EFFICIENT REINFORCEMENT LEARNING EXPERIMENTATION IN PYTORCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Deep reinforcement learning (RL) has proved successful at solving challenging environments but often requires long training times and very many samples. Furthermore, advancing artificial intelligence requires to easily prototype new methods, yet avoiding impractically slow experimental turnaround times. To this end, we present a PyTorch-based library for RL with a modular design that allows composing agents based on three components types: actors, storages and algorithms. Additionally, the definition of synchronous and asynchronous architectures is permitted with flexibility and independence of the components. We present several standard use-cases of the library and showcase its potential by obtaining the highest to-date test performance on the Obstacle Tower Unity3D challenge environment. In summary, we believe that this work helps accelerate experimentation of new ideas, simplifying research and enabling to tackle more challenging RL problems.

1 INTRODUCTION

In modern reinforcement Learning (RL) research, agents are usually implemented as a combination of multiple interacting, heterogeneous algorithmic components (e.g. data buffers, neural networks, training loss, etc.). In addition, RL relies heavily on empirical experimentation both to solve new environments and to test new research ideas. Consequently, investigating how to solve nontrivial environments with current methods sometimes becomes a painstakingly slow process.

Modular code bases are critical for deep RL research because many RL components are extensions of others proposed in previous works. Furthermore, individual components are easier to read, understand and modify, facilitating code reuse and therefore more reliable performance comparisons between methods and easier reproducibility. Composable agents are the best option to enable method experimentation. In this work we present a PyTorch-based (Paszke et al., 2019) RL library designed to address these issues and help accelerate RL experimentation. Agents are defined by selecting and combining different independent and replaceable RL components. 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 internal code. As a result, this RL library allows for a simplified definition of RL agents and fast prototyping of new methodological RL components.

The poor resource efficiency of most RL methods stems from the fact that its original problem formulation is based on the single-threaded execution of operations with heterogeneous computation patterns. To accelerate RL training times, in recent years numerous methods have been conceived around the idea of relaxing this constraint by parallelizing individual operations and decoupling the execution of consecutive operations into different processes, allowing both higher resource efficiency and the ability to distribute computational workload across multiple machines (Recht et al., 2011; Mnih et al., 2016; Espeholt et al., 2018; Wijmans et al., 2019; Heess et al., 2017; Horgan et al., 2018). In order to accelerate method development, this RL library frees practitioners from the burden of implementing their own scalable architectures, and provides a reliable manner to run RL agents in distributed regimes. This approach enables agent comparisons without worrying about performance differences being caused by quirks in different standalone implementation of distributed components. Furthermore, we provide the flexibility to define multiple distributed schemes that can be coupled with any RL agent. Organizations behind recent important contributions such as OpenAI (OpenAI, 2018) and DeepMind (Silver et al., 2018) use their own internal, scalable RL frameworks which are

all distributed and offer the possibility to separate sampling, gradient computations and policy updates both in time and hardware. Unfortunately, these libraries are largely proprietary. It is indicative that two recent Unity3D challenges in RL, Animal Olympics (Crosby et al., 2019) and Obstacle Tower Challenge (Juliani et al., 2019) were won by people who developed custom RL implementations instead of using any available library.

The remainder of the paper is organized as follows. An overview of the most relevant related work is presented in Section 2. In section 3 we detail the design choices and describe its main components. Following, some interesting use-cases are presented in section 4. More specifically, subsection 4.1 explores the possibilities offered within the computational budget of a single machine, subsection 4.2 investigates the scalability of multiple distributed training schemes, and finally, subsection 4.3 demonstrates a practical application by reaching the state-of-the-art agent performance results on the Obstacle Tower Unity3D challenge environment (Juliani et al., 2019).

2 Related work

A relatively large number of reinforcement learning libraries has been proposed to help with the challenges of implementing, training and testing the set of existing RL methods and their constantly increasing number of improvements (Gauci et al., 2018; Castro et al., 2018; Loon et al., 2019; Schaarschmidt et al., 2019; Dhariwal et al., 2017; Caspi et al., 2017; Kuhnle et al., 2017).

