Parallelization of the DQN Algorithm

Robert Moore, Jacob Hofstein, Colin Chu

April 26, 2021

1 Abstract

We present an open-source implementation of a distributed Deep Q-Network algorithm (DQN). The design is based on DeepMind's *Gorila* architecture [1], which they used with a massively distributed system to outperform GPU training. We replicate their experimental evaluations with a smaller cluster to determine whether the gains from parallelization are significant enough to accelerate machine learning even on modest hardware.

2 Introduction

Deep learning models have been used to achieve state-of-the-art results across numerous domains and tasks, ranging from 3D object detection [2] to text generation [3] to speech recognition [4]. A sufficiently complex neural network can accurately model an arbitrary function, but deep supervised learning methods are limited by the need for massive amounts of human-labeled training data. Reinforcement learning techniques avoid this limitation by enabling an agent to generate its own training data by interacting with a simulated environment. The groundbreaking success of programs like *AlphaGo* [5] and its successors like *MuZero* [6] has brought reinforcement learning to the forefront of research as a powerful approach for difficult problems.

Reinforcement learning techniques avoid the need to manually gather a large training data set, but they can still be limited by the agent's speed in interacting with the environment. Distributed architectures can simulate many agents and generate training data in parallel, thereby exploring a greater portion of the total state space during each training step. The parameters of the neural network can also be stored and updated in a distributed fashion, providing multiple avenues for parallelization to speed up the training process.

3 Related Work

This paper is principally a replication of the work described in Nair et al 2015 [1]. Other closely related work include Ong et al. 2015 [7] and Dean et al. 2012

[8].

4 Background

4.1 DQN Components

The DQN algorithm combines the traditional Q-learning algorithm with a neural network. Q-learning is a reinforcement learning algorithm that adopts a policy of maximizing total reward over all actions. Interestingly, though, Q-learning is off-policy; our target policy is a greedy policy that optimizes reward, but the behavior policy for learning is about exploration and contains randomness. The Q comes from the "Q-function", an equation that calculates the expected future reward. The Q-learning algorithm is model-free and relies on a table that stores reward values for every combination of action and state. By implementing a neural network (Q-network) instead of a table, we can make it work for more complex problems.

An "experience replay memory" is used to store the experiences of the agent. The agent decides its next action based on a random sample from the replay memory, rather than the most recent action, to reduce correlation between actions (higher correlation would make the learning process inefficient, as the agent would explore less). Another important feature of DQN is the target network. This neural network is separate from the Q-network that is trained after every action. The Q equation is recursive, which can lead to exploding gradients and instability in training. To mitigate this issue, a target network is used for the prediction of future rewards. This network stays fixed to keep the predictions stable, and is periodically sychronized to the Q-network.

4.2 Parallelizing DQN

There are many ways of parallelizing the DQN algorithm. We chose to implement three different methods and measure their performance increases objectively and in comparison with each other. All of them involve the use of multiple agents running simultaneously and learning in their own environments. In the first, we implement a global replay memory shared by all agents. This gives the agents a wider pool of experience to learn from, ideally improving overall learning speed and reward outcomes. In the second, we implement a global target network shared by all the agents. After a set number of global actions, the values from the local Q-networks of each agent will be aggregated and used to update the global target network, and then these Q-networks will be updated to the new target network. Theoretically, by pooling the gradient values from the Q-networks into one global network, the agents should learn more efficiently. In the third, we combine the previous two methods into the same algorithm. This should improve learning efficiency and reward values more than either of the

previous methods. It is also necessary to have each parallel agent take actions with a different policy from each other, so that they do not learn in exactly the same way. This is accomplished automatically by our current behavior policy; because the agents learn from a random sample of past experiences, their actions will never be identical, and they will always explore differently.

5 Project Goal

The goal of our project is to use the principles of concurrency and parallelization to speed up the standard DQN algorithm. We want to replicate the solution proposed in the paper *Massively Distributed Methods For Deep Reinforcement Learning* mentioned previously [1]. The paper uses a massively distributed system to accelerate the training of a machine learning model. The scope of our project will be limited to our personal machines, as opposed to a cluster of servers, but we hope to show that the results of the paper still stand. In order to recreate a similar system, we will be using processes on our own computers, implemented in the Python programming language.

