool	7,723	16,024	10,415
DGX-A100	Speedup	1.74×	1.45×	2.07×

We implemented `ActionBufferQueue` with a lock-free circular buffer. A buffer with a size of $2N$ is allocated. We use two atomic counters to keep track of the head and tail of the queue. The counters modulo $2N$ is used as the indices to make the buffer circular. We use a `semaphore` to coordinate enqueue and dequeue operations and to make the threads wait when there is no action in the queue.

D.2 StateBufferQueue

`StateBufferQueue` is in charge of receiving data produced by each environment. Like the `ActionBufferQueue`, it is also tailored exactly for RL environments. `StateBufferQueue` is a lock-free circular buffer of memory blocks, each block contains a fixed number of slots equal to `batch_size`, where each slot is for storing data generated by a single environment.

When one environment finishes its step inside `ThreadPool`, the corresponding thread will acquire a slot in `StateBufferQueue` to write the data. When all slots are written, a block is marked as ready (see yellow slots in Figure 1). By pre-allocating memory blocks, each block in `StateBufferQueue` can accommodate a batch of states. Environments will use slots of the pre-allocated space in a first come first serve manner. When a block is full, it can be directly taken as a batch of data, saving the overhead for batching. Both the allocation position and the write count of a block are tracked by atomic counters. When a block is ready, it is notified via a semaphore. Therefore the `StateBufferQueue` is also lock-free and highly performant.

Data Movement The popular Python vectorized environment executor performs memory copies at several places that are saved in EnvPool. There is one inter-process copy for collecting the states from the worker processes, and one copy for batching the collected states. In EnvPool, these copies are saved thanks to the `StateBufferQueue` because:

- We pre-allocate memory for a batch of states, the pointer to the target slot of memory is directly passed to the environment execution and written from the worker thread.
- The ownership of the block of memory is directly transferred to Python and converted into NumPy arrays via pybind11 when the block of memory is marked as ready.

E Jitting for JAX

Jitting the environment simulation code with the neural networks is supported via a set of jittable functions in EnvPool:

```
import envpool
import jax.lax as lax

env = envpool.make(..., env_type="gym" | "dm")
handle, recv, send, step = env.xla()

def actor_step(iter, loop_var):
  handle0, states = loop_var
  action = policy(states)
  # for gym
  handle1, (new_states, rew, done, info) = step(handle0, action)
  # for dm
  # handle1, new_states = step(handle0, action)
  return (handle1, new_states)

@jit
def run_actor_loop(num_steps, init_var):
  return lax.fori_loop(0, num_steps, actor_step, init_var)

states = env.reset()
run_actor_loop(100, (handle, states))
```

Currently, EnvPool supports jitting for CPU and GPU when used with JAX [3]. All jittable functions are implemented via XLA's custom call mechanism [8]. When the environment code is jitted, the control loop of the actor is lowered from Python code to XLA's runtime, allowing the entire control loop to run on a native thread and freeing the Python Global Interpreter Lock (GIL). Note that when the EnvPool function is jitted for GPU, the environment simulation is still executed on the CPU, as EnvPool only supports existing environments written in C/C++. We implemented GPU jitting by wrapping the CPU code in a GPU custom call, with memory transfers between two devices.

F Complete Results of End-to-end Agent Training

F.1 CleanRL Training Results

This section presents the complete training results using CleanRL's PPO and EnvPool. CleanRL's PPO closely matches the performance and implementation details of `openai/baselines`' PPO [13]. The source code is made available publicly[3]. The hardware specifications for conducting the CleanRL's experiments are as follows:

- OS: Pop!_OS 21.10 x86_64
- Kernel: 5.17.5-76051705-generic
- CPU: AMD Ryzen 9 3900X (24) @ 3.800GHz
- GPU: NVIDIA GeForce RTX 2060 Rev. A
- Memory: 64237MiB

CleanRL's Atari experiment's hyperparameters and learning curves can be found in Table 3 and Figure 7. CleanRL's MuJoCo experiment's hyperparameters and learning curves can be found in Table 5 and Figure 8. CleanRL's tuned Pong experiment's hyperparameters can be found in Table 4.

[3] See https://github.com/vwxyzjn/envpool-cleanrl

Note the CleanRL's EnvPool experiments with MuJoCo use the `v4` environments and the gym's vecenv experiments use the `v2` environments. There are subtle differences between the `v2` and `v4` environments[4].

Table 3: PPO hyperparameters used for CleanRL's Atari experiments (i.e., `ppo_atari.py` and `ppo_atari_envpool.py`). The hyperparameters used is aligned with [28].