Some of these libraries focus on single-threaded implementations and do not consider distributed training (Dhariwal et al., 2017; Castro et al., 2018; Caspi et al., 2017; Kuhnle et al., 2017). We provide both local training, which can be useful to debug method components, and distributed training. More importantly, the library allows the switch from one option to the other with minimal changes in the training script and no changes in the agent components.

Some other libraries offer scalability. However, they often rely on communication between independent program replicas, and require specific additional code to coordinate them (Schaarschmidt et al., 2019; Loon et al., 2019; Espeholt et al., 2018; Falcon, 2019). While this programming model is efficient to scale supervised and unsupervised deep learning, where training has a relatively constant compute pattern bounded only by GPU power, it is not ideal for RL. In RL, different operations can have very diverse resource and time requirements, often resulting in CPU and GPU capacity being underexploited during alternated periods of time. Conversely, we break down the training process into sub-tasks and uses independent compute units called Workers to execute them. Workers have access to separate resources within the same cluster and define hierarchical communications patterns among them, enabling coordination from a logically centralised script. This approach, while very flexible, requires a programming model allowing to easily create and handle Workers with defined resource boundaries. We currently use Ray (Moritz et al., 2018) for that purpose.

Libraries such as RLlib (Liang et al., 2017) and RLgraph (Schaarschmidt et al., 2019) use Ray to obtain highly efficient distributed reinforcement learning implementations, with logically centralized control, task parallelism and resource encapsulation. However, both RLlib and RLgraph advocate for high level implementations, creating agents with single function calls that accept many parameters and handle the instantiation of method components "under the hood". This design choice makes sure method components match, preventing errors and allowing for a more general use of their RL APIs but difficult code understanding and method experimentation. We aim to allow for composable agents, using RL core components such as the rollout buffer, the algorithm or the actor model as building blocks. Besides the components already implemented, new components can be created and combined for experimentation without the need to change the library code or anything else in the train script. While this approach requires a deeper knowledge on the field to make sure the selected components work well together, it is much more convenient for research.

Probably the closest work to the one presented in this paper is ACME (Hoffman et al., 2020) from DeepMind, which also focuses on simplifying research with composable agents while allowing these agents to scale. It also allows to run agents both as a single-threaded process or in distributed regimes using the same core modules. However, ACME does not specifically distinguish between all possible distributed architectures, fully decoupling the construction of the RL agent from the choice of a distributed architecture used to scale. Furthermore, up to the best of our knowledge ACME's distributed components are proprietary code and have not been open sourced, restricting it in practice to the serial case.

Table 1: **Currently implemented RL components**

	Description	Policy
Algorithm	*A2C*: Advantage Actor-Critic (Mnih et al., 2016)	on
	DDQN: Double Deep Q-Learning (Van Hasselt et al., 2016)	off
	PPO: Proximal Policy Optimization (Schulman et al., 2017)	on
	DDPG : Deep Deterministic Policy Gradient (Lillicrap et al., 2015)	off
	TD3: Twin Delayed Deep Deterministic Policy Gradient (Fujimoto et al., 2018)	off
	MPO: Maximum a Posteriori Policy Optimisation (Abdolmaleki et al., 2018)	off
	SAC: Soft Actor-Critic (Haarnoja et al., 2018)	off
Actor	*OnPolicyActor*: actor class for on-policy RL agents (Montague, 1999)	on
	OffPolicyActor: actor class for off-policy RL agents (Montague, 1999)	off
Storage	*VanillaOnPolicyBuffer*: minimal data storage component for on-policy algorithms (Sutton et al., 2000)	on
	ReplayBuffer: minimal data storage component for off-policy algorithms (Mnih et al., 2015)	off
	GAEBuffer: data storage component with Generalized Advantage Estimation (Schulman et al., 2015)	on
	VTraceBuffer: data storage component with V-Trace correction (Espeholt et al., 2018)	on
	NStepBuffer: multi-step leaning replay buffer (Hessel et al., 2018)	off
	PERBuffer: Prioritized Experience Replay buffer (Andrychowicz et al., 2017b)	off
	EREBuffer: Emphasizing Recent Experience replay buffer (Wang & Ross, 2019)	off
	HERBuffer: Hindsight Experience Replay buffer (Andrychowicz et al., 2017a)	off
Envs	OpenAI gym environments (Brockman et al., 2016)	-
	Obstacle Tower Unity3D challenge environment (Juliani et al., 2019)	-
	Animal Olympics Unity3D challenge environment (Crosby et al., 2019)	-
	Habitat environment (Savva et al., 2019)	-
	Trifinger Real Robot Challenge environment (Wüthrich et al., 2020)	-

