Exploring Distributed Reinforcement Learning

Sean J. Parker
Department of Computer Science
University of Cambridge
sjp240@cam.ac.uk

Abstract

Distributed reinforcement learning is a field that combines two fields that have experienced remarkable growth over the past decade. It has been applied to various topics and with enough computation power, achieving state of the art results. Due to the nature of RL algorithms, they often require thousands of hours of training, which can only reasonably be achieved by training over a network of extremely powerful machines. This paper investigates three popular frameworks, TensorFlow [1], PyTorch [2], and Ray [3]. Furthermore, we explore the three prototypes' support for distributed learning and each framework's performance in such an environment. The code repository can be found on GitHub, https://github.com/seanjparker/distributed-rl.

1 Introduction

Computational requirements of machine learning models have seen a marked increase in recent years. Modern machine learning models have been developed with more layers, up to billions of parameters. For example, GPT-3 [4] introduced by OpenAI, contains 175 billion parameters. As it is unrealistic to assume the whole model can fit on a single device even with the state of the art model compression techniques, researchers developed various methods to spread the computation over multiple GPUs and even multiple distinct machines. Similarly, in the field of reinforcement learning, often due to the poor sample efficiency of model-free approaches, training RL algorithms require hundreds of hours of compute time [5, 6, 7]. Distributed RL is emerging to be critical for efficiently training multiple models in parallel, the weights learned can be shared between nodes to improve the overall efficiency [5].

Reinforcement learning is a rapidly evolving area where new algorithms are being developed to solve problems in complex, stochastic simulated environments or in the real-world [8]. It is commonly known that most model-free algorithms require an enormous amount of computation time to solve the current benchmarks for RL, such as Atari. Therefore, due to time and resource constraints, PPO [9] was chosen, primarily as it is computationally efficient, simple, and scalable to multiple nodes.

This paper explores distributed reinforcement learning and the level of support and efficiency provided by the most common deep learning frameworks. Additionally, we present both a qualitative and quantitative analysis of using the framework to build and train an RL model. There are two key questions we wish to explore in this paper. Firstly, what are the current offerings for taking a deep learning model and using it in a distributed manner? Secondly, given we have a model defined in the framework, how much work is required to use the model in a distributed setting, and what is the performance impact or gain? In this paper, we explore three different, popular deep learning frameworks, which are Tensorflow [1], PyTorch [2], and Ray [3].

The rest of the paper is organised as follows. Section 2 provides a brief background for reinforcement learning as its extension into distributed RL. Section 3 provides a detailed explanation of details for implementing the project in the various deep learning frameworks. Penultimately, Section 4 presents the quantitative results gathered while exploring the prototypes with a comparison of the performance metrics. Finally, in Section 5, we provide a concise review of the paper and provide closing thoughts on the investigated prototypes

2 Background

Reinforcement learning has been used to solve areas that were once thought were decades away from being solved. Notably, researchers at DeepMind used reinforcement learning to defeat the world champion in the game of Go. More recently, RL has seen applications in protein folding, a critical step to gain a deeper understanding of many biological processes. However, there are examples of RL agents showing what could be considered creativity, exploration, and discovery of new ideas. Nevertheless, the amount of computation required has been doubling every 3 to 4 months [6].

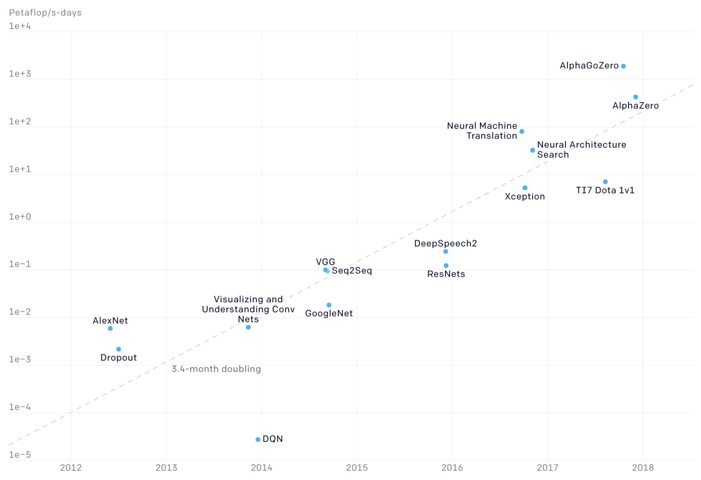


Figure 1: The total amount of compute, in petaflop/s-days, used to train selected results that are relatively well known, and used a lot of compute for their time. OpenAI [6]

Figure 1, produced by OpenAI, shows the computation required for the current state of the art algorithms. It shows the number of Petaflops/day required for training; we can see that the level of compute is not achievable using a single machine. Instead, it requires clusters of powerful machines, working in tandem for multiple days or weeks to train the model, such as AlphaZero and AlphaGoZero. Between the introduced on AlexNet in 2012 to AlphaZero in 2016, it represents a 300,000x increase in compute.