Parameter Names	Parameter Values
N_{total} Total Time Steps	10,000,000
α Learning Rate	0.00025 Linearly Decreased to 0
N_{envs} Number of Environments	8
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 [5]	True

Table 4: PPO *tuned* hyperparameters used for CleanRL's Pong experiments in Figure 6.

Parameter Names	Parameter Values
N_{total} Total Time Steps	7,000,000
α Learning Rate	0.002
N_{envs} Number of Environments	64
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)	2.24
c_2 (Entropy Coefficient)	0.0
ω (Gradient Norm Threshold)	1.13
Value Function Loss Clipping	False

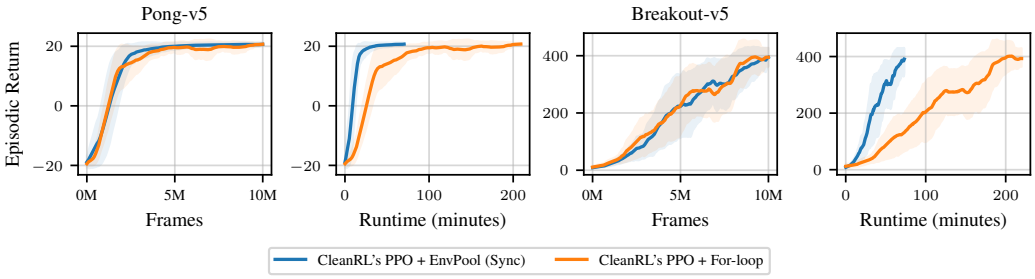

Figure 7: CleanRL example runs with Python vectorized environments and with EnvPool, using the same number of parallel environments $N = 8$.

[4] See https://github.com/openai/gym/pull/2762#issuecomment-1135362092

[5] See "Value Function Loss Clipping" in [13]

Table 5: PPO hyperparameters used for CleanRL's MuJoCo experiments (i.e., `ppo_continuous_action.py` and `ppo_continuous_action_envpool.py`). Note that [28] uses $N_{\text{envs}} = 1$ so we needed to find an alternative set of hyperparameters.

Parameter Names	Parameter Values
N_{total} Total Time Steps	10,000,000
α Learning Rate	0.00295 Linearly Decreased to 0
N_{envs} Number of Environments	64
N_{steps} Number of Steps per Environment	64
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)	2
ε (PPO's Clipping Coefficient)	0.2
c_1 (Value Function Coefficient)	1.3
c_2 (Entropy Coefficient)	0.0
ω (Gradient Norm Threshold)	3.5
Value Function Loss Clipping	False

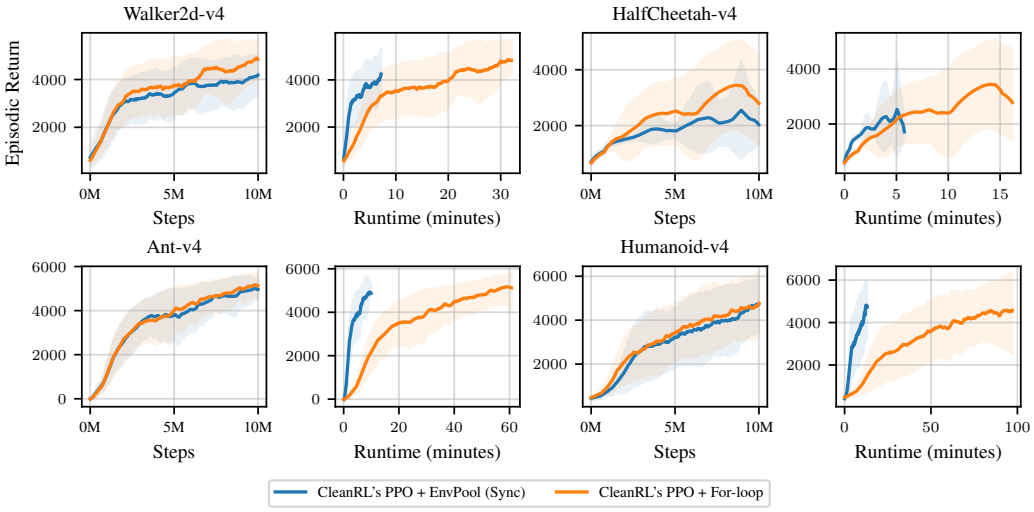

Figure 8: CleanRL example runs with Python vectorized environments and with EnvPool, using the same number of parallel environments $N = 64$.