3 EFFICIENT REINFORCEMENT LEARNING

We present a library for efficient reinforcement learning experimentation. It is designed to enable simple construction of RL agents, which can subsequently be executed both in local settings or in parameterizable distributed regimes using the exact same agent building blocks. In this section we discuss how RL agent can be defined, we will motivate the need for a flexible and simple module to define training architectures and we will detail how this module, called *Scheme*, works.

3.1 RL AGENTS

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 three types of agent components implemented as different Python classes, the combination of which creates a complete RL agent: the *Actor*, which implements the deep neural networks used as function approximators and provides the methods to update them and to generate predictions (most notably, but not exclusively, next action predictions), the *Storage*, which handles the storage, processing, and retrieval of environment transitions data, and the *Algorithm*, which manages loss and gradient computations from data. With one instance of each component, a single-threaded Worker can fully define the logic of modern RL methods. Our initial release includes multiple components of each type from recent RL literature, listed in Table 1 and also displayed in Figure 1. We find it more convenient for code simplicity to distinguish between on-policy and off-policy components due to the fundamental underlying methodological differences of these two families of RL approaches (Montague, 1999), and simply construct agents of each family with different components.

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. Following a modular paradigm, already existing components are derived from these superclasses, which are backed by rich code documentation to facilitate new implementations. Already implemented components can also serve as a superclass to new ones, facilitating code reuse and reducing development time. Experimenting with different neural network architectures is also easy. The *Actor* superclass requires all its descendants to accept PyTorch neural networks as an optional input parameters, to be used to extract features from environment data. Finally, a special type of component called *VecEnv* allows to stack multiple independent environments into a single vectorized environment to make more efficient use of computing resources during inference time. To do that *VecEnv* requires to be provided with a function allowing to create individual copies of the environment under consideration. Therefore training can be both vectorized and distributed for maximum resource efficiency.

Finally, as further detailed in sections 3.1 and 3.2, we aim also at facilitating research with distributed training architectures. Training in distributed regimes can require RL components to be instantiated multiple times in different processes. To that end, instead of directly creating single copies of each component, we use functions allowing to spawn component instances in every process requiring them. We refer to such functions as *component factories*, since they can be called to create identical copies of individual components. A *component factory* can be generated for any given agent component, including *VecEnv*, via their class method called *create_factory*, inherited from all superclasses to ensure package consistency. Working with *component factories* is the critical strategy allowing us to flexibly define a large spectrum of distributed architectures in clusters of arbitrary size.

3.2 Distributed Training

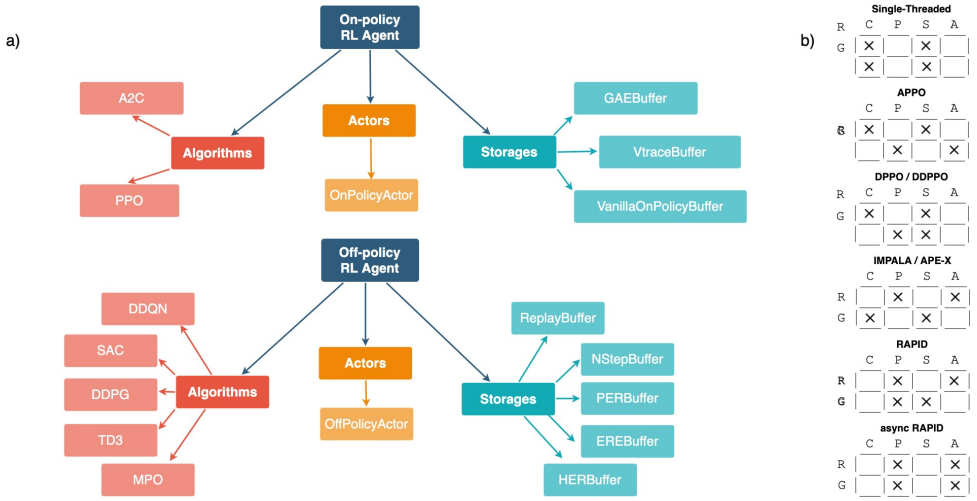

Figure 1: **(a)** Design trees of agents. **(b)** A non-exhaustive set of distributed training schemes used by popular RL agents and easily configurable. Execution of Rollouts collection (R) and Gradient computation (G) tasks can be either Centralised (C, executed only in 1 Worker) or Parallelized (P), and either Synchronous (S) or Asynchronous (A). *Scheme* component allows to define training architectures.

