Off-Policy Reinforcement Learning that Prioritizes Value Function Improvement for Speedup

Ethan Brooks
Computer Science
University of Michigan
Ann Arbor, USA
ethanbro@umich.edu

Idris Hanafi
Computer Science
University of Michigan
Ann Arbor, USA
ihanafi@umich.edu

Jake DeGasperis
Computer Engineering
University of Michigan
Ann Arbor, USA
jakedeg@umich.edu

Richard Higgins
Computer Science
University of Michigan
Ann Arbor, USA
relh@umich.edu

***Abstract*—Parallelizing Reinforcement Learning (RL) is a pertinent venture in an age where leading research groups dedicate thousands of years worth of simulation time to tasks. Existing work explores how to distribute components of these systems across machines to successfully train them. We explore an orthogonal idea, mainly how to divide up a limited amount of computational resources asynchronously and effectively when training an RL agent. One goal is to train a network faster than with a standard division or allocation of resources. The other goal is to enable an off-policy reinforcement learning algorithm that can scale to large computational allocations. We implement our system as a version of Deep Deterministic Policy Gradient (DDPG) and run it in both synchronous, multiprocess, and distributed settings. We show that allocating more GPU capacity to an asynch value process, by using a large batch size, leads to a faster, improved, and more stable training regimen.**

***Index Terms*—reinforcement learning, parallel computing, neural networks, machine learning**

I. Introduction

Reinforcement learning is a sub-discipline of machine learning focused on learning behaviors – mapping observations to actions that maximize cumulative reward from a given reward function. Deep reinforcement learning leverages the expressive capabilities of neural networks to learn behaviors in environments with high-dimensional observation spaces. Deep reinforcement learning has recently demonstrated mastery of complex games, previously thought intractable for artificial intelligence. For example, in a recent blog-post [16], OpenAI reported state-of-the-art performance on the video game *Defense of the Ancients (DOTA) 2* by deploying large fleets of parallel learning agents on 256 GPUs and 128,000 CPU cores. Similarly, Google DeepMind demonstrated state-of-the-art performance on StarCraft II [9] pitting hundreds of reinforcement learning agents against each other, training on 16 TPUs apiece.

Both of these examples use *on-policy* reinforcement learning vs. *off-policy* reinforcement learning. On-policy reinforcement learning is restricted to training on experience *drawn from the current policy*. This means that older experience cannot be reused after learning has resulted in changes to the policy. In general, on-policy algorithms work by performing gradient descent on an unbiased sample estimates. These sample estimates grow more accurate with more samples in accordance with the law of large numbers. Consequently, on-policy algorithms scale naturally, since large fleets of agents can be used to concurrently generate experience and improve the accuracy of sample estimates.

In contrast, off-policy algorithms usually work by climbing the gradient of a *value function estimate*. The value function maps observations to *values*, which roughly measure the goodness of the state that the observation represents. It is always best for an agent to take the action that leads to the state with the highest expected value. However, the value of a state is in general unknown, and must be estimated (by a neural network, if we are using deep reinforcement learning). Since these estimates are approximate and tend to be inaccurate, climbing the gradient of the neural network estimator will improve the *estimated* value but may not improve the actual value.

Furthermore, because off-policy algorithms are able to reuse old experiences, they tend to be very sample-efficient, meaning that they require little experience to train, relative to on-policy algorithms. Both the inaccuracy of value estimates and the sample-efficiency of off-policy algorithms make large fleets of experience-generating agents like those used by OpenAI and DeepMind less beneficial.

That said, off-policy learning has unique benefits that are not easily reproducible by on-policy learning. If generating large amounts of experience is prohibitively slow or costly (as can be the case with physical robots), off-policy training may be the only feasible solution. OpenAI's DOTA agent plays 180 years worth of games against itself every day by accelerating the game engine. Obviously this quantity of experience is not accessible to agents acting in the real-world.

Our project asks the question, how do we use hardware parallelization to leverage large amounts of computation with off-policy learning? We posit that, just as more accurate gradient samples improve learning for on-policy agents, more accurate value estimates will accordingly improve learning for off-policy agents.