Furthermore, it is possible to scale algorithms over multiple machines, CPU cores, and GPUs. When scaling an RL algorithm over such heterogeneous systems, some trade-offs must be considered. For example, when scaling over multiple GPUs on a single machine, we must consider the bandwidth of the bridge between the units, which strategy will be used (master/slave, ring-based), and which connection technology will be used? The situation becomes more complex when it involves multiple nodes, each with multiple GPUs and CPUs that must be orchestrated to prevent bottlenecks due to communication latency or compute barriers. Consequently, researchers have focused on using asynchronous gradient updates and weight sharing to improve the efficiency over sequential algorithms.

Concerning the three deep learning frameworks explored in this paper, all provide offerings for taking a model written using their respective abstractions and distributing the computation over a set of pre-defined workers. TensorFlow, for example, provides the `tf.distribute.Strategy` package which accepts a TensorFlow model and can distribute it over multiple GPUs and machines. Additionally, PyTorch and Ray have similar offerings as TensorFlow. However, the details as to how they distribute the models is explored further in Section 3.

3 Implementation

3.1 PPO

The following section describes the experiments' implementation using the prototypes of the frameworks as described in the prior section. Firstly, we describe the main algorithm, Proximal Policy Optimisation, commonly referred to as PPO, which was implemented in all three frameworks. There are two main versions of PPO, one using a clipped objective function, described below, with another being using KL-penatly. Also, we use Generalised Advantage Estimation (GAE) [10], which is used to calculate the advantages. Advantages are imperative to PPO as they are used to optimise the value function; advantages indicate if taking an action in some state could provide a better reward. However, as we use policy clipping, it prevents the policy from moving too far from the policy in the previous step using a "threshold" parameter, ϵ.

Algorithm 1: PPO with Clipped Objective

```
Input: initial policy parameters $\theta_0$, clipping threshold $\epsilon$
for $k$ = 0, 1, 2, ... do
    Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
    Estimate advantages $\hat{A}_t^{\pi_k}$ using GAE with the value function $V_{\phi_k}$
    Compute policy update
        $$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}_{\theta_k}^{\text{CLIP}}(\theta)$$
     by taking $K$ steps of minibatch SDG (using Adam), where
        $$\mathcal{L}_{\theta_k}^{\text{CLIP}}(\theta) = \mathbb{E}_{\tau\sim\pi_k}\left[\sum_{t=0}^{T}\left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t^{\pi_k})\right]\right]$$
     Fit value function using using MSE loss using minibatch SDG
        $$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T}\sum_{t=0}^{T}\left(V_{\phi}(s_t) - \hat{R}_t\right)^2$$
end
```

Using PPO brought three key advantages to the project, it was deliberately designed to be sample efficient, easy to implement, and stable to a wide range of hyperparameter choice. Its predecessors, such as TRPO [11], required off-policy learning using replay memory, which is often challenging to implement efficiently. Moreover, PPO is an on-policy algorithm that is compatible with SDG, meaning that it can be used in deep learning frameworks in the standard way.

After the introduction of PPO by Schulman et al. [9], there were further improvements to the algorithm, notably, it was adjusted to take advantage of GPUs by leveraging vectorised environments that use rollout workers to gather a collection of environment steps at once, it brought a 2-3x speedup over the original implementation. Additionally, DD-PPO[12] was introduced to take advantage of distributed workers, by making the gradient updates asynchronous using a parameter server, it removes the slowdowns due to barriers in gradient synchronisation across workers.

