Parallelization of the DQN Algorithm

Robert Moore, Jacob Hofstein, Colin Chu

March 15, 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 target network is updated periodically to match the agent's Q-network after a predefined number of steps. We use the target network to train the agent; the Q-learning network changes too rapidly for the agent to learn from it reliably.

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 using concurrency to speed up Deep Q-learning. Deep Q-learning basically a model free deep reinforcement learning algorithm. To put it in simple words, Q-learing create a cheat sheet(using state and action) for the AI, so they know what action needs to be taken. We want to see if there is any way to either speed up the performance or produces better results. After some research we decided to go with the solution proposed in the paper Massively Parallel Methods for Deep Reinforcement Learning. The paper uses a massively distributed system to accelerate the training of a machine learning model compared to using a single high-performance GPU. What we want is to be able to validate the result of the paper, proving that using distributed replay memory and distributed neural network will provide better performance doing deep Q learning on a single machine. Of course we won't be able to replicate the whole network architecture completely since they have many machines, while we each have one. In order to recreate a similar environment, we will be using threads or processes theose will represents an individual machine. This way we can simulate the network acrchitecture of the paper.

6 Implementation

–more detailed explanation of how we implemented the architecture, what libraries were used, etc

7 Experimental Procedures

–what tests we run, how long we train model on what type of CPUs, etc

8 Results

–tables, charts, brief analysis

9 Appendix

9.1 Challenges

Jacob: I came into the project with no background in deep learning, so it was difficult to acquire the knowledge needed to start the project. Learning two new libraries was also a bit tough, but luckily we only needed a small portion of their functionality for the project. In addition, managing a shared data structure between agents running in parallel was not trivial.

Robert: I had some experience with deep learning from previous classes, but I had not specifically worked with deep reinforcement learning. Significant research was required before I understood the available tools and their mutual compatibilities. We also chose Python as the language due to the useful libraries available for machine learning, but Python multithreading is limited by the interpreter. To leverage multicore processors we have to spawn new processes, which affects how memory can be shared between agents.

Colin: I had no prior knowledge of machine learning. Also, I was unsuccessful with installing TensorFlow on my local machine, because of package mismatch and other reasons, so I ended up switching to Google Colab.

9.2 Completed Tasks

– Implement the base DQN algorithm
– Implement DQN with a shared replay memory
– Implement DQN with a global target network
– Implement DQN with both techniques

9.3 Remaining Tasks

– Train the models
– Evaluate their performance
– Compile data and analyze results

10 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.