Sample Factory: Egocentric 3D Control from Pixels at 100000 FPS with Asynchronous Reinforcement Learning

Aleksei Petrenko [1 2]   **Zhehui Huang** [2]   **Tushar Kumar** [2]   **Gaurav Sukhatme** [2]   **Vladlen Koltun** [1]

Abstract

Increasing the scale of reinforcement learning experiments has allowed researchers to achieve unprecedented results in both training sophisticated agents for video games, and in sim-to-real transfer for robotics. Typically such experiments rely on large distributed systems and require expensive hardware setups, limiting wider access to this exciting area of research. In this work we aim to solve this problem by optimizing the efficiency and resource utilization of reinforcement learning algorithms instead of relying on distributed computation. We present the "Sample Factory", a high-throughput training system optimized for a single-machine setting. Our architecture combines a highly efficient, asynchronous, GPU-based sampler with off-policy correction techniques, allowing us to achieve throughput higher than 10^5 environment frames/second on non-trivial control problems in 3D without sacrificing sample efficiency. We extend Sample Factory to support self-play and population-based training and apply these techniques to train highly capable agents for a multiplayer first-person shooter game. Github: `https://github.com/alex-petrenko/sample-factory`

1. Introduction

Training agents in simulated environments is a cornerstone of contemporary reinforcement learning research. Substantial progress has been made in recent years by applying reinforcement learning methods to train agents in these fast and efficient environments, whether it is to solve complex computer games (Dosovitskiy & Koltun, 2017; Jaderberg et al., 2019; Vinyals et al., 2019) or sophisticated robotic control problems via sim-to-real transfer (Müller et al., 2018;

Hwangbo et al., 2019; Molchanov et al., 2019; Andrychowicz et al., 2020).

Despite major improvements in the sample efficiency of modern learning methods, most of them remain notoriously data-hungry. For the most part, the level of results in recent years has risen due to the increased scale of experiments, rather than the efficiency of learning. Billion-scale experiments with complex environments are now relatively commonplace (Horgan et al., 2018; Espeholt et al., 2018; Kapturowski et al., 2019), and the most advanced efforts consume trillions of environment transitions in a single training session (Berner et al., 2019).

To minimize the turnaround time of these large-scale experiments, the common approach is to use distributed supercomputing systems consisting of hundreds of individual machines (Berner et al., 2019). Here, we show that by optimizing the architecture and improving the resource utilization of reinforcement learning algorithms, we can train agents on billions of environment transitions even on a *single* compute node. We present the **"Sample Factory"**, a high-throughput training system optimized for a single-machine scenario. Sample Factory, built around an Asynchronous Proximal Policy Optimization (APPO) algorithm, is a reinforcement learning architecture that allows us to aggressively parallelize the experience collection and achieve throughput as high as 130000 FPS (environment frames per second) on a single multi-core compute node with only one GPU. We describe theoretical and practical optimizations that allow us to achieve extreme frame rates on widely available commodity hardware.

We evaluate our algorithm on a set of challenging 3D environments and demonstrate how to leverage vast amounts of simulated experience to train agents that reach high levels of skill. We then extend Sample Factory to support self-play and population-based training and apply these techniques to train highly capable agents for a full multiplayer game of Doom (Kempka et al., 2016).

2. Prior Work

The quest for performance and scalability has been ongoing since before the advent of deep RL (Li & Schuurmans,

---

[1]Intel Labs [2]University of Southern California. Correspondence to: Aleksei Petrenko <petrenko@usc.edu>.

2011). Higher throughput algorithms allow for faster iteration and wider hyperparameter sweeps for the same amount of compute resources, and are therefore highly desirable.

The standard implementation of a policy gradient algorithm is fairly simple. It involves a (possibly vectorized) sampler that collects environment transitions from $N_{envs} \geq 1$ copies of the environment for a fixed number of timesteps T. The collected batch of experience – consisting of $N_{envs} \times T$ samples – is aggregated and an iteration of SGD is performed, after which the experience can be collected again with an updated policy. This method has acquired the name Advantage Actor-Critic (A2C) in the literature (Beeching et al., 2019). While it is straightforward to implement and can be accelerated with batched action generation on the GPU, it has significant disadvantages. The sampling process has to halt when the actions for the next step are being calculated, and during the backpropagation step. This leads to a significant under-utilization of system resources during training. Other algorithms such as TRPO (Schulman et al., 2015) and PPO (Schulman et al., 2017) are usually also implemented in this synchronous A2C style (Dhariwal et al., 2017).

Addressing the shortcomings of the naive implementation, the Asynchronous Advantage Actor-Critic (A3C) (Mnih et al., 2016) proposed a distributed scheme consisting of a number of independent actors, each with its own copy of the policy. Every actor is responsible for environment simulation, action generation, and gradient calculation. The gradients are asynchronously aggregated on a single parameter server, and actors query the updated copy of the model after each collected trajectory.

GA3C (Babaeizadeh et al., 2017) recognized the potential of using a GPU in an asynchronous implementation for both action generation and learning. A separate learner component is introduced, and trajectories of experience are communicated between the actors and the learner instead of parameter vectors. GA3C outperforms CPU-only A3C by a significant margin, although the high communication cost between CPU actors and GPU predictors prevents the algorithm from reaching optimal performance.

IMPALA (Espeholt et al., 2018) uses an architecture conceptually similar to GA3C, extended to support distributed training. An efficient implementation of GPU batching for action generation leads to increased throughput, with reported training frame rate of 24K FPS for a single machine with 48 CPU cores, and up to 250K FPS on a cluster with 500 CPUs.

