Integrating Distributed Architectures in Highly Modular RL Libraries

Anonymous Author(s)
Affiliation
Address
email

Abstract

Advancing reinforcement learning (RL) requires tools that are flexible enough to easily prototype new methods while avoiding impractically slow experimental turnaround times. To match the first requirement, the most popular RL libraries advocate for highly modular agent composability, which facilitates experimentation and development. To solve challenging environments within reasonable time frames, scaling RL to large sampling and computing resources has proved a successful strategy. However, this capability has been so far difficult to combine with modularity. In this work, we explore design choices to allow agent composability both at a local and distributed level of execution. We propose a versatile approach that allows the definition of RL agents at different scales through independent reusable components. We demonstrate experimentally that our design choices allow us to reproduce classical benchmarks, explore multiple distributed architectures, and solve novel and complex environments while giving full control to the user in the agent definition and training scheme definition. We believe this work can provide useful insights to the next generation of RL libraries.

1 Introduction

The domain of reinforcement learning (RL) has experienced significant advancements in recent years, transitioning from game-based applications [1; 2; 3; 4] to simulators [5; 6; 7; 8; 9] and to addressing practical challenges encountered in real-world scenarios such as drug discovery [10], controlling cooling systems [11] or chip design [12]. This progress has propelled RL to emerge as an increasingly influential discipline. However, the acceptance and widespread utilization of general RL libraries has been slow and fragmented. One contributing factor to this phenomenon is the intricate nature of modern RL algorithms, which typically consist of numerous interdependent and heterogeneous algorithmic components. These components include data buffers, neural networks, training loss functions, and more. The complex interplay between these components often poses challenges to developing general well-integrated RL libraries, leading many researchers to resort to writing custom code tailored to their specific applications.

To tackle this challenge, prominent libraries in the field such as Tianshou [13] or RLlib [14] have embraced highly modular methodologies, enabling component reusability and limiting code complexity. This approach provides a practical solution, especially in research settings, by allowing processes to be segregated into independent components. These components can then be combined in a flexible manner to construct RL agents. Libraries such as TorchRL[15] have further advanced this concept by providing completely standalone components and introducing robust data carriers to improve class-to-class communication. These features facilitate seamless interaction between individual components. As a result, practitioners can leverage tested and reusable modules and combine them with novel ideas implemented as new components, offering a balance between leveraging existing knowledge and exploring innovative approaches.

Submitted to 16th European Workshop on Reinforcement Learning (EWRL 2023). Do not distribute.

Nevertheless, an additional still unsolved challenge stems from the intrinsic sample inefficiency of most RL algorithms, which often demand a substantial volume of samples to tackle intricate problems effectively within reasonable time frames. In recent years, researchers have turned to scale-up algorithms by distributing data collection and other computationally demanding tasks, an approach that has shown promise in works like IMPALA [16], R2D2 [17], DPPO[18], DDPPO[19] or RAPID [2]. These results have consequently made scalability a desirable feature in RL libraries. However, integrating distributed training from a modular perspective has proven to be challenging thus far. Currently, most libraries tend to adopt a rigid approach to scalability, even if the rest of the library is highly modular. They rely on high-level methods that handle the distributed intricacies internally, which can limit users' ability to exercise fine-grained control over this relevant training aspect. However, while most distributed works are supported by different underlying distributed schemes, they can be broken down into common reusable components, allowing for the adoption of a modular approach.

In this study, we focus on finding good design choices to integrate distributed training into a modular framework. To achieve this, we start by implementing PyTorch-based [20] agents from the seamless combination of individual agent components, such as storages, losses, or actors. This combination empowers us to create a diverse set of RL methods, including on-policy, off-policy, and model-based approaches. Components can be reused in different agents and new components can be easily designed or extended and combined with those already available without any need to change the library's internal code. This is akin to the most successful RL libraries. Additionally, we develop scheme components, which encompass various functionalities and distributed options. By combining these components, a diverse array of training schemes can be defined with the independence of the agent components, extending the inherent benefits of composable agents to the context of distributed training. We validate our design choices by testing a variety of training schemes against classic benchmarks. We also test their capacity to accelerate training processes in increasingly large clusters and obtain the highest to-date test performance on the Obstacle Tower Unity3D challenge environment [7]. In doing so, our goal is to provide valuable insights into successfully integrating agent and scheme composability and to offer guidance for future projects on making informed design choices.

Our work contributes in several key ways. Firstly, we make an analysis of distributed RL requirements and propose a set of components that allow for the creation of distributed training schemes at various levels of execution following a modular paradigm. Secondly, we investigate and test our design choices, showcasing its ability to accurately replicate classical benchmarks and to effectively address novel problems while granting complete control over the scaling option and being transparent to the user. Lastly, our work demonstrates the advantages of integrating distributed applications using the paradigm of independent, reusable, and interchangeable components, such as code comprehensibility and simplification. We firmly believe our work can offer valuable insights for the future generations of RL libraries, aimed at achieving widespread adoption among the research and development community.

2 Related Work

The advancements in RL have led to the appearance of a wide array of techniques. To enhance accessibility and comparability of RL methods, numerous libraries have been introduced with the objective to address the complexities associated with the implementation, training, and assessment of both existing and emergent RL algorithms [21; 22; 23; 24; 25; 26; 27]. However, despite the abundance of RL libraries, practitioners still face difficulties in finding suitable options that allow for the exploration of new methods while facilitating flexible scalability in training.

For example, many formerly prevalent libraries have gradually fallen into obsolescence, with several no longer receiving regular maintenance [28; 29; 30; 31]. Other libraries fall short in terms of modularity, presenting hurdles for researchers and developers intending to integrate custom algorithmic components for application and method testing [23; 26]. Furthermore, while certain libraries adopt modularity as a design principle, they lack considerations for distributed training, and focus predominantly on single-threaded implementations [32; 33; 28; 13; 24].