6 Shared Replay Memory Implementation

This implementation involved multiple actor processes with a shared replay memory. The shared replay was implemented using the python multiprocessing manager, which created a shared, synchronized list between the processes. Since the process manager data structures are inherently process-safe, no extra safety precautions needed to be taken. Interestingly, even though each actor has its own local network, in order to evaluate how much was learned from the training, the output requires a single model that represents all the different processes together. So, similar to the other implementations, the Ray [9] library was used to create a remote process that stored the final network. Ray is designed to handle asynchronous access itself, so we did not have to implement locks or something similar. Any update made to a local network was also done to this shared network. Unlike the other implementations, however, the shared network was not used to train the actors. Each actor still learned from its local network, and the only purpose of the shared network was to provide an output that could be evaluated and was representative of the learning done by all actors.

7 Parameter Server Implementation

The parameter server holds the master copy of our model. All the gradients that would normally be applied to the local network of each actor are instead applied to the weights stored on the parameter server. Each process has an actor and learner bundled together. The actors will produce experiences and store them in the local replay memory, then the learner will train the model on a minibatch of those experiences and calculate the gradient. The parameter server receives the

gradients from the learners, then applies the gradients using the asynchronous stochastic gradient descent algorithm. Each time the parameter server makes an update, we count it as a global step. Periodically, when the global number of steps hits a certain number, we update the target network.
The way we implemented the parameter server is by using the Ray Libary [9]. The remote actors and functions are used to allow multiple threads access to the parameter server. These tools make it very easy to transition a sequential algorithm to a parallel one.

8 Shared Replay and Parameter Server Implementation

The parameter server and replay memory are both shared between all actors and learners in this implementation. An Agent class encapsulates an arbitrary number of actor and learner processes, and defines one instance of each of the parameter server and replay memory classes. The parameter server wraps a Keras model and provides methods for querying the weights and for applying gradients. The replay memory is implemented as a fixed-size deque with methods for adding replays and for sampling a batch from memory. Actors and Learners are both implemented as Ray remote actors and so spawn their own workers when created. Each Actor maintains a local copy of the Q-network and its own copy of the environment, and provides the run_step() remote method to take an action. Each Learner maintains local copies of both the Q-network and the target network, and provides the learn() remote method to compute a gradient. The Agent creates the necessary number of Actors and Learners and then runs the global training loop. Every Actor and Learner repeatedly runs its task, while the Agent collects the results. Replays generated are stored in the shared replay memory and computed gradients are applied to the global model. The weights of the global model are periodically sent to the Actors and Learners to synchronize their local copies.

9 Experimental Procedures

To evaluate our models, we needed to train them in some kind of environment. We decided to go with Atari game environments, provided by the OpenAI Gym [10] library. The environments we chose were Pong, Breakout, and Space Invaders. These were sufficiently complex, but also simple enough to hopefully see some good results with just a few days of training.

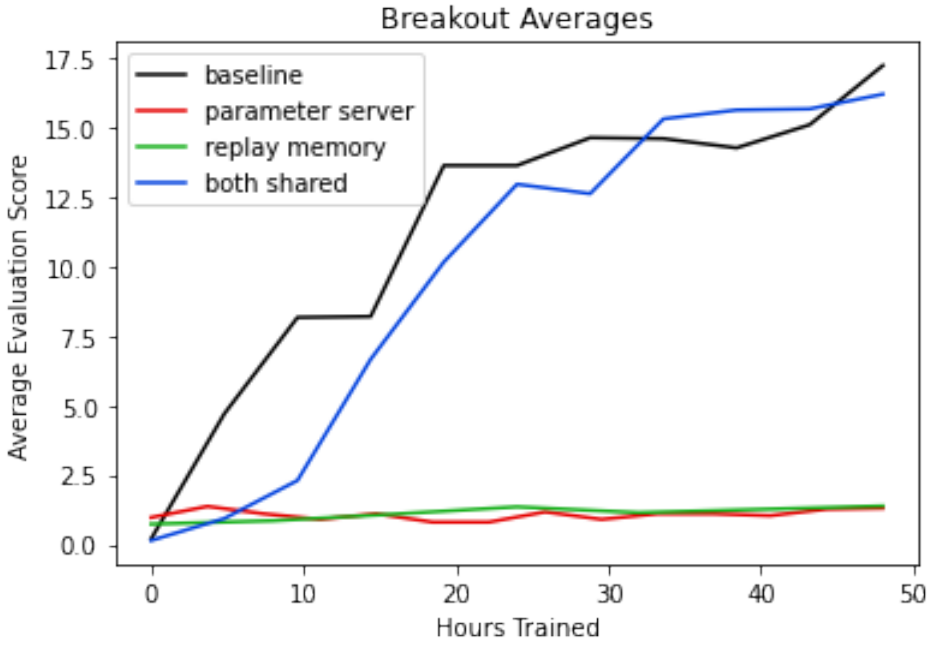
Two important hyperparameters for the training are the learning rate and the exploration rate, both values between zero and one. The benefit of a smaller learning rate means we are less likely to miss the optimal weight values, but will