The need for ever larger-scale experiments has focused attention on high-throughput reinforcement learning in recent publications. Decentralized Distributed PPO (Wijmans et al., 2020) optimizes the distributed policy gradient setup for multi-GPU clusters and resource-intensive environments by parallelizing the learners and significantly reducing the network throughput required. Concurrent with this work, SEED RL (Espeholt et al., 2019) improves upon the IMPALA architecture and achieves high throughput in both single-machine and multi-node scenarios, although unlike Sample Factory it focuses on more expensive hardware setups involving multiple accelerators.

Deep RL frameworks also provide high-throughput implementations of policy gradient algorithms. RLlib (Liang et al., 2018), based on the distributed computation framework Ray (Moritz et al., 2018), and TorchBeast (Küttler et al., 2019) provide optimized implementations of the IMPALA architecture. Rlpyt (Stooke & Abbeel, 2019) implements highly-efficient asynchronous GPU samplers that share some ideas with our work, although currently it does not include asynchronous policy gradient methods such as IMPALA or APPO.

Methods such as APE-X (Horgan et al., 2018) and R2D2 (Kapturowski et al., 2019) demonstrate the great scalability of off-policy RL. While off-policy algorithms exhibit state-of-the-art performance in domains such as Atari (Bellemare et al., 2013), they may be difficult to extend to the full complexity of more challenging problems (Vinyals et al., 2019), since Q-functions may be hard to learn for large multi-headed and autoregressive action spaces. In this work, we focused on policy gradient methods, although there is great potential in off-policy learning. Hybrid methods such as LASER (Schmitt et al., 2019) promise to combine high scalability, flexibility, and sample efficiency.

3. Sample Factory

Sample Factory is an architecture for high-throughput reinforcement learning on a single machine. When designing the system we focused on making all key computations fully asynchronous, as well as minimizing the latency and the cost of communication between components, taking full advantage of fast local messaging.

A typical reinforcement learning scenario involves three major computational workloads: environment simulation, model inference, and backpropagation. Our key motivation was to build a system in which the slowest of three workloads never has to wait for any other processes to provide the data necessary to perform the next computation, since the overall throughput of the algorithm is ultimately defined by the workload with the *lowest* throughput. In order to minimize the amount of time processes spend waiting, we need to guarantee that the new portion of the input is always available, even before the next step of computation is about to start. The system in which the most compute-intensive workload never idles can reach the highest resource utilization, thereby approaching optimal performance.
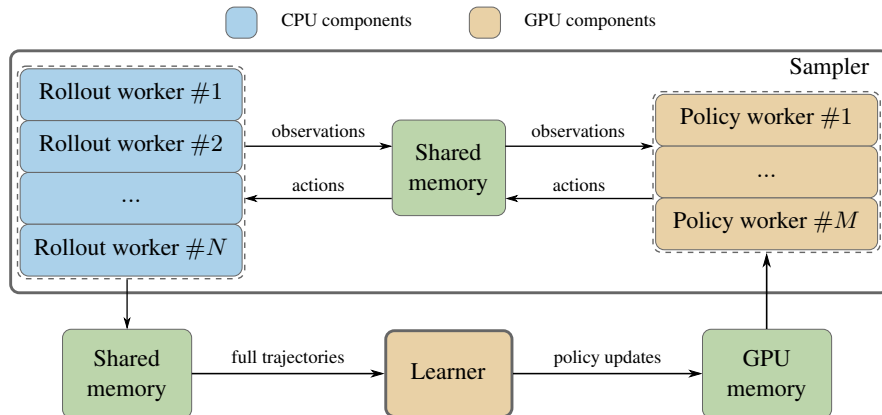
Figure 1. Overview of the Sample Factory architecture. N parallel rollout workers simulate k environments each, collecting observations. These observations are processed by M policy workers, which generate actions and new hidden states via an accelerated forward pass on the GPU. Complete trajectories are sent from rollout workers to the learner. After the learner completes the backpropagation step, the model parameters are updated in shared CUDA memory and immediately fetched by the policy workers.

3.1. High-level design

The desire to minimize the idle time for all key computations motivates the high-level design of the system (Figure 1). We associate each computational workload with one of three dedicated types of components. These components communicate with each other using a fast protocol based on FIFO queues and shared memory. The queueing mechanism provides the basis for continuous and asynchronous execution, where the next computation step can be started immediately as long as there is something in the queue to process. The decision to assign each workload to a dedicated component type also allows us to parallelize them independently, thereby achieving optimized resource balance. This is different from prior work (Mnih et al., 2016; Espeholt et al., 2018), where a single system component, such as an actor, typically has multiple responsibilities. The three types of components involved are rollout workers, policy workers, and learners.

Rollout workers are solely responsible for environment simulation. Each rollout worker hosts $k \geq 1$ environment instances and sequentially interacts with these environments, collecting observations x_t and rewards r_t. Note that the rollout workers do not have their own copy of the policy, which makes them very lightweight, allowing us to massively parallelize the experience collection on modern multi-core CPUs.

The observations x_t and the hidden states of the agent h_t are then sent to the ***policy worker***, which collects batches of x_t, h_t from multiple rollout workers and calls the policy π, parameterized by the neural network θ_π to compute the action distributions $\mu(a_t | x_t, h_t)$, and the updated hidden states h_{t+1}. The actions a_t are then sampled from the distributions μ, and along with h_{t+1} are communicated back to the corresponding rollout worker. This rollout worker uses the actions a_t to advance the simulation and collect the next

set of observations x_{t+1} and rewards r_{t+1}.

Rollout workers save every environment transition to a trajectory buffer in shared memory. Once T environment steps are simulated, the trajectory of observations, hidden states, actions, and rewards $\tau = x_1, h_1, a_1, r_1, ..., x_T, h_T, a_T, r_T$ becomes available to the ***learner***. The learner continuously processes batches of trajectories and updates the parameters of the actor θ_π and the critic θ_V. These parameter updates are sent to the policy worker as soon as they are available, which reduces the amount of experience collected by the previous version of the model, minimizing the average policy lag. This completes one training iteration.