We therefore propose an algorithm that trains value functions *in parallel* with other components of the algorithm (generating experience and training the policy). This allows a larger proportion of resources to be allocated to value function learning. We hypothesize that value function learn-

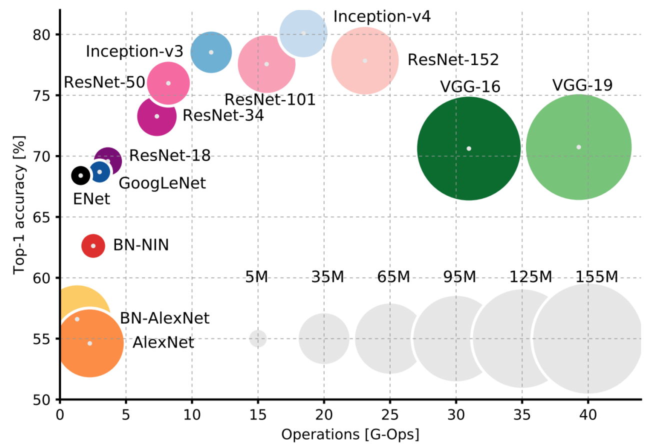

Fig. 1. Different neural network architectures and their image classification performance accuracy compared to the G-Ops they perform. Dot size represents the number of parameters.

ing is the component of off-policy learning that best scales with increased resources and is therefore most amenable to parallelization.

Value functions are trained using a bootstrapped[1] variant of supervised learning. Research [8] has shown that deep learning scales well with large-scale distributed systems. Indeed, in the domain of supervised learning, large numbers of floating-point operations over time have proved capable of radically improving performance. We apply these techniques to the problem of value function learning and find that they yield benefits equivalent to those gained by large-scale parallel actor training with on-policy algorithms.

II. Related Work

Many improvements have been done in parallelizing Reinforcement Learning methods. Some of the earliest ideas date back to [4] in the simplest form of Distributed Dynamic Programming. IMPALA [10] took a different approach to this problem. IMPALA can be broken down into three components. First, the architecture consists of many actors that are continuously generating experiences distributed across different machines. Second, it utilizes GPUs to accelerate learners that retrieve batches of trajectories from said actors. Third, it uses the V-trace, a method to offset the policy-lag between an out-of-date actor that has not been updated and the learner(s). This V-trace is only a requirement for it being an on-policy learner, in proposing an off-policy learner we have discarded it and emphasized an updated value network.

Another popular aspect of parallel reinforcement learning is the parallelization of stochastic gradient descents, such as Soft actor-critic and more [2], [7], [15], [19]. These techniques are part of a rich and active literature on parallel optimization. Recently, the advent of scalable data analysis has brought

[1]The target for supervised learning is a function of the current value estimate. Unlike in traditional supervised learning, these targets are biased estimates, not ground-truth.

Fig. 2. Screenshots from the Pendulum, MountainCar, and HalfCheetah environments we use.

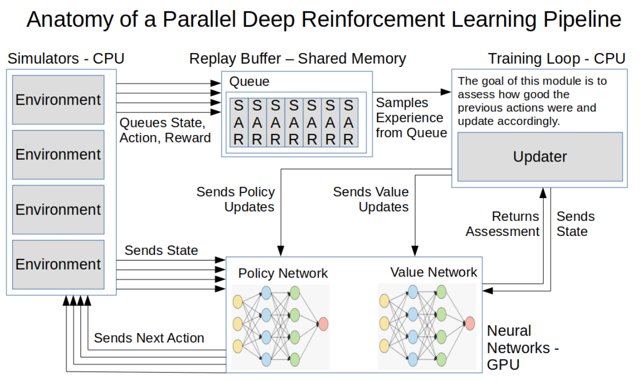

Fig. 3. This overview of the deep reinforcement learning pipeline shows how simulators feed experience to a queue and how this queue is sampled to improve policy and value neural networks.

about the lock overhead issue. Popular attempts to circumvent the lock overhead issue includes HOGWILD! [18] and Round Robin [13]. Besides parallelization, the usage of GPUs [5], [6], [12], [17] has also faced a surge in recent years. Unfortunately, these methods cause a CPU-to-GPU transfer bottleneck when the model does not fit in GPU memory.