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 an inefficient use of the available resources.

To alleviate computational bottlenecks, a more convenient programming model consists on 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 a more efficient use of compute 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.

3.3 Adjustable Training Architectures

The creation and coordination of Workers is managed via the Ray (Moritz et al., 2018) distributed framework. Distributed training schemes achieve shorter training times but at the cost of some deviation from the original problem formulation. Common consequences of such deviations are the introduction of policy lag (Espeholt et al., 2018; Babaeizadeh et al., 2016; Mnih et al., 2016), a situation in which the policy version used for data collection is delayed by some updates with respect to the policy version used for gradient computations, or gradient asynchrony (Mnih et al., 2016; Stooke & Abbeel, 2018), when a similar delay happens between the policy version used for gradient computation and the policy version to which the gradients are applied. Additionally, some schemes also alter the effective value of the batch size (Heess et al., 2017; Wijmans et al., 2019; OpenAI, 2018). That makes choosing the most appropriate architecture for each experiment a sometimes challenging problem. While some RL agents can be in principle more robust than others to the deviations introduced by distributed schemes, generally it is still necessary to better understand how different algorithms, environments, and policy architectures behave when scaled under different training architectures. Furthermore, it is possible to develop methodological improvements to mitigate these sorts of problems. For example, (Espeholt et al., 2018) proposed an algorithmic solution called V-Trace to mitigate the brittleness of on-policy algorithms to policy lag. To facilitate rapid and flexible experimentation in distributed settings, we completely separate the definition of the RL agent from that of the training architecture.

A component called *Scheme* is provided that allows to define the training architecture and handles its instantiation, whether working on a single machine or in a cluster. More specifically, the *Scheme* class allows specifying how many Workers should be allocated to sample data from the environment, and how many Workers should be assigned to compute model gradients from the collected data to perform neural networks updates. Additionally, the CPU, GPU and memory resources assigned to individual Workers of each class can also be specified, as well as their synchronicity. If coordinated synchronously, Workers will wait until the preceding operation has finished to commence a new task and also wait for all other Workers to finish their ongoing parallel tasks before sending on their outputs. In the specific case of gradient Workers, synchronous coordination also means that gradients from all Workers will be averaged and jointly applied to the neural networks. Alternatively, Workers coordinated asynchronously never remain idle, constantly executing tasks and passing results on to the next stage, updating their version of the networks between tasks in an asynchronous manner.

Note that the *Scheme* class does not allow to specify the number of Workers performing model update operations. Each update Worker is associated with a trained model. Under this paradigm, defining multiple update Workers means training multiple agent models at the same time. While that is done in some RL methods and we hope to support it in the future, our initial release focuses on training individual agents, so we limit architectures to contain a single local update Worker. Nonetheless, the *Scheme* components allows executing coordinated updates in a distributed manner as in DDPPO (Heess et al., 2017), a configuration option that can also be specified via the *Scheme* module. Note that this can sometimes be convenient for training speed, but does not alter the RL optimization