Parallelism. As mentioned previously, the rollout workers do not own a copy of the policy and therefore are essentially thin wrappers around the environment instances. This allows them to be massively parallelized. Additionally, Sample Factory also parallelizes policy workers. This can be achieved because all of the current trajectory data $(x_t, h_t, a_t, ...)$ is stored in shared tensors that are accessible by all processes. This allows the policy workers themselves to be stateless, and therefore consecutive trajectory steps from a single environment can be easily processed by any of them. In practical scenarios, 2 to 4 policy worker instances easily saturate the rollout workers with actions, and together with a special sampler design (section 3.2) allow us to eliminate this potential bottleneck.

The learner is the only component of which we run a single copy, at least as long as single-policy training is concerned (multi-policy training is discussed in section 3.5). We can, however, utilize multiple accelerators on the learner through data-parallel training and Hogwild-style parameter updates (Recht et al., 2011). Together with large batch sizes typically required for stable training in complex environments, this gives the learner sufficient throughput to match the ex-
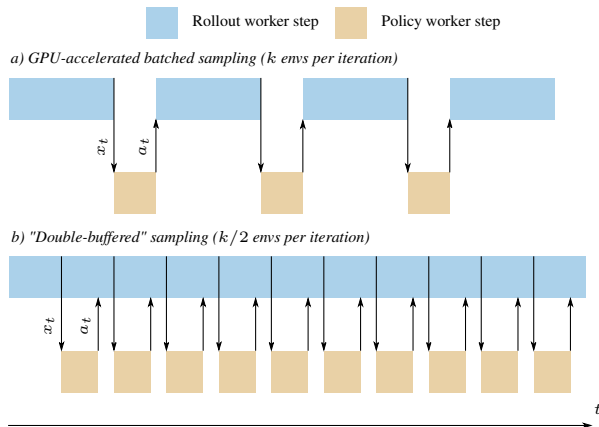
Figure 2. a) Batched sampling enables forward pass acceleration on GPU, but rollout workers have to wait for actions before the next environment step can be simulated, underutilizing the CPU. b) Double-buffered sampling splits k environments on the rollout worker into two groups, alternating between them during sampling, which practically eliminates idle time on CPU workers.

perience collection rate, unless the computational graph is highly non-trivial.

3.2. Sampling

Rollout workers and policy workers together form the sampler. The sampling subsystem most critically affects the throughput of the RL algorithm, since it is often the bottleneck. We propose a specific way of implementing the sampler that allows for optimal resource utilization through minimizing the idle time on the rollout workers.

First, note that training and experience collection are decoupled, so new environment transitions can be collected during the backpropagation step. There are no parameter updates for the rollout workers either, since the job of action generation is off-loaded to the policy worker. However, if not addressed, this still leaves the rollout workers waiting for the actions to be generated by policy workers and transferred back through interprocess communication.

To alleviate this inefficiency we use Double-Buffered Sampling (Figure 2). Instead of storing only a single environment on the rollout worker, we instead store a vector of environments $E_1, ..., E_k$, where k is even for simplicity. We split this vector into two groups $E_1, ..., E_{k/2}$, $E_{k/2+1}, ..., E_k$, and alternate between them as we go through the rollout. While the first group of environments is being stepped through, the actions for the second group are calculated on the policy worker, and vice versa. With a fast enough policy worker and a correctly tuned value for k we can completely mask the communication overhead and ensure full utilization of the CPU cores during sampling, as illustrated in Figure 2. For maximal performance with double-buffered

sampling we want $k/2 > \lceil t_{inf}/t_{env} \rceil$, where t_{inf} and t_{env} are average inference and simulation time, respectively.

3.3. Communication between components

The key to unlocking the full potential of the local, single-machine setup is to utilize fast communication mechanisms between system components. As suggested by Figure 1, there are four main pathways for information flow: two-way communication between rollout and policy workers, transfer of complete trajectories to the learner, and transfer of parameter updates from the learner to the policy worker. For the first three interactions we use a mechanism based on PyTorch (Paszke et al., 2019) shared memory tensors. We note that most data structures used in an RL algorithm can be represented as tensors of fixed shape, whether they are trajectories, observations, or hidden states. Thus we preallocate a sufficient number of tensors in system RAM. Whenever a component needs to communicate, we copy the data into the shared tensors, and send only the indices of these tensors through FIFO queues, making messages tiny compared to the overall amount of data transferred.

For the parameter updates we use memory sharing on the GPU. Whenever a model update is required, the policy worker simply copies the weights from the shared memory to its local copy of the model.

Unlike many popular asynchronous and distributed implementations, we do not perform any kind of data serialization as a part of the communication protocol. At full throttle, Sample Factory generates and consumes more than 1 GB of data per second, and even the fastest serialization/deserialization mechanism would severely hinder throughput.

3.4. Policy lag

Policy lag is an inherent property of asynchronous RL algorithms, a discrepancy between the policy that collected the experience (*behavior policy*) and the *target* policy that is learned from it. The existence of this discrepancy conditions the off-policy training regime. Off-policy learning is known to be hard for policy gradient methods, in which the model parameters are usually updated in the direction of $\nabla \log \mu(a_s|x_s)q(x_s, a_s)$, where $q(x_s, a_s)$ is an estimate of the policy state-action value. The bigger the policy lag, the harder it is to correctly estimate this gradient using a set of samples x_s from the behavior policy. Empirically this gets more difficult in learning problems that involve recurrent policies, high-dimensional observations, and complex action spaces, in which even very similar policies are unlikely to exhibit the same performance over a long trajectory.