We note the work of the asynchronous advantage actor-critic (A3C) [14]. A3C relies on parallelizing learners and collecting updates to improve training stability. Since then, upgrades and work based off of A3C has been done including a hybrid CPU/GPU A3C (GA3C) from [3], distributed BA3C [1] and ultimately, IMPALA. Two notable and successful implementations that used this distributed idea, were AlphaGo, which trained on 40 search threads, 48 CPUs, and 8 GPUs, and the OpenAI Five (for DOTA 2), which trained using a scaled-up version of Proximal Policy Optimization running on 256 GPUs and 128,000 CPU cores.

III. Approach

In IMPALA [10], we see how the authors' use of computation with on-policy algorithms revealed significant parallel speedup opportunities. Indeed, the impressive results that this algorithm demonstrates suggest opportunities for significant improvement through parallelization in other areas in reinforcment learning — most conspicuously, off-policy learning.

Our current hypothesis is that the GPU on which the policy and value networks operate is severely underutilized. Currently, the policy network is only able to operate with small batches of observations. This is because only a few environments can be run in parallel, and most of the common environments run on the CPU. If there are only ever small

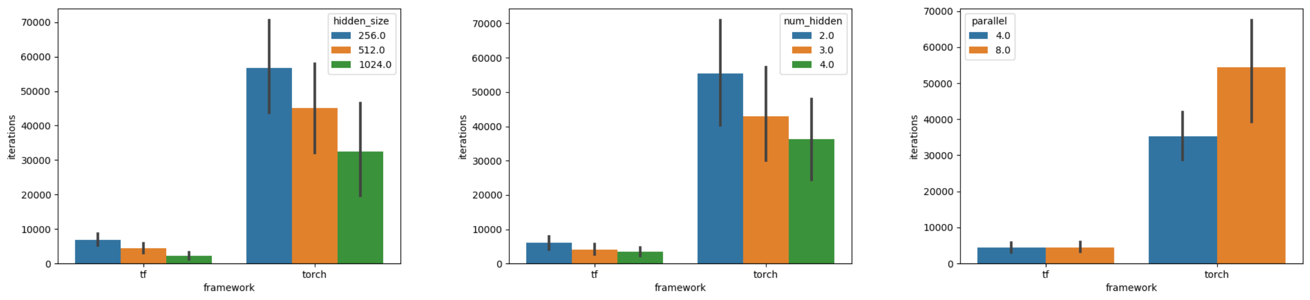

Fig. 4. Completed iterations on GPU given hidden layer size, # of layers, and process count.

batches of policy observations being fed into the policy network, it leaves open the opportunity for using any remaining compute in running larger value network workloads. Typically, updates from both of these networks happen synchronously, with equal batch sizes for policy and value network updates.

Value network training batches usually come from a history of saved experience, called a experience replay buffer. This experience replay lets one train a value network on past experiences with relatively modest demands for new experiences from the environment. We suggest that this imbalance is not being exploited by current approaches, suggesting that we can create different computation configurations of policy and value networks to best utilize the GPU.

Value networks are trained in a semi-supervised manner, regressing toward targets, using gradient descent in most current algorithms. This means that all of the supervised learning literature, from scenarios where one has the data sample and already knows the right answer, applies to optimizing this network. The Reinforcement Learning community doesn't typically stay well-versed in the supervised learning community literature and the optimizations that the Computer Vision community uses in their training pipelines. We then research and implement a few of these optimizations to improve performance. Exploring network architectures here that use more or less parameters but can achieve the same result is also in scope. If we can have a larger network because of compute capabilities that are otherwise bottlenecked due to a slow environment, then it would be worth exploring this change.

Finally we explore concurrent behavior. As humans in the world, we operate on the data we have seen in real time, processing it to make decisions as time unfurls. In the Reinforcement Learning community, agents receive observations from the world, process them, and then make actions that step the environment forward in some manner. We explore running these processes in parallel.