Among the libraries that offer modularity and distributed training, RLlib [14] and RLgraph [34] utilize a distributed framework called Ray [35]. This framework is characterized by logically centralized control, task parallelism, and resource encapsulation. In a similar vein, the ACME

[36] library employs launchpad [37] for executing distributed training. Launchpad is a library specifically designed to simplify distributed programming by representing systems as graphs with nodes representing computational services. Nonetheless, there are two commonalities shared by all these approaches. Firstly, they tend to employ high-level implementations, instantiating agents through single function calls that accept a multitude of parameters and handle agent composition "under the hood". Secondly, these approaches abstract the configuration aspects of the distributed scheme from the user. Such design choices can ensure the compatibility of method components and simplify the nuances of distributed scaling. However, they also make it more difficult to understand which code is being executed, hinder code reusability, increase its complexity, and remove fine-grained control over the distributed scheme definition from the user, thus limiting the available options. The absence of these features complicates the tasks of many RL practitioners and researchers. Finally, TorchRL [15] provides a limited selection of standalone composable distributed components tailored for specific use cases. We believe that our design approach could be employed to expand its range of distributed components, thus encompassing all possible RL use cases.

3 Methods

In this section, we introduce the interchangeable agent components that provide a high level of flexibility to define and extend RL agents. These components can be easily customized to suit the requirements of specific RL problems. Additionally, we delve into the rationale behind our distributed design choices and present a detailed overview of the specific distributed components. We also provide implementation details of these components, showcasing how the combination of agent-reusable components and scheme-reusable components empowers practitioners with a remarkable degree of flexibility in two crucial dimensions: agent definition and training scheme definition.

3.1 Agent Composability

Our first design choice is to allow for agent composability. The main constituents of RL agents can be selected from a pool of options, and the resulting set of components can be seamlessly combined to construct a working RL agent. Each component can be selected independently of the rest and any specific component can be replaced by other candidates of the same type, resulting in a different agent but precluding execution errors. We distinguish between four types of agent components implemented as different Python classes, the combination of which creates a complete RL agent:

Actors. These components implement the deep neural networks used as function approximators in the form of PyTorch nn.Modules and provide the methods to update them and generate predictions (most notably, but not exclusively, next action predictions). Currently, three classes are available for implementation: `OnPolicyActor`, `OffPolicyActor` and `ModelBasedPlannerActor` for on-policy and off-policy and model-based RL methods, respectively.

Storages. These components handle the storage, processing, and retrieval of environment transition data. For on-policy algorithms, the storage options include `VanillaOnPolicyBuffer`, `GAEBuffer` (Generalized Advantage Estimation [38]), and `VTraceBuffer` [16]. For off-policy algorithms, the storage options encompass `ReplayBuffer`, `NStepBuffer` (for multi-step TD-learning), `PERBuffer` (Prioritized Experience Replay [39]), `HERBuffer` (Hindsight Experience Replay [40]), `EREBuffer` (Emphasizing Recent Experience [41]). Finally, for model-based algorithms `MBReplayBuffer` (Model-Based RL Replay Buffer) is available.

Algorithms. These components manage loss and gradient computations from data. The Algorithms include the on-policy methods `A2C` (Advantage Actor-Critic [42]) and `PPO` (Proximal Policy Optimization [43]), for off-policy methods `DQN` [1] (incorporating improved versions like Rainbow [44]), `DDPG` (Deep Deterministic policy gradient [45]), `TD3` (Twin Delayed Deep Deterministic policy gradient [46]), `SAC` (Soft Actor-Critic [47]) ,and `MPO` (Maximum a Posteriori Policy Optimisation [48])

Envs. Vectorized environments to make more efficient use of computing resources during inference time. Can work with any Gym-based environment. Currently, the environments accessible for training include OpenAI `gym` [49] (including MuJoCo [50] and Pybullet [5]), Obstacle Tower [7], Animal Olympics, Habitat [6] and Trifinger Real Robot [51].

Following a modular paradigm, new components can be created and combined with already existing ones as they are rooted in Python abstract superclasses defining all methods requiring implementation by any subclass to ensure correct functionality. While our primary goal is not to function as a reference library, we offer component templates accompanied by comprehensive code documentation to simplify the process of creating new implementations.

Finally, in distributed training scenarios, it is frequently required to instantiate multiple instances of RL components in separate processes. For example, both data collection and gradient computation tasks necessitate a copy of the actor component. To address this challenge, we employ a straightforward solution. All agent components are equipped with a method called *create_factory()* that returns a function capable of instantiating a new copy of the component each time it is called. By utilizing these methods and passing them to the distributed components, we ensure a straightforward and modular bottom-up scheme definition while accommodating the need for interchangeable component instances. Code examples of the components for PPO, SAC, and Model-Based MPC can be found in figures 1, 2 and 3, with color coding to depict different components.

```python
from pytorchrl.agent.actors import OnPolicyActor
from pytorchrl.agent.algorithms import PPO
from pytorchrl.agent.env import VecEnv
from pytorchrl.agent.storages import GAEBuffer

# Define environment
def halfcheetah_env():
    import gym, pybullet_envs

    env = gym.make("HalfCheetahBulletEnv")
    return env
# Define Vector of environments
envs_factory, act_space, obs_space = VecEnv.create_factory(
    vec_env_size=8, env_fn=halfcheetah_env
)
# Define Actor
actor_factory = OnPolicyActor.create_factory(
    obs_space,
    act_space
)
# Define Algo
algo_factory = PPO.create_factory(
    lr=1e-4,
    num_epochs=4,
    clip_param=0.2,
    entropy_coef=0.01,
    value_loss_coef=0.5,
    num_mini_batch=4,
    gamma=0.99,
)
# Define Storage
storage_factory = GAEBuffer.create_factory(
    size=1000,
    gae_lambda=0.95
)
```