Policy lag in an asynchronous RL method can be caused either by acting in the environment using an old policy, or

collecting more trajectories from parallel environments in one iteration than the learner can ingest in a single minibatch, resulting in a portion of the experience becoming off-policy by the time it is processed. We deal with the first issue by immediately updating the model on policy workers, as soon as new parameters become available. In Sample Factory the parameter updates are cheap because the model is stored in shared memory. A typical update takes less than 1 ms, therefore we collect a very minimal amount of experience with a policy that is different from the "master" copy.

It is however not necessarily possible to eliminate the second cause. It is beneficial in RL to collect training data from many environment instances in parallel. Not only does this decorrelate the experiences, it also allows us to utilize multi-core CPUs, and with larger values for k (environments per core), take full advantage of the double-buffered sampler. In one "iteration" of experience collection, n rollout workers, each running k environments, will produce a total of $N_{iter} = n \times k \times T$ samples. Since we update the policy workers immediately after the learner step, potentially in the middle of a trajectory, this leads to the earliest samples in trajectories lagging behind $N_{iter}/N_{batch} - 1$ policy updates on average, while the newest samples have no lag.

One can minimize the policy lag by decreasing T or increasing the minibatch size N_{batch}. Both have implications for learning. We generally want larger T, in the 2^5–2^7 range for backpropagation through time with recurrent policies, and large minibatches may reduce sample efficiency. The optimal batch size depends on the particular environment, and larger batches were shown to be suitable for complex problems with noisy gradients (McCandlish et al., 2018).

Additionally, there are two major classes of techniques designed to cope with off-policy learning. The first idea is to apply trust region methods (Schulman et al., 2015; 2017): by staying close to the behavior policy during learning, we improve the quality of gradient estimates obtained using samples from this policy. Another approach is to use importance sampling to correct the targets for the value function V^π to improve the approximation of the discounted sum of rewards under the *target* policy (Harutyunyan et al., 2016). IMPALA (Espeholt et al., 2018) introduced the V-trace algorithm that uses truncated importance sampling weights to correct the value targets. This was shown to improve the stability and sample-efficiency of off-policy learning.

Both methods can be applied independently, as V-trace corrects our training objective and the trust region guards against destructive parameter updates. Thus we implemented both V-trace and PPO clipping in Sample Factory. Whether to use these methods or not can be considered a hyperparameter choice for a specific experiment. We find that a combination of PPO clipping and V-trace works well across tasks and yields stable training, therefore we decided to use both methods in all experiments reported in the paper.

3.5. Multi-agent learning and self-play

Some of the most advanced recent results in deep RL have been achieved through multi-agent reinforcement learning and self-play (Bansal et al., 2018; Berner et al., 2019). Agents trained via self-play are known to exhibit higher levels of skill than their counterparts trained in fixed scenarios (Jaderberg et al., 2019). As policies improve during self-play they generate a training environment of gradually increasing complexity, naturally providing a curriculum for the agents and allowing them to learn progressively more sophisticated skills. Complex behaviors (e.g. cooperation and tool use) have been shown to emerge in these training scenarios (Baker et al., 2020).

There is also evidence that *populations* of agents training together in multi-agent environments can avoid some failure modes experienced by regular self-play setups, such as early convergence to local optima or overfitting. A diverse training population can expose agents to a wider set of adversarial policies and produce more robust agents, reaching higher levels of skill in complex tasks (Vinyals et al., 2019; Jaderberg et al., 2019).

To unlock the full potential of our system we add support for multi-agent environments, as well as training populations of agents. Sample Factory naturally extends to multi-agent and multi-policy learning. Since the rollout workers are mere wrappers around the environment instances, they are totally agnostic to the policies providing the actions. Therefore to add more policies to the training process we simply spawn more policy workers and more learners to support them. On the rollout workers, for every agent in every multi-agent environment we sample a random policy π_i from the population at the beginning of each episode. The action requests are then routed to their corresponding policy workers using a set of FIFO queues, one for every π_i. The population-based setup that we use in this work is explained in more detail in Section 4.

4. Experiments

4.1. Computational performance

Since increasing throughput and reducing experiment turnaround time was the major motivation behind our work, we start by investigating the computational aspects of system performance. We measure training frame rate on two hardware systems that closely resemble commonly available hardware setups in deep learning research labs. In our experiments, System #1 is a workstation-level PC with a 10-core CPU and a GTX 1080 Ti GPU. System #2 is equipped with a server-class 36-core CPU and a single RTX 2080 Ti.
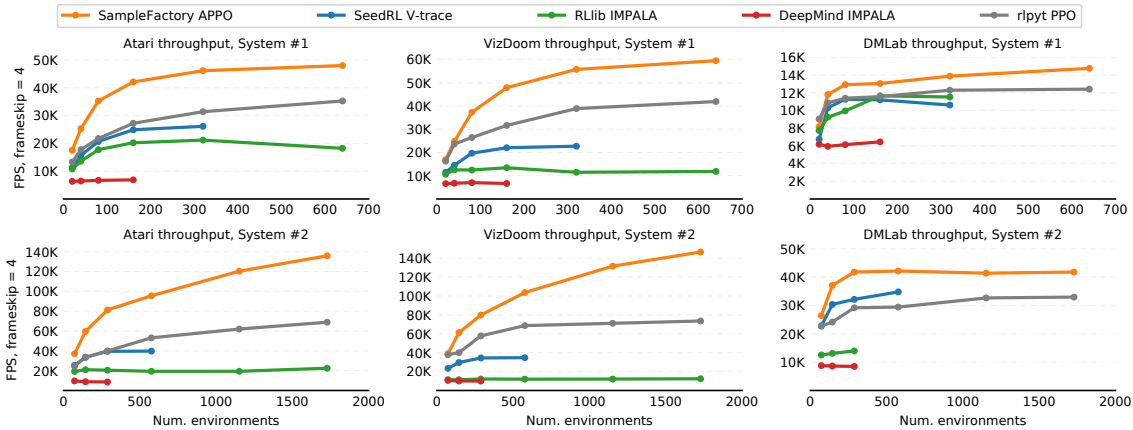
Figure 3. Training throughput, measured in environment frames per second.