Our first goal in this project was to analyze and benchmark some existing reinforcement pipelines. Second, it was to test how our hypothesized new off-policy variant of IMPALA improves the resource utilization, convergence rate of agents, and overall performance of a Reinforcement Learning system with computationally expensive environments and agents.

IV. Initial Experiments

Here we outline our first experiments, analyzing and benchmarking simple reinforcement learning systems. Then we segue into our main experiments, progress, and overall project approach.

A. *Initial Approach*

Initially, we built a minimal reinforcement learning loop in four testbeds. Our testbeds were minimal runtimes running on a CPU/GPU and in PyTorch and Tensorflow. We profiled how many parallel executions of a dummy loop could be completed in a certain wall clock time. We also trained a basic asynchronous Deep Q-Network (DQN) on the MountainCar problem provided by the OpenAI Gym in PyTorch. This system used four separate processes, two running the environment and two updating the value network. We formalized the anatomy of a reinforcement learning pipeline and present it below, as a flow diagram of a typical training loop.

B. *Initial Techniques*

With Tensorflow, we found we can readily use extra servers or processors to increase the available parallelism. Tensorflow supported this mechanism by 1: allowing specification of the cluster tf.train.ClusterSpec and 2: passing the cluster to the tf.train.Server for each task. We profiled this distributed setup and ran experiments to measure time elapsed.

With Pytorch, multiprocessing is supported as a drop in replacement for the Python multiprocessing API. We profiled a simple setup for comparison to Tensorflow and then ran a DQN network in Pytorch on the MountainCar environment. We implemented Hogwild! across multiple threads to observe the convergence rate and wall clock running time for the DQN on MountainCar. In this system, a single neural network is shared across four processes. Two processes are responsible for taking steps in the environment. Two other processes are responsible for taking the experience and making updates to the network with it.

C. *Initial Results*

In our results, we found that PyTorch handily beat Tensorflow across the board. It was able to complete more iterations in the same amount of time, even accounting for startup costs.

This held across both differing batch sizes, number of neurons per hidden layer, number of layers, and number of processes running in parallel. It also held across both the CPU and GPU. We think these initial results were related to inefficienies in our implementations.

We find that a sequential run of a simple tensorflow calculation can be ran about approximately 50,000 times within a minute. Under the distributed version with only 2 nodes, we were able to run the same tensorflow calculation 70,000 times within a minute.

With a single process, we were able to run the calculation 134,392 times in one minute. With 2 and 4 processes, we ran the calculation 243,629 and 337,275 times, respectively, in the same amount of time, showing that this function has a lot of potential for parallelization.

With 8 processors we were able to call the dummy run method 110,000 times in the span of a minute. It took 132 seconds to train 5000 episodes, eventually converging within 1 of the optimal reward value.

D. Selected Next Methods

Although we implemented a basic version of the above diagram in PyTorch, there was more to be done. First, we needed to extend the implementation to include an off-policy component. Then we needed to select/implement both a more difficult environment and larger network in order to explore cases where different ratios of time between environment steps and network updates lead to different training outcomes. Finally, we needed to explore the theoretical impact of our off-policy choices and analyze the paradigm we have created compared to other existing parallel training paradigms such as A2C and A3C.

To resolve the off-policy requirement, we switched to a system called Actor-Critic learning. In Actor-Critic learning, we have two networks, an actor and a critic (policy and value). The actor receives states from the environment and outputs the best action. The critic takes a state and determines how good it is. The critic is vital to training the actor, due to credit assignment. In a long game where the only reward is winning or losing, you need to know where, specifically, you went wrong. This is the job of the critic. Our proposition to improve the parallel training was to devote some processes to the slow running of an environment (sometimes a complex physics simulator) while devoting more to updating the neural network, using saved past historical data.

In our next phase, we benchmark the performance and identify bottlenecks in a multiprocessor and distributed PyTorch RL scenario. We also introduce the above off-policy RL pipeline and analyze how a workflow benefits from parallelization. To perform these experiments, we use our local machines, existing provisioned GPU enabled machines, and a small cluster of GPU machines.