Figure 1: PPO example

```python
from pytorchrl.agent.actors import OffPolicyActor
from pytorchrl.agent.algos import SAC
from pytorchrl.agent.env import VecEnv
from pytorchrl.agent.storages import ReplayBuffer

# Define environment
def halfcheetah_env():
    import gym, pybullet_envs

    env = gym.make("HalfCheetahBulletEnv")
    return env
# Define Vector of environments
envs_factory, act_space, obs_space = VecEnv.create_factory(
    vec_env_size=8, env_fn=halfcheetah_env
)
# Define Actor
actor_factory = OffPolicyActor.create_factory(
    obs_space,
    act_space
)
# Define Algo
algo_factory = SAC.create_factory(
    lr_pi=3e-4,
    lr_q=3e-4,
    lr_alpha=3e-4,
    initial_alpha=0.1,
    gamma=0.99,
    polyak=0.995,
    batch_size=256,
)
# Define Storage
storage_factory = ReplayBuffer.create_factory(
    size=1000000
)
```

Figure 2: SAC example

```python
from pytorchrl.actors.world_models import WorldModel
from pytorchrl.agent.actors import ModelBasedPlannerActor
from pytorchrl.agent.algorithms import MPC_CEM
from pytorchrl.agent.env import VecEnv
from pytorchrl.agent.storages import MBReplayBuffer

# Define environment
def halfcheetah_env():
    import gym, pybullet_envs

    env = gym.make("HalfCheetahBulletEnv")
    return env
# Define Vector of environments
envs_factory, act_space, obs_space = VecEnv.create_factory(
    vec_env_size=8, env_fn=halfcheetah_env
)
# Define Actor
actor_factory = ModelBasedPlannerActor.create_factory(
    obs_space,
    act_space,
    horizon=15,
    n_planner=1000,
    world_model_class=WorldModel
)
# Define Algo
algo_factory = MPC_CEM.create_factory(
    lr=3e-4,
    mb_epochs=100,
    mini_batch_size=256,
)
# Define Storage
storage_factory = MBReplayBuffer.create_factory(
    size=1000000
)
```

Figure 3: MPC example

3.2 Distributed Scheme Composability

In this subsection we detail the unique characteristics and challenges associated with distributed reinforcement learning (RL), to then leverage these insights to define our scheme components.

3.2.1 Distributed Design Choices

Deep RL algorithms are generally based on the repeated execution of three sequentially ordered operations: rollout collection, gradient computation, and policy update. In single-threaded implementations, all operations are executed within the same process, and training speed is limited by the performance that the slowest operation can achieve with the resources available on a single machine. Furthermore, these algorithms do not have regular computation patterns, i.e. while rollout collection is generally limited by CPU capacity, gradient computation is often GPU-bounded, causing inefficient use of the available resources.

To alleviate computational bottlenecks, a more convenient programming model consists of breaking down training into multiple independent and concurrent computation units called compute actors or simply actors, with access to separate resources and coordinated from a higher-level script. To prevent any confusion with the *Actor* component in our RL agents, we refer to these compute units as Workers in the remainder of the paper. Even within the computational budget of a single machine, this solution enables more efficient use of computing resources at the cost of slightly asynchronous operations. Furthermore, if Workers can communicate across a distributed cluster, this approach allows to leverage the combined computational power of multiple machines.

A Worker-based software solution offers two main design possibilities that define implementable training schemes: 1) Any operation can be parallelized, executing it simultaneously across multiple Worker replicas. 2) Coordination between consecutive operations can be synchronous or asynchronous. In other words, it is possible to decouple two consecutive operations by running them in different Workers that never remain idle and coordinate asynchronously, achieving higher resource efficiency. Thus, specifying which operations are parallelized and which operations are asynchronous defines the training schemes that we can implement. Note that single-threaded implementations are nothing but a particular case in which any operation is parallelized or executed asynchronously. Breaking down RL training also enables a composable implementation, allowing for the flexible combination of different task-based components.

3.2.2 Distributed Components

The cornerstone of our distributed design is the Worker class, which is exclusively dedicated to a single operation among the three essential tasks: data collection, gradient computation, or model updates. Each variant of the Worker class is specifically tailored to handle a distinct operation, ensuring a clear and focused role for each Worker within the system. Based on the specific operation they perform, we categorize them as CollectionWorker, GradientWorker, and UpdateWorker, respectively. These Workers can be organized into sets to facilitate the parallel execution of tasks. As a result, the three key distributed components that form the foundation of any distributed architecture are the following:

Collector Workers Sets. The collection worker plays a crucial role in gathering and transmitting data samples. In order to accomplish this task, it necessitates a copy of the environment, the actor, and the buffer. By consolidating multiple collection workers into a unified class, the collection worker effectively manages the process of data collection.

Gradient Workers Sets. A gradient worker receives data rollouts from a set of collection workers that it manages and computes gradients. It requires a copy of the storage, the actor and the algorithm components and provides the flexibility to specify whether the coordination between the collection workers is synchronous or asynchronous. In the synchronous case, the gradient worker waits for all the collection workers to complete their respective data collection before computing the gradients. Subsequently, an updated version of the actor is sent to all collection workers simultaneously. In the asynchronous case, the gradient worker utilizes the data as it becomes available from the collection workers. This ensures that the collection agents are continuously active, sending data and receiving slightly delayed versions of the actor, which we refer to as policy lag. This asynchronous approach enables efficient utilization of the collection workers without unnecessary idle periods. Multiple gradient workers are also grouped into a gradient worker set.