As our testing environments we use three simulators: Atari (Bellemare et al., 2013), VizDoom (Kempka et al., 2016), and DeepMind Lab (Beattie et al., 2016). While that Atari Learning Environment is a collection of 2D pixel-based arcade games, VizDoom and DMLab are based on the rendering engines of immersive 3D first-person games, Doom and Quake III. Both VizDoom and DMLab feature first-person perspective, high-dimensional pixel observations, and rich configurable training scenarios. For our throughput measurements in Atari we used the game Breakout, with grayscale frames in 84×84 resolution and 4-framestack. In VizDoom we chose the environment *Battle* described in section 4.3, with the observation resolution of $128 \times 72 \times 3$. Finally, for DeepMind Lab we used the environment *rooms_collect_good_objects* from DMLab-30, also referred to as *seekavoid_arena_01* (Espeholt et al., 2018). The resolution for DeepMind Lab is kept at standard $96 \times 72 \times 3$. We follow the original implementation of IMPALA and use a CPU-based software renderer for Lab environments. We noticed that higher frame rate can be achieved when using GPUs for environment rendering, especially on System #1 (see appendix). The reported throughput is measured in simulated environment steps per second, and in all three testing scenarios we used traditional 4-frameskip, where the RL algorithm receives a training sample every 4 environment steps.

We compare performance of Sample Factory to other high-throughput policy gradient methods. Our first baseline is an original version of the IMPALA algorithm (Espeholt et al., 2018). The second baseline is IMPALA implemented in RLlib (Liang et al., 2018), a high-performance distributed RL framework. Third is a recent evolution of IMPALA from DeepMind, SeedRL (Espeholt et al., 2019). Our final comparison is against a version of PPO with asynchronous sampling from the rlpyt framework (Stooke & Abbeel, 2019), one of the fastest open-source RL implementations. We use the same model architecture for all methods, a Con-

vNet with three convolutional layers, an RNN core, and two fully-connected heads for the actor and the critic. Full benchmarking details, including hardware configuration and model architecture are provided in the supplementary files.

Figure 3 illustrates the training throughput in different configurations averaged over five minutes of continuous training to account for performance fluctuations caused by episode resets and other factors. Aside from showing the peak frame rate we also demonstrate how the performance scales with the increased number of environments sampled in parallel.

Sample Factory outperforms the baseline methods in most of the training scenarios. Rlpyt and SeedRL follow closely, matching Sample Factory performance in some configurations with a small number of environments. Both IMPALA implementations fail to efficiently utilize the resources in a single-machine deployment and hit performance bottlenecks related to data serialization and transfer. Additionally, their higher per-actor memory usage did not allow us to sample as many environments in parallel. We omitted data points for configurations that failed due to lack of memory or other resources.

Figure 4 demonstrates how the system throughput translates into raw wall-time training performance. Sample Factory and SeedRL implement similar asynchronous architectures and demonstrate very close sample efficiency with equivalent sets of hyperparameters. We are therefore able to compare the training time directly. We trained agents on two standard VizDoom environments. The plots demonstrate a 4x advantage of Sample Factory over the state-of-the-art baseline. Note that direct fair comparison with the fastest baseline, rlpyt, is not possible since it does not implement asynchronous training. In rlpyt the learner waits for all workers to finish their rollouts before each iteration of SGD, therefore increasing the number of sampled environments also increases the training batch size, which significantly affects sample efficiency. This is not the case for SeedRL
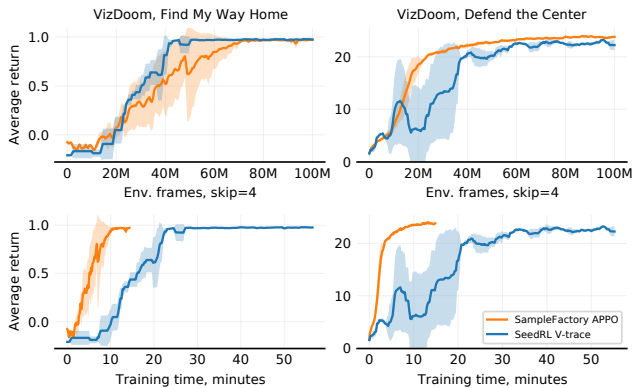
Figure 4. Direct comparison of wall-time performance. We show the mean and standard deviation of four training runs for each experiment.

and Sample Factory, where a fixed batch size can be used regardless of the number of environments simulated.

Finally, we also analyzed the theoretical limits of RL training throughput. By stripping away all computationally expensive workloads from our system we can benchmark a bare-bones sampler that just executes a random policy in the environment as quickly as possible. The framerate of this sampler gives us an upper bound on training performance, emulating an ideal RL algorithm with infinitely fast action generation and learning. Table 1 shows that Sample Factory gets significantly closer to this ideal performance than the baselines. This experiment also shows that further optimization may be possible. For VizDoom, for example, the sampling rate is so high that the learner loop completely saturates the GPU even with relatively shallow models. Therefore performance can be further improved by using multiple GPUs in data-parallel mode, or, alternatively, we can train small populations of agents, with learner processes of different policies spread across GPUs.