V. Design

Our design approach was to implement a Deep Deterministic Policy Gradient (DDPG) system in PyTorch. DDPG is an off-policy and actor-critic algorithm that uses a deterministic target policy. This learning design entails three processes: the environment, value, and policy. The environment process is a physics simulator which maps a state and action pair to a reward. Our goal is to choose the right actions that lead the environment to generate the maximum reward. Every step of the environment, an experience of the form (state, action, reward, next_state) is appended to the experience replay. The experience replay is then randomly sampled from to break up any temporal correlations that exist within training episodes. The actor-critic portion comes into play by having two other processes, a value (critic) and policy (actor) process. A set of value and policy networks, stored within their respective processes, are then treated as target values to calculate the next-state reward value for the environment. This design allows the value and policy process to run separately from the environment but with the condition of frequent network weight exchange. The role of the value and policy process is to then exchange their parameters with each other (a soft update) to slowly track those of the learned networks. For our

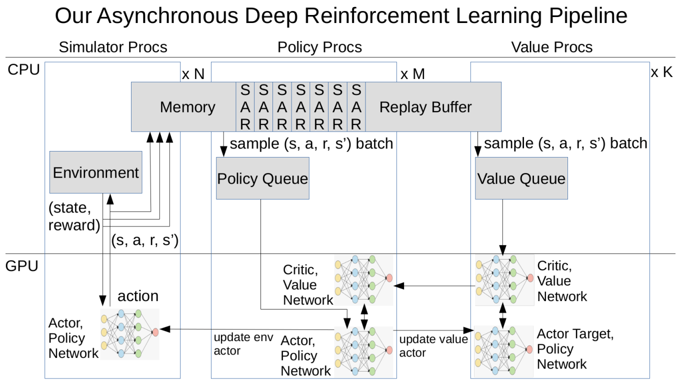

Fig. 5. Anatomy of our asynchronous RL pipeline. Note that we can use any number of simulators, policy, and value processes. We share a memory replay buffer as either a shared object or a distributed database. Separate processes adjacent to networks sample from this shared memory replay buffer to create local queues for consumption by the networks during training.

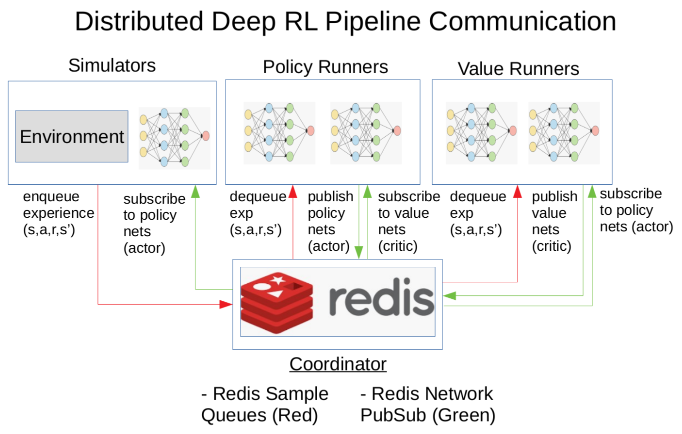
Distributed Deep RL Pipeline Communication
Simulators
Policy Runners
Value Runners
Environment
enqueue experience (s,a,r,s')
subscribe to policy nets (actor)
dequeue exp (s,a,r,s')
publish policy nets (actor)
subscribe to value nets (critic)
dequeue exp (s,a,r,s')
publish value nets (critic)
subscribe to policy nets (actor)
redis
Coordinator
- Redis Sample Queues (Red)
- Redis Network PubSub (Green)

Fig. 6. How our distributed RL pipeline uses queues and a PubSub system to exchange experience and network weights across the internet.

parallelized implementation, the three processes require three different lines of communication, 1: policy process sends their current actor network parameters to the environment process, 2: policy process sends their current actor network parameters to the value process, 3: value process sends their current critic network parameters to the policy processor.