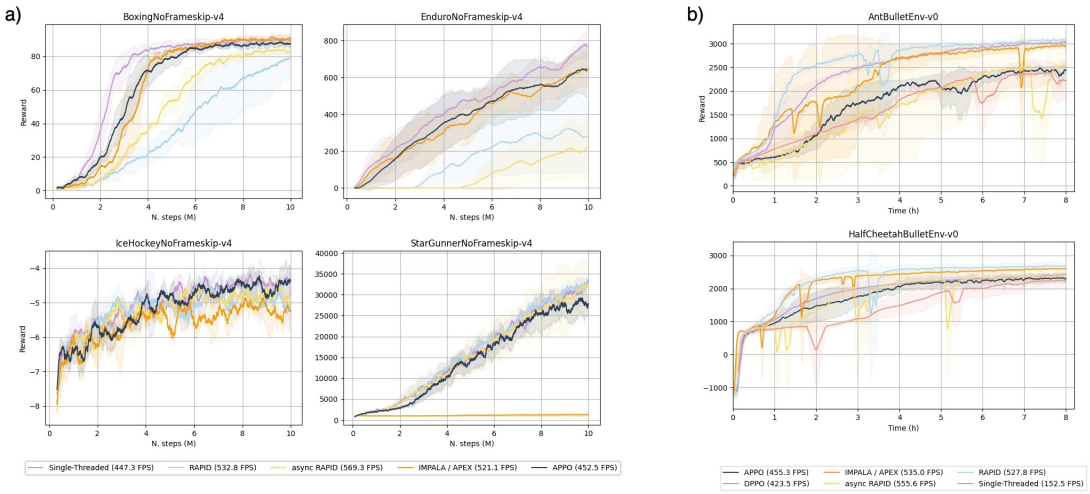

Figure 2: **Experimental results on Atari 2600 (a) and Pybullet (b) environments with different distributed training schemes.** We train each scheme 3 times, with different random seeds and with random weight initialization, and plot the mean training curve. Plots were smoothed using a moving average with window size of 100 points.

problem with respect to the default centralized update option, in which the update Worker itself updates model parameters and subsequently broadcasts them to all gradient Workers. In addition, the DDPPO architecture also proposes a preemption mechanism in which straggling data collection Workers are forced to end early once a percentage of the other Workers finish collecting experience. This option is also available in our architectures.

Figure 1b shows how some of the possible configurations of *Scheme* correspond to architectures used by several popular RL methods. These methods are associated with specific sets of agent components, but we name them only to refer to the underlying distributed scheme they use. Note that these architectures are no longer bound to a fixed set of agent components i.e. DPPO (Heess et al., 2017) underlying distributed scheme can be used, for example, to train a SAC-based agent instead of a PPO-based agent. The training process itself is managed by yet another class named *Learner*. The *Learner* class takes the *Scheme*, the target number of steps to train and, optionally, a path where to store the training logs as inputs and allows to define the training loop.

4 Experiments

4.1 Single machine benchmarks in continuous and discrete domains

We first validate the library on several experiments on a number of Atari 2600 games from the Arcade Learning Environment (Bellemare et al., 2013). We combine a PPO-based agent with GAE with several architecture configurations including a Single-Threaded scheme and other schemes presented in Figure 1b and named after the original agents that introduced them. We opt for serializing operations and 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). DPPO architecture is not tested since, when operation are not parallelized, this scheme is equivalent to a single threaded implementation. RAPID and IMPALA / APEX architectures are also equivalent under these circumstances, but for IMPALA / APEX we use a *Storage* agent component implementing V-Trace (Espeholt et al., 2018) to test the policy lag correction effect it has on a PPO agent (the original IMPALA agent uses an A2C (Mnih et al., 2016) algorithmic component).

We fix all hyperparameters to be equal to the experiments presented in (Schulman et al., 2017), the original work presenting PPO, and use the same policy network topologies. We process the environment observations following (Mnih et al., 2015) and train each architecture three times on each

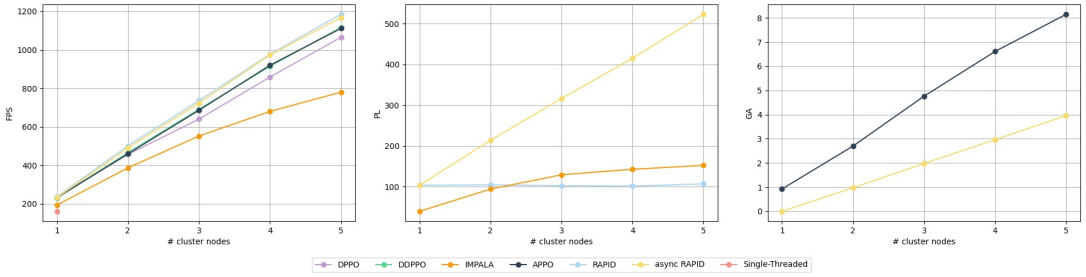

Figure 3: **Scaling Performance with multiple machines.** Frames-per-Second (FPS), Policy Lag (PL) and Gradient Asynchroy (GA) of different training schemes in increasingly large clusters. We do not plot curves that remain constant to 0 as cluster size increases.

environment under consideration with randomly initialized policy weights every time. In (Schulman et al., 2017) the authors used vectors of 8 environments to collect multiple rollouts in parallel. We do the same and use a different seed for each environment. Results are presented in Figure 2a.