4.2. DMLab-30 experiment

IMPALA (Espeholt et al., 2018) showed that with sufficient computational power it is possible to move beyond single-task RL and train one agent to solve a set of 30 diverse pixel-based environments at once. Large-scale multi-task training can facilitate the emergence of complex behaviors,

	Atari, FPS	VizDoom, FPS	DMLab, FPS
Pure simulation	181740 (100%)	322907 (100%)	49679 (100%)
DeepMind IMPALA	9961 (5.3%)	10708 (3.3%)	8782 (17.7%)
RLlib IMPALA	22440 (12.3%)	12391 (3.8%)	13932 (28.0%)
SeedRL V-trace	39726 (21.9%)	34428 (10.7%)	34773 (70.0%)
rlpyt PPO	68880 (37.9%)	73544 (22.8%)	32948 (66.3%)
SampleFactory APPO	**135893 (74.8%)**	**146551 (45.4%)**	**42149 (84.8%)**

Table 1. Peak throughput of various RL algorithms on System #2 in environment frames per second and as percentage of the optimal frame rate.

which motivates further investment in this research direction. To demonstrate the efficiency and flexibility of Sample Factory we use our system to train a population of four agents on DMLab-30 (Figure 5). While the original implementation relied on a distributed multi-server setup, our agents were trained on a single 36-core 4-GPU machine. Sample Factory reduces the computational requirements for large-scale experiments and makes multi-task benchmarks like DMLab-30 accessible to a wider research community. To support future research, we also release a dataset of pre-generated environment layouts for DMLab-30 which contains a sufficient number of unique environments for 10^{10}-sample training and beyond. This dataset removes the need to dynamically generate new layouts during training, which leads to a multifold increase in throughput on DMLab-30.
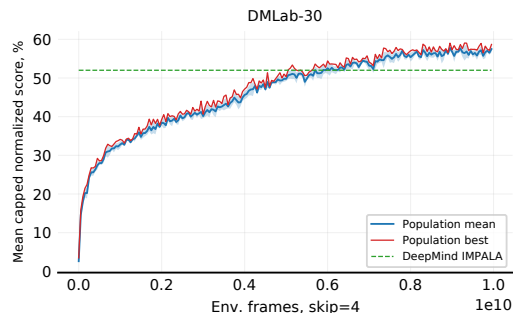

Figure 5. Mean capped human-normalized training score (Espeholt et al., 2018) for a *single-machine* DMLab-30 PBT run with Sample Factory. (Compared to cluster-scale IMPALA deployment.)

4.3. VizDoom experiments

We further use Sample Factory to train agents on a set of VizDoom environments. VizDoom provides challenging scenarios with very high potential skill cap. It supports rapid experience collection at fairly high input resolution. With Sample Factory, we can train agents on *billions* of environment transitions in a matter of hours (see Figure 3). Despite substantial effort put into improving VizDoom agents, including several years of AI competitions, the best reported agents are still far from reaching expert-level human performance (Wydmuch et al., 2019).

We start by examining agent performance in a set of basic environments included in the VizDoom distribution (Figure 6). Our algorithm matches or exceeds the performance reported in prior work on the majority of these tasks (Beeching et al., 2019).

We then investigate the performance of Sample Factory agents in four advanced single-player game modes: *Battle*, *Battle2*, *Duel*, and *Deathmatch*. In *Battle* and *Battle2*, the goal of the agent is to defeat adversaries in an enclosed maze while maintaining health and ammunition. The maze in *Bat-*
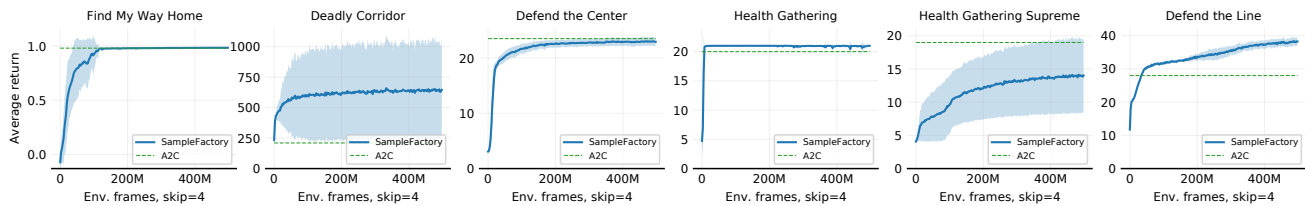
Figure 6. Training curves for standard VizDoom scenarios. We show the mean and standard deviation for ten independent experiments conducted for each scenario.

tle2 is a lot more complex, with monsters and healthpacks harder to find. The action set in the battle scenarios includes five independent discrete action heads for moving, aiming, strafing, shooting, and sprinting. As shown in Figure 7, our final scores on these environments significantly exceed those reported in prior work (Dosovitskiy & Koltun, 2017; Zhou et al., 2019).

We also introduce two new environments, *Duel* and *Deathmatch*, based on popular large multiplayer maps often chosen for competitive matches between human players. Single-player versions of these environments include scripted in-game opponents (bots) and can thus emulate a full Doom multiplayer gameplay while retaining high single-player simulation speed. We used in-game opponents that are included in standard Doom distributions. These bots are programmed by hand and have full access to the environment state, unlike our agents, which only receive pixel observations and auxiliary info such as the current levels of health and ammunition.

For *Duel* and *Deathmatch* we extend the action space to also include weapon switching and object interaction, which allows the agent to open doors and call elevators. The augmented action space fully replicates a set of controls available to a human player. This brings the total number of possible actions to $\sim 1.2 \times 10^4$, which makes the policies significantly more complex than those typically used for Atari or DMLab. We find that better results can be achieved in these environments when we repeat actions for two consecutive frames instead of the traditional four (Bellemare et al., 2013), allowing the agents to develop precise movement and aim. In *Duel* and *Deathmatch* experiments we use a 36-core PC with four GPUs to harness the full power of Sample Factory and train a population of 8 agents with population-based training. The final agents beat the in-game bots on the highest difficulty in 100% of the matches in both environments. In *Deathmatch* our agents defeat scripted opponents with an average score of 80.5 versus 12.6. In *Duel* the average score is 34.7 to 3.6 frags per episode (Figure 8).