Update Workers. Finally, an update worker manages a set containing one or more gradient workers and solely requires a copy of the actor component to perform weight updates. It allows us to define the coordination strategy between the gradient workers. In the case of synchronous coordination, gradients from all workers are averaged and collectively applied to the neural networks. This ensures a synchronized update across all workers, promoting consistent learning progress. On the other hand, in asynchronous coordination, the workers operate continuously and some delay between the weights used for gradient computation and the weights to which the gradients are applied can is to be expected. This phenomenon is called gradient asynchrony. In our approach, we have made a deliberate decision to limit the size of the updater worker sets to a single worker. This choice is motivated by our focus on simplicity, excluding multi-agent applications from our current considerations.

By leveraging these components, researchers can design and configure distributed RL systems that effectively share the workload among multiple workers, optimizing the overall efficiency and performance of the training process. For instance, as shown in Figure 5, by defining a collection worker set consisting of multiple workers and configuring them to communicate asynchronously, one can achieve schemes such as IMPALA, APE-X, or R2D2. Alternatively, employing multiple gradient collectors with synchronous coordination allows for approaches like DPPO and DDPPO to be utilized. We have chosen to implement our Workers utilizing the Ray [35] distributed framework, which provides a robust and efficient infrastructure for distributed computing. Please note that the modularity of this approach would allow scheme components to be replaced by others. For example, alternative components utilizing backends other than Ray, such as Submittit[52], could be incorporated into the pool of available options to collect, compute gradients, or update networks.

In summary, we formulate a comprehensive definition of a training process using a bottom-up approach. Initially, we select the agent components. Multiple example are shown in figures 1, 2 and 3. Next, as illustrated in figure 4 we establish the collector workers, followed by the gradient workers, and finally the update worker. Different scheme architectures can be configured as explained in Figure 5. It is important to note that the composition of workers follows a hierarchical structure. Specifically, a gradient worker can coordinate multiple collection workers, and an update worker can manage multiple gradient workers.

```python
from pytorchrl.scheme import CWorkerSet, GWorkerSet, UWorker
# Collector workers
col_workers_factory = CWorkerSet.create_factory(
    num_workers=1,
    algo_factory=algo_factory,
    actor_factory=actor_factory,
    storage_factory=storage_factory,
    test_envs_factory=test_envs_factory,
    train_envs_factory=train_envs_factory,
    col_worker_resources={"num_cpus": 1, "num_gpus": 0.25},
)
# Gradient Workers
grad_workers_factory = GWorkerSet.create_factory(
    num_workers=1,
    col_communication=prl.SYNC,
    col_workers_factory=col_workers_factory,
    grad_worker_resources={"num_cpus": 1, "num_gpus": 0.25},
)
# Model Updater
update_worker = UWorker(
    grad_communication=prl.SYNC,
    grad_workers_factory=grad_workers_factory,
)
collected_steps = 0
while collected_steps < target_steps:
    # Collect data, take one grad step, update model parameters
    info = update_worker.step()
```

Figure 4: Scheme components offer a simple and composable way to define the number of workers of each type, worker resources, and worker coordination patterns with independence of agent components.

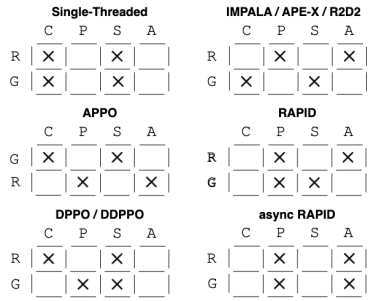

Figure 5: Execution of Rollouts collection (R) and Gradient computation (G) tasks can be either Centralised (C, executed only in 1 Worker) or Parallelized (P, multiple workers), and either Synchronous (S) or Asynchronous (A). Different parameter combination were used by DPPO [18], DDPPO [19], APPO [53], IMPALA[16], Ape-X [54] R2D2[17] or Rapid [2].

4 Experiments

Distributed training schemes offer the advantage of significantly reducing training times, making them an appealing choice for accelerating the learning process. However, it is important to recognize that these schemes come with their own set of challenges and deviations from the original problem formulation. One common consequence is the introduction of policy lag when data collection is asynchronous, where there is a delay between the policy version used for data collection and the one used for gradient computations. Another deviation is gradient asynchrony, where the delay occurs between the policy version used for gradient computation and the application of the computed gradients. Additionally, certain schemes may alter the effective value of the batch size, further modifying the training process. These deviations pose a challenge in selecting the most suitable training architecture for each experiment.

Access to a variety of training scheme options, including both distributed and non-distributed approaches, allows practitioners to explore to identify the most appropriate scheme for each reinforcement learning application. In this section, we test whether or not our design choices allow for flexible experimentation at different levels of execution with independence to the agent components.

4.1 Single Machine Benchmarks

We validate our design choices by training a number of PPO-based agents on Atari 2600 games from the Arcade Learning Environment [55]. We combine the agent with several architecture configurations including a Single-Threaded scheme and other schemes presented in Figure 5. We refer to each scheme by the name of the original work that introduced it. We keep the number of collection and gradient Workers to 1, with the idea to test if training speed arises solely from decoupling operations. We measure training speed as the number of environment frames being processed, on average, in one second of clock time, or Frames-Per-Second (FPS). We fix all agent hyperparameters to be equal to