A. Multiprocessing

The first domain of concurrency we explored was multiprocessing. We separated the training process into an environment, policy, and value process that could be run semi-independently. All processes used a shared memory queue for storing experience replay. We used the Python and PyTorch multiprocessing libraries for this section of the project. We used a Python queue for storing experience replay, while using shared memory to share the actor and critic neural networks between the policy and value processes. This shared memory paradigm for both queues and network weights allowed us to have significant control over the batch size across the policy and value processes.

B. Distributed

The second domain of concurrency we explored was employing a distributed system to achieve better utilization of computational resources. As previously stated, we separated the training process into an environment, policy, and value process that could be run semi-independently. However, these three processes still needed to communicate with one another, but doing so over multiple machines required additions to our DDPG pipeline.

First, the environment needed to be able to provide the value and policy processes with observations through the replay buffer. Using a ring buffer or thread-safe queue is no longer an option when running processes in a distributed setting. To solve this, we used a Redis database server that all the processes could access. Given that all database systems abide by the ACID (Atomicity, Consistency, Isolation, Durability) transaction properties, our replay buffer need not be concerned about any memory consistency issues. The environment process would insert observations into the database along with a constantly incrementing key, while the policy and value processes would generate random key values and retrieve the data that was stored at the aforementioned keys. This provides the added benefit of always having all observations available to be sampled, whereas a ring buffer eventually must overwrite older observations. This is especially important when using an off-policy algorithm like DDPG as adjacent experience samples are highly correlated and thus violate the independent and identically distributed assumption made by supervised learning. To avoid potential bottlenecks caused by the database, we also implemented two daemon processes that utilize queues to insert into and sample from the database. This removed the database from the critical path of our distributed DDPG pipeline.

Additionally, instead of having each process communicating via storing message values in the database, we opted for a message passing interface (MPI) provided from redis pubsub. Redis pubsub is comprised of two components: a publisher (sender) and a subscriber (listener); synonymous to the multiprocessor setting, the environment process subscribes (i.e. listens) for the policy network parameters published (i.e. sent) by the policy process. On the other hand, the value process subscribes for the policy network parameters published by the policy processor, and vice versa. The only downside of utilizing an MPI is that messages are sent across a distributed network. Therefore, bottlenecks may occur if one of the systems experiences high network traffic.

VI. Experimentation Methodology

To run the above proposed experiments, we used a variety of hardware resources. We used group members' GPU-enabled machines for multiprocessing testing, as well as the University of Michigan's RLD6 and Omen cluster. These multi-GPU machines were an ideal testbed for running our asynchronous experimentation. We ran our environmental processes, experience replay processes, and Redis server on Omen's and RLD6's CPUs. Meanwhile, we ran our value and policy processes on different GPUs of the machines. This allowed us to maximize the computational resources being utilized by these processes.

We primarily evaluated our algorithms by observing the returns (cumulative rewards) and plotting these against time and time-steps.[2] To monitor these values, we used printouts to the terminal and also relied heavily on Tensorboard, a free software tool for logging and plotting values from currently running processes. It should be noted that both clock time and time-steps are problematic, particularly when applied to parallel algorithms. Clock-time is heavily dependent on the specific hardware supporting the algorithm and can be impacted by other processes running on the same machine. Meanwhile time-steps, the usual substitute for clock-time, are less meaningful in a parallel setting when one environment time-step can correspond to a variable number of concurrent network updates.

A rigorous presentation of performance for a parallel algorithm would need to consider time-steps along with the number of gradient updates, as well as clock-times for runs on pristine GPUs not shared by other processes. We defer this to future work.

VII. Results

Our main experiments explored the impact of varying the batch-size for the value process. The value process batch-size is the number of data points (each data point corresponding to one environment time-step of experience) per gradient update of the value network. Supervised learning suggests that larger batch-sizes yield more accurate models at the expense of increased computation. We chose to focus on scaling batch-sizes instead of increasing the number of environments and the

[2]Reinforcement learning generally discretizes time in the training environment. Environment steps are the most common value to measure performance against, since they abstract away computational details that affect clock-time.