Self-play experiment. Using the networking capabilities of VizDoom we created a Gym interface (Brockman et al., 2016) for full multiplayer versions of *Duel* and *Deathmatch* environments. In our implementation we start a separate

environment instance for every participating agent, after which these environments establish network connections using UDP sockets. The simulation proceeds one step at a time, synchronizing the state between the game instances connected to the same match through local networking. This environment allows us to evaluate the ultimate configuration of Sample Factory, which includes both multi-agent and population-based training.

We use this configuration to train a population of eight agents playing against each other in 1v1 matches in a *Duel* environment, using a setup similar to the "For The Win" (FTW) agent described in (Jaderberg et al., 2019). As in
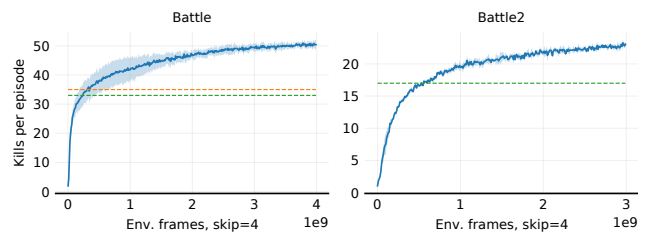

Figure 7. VizDoom battle experiments. We show the mean and standard deviation for four independent runs. Here as baselines we provide scores reported for Direct Future Prediction (DFP) (Dosovitskiy & Koltun, 2017), and a version of DFP with additional input modalities such as depth and segmentation masks, produced by a computer vision subsystem (Zhou et al., 2019). The latter work only reports results for *Battle*.
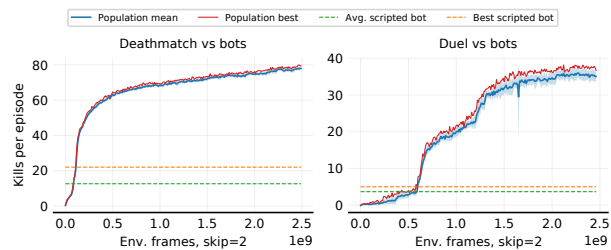

Figure 8. Populations of 8 agents trained in *Deathmatch* and *Duel* scenarios. On the y-axis we report the average number of adversaries defeated in an 4-minute match. Shown are the means and standard deviations within the population, as well as the performance of the best agent.

scenarios with scripted opponents, within one episode our agents optimize environment reward based on game score and in-game events, including positive reinforcement for scoring a kill or picking up a new weapon and penalties for dying or losing armor. The agents are meta-optimized through hyperparameter search via population-based training. The meta-objective in the self-play case is simply winning, with a reward of $+1$ for outscoring the opponent and 0 for any other outcome. This is different from our experiments with scripted opponents, where the final objective for PBT was based on the total number of kills, because agents quickly learned to win 100% of the matches against scripted bots.

During population-based training we randomly mutate the bottom 70% of the population every 5×10^6 environment frames, altering hyperparameters such as learning rate, entropy coefficient, and reward weights. If the win rate of the policy is less than half of the best-performing agent's win rate, we simply copy the model weights and hyperparameters from the best agent to the underperforming agent and continue training.

As in our experiments with scripted opponents, each of the eight agents was trained for 2.5×10^9 environment frames on a single 36-core 4-GPU server, with the whole population consuming ~ 18 years of simulated experience. We observe that despite a relatively small population size, a diverse set of strategies emerges. We then simulated 100 matches between the self-play (FTW) agent and the agent trained against scripted bots, selecting the agent with the highest score from both populations. The results were 78 wins for the self-play agent, 3 losses, and 19 ties. This demonstrates that population-based training resulted in more robust policies (Figure 9), while the agent trained against bots ultimately overfitted to a single opponent type. Video recordings of our agents can be found at https://sites.google.com/view/sample-factory.

5. Discussion

We presented an efficient high-throughput reinforcement learning architecture that can process more than 10^5 environment frames per second on a single machine. We aim to democratize deep RL and make it possible to train whole populations of agents on billions of environment transitions using widely available commodity hardware. We believe this is an important area of research, as it can benefit any project that leverages model-free RL. With our system architecture, researchers can iterate on their ideas faster, thus accelerating progress in the field.

We also want to point out that maximizing training efficiency on a single machine is equally important for *distributed* systems. In fact, Sample Factory can be used as a single node in

Figure 9. Behavior of an agent trained via self-play. **Top:** Agents tend to choose the chaingun to shoot at their opponents from longer distance. **Bottom:** Agent opening a secret door to get a more powerful weapon.

a distributed setup, where each machine has a sampler and a learner. The learner computes gradients based on locally collected experience only, and learners on multiple nodes can then synchronize their parameter updates after every training iteration, akin to DD-PPO (Wijmans et al., 2020).

We showed the potential of our architecture by training highly capable agents for a multiplayer configuration of the immersive 3D game Doom. We chose the most challenging scenario that exists in first-person shooter games – a duel. Unlike multiplayer deathmatch, which tends to be chaotic, the duel mode requires strategic reasoning, positioning, and spatial awareness. Despite the fact that our agents were able to convincingly defeat scripted in-game bots of the highest difficulty, they are not yet at the level of expert human players. One of the advantages human players have in a duel is the ability to perceive sound. An expert human player can hear the sounds produced by the opponent (ammo pickups, shots fired, etc.) and can integrate these signals to determine the opponent's position. Recent work showed that RL agents can beat humans in pixel-based 3D games in limited scenarios (Jaderberg et al., 2019), but the task of defeating expert competitors in a full game, as played by humans, requires additional research, for example into fusing information from multiple sensory systems.