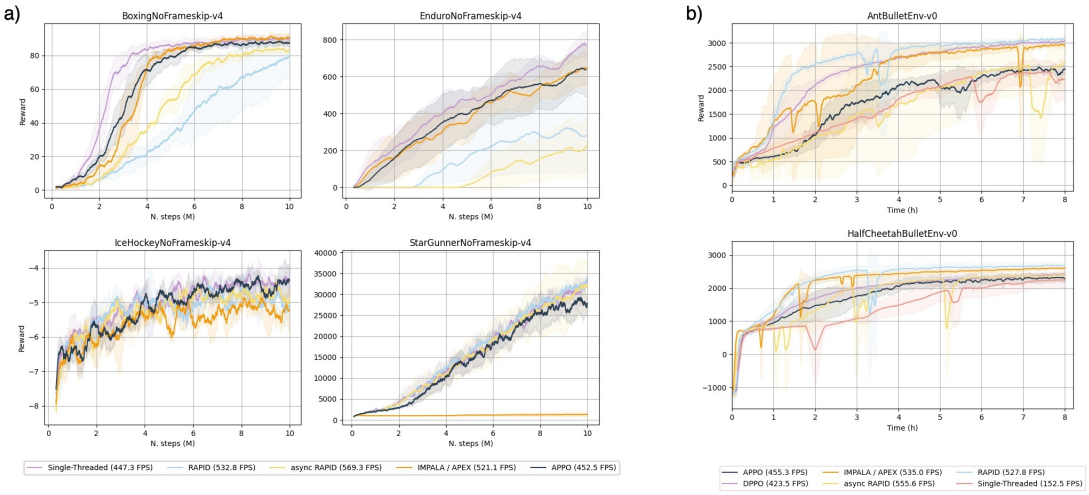

Figure 6: **Experimental results on Atari 2600 with PPO-based agents (a) and Pybullet with SAC-based agents (b) environments with different distributed training schemes.** We train each scheme 3 times, with different random seeds and with random weight initialization, and plot the mean training curve. Plots were smoothed using a moving average with a window size of 100 points.

the experiments presented in [43], the original work presenting PPO, and use the same policy network topologies. We process the environment observations following [1] and train each architecture three times on each environment under consideration with randomly initialized policy weights every time. In [43] the authors used vectorized environments of size 8 to collect multiple rollouts in parallel. We do the same and use a different seed for each environment. Results are presented in Figure 6a. Our single-threaded agents, and most of our other agents, present training curves close to those from [43]. Notably, even within the exact same computational budget, by moving from a completely sequential execution of operations (single-threaded) to a completely asynchronous one (async Rapid), we achieve a speed increase of over 25%, from 447,34 FPS to 569,29 FPS.

We also validate the flexibility of our approach in several experiments on a number of PyBullet [5] environments with a SAC-based agent. It is common for RL agents to be bounded by single GPU memory, even when the machine has RAM memory and CPU resources available. However, employing a worker-based approach enables the allocation of distinct workers to different GPUs. Therefore, we design an experiment to test speed increases in a single machine using multiple GPUs and experiment with schemes that have 2 gradient Workers managing 2 data collection workers each. We train each architecture 3 times in each environment under consideration for 8h, using 3 different random environment seeds and random parameter initialization in each case. Results are displayed in Figure 6b. Hyperparameters can be found in the appendix material. We observe faster convergence to similar or superior reward values compared to the single-threaded experiment in almost all the cases. As the sole exception, architectures that introduce gradient asynchrony do not seem to improve the single-threaded training for the AntBulletEnv-v0, yet they do so for the HalfCheetahBulletEnv-v0 environment. Additionally, all architectures achieve training speed increases between 2.77 and 3.64 times that of the single-threaded baseline.

4.2 Multiple Machines Benchmarks

The rest of our experiments were conducted in the Obstacle Tower Unity3D challenge environment [7] in order to test on more challenging environments. Obstacle Tower is a procedurally generated 3D world where an agent must learn to navigate an increasingly difficult set of floors (up to 100) with varying lighting conditions and textures. Each floor can contain multiple rooms, and each room can contain puzzles, obstacles, or keys enabling one to unlock doors. The number and arrangement of rooms on each floor for a given episode can be fixed with a seed parameter. This environment is far more complex than the previously tested and requires significantly more data only to start solving the initial rooms.

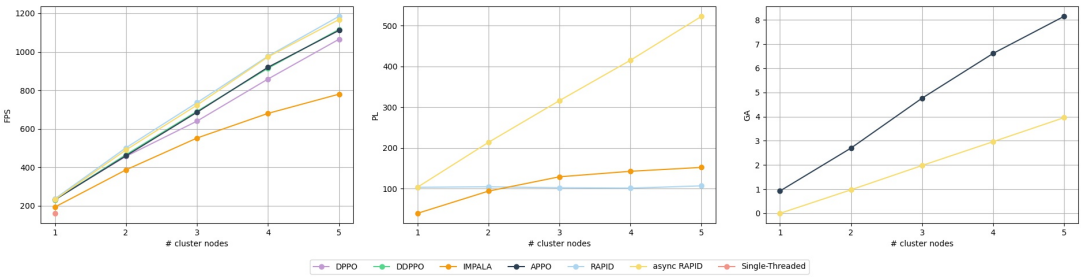

Figure 7: **Scaling Performance with multiple machines.** Frames-per-Second (FPS), Policy Lag (PL) and Gradient Asynchrony (GA) of different training schemes in increasingly large clusters. We do not plot curves that remain constant to 0 as cluster size increases.

We design our second experiment to test the capacity of our implementations to accelerate training processes in increasingly large clusters. We benchmark the FPS, the average policy lag (PL) and the average gradient asynchrony (GA) when training a PPO-based on-policy agent in clusters composed of 1 to 5 machines. We experiment with the same training scheme as in the previous section but also include a variant of DPPO, DDPPO. In DDPPO, once a predefined percentage of the parallel data collection Workers have finished their task, the straggling Workers are forced to early stop their task and the training proceeds. We set a preemption threshold of 80%, a parameter available in our gradient workers. Agent hyperparameters are held constant throughout the experiment. For each specific cluster size and training scheme combination, we select the number of collection and gradient Workers that maximize the training speed. We used machines with 32 CPUs and 3 GPUs, model GeForce RTX 2080 Ti. We could use two GPUs to obtain similar results if the environment instances could be executed in an arbitrarily specified device. However, currently, Obstacle Tower Unity3D challenge instances run by default in the primary GPU device, and thus we decide to devote it exclusively to this task. Our results are plotted in Figure 7.