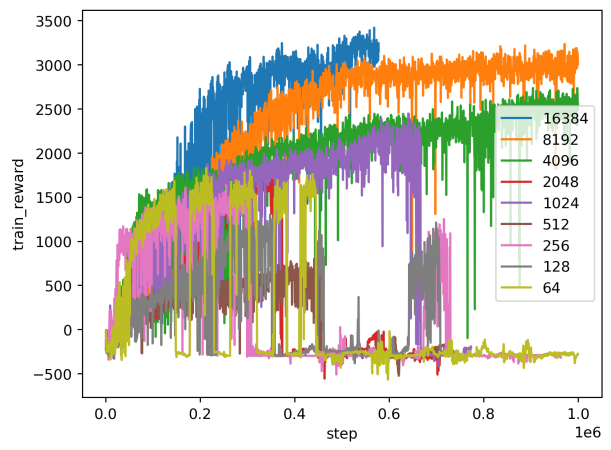

Fig. 7. Improved and more stable RL training performance as batchsize and resources increase.

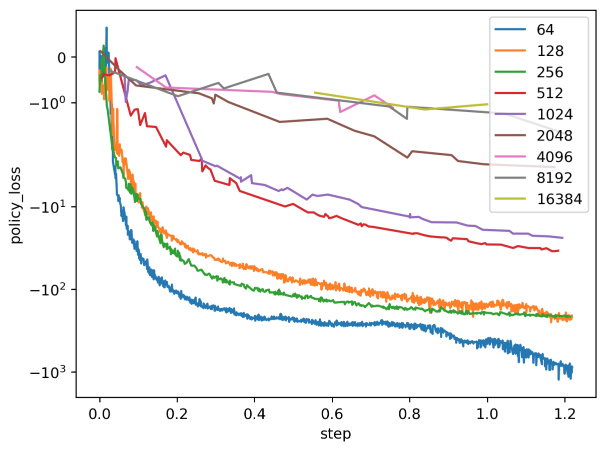

Fig. 9. Multiprocess policy loss as a function of normalized step count.

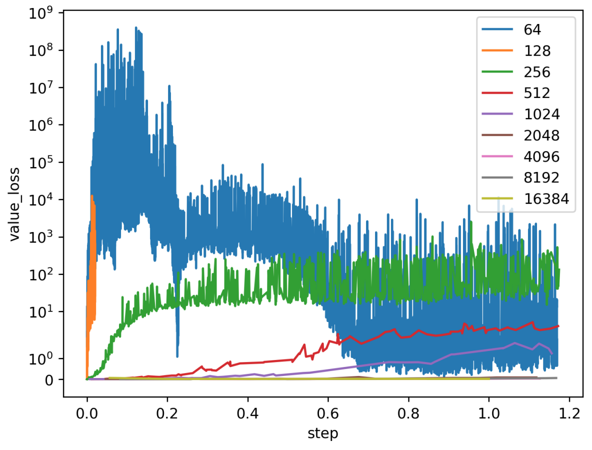

Fig. 8. Multiprocess value loss as a function of normalized step count.

quantity of behavior generated because, in general, off-policy learning algorithms are relatively sample-efficient because of their ability to reuse past-experiences. Of the two networks, we chose the value network to target with batch increases because the effectiveness of policy network updates is dependent on the accuracy of the value function — the policy network tries to improve the values of the actions it generates, which is only beneficial if those values are accurate.

For the most part, our experiments confirmed these assumptions. First note that in figure 8, batch-size dramatically affects the variance of the value loss. This seems to correlate, in figure 7, with the propensity of an algorithm to diverge (indicated by lines that slump to zero and stop learning). Relative to value losses, policy losses (figure 9 are well behaved for all batch-sizes, which is to say that increased policy batch-sizes would probably slow learning without much benefit in terms of stability. However, this is an assumption that must be confirmed by future work. Most importantly, we note that returns using large value batches consistently outperformed returns from smaller value batches in figure 7.

VIII. Conclusion

The outcome of this project is a working algorithm capable of scaling to large computational resources. Both implementations, multiprocessing and distributed, allow adjustments to the value and policy batch sizes (which slows down gradient updates for the associated networks) without blocking the forward progress of training for the other processes. For example, with a very large batch-size of several thousand, gradient updates for the value process can take seconds. If environment steps and policy-update steps were blocked during this time, the delay would be intolerable.