F.2 rl_games Training Results

This section presents the complete training results using rl_games' PPO and EnvPool. The hyperparameters configuration can be found in rl_games' repository.[6] The hardware specifications for conducting the rl_games' experiments are as follows:

- OS: Ubuntu 21.10 x86_64
- Kernel: 5.13.0-48-generic
- CPU: 11th Gen Intel i9-11980HK (16) @ 4.900GHz
- GPU: NVIDIA GeForce RTX 3080 Mobile / Max-Q 8GB/16GB
- Memory: 64,012MiB

The commit used to run the experiments is `7f259a6436f396274c9931d0bd7004cee2ecabfa`, and the hyperparameters are tuned per environment and are available at

- Ant: rl_games/configs/MuJoCo/ant_envpool.yaml
- Walker2D: rl_games/configs/MuJoCo/walker2d_envpool.yaml
- HalfCheetah: rl_games/configs/MuJoCo/halfcheetah_envpool.yaml
- Humanoid rl_games/configs/MuJoCo/humanoid_envpool.yaml
- Breakout rl_games/configs/atari/ppo_breakout_envpool.yaml
- Pong rl_games/configs/atari/ppo_pong_envpool.yaml

We compare the training performance of rl_games using EnvPool against using Ray [23]'s parallel environments. In Figure 10, it's observed that EnvPool can boost the training system with multiple times training speed compared to Ray's integration.

For example using well-tuned hyperparameters we can train Atari Pong game in under 2 min to 18+ training and 20+ evaluation score on a laptop.

Well-established implementations of SAC [14, 36] can only train Humanoid to a score of 5,000 in three to four hours. It's worth highlighting that we can now train Humanoid to a score over 10,000 just in 15 minutes with a laptop.

rl_games's Atari experiment's learning curves can be found in and Figure 9. rl_games's MuJoCo experiment's learning curves can be found in Figure 10. Table 6, Table 7 and Table 8 are the hyperparameters.

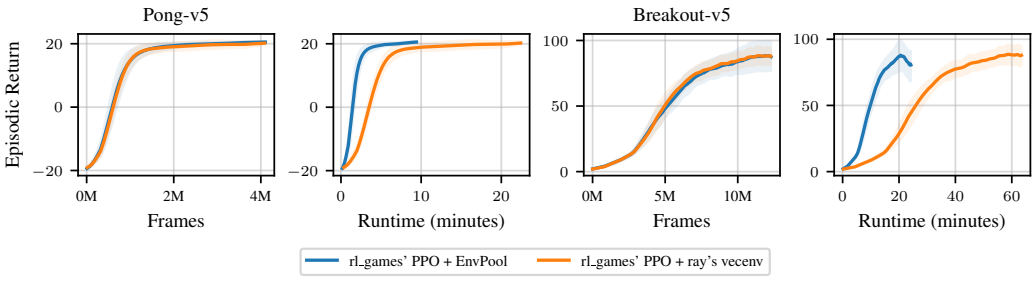

Figure 9: rl_games example runs with Ray environments and with EnvPool, using the same number of parallel environments $N = 64$.

[6] See the files postfixed with `envpool` in https://github.com/Denys88/rl_games/tree/master/rl_games/configs/atari.

Table 6: PPO baseline hyperparameters used for rl_games's MuJoCo experiments. Some environments use different neural network architectures.

Parameter Names	Parameter Values
N_{envs} Number of Environments	64
N_{steps} Number of Steps per Environment	256 for HalfCheetah 64 for Ant 128 for Humanoid and Walker2D
γ (Discount Factor)	0.99
λ (for GAE)	0.95
N_{mb} Number of Mini-batches	2
K (Number of PPO Update Iteration Per Epoch)	4 for Ant 5 for HalfCheetah, Walker2D and Humanoid
ε (PPO's Clipping Coefficient)	0.2
c_1 (Value Function Coefficient)	2.0
c_2 (Entropy Coefficient)	0.0
ω (Gradient Norm Threshold)	1.0
α Learning Rate	0.0003 dynamically adapted based on ν
ν KL Divergence threshold (for α)	0.008
Value Function Loss Clipping	True
Value Bootstrap on Terminal States	True
Reward Scale	0.1
Smooth Ratio Clamp	True
Observation Normalization	True
Value Normalization	True
MLP Sizes	$[128, 64, 32]$ for HalfCheetah $[256, 128, 64]$ for Ant and Walker2D $[512, 256, 128]$ for Humanoid
MLP Activation	Elu
Shared actor critic network	True
MLP Layer Initializer	Xavier

Table 7: PPO hyperparameters used for rl_games's Atari Breakout experiments.