3.2 Communication Backend

PyTorch has out of the box support for both GLOO and NCCL, as well as MPI when built from source [13]. When using the `torch.distributed` package, the author can choose to use one of the supported communication backends. On the other hand, TensorFlow is more limited in terms of the supported backends. By default it uses gRPC via HTTP for performing distributed operations; TensorFlow also supports using NVIDIA NCCL when performing `AllReduce` across GPUs [14]. Often, TensorFlow will choose the optimal communication technology available, or the author can write their own implementation. However, similar to PyTorch, it is possible to use the Python package `mpi4py` to manually perform distributed operations. Moreover, MPI supports both using the operations on both CPU and GPU devices. In this project we use the PPO algorithm on CPU cores, thus, it is important we use a protocol that supports such operations.

3.3 Distribution Strategies

Both TensorFlow and PyTorch offer excellent flexibility in terms of the way in which computation is distributed over a set of workers, this comes at the cost of effort for the author as they need to design and implement the distribution strategy. On the other hand, Ray abstracts all the complexity of communication and synchronisation behind an API with which the research can interact. In this project, TensorFlow and PyTorch use the MPI (Message Passing Interface), via OpenMPI [15], which offers support for both parallel processing on a single machine and across separate machines via SSH. The main functions which we use, `AllReduce`, is highly efficient and allows all workers to receive the same result from a computation using data from every worker. Figure 2 summarises the architecture of the Ray Actor and Workers.

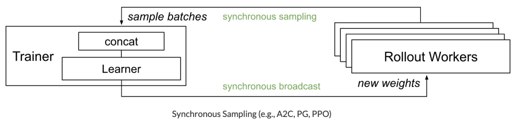


Figure 2: Architecture of how Ray uses Rollout workers to sample a minibatch

Ray takes a different approach. We first introduce the idea of a “Rollout Worker”, this is a separate process, linked to each worker process that gathers a minibatch of data from the environment by taking a number of predefined steps which is used in the worker process to perform SDG to train the policy and value networks. Furthermore, internally, Ray can schedule the actors much more efficiency and also handles message passing and synchronisation that optimises message delivery, by exploiting worker locality, as well as disk operations. Importantly, Ray can support heterogeneous systems where a system may have both CPUs and GPUs, it can take full advantage of the hardware whereas a library such as OpenMPI [15] often requires symmetric, homogeneous structures unless the programmer explicitly handles coordination and fault tolerance.

4 Experiments

In the following sections, we present the experimental results produced throughout the project. We describe and present the experiments’ results for training PPO to act in the OpenAI gym environment called “Lunar Lander”. This environment was chosen as it provides the right balance between the environment’s complexity and observation space. It is not necessary to use a CNN to learn directly from the raw frames. Instead, we can directly access the lander’s position in the world, which means both the policy and value networks can be simple fully-connected networks. We consider that this environment is “solved” when an agent achieves an average score of 200 [1].

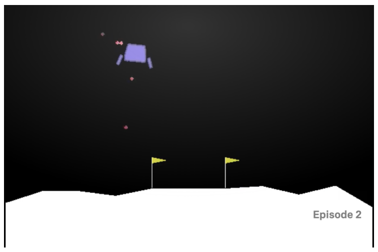
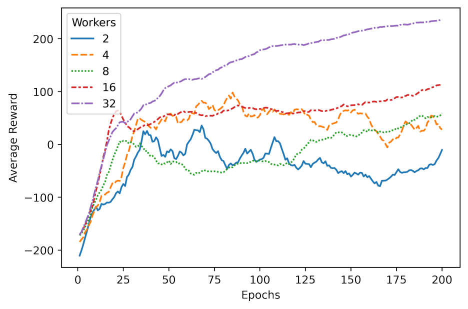


Figure 3: Frame from the `LunarLander-v2` environment in OpenAI Gym

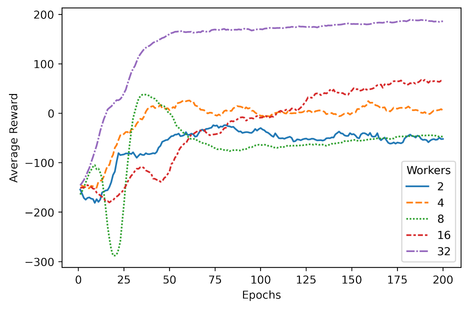
Figure 4a shows the results for training our actor to play Lunar Lander using PPO in PyTorch. It is trained using a range of 2 to 32 workers, the workers are split among multiple 8 core machines in Google Cloud. Specifically, we used custom-sized E2 machines with 8 physical cores and 4GB

