11-633 Independent Study: Extending Learn-to-Race with Distributed Deep Reinforcement Learning

James Herman JAMESHER@CS.CMU.EDU
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213, USA

Eric Nyberg[1] EHN@CS.CMU.EDU
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213, USA

Abstract

Reinforcement learning approaches suffer from poor sample complexity. Many popular benchmarks involve millions of agent-environment interactions, and state-of-the-art control extends to the scale of billions. Reaching this scale becomes particularly problematic when using high-fidelity, computationally expensive simulation environments. In this work, we extend the Learn-to-Race framework to the distributed setting with the implementation of a generalized one-to-many, multi-node distributed system for off-policy reinforcement learning. Furthermore, we present numerous opportunities to improve the performance of the system. The code used for this paper is privately available at https://github.com/hermgerm29/learn-to-race-cmu.

1. Introduction

Reinforcement learning algorithms are sample inefficient as the number of agent-environment interactions used for popular benchmarks are on the order of *millions*. This is one of the primary reasons why agents are trained in simulated environments, where the interaction time interval is bounded by the computation time, while physical environments operate in real-time. There exist numerous approaches to distributing reinforcement learning to accelerate training. For time-invariant simulation environments, a common and powerful approach is environment vectorization, conceptually similar to any general vectorized operation, which can be performed on a single compute node. More recent works have gone further by customizing computationally expensive simulators to accept batch requests [1]. Multi-node, distributed reinforcement learning approaches vary based on the algorithm family, where learning occurs, and how data is communicated.

1. Eric Nyberg advised this independent study.

2. Learn-to-Race

Learn-to-Race [2] is a multimodal control environment for autonomous racing where agents interact with a high-fidelity racing simulator. The Learn-to-Race framework provides a variety of interfaces that allow agents to interact with the racing simulator in discrete steps and detailed racing metrics as well as a racing simulator with vehicle dynamics and configuration, accurate sensor models, and real-world inspired racing environments.

Autonomous racing presents unique challenges to agents and rich opportunities for research, including:

- Making real-time, safety-critical, decisions in a dynamic environment
- Perceiving the world from multiple, asynchronous input modalities
- Operating hardware near its physical limits resulting in significant error propagation
- Racing against intelligent agents with competing goals
- Generalizing to new racing environments with limited interaction

Challenging agents to overcome many of the above challenges, the Learn-to-Race task requires agents to race on unseen racetracks with limited interaction, similar to human race car drivers having limited practice sessions on a new track before competition.

While the racing simulator used in Learn-to-Race is able to provide agents with a complex and highly realistic learning environment, it comes with significant computational cost. The simulator requires a moderately powerful graphics card to render the environment which is necessary for virtual cameras on the vehicle, and overall, is both GPU and CPU resource intensive. Another limitation of the simulator is that it is not time-invariant and should be treated as inference-only since milliseconds matter in high-speed racing.

3. Distributed Training System

Due to considerable constraints and complexities associated with the racing simulator, we designed a customized distributed training system. All the components of our system are isolated into containers, using Docker, and our system is orchestrated with Kubernetes, a widely available, production-quality container-orchestration system that automates deployment and scaling. Together, this results in a highly portable system that can be easily deployed on both public and private compute clusters. Finally, we maintain some degree of persistence by writing periodic model checkpoints and other logging information external storage using Amazon's Simple Storage Service. Furthermore, our system has limited external dependencies as it is built with only the Python Standard Library and Tianshou [3], a fast and elegant PyTorch reinforcement learning framework. The system is also flexible as it is both policy and replay buffer agnostic.

3.1 Centralized Learning

Conceptually, our architecture is highly similar to ACME [4] and can be described as centralized, offline, asynchronous, and off-policy. At present, there is a one-to-many relationship

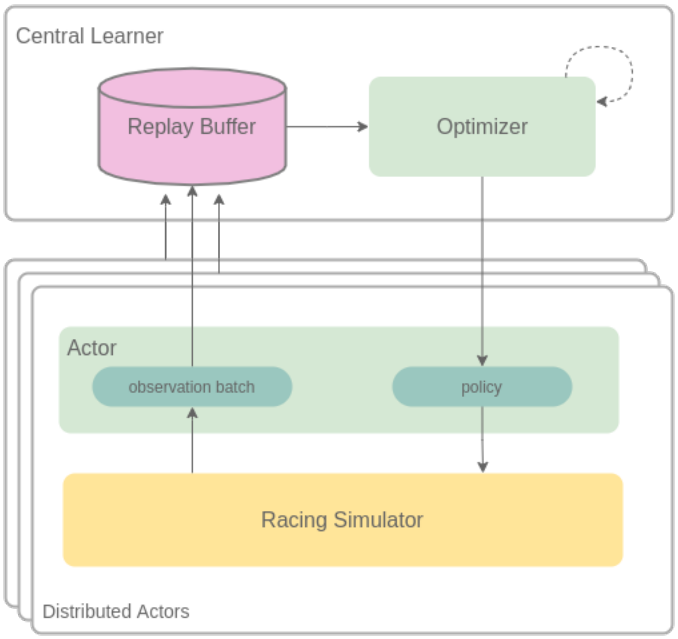

Figure 1: Distributed actors collect batches of observation data, send them to the learning node, and in response, they get an updated policy. The centralized learner uses an in-memory replay buffer, takes all gradient steps, and maintains the most up-to-date policy.

between an optimizer node, which performs gradient updates, and actor nodes which collect experience through interaction with the racing simulator. Rollout workers require a GPU to be able to run the racing simulator, and they perform inference locally to avoid costly network communication. Rollout workers send batches of experience and, in return, receive an updated policy from the optimizer as shown in Figure 1. Rather than waiting for gradient updates, the worker simply adds gradient steps to the learner's learning queue and receives the current policy. If the learning node's learning queue gets too long, it will force workers to wait, resulting in wasted resources. The optimizer is implemented as a socket server and data, namely batches of experience and policy weights, is communicated using TCP protocol.