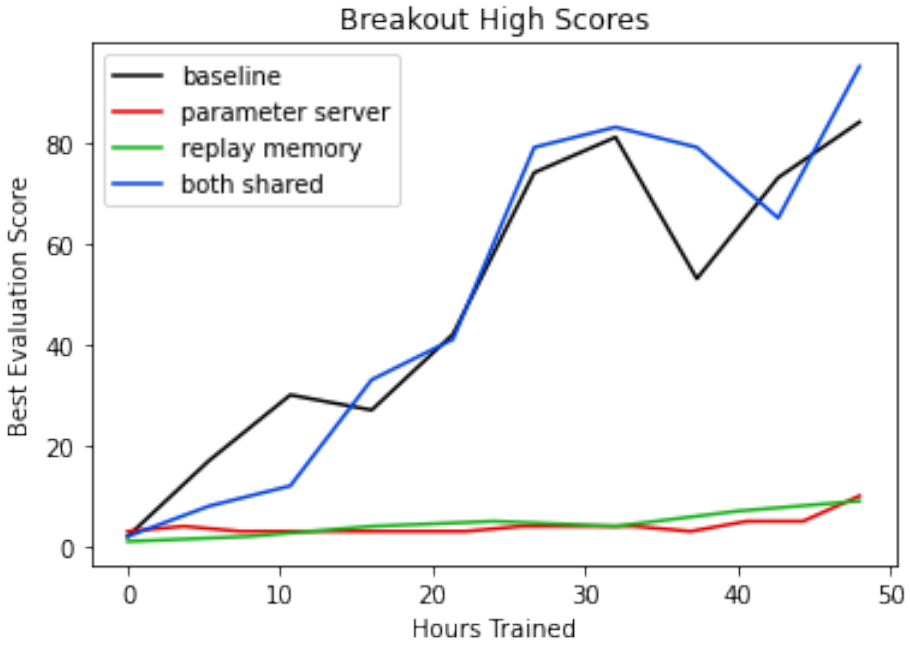
require a longer training time and are more likely to get stuck at a local optimum. A larger learning rate can potentially get to the optimal model faster, but has a higher chance of repeatedly overshooting and bouncing around it without ever reaching it. Furthermore, Q-learning is prone to instability due to its recursive nature which a higher learning rate can exacerbate. The learning rates for all our implementations were set between 0.001 and 0.005, since we each adjusted the values independently to try and find the best result. The exploration rate determines how often an actor will choose to take a random action instead of the predicted best one. An initially high exploration rate is desirable so that the agent sees a diverse set of states before prematurely optimizing. We chose an exploration rate beginning at 1.0 and decaying by a factor of 0.999 after every action, down to a minimum of 0.1.

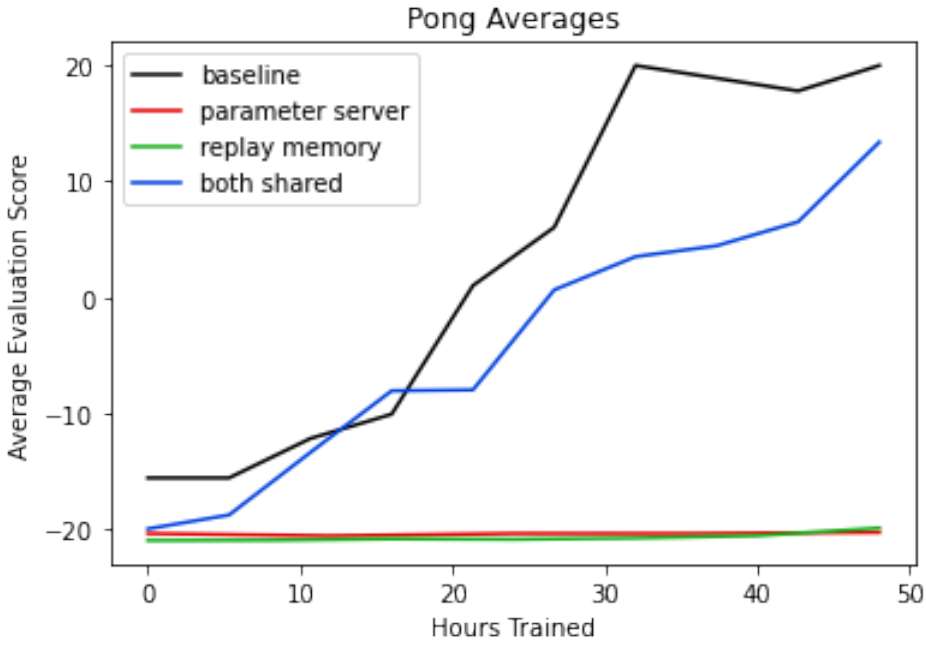
All parallelized implementations were trained using three actor processes and one learner process. We chose an example from the Keras documentation [11] as our baseline sequential implementation. All implementations were trained using GPU-enabled machines through Google Colab. Every implementation was trained for 48 hours in each environment, and the weights were saved periodically. We then loaded the weights from each stage of training, ran the agent for 30 episodes with no exploration or learning, and recorded the maximum and average scores.

