Analysis of Concurrent Reinforcement Learning Algorithms

Cathlyn Stone
ECE 5510 - Multiprocessor Programming
Virginia Tech
Blacksburg, VA
stonecat@vt.edu

Karan Sachdev
ECE 5510 - Multiprocessor Programming
Virginia Tech
Blacksburg, VA
karansachdev1012@vt.edu

I. Introduction

Reinforcement learning has found success in a wide variety of applications, ranging from playing games like AlphaGo, to automating neural network design, to even solving problems in domains such as finance and healthcare [1]. To enable faster learning, several methods have been proposed for implementing efficient and highly parallel versions of RL algorithms [2]–[5].

For this project, we analyze and evaluate the efficacy of concurrent RL algorithms. Specifically, we aim to understand the operation of the sequential versions of a few RL algorithms, how they might be parallelized, and the challenges and bottlenecks that arise when implementing them concurrently. To evaluate, we intend to examine how well these algorithms scale as more processors are added. Through this work, we hope to gain a clearer knowledge of the synchronization techniques useful in parallelizing these types of algorithms and the tradeoffs of various design choices considered in their implementation.

We have implemented three different parallel versions of RL algorithms in Python: Deep Q-learning (DQN), Asynchronous Advantage Actor Critic (A3C), and Dyna-Q. We apply these algorithms to solve Atari games, so that a trained model can skillfully beat the game. We found that overall, adding additional processors helped to improve training time to an extent. Too many processors resulted in much overhead in synchronizing global network weights. Furthermore, DQN generally gave the best results among all the RL algorithms we considered for the game we trained on.

II. Background

Reinforcement Learning algorithms generally involve an agent that interacts with its environment in order to find a policy that will maximize its future reward. Furthermore, two main types of these algorithms exist: model-free and model-based methods. Model-free methods directly try to predict the next action that should be taken given previous experiences. Model-free methods can further be categorized as being value-based or policy-based. Value-based approaches try to approximate the optimal value function (a mapping from an action to a value). Higher values represent better actions. In contrast, policy-based methods attempt to learn an optimal policy directly rather than using a Q-value to determine the action that should be taken.

Q-learning [6], [7] is a model-free value-based RL algorithm that learns to find the best action for each state in order to obtain the greatest total reward. A Q-table, indexed by each possible action and state, is employed to hold the expected reward for each (action, state) pair. This table is then updated as the model trains, such that eventually the table gives a good estimate of the reward that can be expected for each pair. This way, if the agent takes the action that gives the maximum reward value for each state that is encountered as the agent interacts with its environment, it should obtain the maximum reward total. Of course, there are many other subtleties here, such as sometimes taking actions with lesser expected rewards during training in order to fully explore the environment and not overlook an action that may actually lead to a state that gives a better reward total.

Deep Q-learning [8] expands upon Q-learning by replacing the grid that holds the Q-table with a neural network. This is very useful especially in cases where we might not have a set of discrete state values. For example, an agent playing an Atari game might observe a state that is represented by a vector holding continuous values. If we wanted to use Q-learning in this case, we would then have to create a mapping of this continuous state space to separate states, so we can index our table. However, a neural net can be used instead to take as input this state vector, and have nodes representing the expected reward for each action as output. Hidden layers can be added as desired and other choices can be made regarding the net's architecture (like choice of activation function, etc...) which may also improve results. Over time, by taking different actions and back-propagating error in order to the train the network weights, we should hopefully obtain a network that will give a good estimate of the expected reward of taking each action given this continuous state. In addition, as an agent interacts with it's environment, it stores its experiences in a replay memory. This way, experiences from the memory can be randomly sampled in order to train the Q-network in batches.