One noteworthy aspect is that we conducted this experiment with minimal adjustments to our training script, highlighting the advantages of modular code. By keeping the agent components intact and solely modifying the configuration of our scheme components, we achieved great flexibility and control while effectively managing code complexity. This capacity to seamlessly test multiple distributed schemes offers significant benefits in understanding the behaviors exhibited by different training approaches, empowering practitioners to make informed design decisions.

4.3 Scaling to solve computationally demanding environments

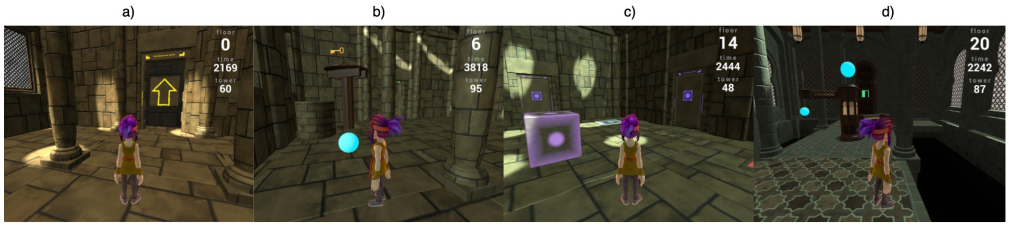

Figure 8: **Obstacle tower Unity3D challenge environment.**

In our last experiment, we aim to demonstrate a training scenario that requires the integration of all key features sought by practitioners in RL libraries. These features include the ability to flexibly experiment with agent components, experiment with scheme components, and scale the solution without the need for reimplementation. The obstacle environment consists of navigating a series of rooms connected by one or more doors. The action space is discrete (e.g. move forward, turn right, jump, move back, etc). The observation is an image of 84x84x3, no map is available. From floors 1-5 the only task is to be able to navigate rooms towards the next floor (see Figure 8a). There is a time limit that is extended every time a floor is completed. When a floor is passed the environment returns a +1 reward, and intermediate doors are given a smaller reward. There can be closed rooms where the player needs to navigate back to the previous room and pass another door. From floors 5 to 10, some doors are closed and require a key that can be picked up somewhere (see Figure 8b).

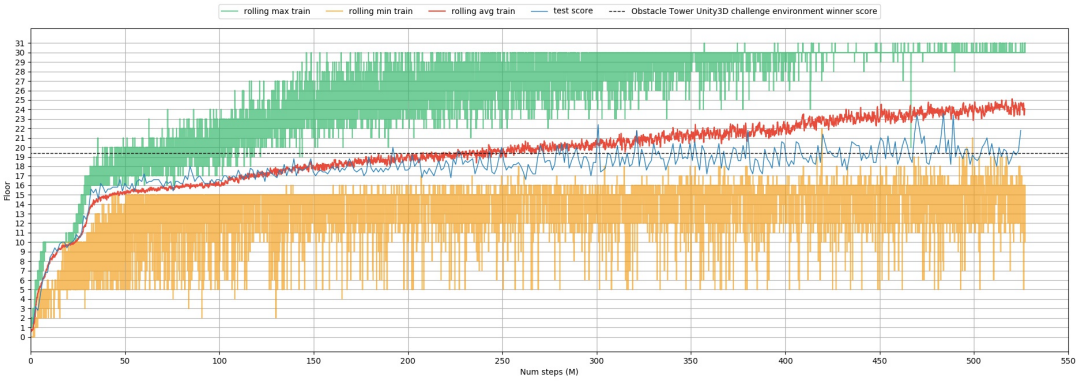

Figure 9: **Train and test curves on the Obstacle tower Unity3D challenge environment.** We use a rolling window size of size 20 to plot the maximum and the minimum obtained scores during training, and a window of size 250 for the training mean. A video recording of the best policy is available at https://youtu.be/L442rrVnDr4

Sometimes the key is on the ground but sometimes picking the key requires coordinated movements to jump up moving platforms or stairs. From floors 10-15 a box needs to be pushed over a platform on the ground in order to open doors (see Figure 8c). Both the position of the box and platform are random and can be in any room on the floor, so the player needs to navigate there first. This level was only solved by two participants in the original challenge. Above 15, the player can also fall into holes and die, the texture of the walls can change and rooms can become very complex (see Figure 8d).

We experiment with diverse agents and finally settle for an on-policy agent composed of our *PPO*, *GAEBuffer* and *OnPolicyActor* components, and use *VecEnv* to define environment vectors of size 16. As a training scheme, we use a distributed RAPID-like scheme as it offers high training speed while keeping PL and GA metrics low and stable. Nonetheless, we observe low sensitivity to the increase of these metrics in The Obstacle Tower Unity3D environment. We use 4 gradient Workers and 8 collection Workers. Specific hyperparameter values, neural network typologies and a description of the reward shape mechanism are provided in the supplementary material.

We train an RL agent on the Obstacle Tower Unity3D environment over a cluster with 64 CPU cores and 6 GPUs and compare it against the state-of-the-art, obtained during the challenge competition organized by Unity Technologies upon the environment release. We train an agent on a fixed set of seeds [0, 100) for approximately 11 days and test its behavior on seeds 1001 to 1005, a procedure designed by the authors of the challenge to evaluate weak generalization capacities of RL agents [7]. The maximum average test score is 23.6, which supposes a significant improvement with respect to 19.4, the previous state-of-the-art obtained by the winner of the competition. Our final results are presented in Figure 9 showing that we are also consistently above 19.4. A video recording of our model performance is available at https://youtu.be/L442rrVnDr4, showing that the agent reached high levels of skill after being trained with over 500 million frames of experience.