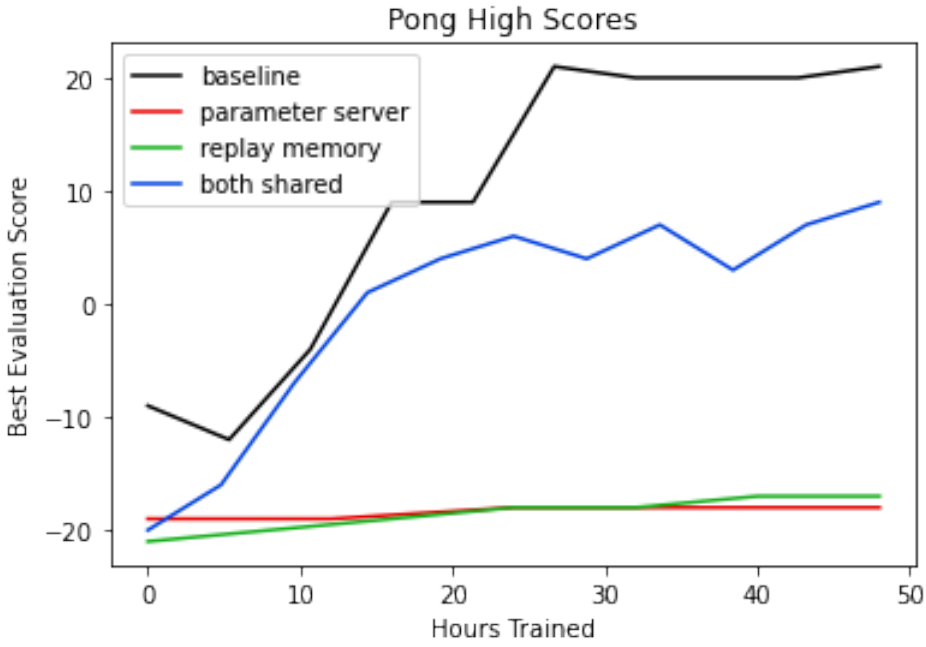
10 Results

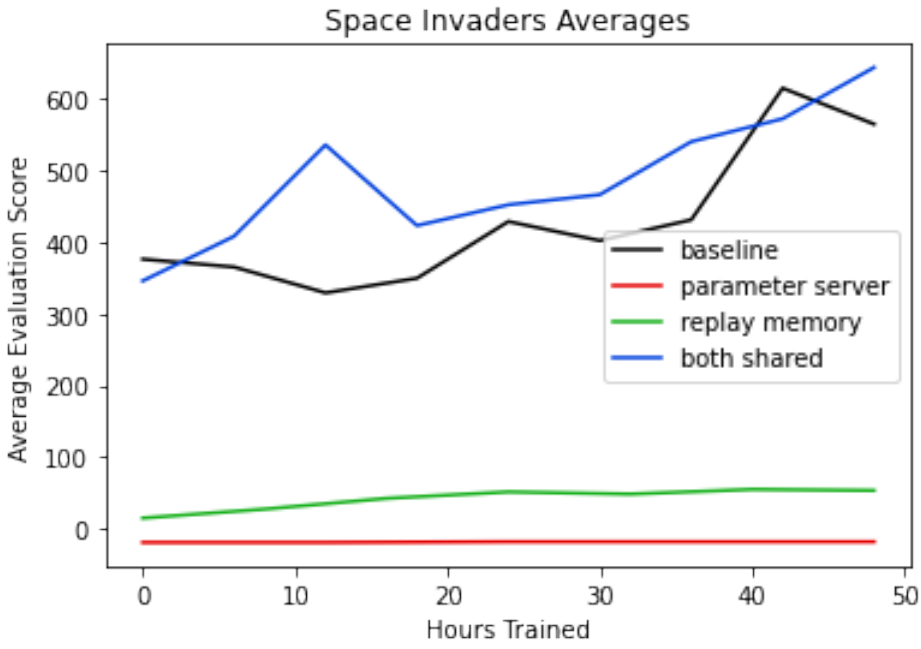
Results were mixed, in part due to particularities of Deep-Q Learning that did not become apparent until training was underway. While we did eventually tune the model to get good performance, there was not time to retrain every implementation so some were working suboptimally. The best-performing implementation outperformed the baseline in some regards, but was still hampered by the high cost of parallelization relative to the number of available CPUs.