The distributed implementation allows all of the resources of a GPU or CPU to be allocated to any one of the three processes, allowing much larger batch-sizes than are possible if all processes have to share the resources of a single machine.

IX. Future Work

Nevertheless, the current setup is not able to scale any one of the processes to multiple machines, in the sense that multiple machines cannot participate in providing updates to a shared value network, for example. Though the actual update must be performed on a single machine, it is possible to have multiple machines generate the gradients to be accumulated and applied to the network on a single machine, a la [14]. Implementing this feature would be an important next step to a truly scalable off-policy reinforcement learning algorithm.

Also, extensive hyperparameter search needs to be done to ascertain the right balance of computation between value, policy, and environment processes. We have offered evidence that increasing the batch size of value function update can improve performance, but we have not had the opportunity to investigate how scaling the other two processes — large policy batch sizes or multiple concurrent environment steps — might benefit learning or potentially interact.

Finally, before submitting this work for publication, it will be worth applying the paradigm to a state of the art algorithm like SAC [11] instead of DDPG. We will need to compare the results with existing nonparallel approaches to reinforcement learning.

X. Contributions

Ethan: Developed the initial benchmark Torch/Tensorflow code. Wrote the distributed implementation. Ran the final distributed and multiprocessing experiments. Developed the performance graphs. Wrote several sections of the report.

Idris: Ran initial profiling for distributed Tensorflow. Focused primarily on the distributed portion of the implementation, more specifically, the message passing interface (redis pubsub) between value and policy process. Then later focused on the algorithm speedup by pipelining database accesses. **Jake:** Ran initial profiling for multiprocess Tensorflow. Worked with exchanging experience between environment process and policy/value processes for distributed implementation. Wrote portions of report.

Richard: Ran initial profiling for PyTorch. Developed initial implementation of DQN in PyTorch. Developed synchronous, main, and multiprocessing implementation of code in PyTorch. Wrote portions of proposal, milestone report, milestone slides, and final report. Made diagrams. Made large portions of the poster. Spent too much time arranging figures.

References

[1] Igor Adamski, Robert Adamski, Tomasz Grel, Adam Jedrych, Kamil Kaczmarek, and Henryk Michalewski. Distributed deep reinforcement learning: Learn how to play atari games in 21 minutes. In *International Conference on High Performance Computing*, pages 370–388. Springer, 2018.

[2] A. Agarwal and J. C. Duchi. Distributed delayed stochastic optimization. In *Decision and Control (CDC)*, page 54515452. 2012.

[3] Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. Ga3c: Gpu-based a3c for deep reinforcement learning. *CoRR abs/1611.06256*, 2016.

[4] Dimitri P. Bertsekas. *Distributed dynamic programming*. Automatic Control, IEEE Transactions on, 27(3):610 616, 1982.

[5] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber. *Deep big simple neural nets excel on handwritten digit recognition*. CoRR, 2010.

[6] G. Dahl, D. Yu, L. Deng, and A. Acero. *Context-dependent pre-trained deep neural networks for large vocabulary speech recognition*. IEEE Transactions on Audio, Speech, and Language Processing, 2012.

[7] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng. Large scale distributed deep networks. In P. Bartlett, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, page 12321240. 2012.

[8] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[9] Deepmind. Alphastar: Mastering real time strategy game starcraft ii. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/, 2018.

[10] 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.

[11] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[12] G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. *Deep neural networks for acoustic modeling in speech recognition*. IEEE Signal Processing Magazine, 2012.

[13] J. Langford, A. J. Smola, and M. Zinkevich. Slow learners are fast. In *Advances in Neural Information Processing Systems*. 2009.

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

[15] F. Niu, B. Recht, C. Re, and S. J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *arXiv preprint arXiv:1106*, 5730, 2011.

[16] OpenAI. Openai five. https://blog.openai.com/openai-five/.

[17] R. Raina, A. Madhavan, and A. Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *ICML*. 2009.

[18] 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*, pages 693–701, 2011.

[19] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems*, page 25952603. 2010.