Our single-threaded agents, and most of our other agents, present training curves close to those from (Schulman et al., 2017). Notably, even withing 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 observe, for a few environment and agent cases, deviations with respect to the reference training curves. Whether this depends on the environment itself, on the policy architecture or on the chosen hyperparameters or a combinations of all factors is unclear. In addition, our agent using V-Trace, IMPALA / APEX, seems to perform better than RAPID in most environments, but not in all of them.

We also validate the library on several experiments on a number of PyBullet (Coumans & Bai, 2017) environments. It is common for RL agent to be bounded by single GPU memory, even when the machine has RAM memory and CPU resources available. Therefore, we design an experiment to test speed increases in a single machine using multiple GPU's. We use an off-policy agent composed of our *SAC*, *ReplayBuffer* and *OffPolicyActor* components and configure all architectures to have 4 data collection Workers. We configure the *Scheme* class to allow fitting 2 Workers to each GPU, whether collection or gradient Workers. We define a single gradient Worker for all architectures except for RAPID and its gradient asynchronous version, async RAPID, with 2. In any case, 3 GPU are sufficient to fit the training architecture. 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 2b. Hyperparameters can be found in the supplementary 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 baseline.

4.2 Scaling to solve computationally demanding environments

The rest of our experiments were conducted in the Obstacle Tower Unity3D challenge environment (Juliani et al., 2019), a procedurally generated 3D world where an agent must learn to navigate an increasingly difficult set of floors (up to 100) with varying lightning conditions and textures. Each floor can contain multiple rooms, and each room can contain puzzles, obstacles or keys enabling to unlock doors. The number and arrangement of rooms in 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 a significantly more data only to start solving the initial rooms.

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 DDPPO. For DDPPO we set a preemption threshold of 0.8, meaning that once 80% of

the parallel data collection Workers have finished their task, the straggling Workers are forced to early stop their task and training proceeds. 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 maximise 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 3.

4.3 Achieving state-of-the-art performance on Obstacle tower Unity3D challenge environment

Figure 4: **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

In our last experiment we train a 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 behaviour on seeds 1001 to 1005, a procedure designed by the authors of the challenge to evaluate weak generalization capacities of RL agents (Juliani et al., 2019). The obstacle environment consists on 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 5a). There is a time limit which is extended every time a floor is completed. Time balls are available along the way to marginally extend time as well. When a floor is passed the environment returns a +1 reward, intermediate doors are given a smaller reward. The reward structure is therefore quite sparse. 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 which can be picked up somewhere (see Figure 5b). Sometime the key are on the ground but sometime 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 5c). Both the position of the box and platform are random and can be in any room in 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, texture of the walls can change and rooms can become very complex (see Figure 5d). The Obstacle environment has several difficulties which means that a direct use of PPO would lead to an average floor of just around 5.

We define an on-policy agent composed of our *PPO*, *GAEBuffer* and *OnPolicyActor* components, and use *VecEnv* to define environment vectors of size 16. We use a RAPID-like scheme as it offers high training speed while keeping PL and GA metrics low and stable. However, we observe low sensibility to the increase of these metrics in The Obstacle Tower Unity3D environment. We use 4 gradient