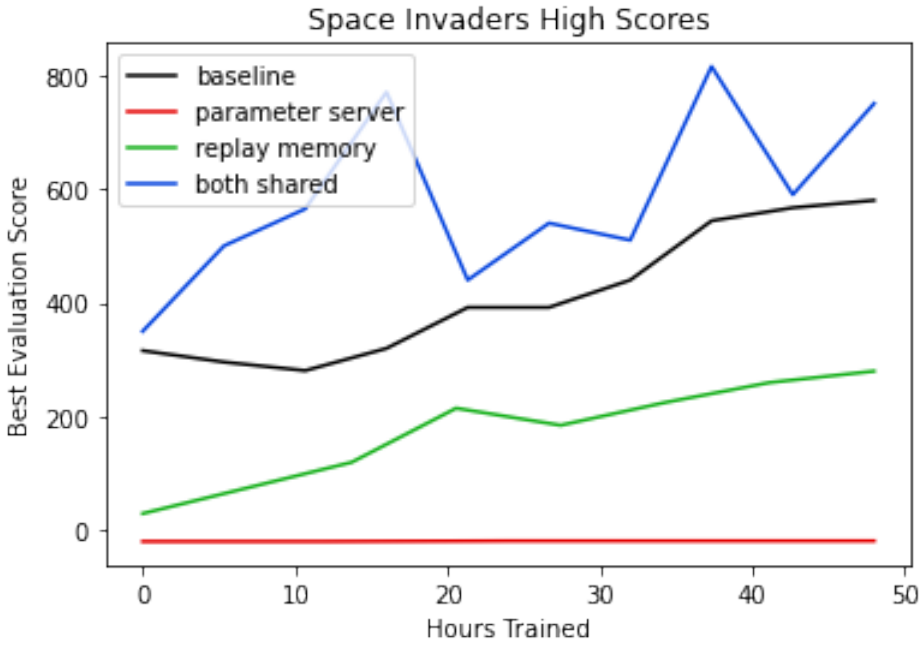

Breakout Averages
baseline
parameter server
replay memory
both shared
Average Evaluation Score
Hours Trained
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
10
20
30
40
50


Breakout High Scores
baseline
parameter server
replay memory
both shared
Best Evaluation Score
Hours Trained
0
20
40
60
80
0
10
20
30
40
50


Pong Averages
baseline
parameter server
replay memory
both shared
Average Evaluation Score
Hours Trained
20
10
0
−10
−20
0
10
20
30
40
50


Pong High Scores
baseline
parameter server
replay memory
both shared
Best Evaluation Score
Hours Trained
20
10
0
−10
−20
0
10
20
30
40
50


Space Invaders Averages
Average Evaluation Score
Hours Trained
baseline
parameter server
replay memory
both shared
0
100
200
300
400
500
600
0
10
20
30
40
50


Space Invaders High Scores
Best Evaluation Score
Hours Trained
baseline
parameter server
replay memory
both shared
0
200
400
600
800
0
10
20
30
40
50

11 References

References

[1] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

[2] Wanli Peng, Hao Pan, He Liu, and Yi Sun. Ida-3d: Instance-depth-aware 3d object detection from stereo vision for autonomous driving. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[3] 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.

[4] Zixing Zhang, Jürgen Geiger, Jouni Pohjalainen, Amr El-Desoky Mousa, Wenyu Jin, and Björn Schuller. Deep learning for environmentally robust speech recognition: An overview of recent developments. *ACM Trans. Intell. Syst. Technol.*, 9(5), April 2018.

[5] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[6] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, and et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, Dec 2020.

[7] Hao Yi Ong, Kevin Chavez, and Augustus Hong. Distributed deep q-learning, 2015.

[8] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[9] 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.

[10] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[11] Jacob Chapman and Mathias Lechner. Deep q-learning for atari breakout. https://keras.io/examples/rl/deep_q_network_breakout/.