5 Conclusion

In this paper, we present an approach that enables the flexible and explicit definition of RL agents and training schemes across one or more machines. By introducing modularity, code reusability, and composition into the domain of distributed training, we address a longstanding challenge wherein such aspects have traditionally been handled internally and abstracted away from users in popular RL libraries. Through our experimental results, we showcase the efficacy of our design choices in granting users complete freedom to explore distributed schemes while maintaining fine-grained control. Moreover, we demonstrate the practical application of our approach by significantly improving the state-of-the-art performance in a challenging RL problem. We firmly believe that the integration of scalable training into highly modular libraries is a crucial step forward. We posit that our work offers valuable insights and practical guidance to those interested in contributing to the development of the next generation of libraries that cater to the needs of a broader user base. Our codebase is available on github.

References

[1] 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.

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

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[4] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[5] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation in robotics, games and machine learning, 2017.

[6] Matthew Crosby, Benjamin Beyret, and Marta Halina. The animal-ai olympics. *Nature Machine Intelligence*, 1(5):257–257, 2019.

[7] Arthur Juliani, Ahmed Khalifa, Vincent-Pierre Berges, Jonathan Harper, Ervin Teng, Hunter Henry, Adam Crespi, Julian Togelius, and Danny Lange. Obstacle tower: A generalization challenge in vision, control, and planning. *arXiv preprint arXiv:1902.01378*, 2019.

[8] William H Guss, Cayden Codel, Katja Hofmann, Brandon Houghton, Noboru Kuno, Stephanie Milani, Sharada Mohanty, Diego Perez Liebana, Ruslan Salakhutdinov, Nicholay Topin, et al. The minerl competition on sample efficient reinforcement learning using human priors. *arXiv preprint arXiv:1904.10079*, 2019.

[9] Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, et al. Habitat: A platform for embodied ai research. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9339–9347, 2019.

[10] Thomas Blaschke, Josep Arús-Pous, Hongming Chen, Christian Margreitter, Christian Tyrchan, Ola Engkvist, Kostas Papadopoulos, and Atanas Patronov. Reinvent 2.0: an ai tool for de novo drug design. *Journal of chemical information and modeling*, 60(12):5918–5922, 2020.

[11] Jerry Luo, Cosmin Paduraru, Octavian Voicu, Yuri Chervonyi, Scott Munns, Jerry Li, Crystal Qian, Praneet Dutta, Jared Quincy Davis, Ningjia Wu, et al. Controlling commercial cooling systems using reinforcement learning. *arXiv preprint arXiv:2211.07357*, 2022.

[12] 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.

[13] Jiayi Weng, Huayu Chen, Dong Yan, Kaichao You, Alexis Duburcq, Minghao Zhang, Yi Su, Hang Su, and Jun Zhu. Tianshou: A highly modularized deep reinforcement learning library. *Journal of Machine Learning Research*, 23(267):1–6, 2022. URL `http://jmlr.org/papers/v23/21-1127.html`.

[14] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E Gonzalez, Michael I Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. *arXiv preprint arXiv:1712.09381*, 2017.

[15] Vincent Moens. TorchRL: an open-source Reinforcement Learning (RL) library for PyTorch, 2023. URL `https://github.com/pytorch/rl`.

[16] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

[17] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. 2018.

[18] Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, SM Eslami, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

[19] 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. *arXiv*, pages arXiv–1911, 2019.

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[21] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL `http://jmlr.org/papers/v22/20-1364.html`.

[22] DI engine Contributors. DI-engine: OpenDILab decision intelligence engine. `https://github.com/opendilab/DI-engine`, 2021.

[23] 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, 2022. URL `http://jmlr.org/papers/v23/21-1342.html`.

[24] The garage contributors. Garage: A toolkit for reproducible reinforcement learning research. `https://github.com/rlworkgroup/garage`, 2019.

[25] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. OpenSpiel: A framework for reinforcement learning in games. *CoRR*, abs/1908.09453, 2019. URL `http://arxiv.org/abs/1908.09453`.

[26] Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G Bellemare. Dopamine: A research framework for deep reinforcement learning. *arXiv preprint arXiv:1812.06110*, 2018.

[27] Sergio Guadarrama, Anoop Korattikara, Oscar Ramirez, Pablo Castro, Ethan Holly, Sam Fishman, Ke Wang, Ekaterina Gonina, Neal Wu, Efi Kokiopoulou, Luciano Sbaiz, Jamie Smith, Gábor Bartók, Jesse Berent, Chris Harris, Vincent Vanhoucke, and Eugene Brevdo. TF-Agents: A library for reinforcement learning in tensorflow. `https://github.com/tensorflow/agents`, 2018. URL `https://github.com/tensorflow/agents`. [Online; accessed 25-June-2019].

[28] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. Tensorforce: a tensorflow library for applied reinforcement learning. Web page, 2017. URL `https://github.com/tensorforce/tensorforce`.

[29] Matthias Plappert. keras-rl. `https://github.com/keras-rl/keras-rl`, 2016.

[30] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control, 2016.

[31] Itai Caspi, Gal Leibovich, Gal Novik, and Shadi Endrawis. Reinforcement learning coach, December 2017. URL `https://doi.org/10.5281/zenodo.1134899`.

[32] Carlo D'Eramo, Davide Tateo, Andrea Bonarini, Marcello Restelli, and Jan Peters. Mushroomrl: Simplifying reinforcement learning research. *Journal of Machine Learning Research*, 22(131): 1–5, 2021. URL `http://jmlr.org/papers/v22/18-056.html`.

[33] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

[34] Michael Schaarschmidt, Sven Mika, Kai Fricke, and Eiko Yoneki. RLgraph: Modular Computation Graphs for Deep Reinforcement Learning. In *Proceedings of the 2nd Conference on Systems and Machine Learning (SysML)*, April 2019.