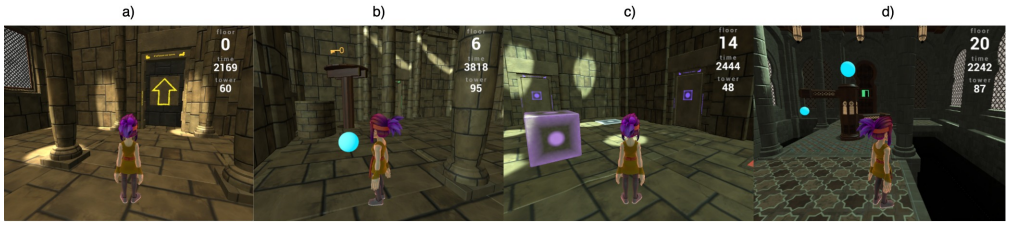

Figure 5: **Obstacle tower Unity3D challenge environment.**

Workers and 8 collection Workers. We use the network architecture proposed in (Espeholt et al., 2018) but we initialize its weights according to Fixup (Zhang et al., 2019). We end our network with a gated recurrent unit (GRU) (Cho et al., 2014) with a hidden layer of size 256 neurons. Gradients are computed using Adam optimizer (Kingma & Ba, 2014), 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 parameters is set to 800 steps and rollout collections are parallelized using environment vectors of size 16. Gradients are computed in minibatches of size 1600 for 2 epochs. Finally we use generalized advantage estimation (Schulman et al., 2015) lambda of 0.95. We use 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 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.

Key to solving the environment to higher floors is to provide a denser reward structure. Given the complexity of the tasks, it is still surprising the simplicity of these rewards. 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 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 are capable to reach the state-of-the-art by brute force sampling of the environment for hundreds of million of steps.

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 4 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. The source code is made available, including the reward shaping strategy and the resulting model in the code repository. It is impressive that the agent can solve these tasks without any imitation learning, only by itself. On the contrary, the previous state-of-the-art used extensively trajectory demonstrations.

5 Conclusion

We present and release a modular codebase for fast deep RL research that allows agent composability. In the present paper, we demonstrate that the implementations provided in it are reliable and can run in clusters, enabling further research on accelerated RL. We also show experiments can be flexibly implemented with minimal code, with RL agents that can be assembled from independent and extendable components. We believe that our library can be used as a tool to do research dealing with more challenging and complex RL environments than popular current benchmarks. The performance is further highlighted by achieving the highest to-date test score on the Obstacle Tower Unity3D challenge environment.

6 REPRODUCIBILITY STATEMENT

To facilitate reproducibility of our results, the supplementary material includes a list of the hyperparameter values as well as the training scripts used in all our experiments. The complete code to reproduce the experiments is included in the github repository of the library, which will be disclosed upon acceptance to prevent breaking anonymity, together with the online documentation. These resources also provide detailed information about package installation and the complete list of dependencies of the python environment used to run the experiments.

REFERENCES

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.

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in neural information processing systems*, pp. 5048–5058, 2017a.

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *arXiv preprint arXiv:1707.01495*, 2017b.

Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. Reinforcement learning through asynchronous advantage actor-critic on a gpu. *arXiv preprint arXiv:1611.06256*, 2016.

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.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

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

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.

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.

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

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

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.

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.

WA Falcon. Pytorch lightning. *GitHub. Note: https://github. com/williamFalcon/pytorch-lightning Cited by*, 3, 2019.

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

Jason Gauci, Edoardo Conti, Yitao Liang, Kittipat Virochsiri, Yuchen He, Zachary Kaden, Vivek Narayanan, Xiaohui Ye, Zhengxing Chen, and Scott Fujimoto. Horizon: Facebook's open source applied reinforcement learning platform. *arXiv preprint arXiv:1811.00260*, 2018.

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.

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.

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. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

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.

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.

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.

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

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

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.

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.

Keng Wah Loon, Laura Graesser, and Milan Cvitkovic. Slm lab: A comprehensive benchmark and modular software framework for reproducible deep reinforcement learning. *arXiv preprint arXiv:1912.12482*, 2019.

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

Volodymyr Mnih, 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*, pp. 1928–1937, 2016.

P Read Montague. Reinforcement learning: an introduction, by sutton, rs and barto, ag. *Trends in cognitive sciences*, 3(9):360, 1999.

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)*, pp. 561–577, 2018.

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

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*, pp. 8024–8035, 2019.

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pp. 693–701, 2011.

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*, pp. 9339–9347, 2019.

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.

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.

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

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.

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

Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pp. 1057–1063, 2000.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

Che Wang and Keith Ross. Boosting soft actor-critic: Emphasizing recent experience without forgetting the past. *arXiv preprint arXiv:1906.04009*, 2019.

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*, pp. arXiv–1911, 2019.

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.

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