[1] `https://gym.openai.com/envs/LunarLander-v2/`

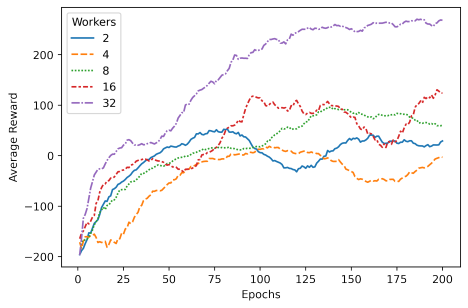
RAM per VM. i.e. with 32 workers, we have 4 seperate VMs running (4x8 geometry), where one is designated manager, the other three are workers [2].



(a) Torch



(b) TensorFlow



(c) Ray

Figure 4

We can see from the results, shown in Figure 4a, that PPO offers significant improvements when scaled to over 16 workers. However, the agents are only trained for a modest budget of 200 epochs, with $^{4000}/_{\text{Workers}}$ steps per epoch per worker which in total results in 800,000 environment frames. We can see that we solve the environment with 32 workers in all three frameworks, with the second highest scoring agents being those trained with 16 workers in all cases. Simply, when we increase the number of workers, we see a significant improvement in the mean score. Although we use the same number of steps in the environment no matter the number of workers, as each worker is randomly seeded, the workers generate unique experiences. Therefore, by averaging the gradients every minibatch across all workers, it helps to improve the algorithm's stability and performance. Hence, we can infer that scalability is key to the improving performance of the PPO algorithm.

We also performed tested the TensorFlow PPO implementation, with the results in Figure 4b. We can see a similar trend to that of the PyTorch implementation from the data, showing that both frameworks offer excellent support for horizontal scalability. However, in terms of the ease of implementation, in my opinion, the TensorFlow implementation was far more complicated as we needed to average the gradients across the workers in the optimiser manually. Therefore, since TensorFlow does not offer direct access to the gradients, we needed to rewrite the Adam optimiser to apply the averaged gradients and ensure the gradients were successfully synced every N steps.

Finally, Figure 4c shows the results using the same experiments using 2, 4, 8, 16 and 32 workers. Notably, when using 32 workers in Ray, we achieve the highest score from all experiments, a reward of over 250 points on average, far exceeding the requirement for solving the environment. It is possible that the way in which Ray collect samples from the environments using multiple rollout workers which are used to compute the average gradients contributes towards this improved performance. On the other hand, we observe slightly unusual behaviour with some workers that have a sharp decrease in performance before improving, we do not observe the same results in TensorFlow or PyTorch. After

[2]It is possible to launch a single node with over 32 logical cores in GCP, however, to demonstrate the frameworks ability to distribute workload, we selected to run more workers, to reduce costs. `https://cloud.google.com/compute/docs/machine-types#e2_custom_machine_types`

further inspection of the Ray PPO implementation we observe that the original authors performed substantial optimisations that are not present in the PyTorch and TensorFlow implementations such as Advantage normalisation, value-function gradient clipping and, importantly, larger batch sizes.

5 Conclusion

In conclusion, we have explored the three different frameworks for deep learning and distributed learning. Using an example environment from OpenAI Gym, we have shown that we can achieve far better improvements over using a single machine or thread with the correct optimisations and algorithms by scaling an algorithm over distributed systems. Furthermore, we also note that when scaling across machines, although we pay a penalty in terms of communication due to gradient transfer and synchronisation, the algorithms trains faster due to using fewer minibatch steps per worker.

In terms of future work, it would be interesting to explore other features of the frameworks such as parameter servers, in combination with DD-PPO it may offer significant improvements. However, at the time of writing, such features are considered experimental in both the TensorFlow and PyTorch frameworks.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[3] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications, 2018.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[5] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks, 2017.

[6] Dario Amodei and Danny Hernandez. AI and Compute. `https://openai.com/blog/ai-and-compute/`, May 2018.

[7] 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, 2015.

[8] OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving Rubik's Cube with a Robot Hand, 2019.

[9] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, 2017.

[10] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

[11] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[12] 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, 2020.

[13] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch Distributed: Experiences on Accelerating Data Parallel Training, 2020.

[14] Abhinav Vishnu, Charles Siegel, and Jeffrey Daily. Distributed TensorFlow with MPI, 2017.

[15] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.