[35] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018.

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

[37] Fan Yang, Gabriel Barth-Maron, Piotr Stańczyk, Matthew Hoffman, Siqi Liu, Manuel Kroiss, Aedan Pope, and Alban Rrustemi. Launchpad: A programming model for distributed machine learning research. *arXiv preprint arXiv:2106.04516*, 2021. URL `https://arxiv.org/abs/2106.04516`.

[38] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[39] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

[40] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay, 2018.

[41] Che Wang and Keith Ross. Boosting soft actor-critic: Emphasizing recent experience without forgetting the past, 2019.

[42] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

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

[44] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017.

[45] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[46] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596. PMLR, 2018.

[47] T. Haarnoja, Aurick Zhou, Kristian Hartikainen, G. Tucker, Sehoon Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine. Soft actor-critic algorithms and applications. *ArXiv*, abs/1812.05905, 2018.

[48] Abbas Abdolmaleki, Jost Tobias Springenberg, Yuval Tassa, Remi Munos, Nicolas Heess, and Martin Riedmiller. Maximum a posteriori policy optimisation. *arXiv preprint arXiv:1806.06920*, 2018.

[49] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[50] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[51] Manuel Wüthrich, Felix Widmaier, Felix Grimminger, Joel Akpo, Shruti Joshi, Vaibhav Agrawal, Bilal Hammoud, Majid Khadiv, Miroslav Bogdanovic, Vincent Berenz, et al. Trifinger: An open-source robot for learning dexterity. *arXiv preprint arXiv:2008.03596*, 2020.

[52] Facebook Incubator. Submitit. `https://github.com/facebookincubator/submitit`, 2021. [GitHub repository].

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

[54] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

[55] 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.

[56] Hongyi Zhang, Yann N Dauphin, and Tengyu Ma. Fixup initialization: Residual learning without normalization. *arXiv preprint arXiv:1901.09321*, 2019.

[57] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[58] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

A Appendix

A.1 Experiments on Atari 2600 environments

We used the ensuing parameters, following [43]. We also used the same CNN network architecture:

Table 1: PPO + GAE Agent - hyperparameters values.

PARAMETER	VALUE
ADAM LR	$2.5e-4$
CLIP-PARAM	0.15
GAMMA	0.99
FRAME-SKIP	4
FRAME-STACK	4
NUM-STEPS	128
NUM-MINI-BATCH	4
NUM-EPOCHS	3
ENTROPY-COEF	0.01
VALUE-LOSS-COEF	1.0
ENVIRONMENT VECTOR LENGTH	8
GAE-LAMBDA	0.95
MAX-GRAD-NORM	0.5

The experiment was executed with the following training script. During training, the learning rate and the PPO clipping parameter are decayed also following [43].

A.2 Experiments on PyBullet environments

We used the following parameters, hand-picked to perform well on the single-threaded regime. As a feature extractor, we used a multilayer perceptron with two hidden layers of 256 each neurons and ReLU activation function:

Table 2: SAC Agent - hyperparameters values.

PARAMETER	VALUE
ADAM LR Q NETWORKS	$1e-3$
ADAM LR POLICY	$1e-4$
ADAM LR ALPHA	$1e-05$
GAMMA	0.98
INITIAL ALPHA	0.2
ENVIRONMENT VECTOR LENGTH	1
POLYAK UPDATE	0.995
BUFFER SIZE	1500000
MINI-BATCH SIZE	256
START STEPS	10000
NUM UPDATES	32
UPDATEEVERY	128

A.3 Obstacle Tower 3D Hyperparameters and Rewad Shaping

For the Obstacle Tower 3D environment we use the network architecture proposed in [16] but we initialize its weights according to Fixup [56] and end our network with a gated recurrent unit (GRU) [57] with a hidden layer of size 256 neurons. Gradients are computed using Adam optimizer [58], with a starting learning rate of 4e-4 decayed by a factor of 0.25 both after 100 million steps and 400 million steps. The value coefficient, the entropy coefficient, and the clipping parameters of the PPO algorithm are set to 0.2, 0.01, and 0.15 respectively. We use a discount factor gamma of 0.99. Furthermore, the horizon parameter is set to 800 steps and rollout collections are parallelized using environment vectors of size 16. Gradients are computed in mini-batches of size 1600 for 2 epochs.

Finally, we use generalized advantage estimation [38] lambda of 0.95, frame skip 2 and frame stack 4. We restart each new episode at a randomly selected floor between 0 and the higher floor reached in the previous episode. During training, we restart each new episode at a randomly selected floor between 0 and the higher floor reached in the previous episode and regularly save policy checkpoints to evaluate the progression of test performance. Test performance is measured as the highest averaged score on the five test seeds obtained after 5 attempts, due to some intrinsic randomness in the environment. We also experiment with several reward-shaping modifications, which we detail in the appendix.

We experiment with several reward-shaping modifications and finally find a surprisingly simple reward structure that helps our agent reach higher floors. The extrinsic reward upon the agent completing a floor is $+1$, and $+0.1$ is provided for opening doors, solving puzzles, or picking up keys. We additionally reward the agent with intrinsic reward: an extra $+1$ to pick up keys, $+0.002$ if a box is in its field of view, $+0.001$ if a platform is in its field of view, $+1.5$ to place the boxes on the platform. We use a simple color detection on the observation image to classify if the box or the platform are in view. We also reduce the action set from the initial 54 actions to 6 (rotate the camera in both directions, go forward, and go forward while turning left, turning right or jumping). The code of our training script is provided in the supplementary material. This simple set of modifications is capable to reach the state-of-the-art by brute force sampling of the environment for hundreds of millions of steps.