The Advantage Actor Critic (A2C) algorithm [9] is another popular model-free RL algorithm. In this design, there is a "critic" which estimates the value function (either the Q-value or V-value), and an "actor" which updates the policy based on the gradients given by the critic. It differs from Q-learning in that while Q-learning is value-based, A2C is both policy-based and value-based. The policy adjusts action probabilities based on the current estimated advantage of taking that action. Again, neural networks are useful for implementing the actor and critic models. While for brevity we leave out the details regarding this approach, more details can be found in [10].

In addition to predicting the next action that should be taken in order to maximize reward, model-based methods also try to learn a model during training that represents the environment the agent interacts with. This way, the algorithm can make a prediction about the next state that will be observed given a current state and action taken from that state. Dyna-Q [11] is a model-based approach that also incorporates Q-learning. A model is created which describes the environment and is updated over time during training. Furthermore, with Deep Dyna-Q [12], the model of the environment consists of a neural network. In the case of playing Atari games, this model can be thought of as representing the Atari game itself. This model takes as input (state, action) pairs and gives as output a combination of (next state, reward, done), where done represents whether or not the episode has completed. So, after executing each action and updating the Q network (as in deep Q-learning), the experience (the state the agent was in, the action it took, the next state it got to, and the resulting reward) is fed to the model in order to train it. Next, the model can be used to "imagine" what would have happened if the agent had taken different actions from different states, based on the model that has been learned so far. From a random sample of the agent's memory of previously encountered states, different actions are taken by using the model to find the outcome of those actions. This results in new experiences that the Q-network can then learn from.

III. Related Work

While Deep Q-learning works well, it can be time consuming for a single agent to train. The Gorila framework, proposed by Nair et. al [5], outlines how a parallel version of the DQN algorithm could be designed. This parallel implementation outperformed DQN in terms of the total time taken to train by an order of magnitude on most of the 49 Atari games, an impressive result.

Rather than using a single agent which interacts with the environment, this framework proposes that multiple agents interact with separate instances of the environment simultaneously. Multiple agents can train and explore different instances of the environment, updating a local instance of a Q-network as they train. This allows faster exploration of the state space of the environment than if just a single agent were run. Then, agents can occasionally synchronize their learned network weights, so that network weights converge to optimal values more quickly.

A parameter server is created to synchronize the parameters of the individual learners' networks. After each mini-batch of experiences is run through each learner's network, and the loss backpropagated to their local Q-networks, the gradients which have been computed are sent to the parameter server. The accumulated gradients sent by the learners can then be used to update the parameter vector held in the parameter server by an asynchronous stochastic gradient descent (or other optimization) algorithm. After N global time steps have passed, each learner's local network parameters can be synchronized with the parameter server vector.To allow greater parallelism, the parameter server can be broken into shards that can be updated concurrently. For example, if the gradients are contained in a vector of size k, k different shards can be created to hold and compute updates from gradient.

However, the parameter server suggested in this paper is not the only way in which network weights can be synchronized. For example, the HogWild! [13] algorithm proposes a lock-free parallel SGD algorithm. HogWild! suggests that multiple processors can operate on a gradient vector held in shared memory. The idea is that if data is sparse, it will be rare enough for processors to overwrite one another as updates happen. Furthermore, if they do, this shouldn't hurt results much. While theoretical guarantees and proofs are given more formally in the paper, the appeal of this algorithm is the near linear speedup that it provides. Additionally, many other algorithms for parallel optimization have been proposed, such as HogWild++ [14] and DogWild! [15].

The Gorila framework also states that the replay memory that is created to store the experiences of the learners can be created either locally or globally. Storing locally alleviates the need to synchronize this memory between multiple processes, but means that actors can only learn from their own experiences and not the experiences from the group as a whole. If the total space size of the environment is large and might take each actor a while to explore in full, it may be desirable to use a global replay memory, despite the additional overhead of synchronization, so that actors can learn from the combined experiences of the group.

The Asynchronous Advantage Actor Critic (A3C) algorithm [16] is a parallelized version of A2C. As in Parallel DQN, multiple agents can be created to interact with copies of the environment in parallel. Each agent contains a local policy network and local value network. Occasionally, local network parameters of agents are synchronized with the parameters of a global policy and value network.