Parameter Names	Parameter Values
N_{envs} Number of Environments	64
N_{steps} Number of Steps per Environment	128
γ (Discount Factor)	0.999
λ (for GAE)	0.95
N_{mb} Number of Mini-batches	4
K (Number of PPO Update Iteration Per Epoch)	2
ε (PPO's Clipping Coefficient)	0.2
c_1 (Value Function Coefficient)	2.0
c_2 (Entropy Coefficient)	0.01
ω (Gradient Norm Threshold)	1.0
α Learning Rate	0.0008
Value Function Loss Clipping	False
Observation Normalization	False
Value Normalization	True
Neural network	Nature CNN
Activation	ReLU
Shared actor critic network	True
Layer Initializer	Orthogonal

Table 8: PPO hyperparameters used for rl_games's Atari Pong experiments.

Parameter Names	Parameter Values
N_{total} Total Time Steps	8,000,000
N_{envs} Number of Environments	64
N_{steps} Number of Steps per Environment	128
γ (Discount Factor)	0.999
λ (for GAE)	0.95
N_{mb} Number of Mini-batches	8
K (Number of PPO Update Iteration Per Epoch)	2
ε (PPO's Clipping Coefficient)	0.2
c_1 (Value Function Coefficient)	2.0
c_2 (Entropy Coefficient)	0.01
ω (Gradient Norm Threshold)	1.0
α Learning Rate	0.0003 dynamically adapted based on ν
ν KL Divergence threshold (for α)	0.01
Value Function Loss Clipping	True
Observation Normalization	True
Value Normalization	True
Neural network	Nature CNN
Activation	Elu
Shared actor critic network	True
Layer Initializer	Orthogonal

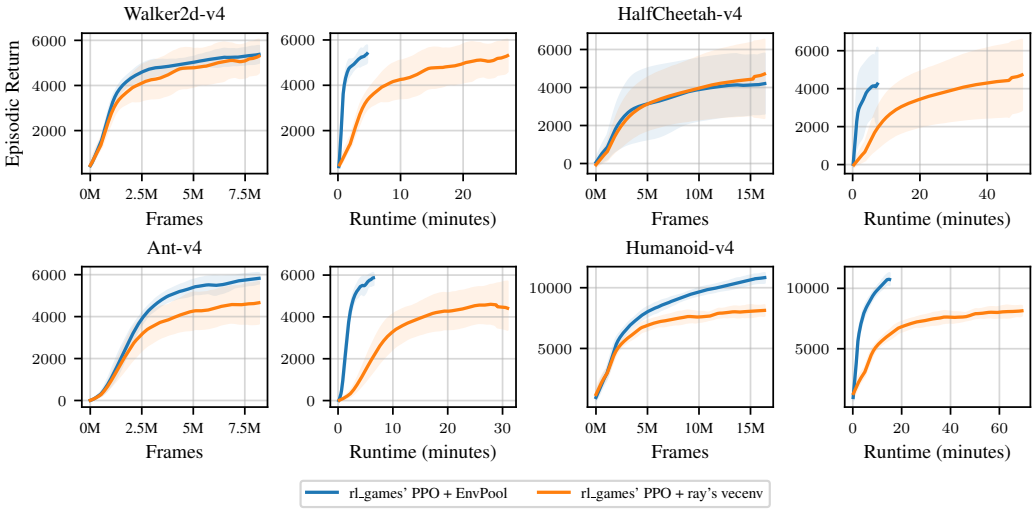

Figure 10: rl_games example runs with Ray environments and with EnvPool, using the same number of parallel environments $N = 64$.

F.3 Acme-based Training Results

We integrate EnvPool with Acme [12] for experiments of PPO [28] in MuJoCo tasks, to show EnvPool's efficiency with different `num_envs` and its advantage over other vectorized environments such as Stable Baseline's *DummyVecEnv* [26]. The codes and hyperparameters can be found in our open-sourced codebase.

All the experiments were performed on a standard TPUv3-8 machine on Google Cloud with the following hardware specifications:

- OS: Ubuntu 20.04 x86_64
- Kernel: 5.4.0-1043-gcp
- CPU: Intel(R) Xeon(R) CPU @ 2.00GHz
- TPU: v3-8 with v2-alpha software
- Memory: 342,605MiB

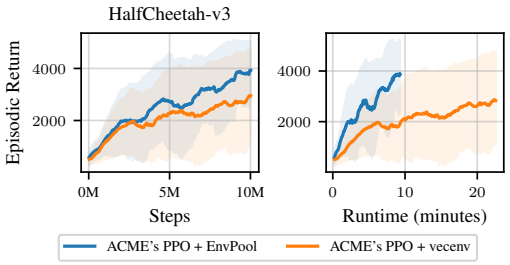

Figure 11: Comparison of EnvPool and DummyVecEnv using Acme's PPO implementation on MuJoCo HalfCheetah-v3 environment. Both settings use `num_envs` of 32.