3.2 Performance

We measure the system's performance based on the following metrics:

- *Overall Throughput* (OT), measured in environment interactions per hour
- *Mean Handling Time* (MHT), the average total time of a worker's request
- *Learner Utilization* (LU), the ratio of learner busy to server duration

We have only completed a small set of experiments with the first two using 12 worker pods and 1 learner using an Nvidia T4 GPU and the last consisting of 64 workers and 1 learner. For the latter, the cluster used had 3 compute nodes with 8 Nvidia RTX 2080 Ti

Rollout Length	Workers	OT	LU	MHT
2048 timesteps	12	305,000	62%	11500 ms
1 epsiode	12	225,000	56%	5200 ms
1 episode, lock-free	64	1,420,000	42%	280 ms

Table 1: Performance of the system. The 2048 timestep rollout, despite having higher throughput, results in slower overall learning because workers have less up-to-date policies. Removing locks as in [5] boosts performance massively with an estimated capacity of more than 100 workers and over 3 million timesteps per hour.

GPUs each. Because Kubernetes does not support GPU sharing, we bypassed the resource limit and instead launched separate replica sets with each one assigned to a specific GPU. We found that each of these GPUs can handle at least 4 pods, but need to explore this limit further. We present findings in Table 1. The model architecture used in this experiment was an input with 164 dimensions fed through a 2 layer multi-layer perceptron with hidden sizes of 256 neurons each and a fully connected layer to output a 2 dimensional action. A common practice in reinforcement learning is taking one or multiple gradient updates at each timestep, but this is not feasible in the L2R environment. The 2048 timestep rollout results in near-linear scaling while the single episode rollout is worse at roughly 80% of linear scaling. It is important to note that simulation time is by far the greatest bottleneck of the system. For reference, a complete, human-performance episode takes approximately 180 seconds in the simulator. Some notable points about performance include:

- Single episode rollouts are preferred to keep policies more up-to-date
- The largest gains are from the removal of locks which can be done even if the buffer is not thread-safe by performing addition and sampling in the same thread as in [5]
- This system and model can easily scale to more than 100 workers with a RTX 2080 Ti learner GPU and more than 3 million timesteps per hour
- More computationally expensive models, such as recurrent networks, will increase learner utilization

4. Future Work

There are significant opportunities for future work. With sufficiently large compute clusters, learner utilization will become too high, bounding overall throughput. Extending this off-policy architecture to boost throughput may involve including intermediaries between workers and the learner like [6], using a higher grade or multi-GPU learner, or implementing multi-learner architectures [7, 8, 9]. Furthermore, the system should have broader coverage of reinforcement learning algorithms including on-policy learning, which requires some degree of synchronization, and in particular, distributed PPO as in [10, 11].

References

[1] Brennan Shacklett, Erik Wijmans, Aleksei Petrenko, Manolis Savva, Dhruv Batra, Vladlen Koltun, and Kayvon Fatahalian. Large batch simulation for deep reinforcement learning, 2021.

[2] James Herman, Jonathan Francis, Siddha Ganju, Bingqing Chen, Anirudh Koul, Abhinav Gupta, Alexey Skabelkin, Ivan Zhukov, Max Kumskoy, and Eric Nyberg. Learn-to-race: A multimodal control environment for autonomous racing, 2021.

[3] Alexis Duburcq Kaichao You Minghao Zhang Dong Yan Hang Su Jun Zhu Jiayi Weng, Huayu Chen. Tianshou. `https://github.com/thu-ml/tianshou`, 2020.

[4] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Alex Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Caglar Gulcehre, Tom Le Paine, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*, 2020.

[5] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. Seed rl: Scalable and efficient deep-rl with accelerated central inference. 2019.

[6] Amir Yazdanbakhsh and Junchao Chen. Massively large-scale distributed reinforcement learning with menger, 2020.

[7] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019.

[8] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively parallel methods for deep reinforcement learning. *CoRR*, abs/1507.04296, 2015.

[9] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. *CoRR*, abs/1802.01561, 2018.

[10] Erik Wijmans, Abhishek Kadian, Ari S. Morcos, Stefan Lee, Irfan Essa, Devi Parikh, M. Savva, and Dhruv Batra. Dd-ppo: Learning near-perfect pointgoal navigators from 2.5 billion frames. In *ICLR*, 2020.

[11] Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, S. M. Ali Eslami, Martin A. Riedmiller, and

David Silver. Emergence of locomotion behaviours in rich environments. *CoRR*, abs/1707.02286, 2017.