References

Andrychowicz, M., Baker, B., Chociej, M., Józefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., Schneider, J., Sidor, S., Tobin, J., Welinder, P., Weng, L., and Zaremba, W. Learning dex-

terous in-hand manipulation. *The International Journal of Robotics Research*, 39(1), 2020.

Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J., and Kautz, J. Reinforcement learning through asynchronous advantage actor-critic on a GPU. In *ICLR*, 2017.

Baker, B., Kanitscheider, I., Markov, T., Wu, Y., Powell, G., McGrew, B., and Mordatch, I. Emergent tool use from multi-agent autocurricula. In *ICLR*, 2020.

Bansal, T., Pachocki, J., Sidor, S., Sutskever, I., and Mordatch, I. Emergent complexity via multi-agent competition. In *ICLR*, 2018.

Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V., Sadik, A., Schrittwieser, J., Anderson, K., York, S., Cant, M., Cain, A., Bolton, A., Gaffney, S., King, H., Hassabis, D., Legg, S., and Petersen, S. DeepMind Lab. *CoRR*, abs/1612.03801, 2016.

Beeching, E., Wolf, C., Dibangoye, J., and Simonin, O. Deep reinforcement learning on a budget: 3D control and reasoning without a supercomputer. *CoRR*, abs/1904.01806, 2019.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. In *IJCAI*, 2013.

Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., de Oliveira Pinto, H. P., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F., and Zhang, S. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. OpenAI Gym. *CoRR*, abs/1606.01540, 2016.

Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y., and Zhokhov, P. OpenAI baselines. https://github.com/openai/baselines, 2017.

Dosovitskiy, A. and Koltun, V. Learning to act by predicting the future. In *ICLR*, 2017.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., and Kavukcuoglu, K. IMPALA: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *ICML*, 2018.

Espeholt, L., Marinier, R., Stanczyk, P., Wang, K., and Michalski, M. SEED RL: Scalable and efficient deep-rl with accelerated central inference. *CoRR*, abs/1910.06591, 2019.

Harutyunyan, A., Bellemare, M. G., Stepleton, T., and Munos, R. Q(λ) with off-policy corrections. In *Algorithmic Learning Theory, ALT*, 2016.

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H., and Silver, D. Distributed prioritized experience replay. In *ICLR*, 2018.

Hwangbo, J., Lee, J., Dosovitskiy, A., Bellicoso, D., Tsounis, V., Koltun, V., and Hutter, M. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26), 2019.

Jaderberg, M., Czarnecki, W. M., Dunning, I., Marris, L., Lever, G., Castañeda, A. G., Beattie, C., Rabinowitz, N. C., Morcos, A. S., Ruderman, A., Sonnerat, N., Green, T., Deason, L., Leibo, J. Z., Silver, D., Hassabis, D., Kavukcuoglu, K., and Graepel, T. Human-level performance in 3D multiplayer games with population-based reinforcement learning. *Science*, 364(6443), 2019.

Kapturowski, S., Ostrovski, G., Quan, J., Munos, R., and Dabney, W. Recurrent experience replay in distributed reinforcement learning. In *ICLR*, 2019.

Kempka, M., Wydmuch, M., Runc, G., Toczek, J., and Jaskowski, W. Vizdoom: A Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games*, 2016.

Küttler, H., Nardelli, N., Lavril, T., Selvatici, M., Sivakumar, V., Rocktäschel, T., and Grefenstette, E. TorchBeast: A PyTorch platform for distributed RL. *CoRR*, abs/1910.03552, 2019.

Li, Y. and Schuurmans, D. MapReduce for parallel reinforcement learning. In *European Workshop on Reinforcement Learning*, 2011.

Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Goldberg, K., Gonzalez, J., Jordan, M. I., and Stoica, I. RLlib: Abstractions for distributed reinforcement learning. In *ICML*, 2018.

McCandlish, S., Kaplan, J., Amodei, D., et al. An empirical model of large-batch training. *CoRR*, abs/1812.06162, 2018.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

Molchanov, A., Chen, T., Hönig, W., Preiss, J. A., Ayanian, N., and Sukhatme, G. S. Sim-to-(multi)-real: Transfer of low-level robust control policies to multiple quadrotors. In *IROS*, 2019.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., and Stoica, I. Ray: A distributed framework for emerging AI applications. In *USENIX Symposium on Operating Systems Design and Implementation*, 2018.

Müller, M., Dosovitskiy, A., Ghanem, B., and Koltun, V. Driving policy transfer via modularity and abstraction. In *Conference on Robot Learning (CoRL)*, 2018.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An imperative style, high-performance deep learning library. In *Neural Information Processing Systems*, 2019.

Recht, B., Ré, C., Wright, S. J., and Niu, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Neural Information Processing Systems*, 2011.

Schmitt, S., Hessel, M., and Simonyan, K. Off-policy actor-critic with shared experience replay. *CoRR*, abs/1909.11583, 2019.

Schulman, J., Levine, S., Abbeel, P., Jordan, M. I., and Moritz, P. Trust region policy optimization. In *ICML*, 2015.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

Stooke, A. and Abbeel, P. rlpyt: A research code base for deep reinforcement learning in PyTorch. *CoRR*, abs/1909.01500, 2019.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575 (7782), 2019.

Wijmans, E., Kadian, A., Morcos, A., Lee, S., Essa, I., Parikh, D., Savva, M., and Batra, D. DD-PPO: Learning near-perfect PointGoal navigators from 2.5 billion frames. In *ICLR*, 2020.

Wydmuch, M., Kempka, M., and Jaskowski, W. Vizdoom competitions: Playing Doom from pixels. *IEEE Transactions on Games*, 11(3), 2019.

Zhou, B., Krähenbühl, P., and Koltun, V. Does computer vision matter for action? *Science Robotics*, 4(30), 2019.