In Figure 11, we compare EnvPool with another popular batch environment *DummyVecEnv*, which is argued to be more efficient than its alternative *SubprocVecEnv* for light environments, both provided by Stable Baseline 3 [26]. Using the same number of environments, EnvPool spent less than half of the wall time of *DummyVecEnv* to achieve a similar final episode return, proving our efficiency.

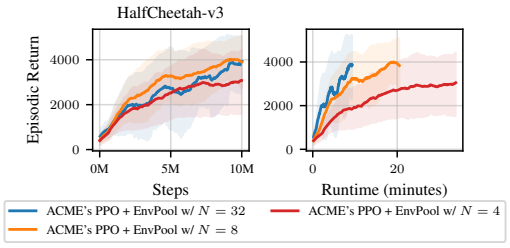

Figure 12: Training curves of Acme's PPO implementation on MuJoCo HalfCheetah-v3 environment using EnvPool of a different number of parallel environments.

Figure 12 shows that under the same environment interaction budget, tuning the `num_envs` can greatly reduce the training time while maintaining similar sample efficiency. We note that the key to maintaining the sample efficiency is to keep the same amount of environment steps under the same set of policy parameters. In our case, we simply maintain `num_envs` $\times$ `batch_size` a constant.

G Hyper-parameters Tuning

The motivation to find a set of hyperparameters is well-explained in Section 4.2 — the idea is to use a large N such as 32 or 64 to better leverage EnvPool's throughput and use less stale data. This section explains the process of tuning hyperparameters in this paper.

Specifically, regarding the CleanRL + Atari experiments in Figure 4, we have used the same hyperparameters as in the original PPO paper [28, Table 5]. CleanRL's hyperparameters in Figure 8 were tuned via Weights and Biases' automated hyperparameters search that optimizes average normalized scores in HalfCheetah-v4, Walker2d-v4, and Ant-v4 for 3 random seeds using Bayesian optimization. CleanRL's hyperparameters in Table 4 were tuned via a similar procedure to optimize just for Pong-v5.

All the hyperparameters used rl_games were tuned through trial and error, following the same practice in IsaacGym [21].

H License of EnvPool and RL Environments

EnvPool is under Apache2 license. Other third-party source codes and data are under their corresponding licenses.

I Data Collection Process and Broader Social Impact

All the data outputted by EnvPool is generated by the underlying simulators and game engines. Wrappers (e.g., transformation of the inputs) implemented in EnvPool conduct data pre-processing for the learning agents. EnvPool provides an effective way to parallel execution of the RL environments and does not have the typical supervised learning data labelling or collection process.

EnvPool is an infrastructure component to improve the throughput of generating RL experiences and the training system. EnvPool does not change the nature of the underlying RL environments or the training systems. Thus, to the best of the authors' knowledge, EnvPool does not introduce extra social impact to the field of RL and AI apart from our technical contributions.

J Author Contributions

Jiayi Weng and Min Lin designed and implemented the core infrastructure of EnvPool.

Shengyi Huang originally demonstrated the effectiveness of end-to-end agent training with Atari Pong and Breakout using EnvPool.

Jiayi Weng conducted pure environment simulation experiments.

Bo Liu conducted environment alignment test and contributed to EnvPool bug reports and debugging.

Shengyi Huang, Denys Makoviichuk, Viktor Makoviychuk, and Zichen Liu conducted end-to-end agent training experiments with CleanRL, rl_games, Ray, and Acme with Atari and MuJoCo environments.

Jiayi Weng developed Atari, ViZDoom, and OpenAI Gym Classic Control and Toy Text environments in EnvPool.

Jiayi Weng, Bo Liu, and Yufan Song developed OpenAI Gym MuJoCo and DeepMind Control Suite benchmark environments in EnvPool.

Jiayi Weng and Ting Luo developed OpenAI Gym Box2D environments in EnvPool.

Yukun Jiang developed OpenAI Procgen environments in EnvPool.

Shengyi Huang implemented CleanRL's PPO integration with EnvPool.

Denys Makoviichuk and Viktor Makoviychuk implemented rl_games integration with EnvPool.

Zichen Liu contributed to EnvPool Acme integration.

Min Lin and Zhongwen Xu led the project from its inception.

Shuicheng Yan advised and supported the project.

Jiayi Weng, Zhongwen Xu, Min Lin, Shengyi Huang, Bo Liu, Denys Makoviichuk, and Viktor Makoviychuk wrote the paper.