Finally, in certain scenarios, we may have only one "real" environment to interact with. While Atari games we can play multiple instances of simultaneously, in other types of problems, this may not be possible. For example, in stock market prediction, there's only one real environment to train on. This is where model-based methods can assist in providing a way to create a model of this environment, such that agents can interact with it in parallel and generate additional simulated experiences to achieve faster training times.Parallel Dyna-Q has been proposed by [17]. Here, multiple parallel Dyna-Q agents help to balance exploration and exploitation, and works similar to how the other parallel RL algorithms we have studied. This could also be extended to a deep Dyna-

Q implementation. We describe our implementation of this in more detail in the following section.

IV. Design and Implementation

Our implementation can be found at https://github.com/altwave/mpp-project.

We selected to use Python to implement our solution, due to the large number of machine learning libraries available, and it's widespread adoption in machine learning. We found the Torch library [18] useful for implementing the neural nets contained within our algorithms. To train on Atari games, the OpenAI Gym library [19] provided an extremely useful interface to create a game environment that the algorithm could interact with. Furthermore, it allows the game to be viewed (if desired, though generally we left this off while training to not slow down the computer with having to also render the game), which can be fun and rewarding to watch as the algorithm plays the game more and more skillfully. Specifically, we elected to use the "LunarLander" game for training and evaluating these algorithms.

A. Python's Concurrency Model

While the functionality available in these libraries served as a compelling reason to use Python versus Java or other languages, working with Python's concurrency and shared memory model proved to be a bit tricky. Python provides both a "multithreading" and "multiprocessing" library. While the multithreading library is lightweight and provides shared memory, multiple threads that are created still live within the same process. This is due to Python's Global Interpreter Lock, which only allows one thread to access the lock at a time in a given process. Thus, threads created within a process are actually time-sliced to run on a single core versus actually running on different cores in parallel. Therefore, we found it necessary for this goals of this project to rely on the multiprocessing library instead, which provides for the creation of separate processes that can be run on multiple cores. However, each process is given a separate memory space, which makes synchronizing shared data between threads a challenge. This is in opposition to how Java's Thread model works, where objects created on the heap can be shared by multiple threads.

In order to communicate data between separate processes, there are a few techniques available. One way is through a Queue, in which objects can be placed on and removed from by different processes. Another way is through a Pipe, which acts like a socket connection to send messages. Messages can be sent through this pipe from one process to another. A slightly more challenging way to share data is via shared memory that can created through Python ctypes objects. Finally, there are a few synchronization primitives (like locks, barriers, semaphores, etc...) which can be used to implement mutual exclusion among processes.

Additionally, Torch provides a multiprocessing module, which eases some of the difficulties of sharing memory. This module builds upon the functionality provided by Python's multiprocessing by allowing certain types of objects (like tensors and nets) to be declared as shared memory objects. There are also a set of "best practices" outlined by Torch's multiprocessing library to follow to enhance performance [20]. Some of these suggestions include using a Queue to communicate shared memory objects, and reusing the buffers created for shared memory object versus creating new buffers. In general, we tried to follow these suggestions to help our performance, and found this multiprocessing library a useful interface versus trying to rely on the base Python library alone.

B. Overhead of Torch's Shared Memory

We found there was significant overhead when accessing data placed into Torch's shared memory. Simply operating on shared memory added extra time for each episode to run, even when locks were not used, as in the HogWild! implementation. Further research revealed to us the specific overheads due to using this memory, further described in [21]. In order to allow multiple processes to share data, Torch creates a file that is then mmap'ed into the address space of both processes, so both processes can access the data. However, this means that reading and writing data takes longer than just operating on objects placed on the heap. Furthermore, it adds the additional constraint that objects in shared memory cannot be resized. This makes it difficult for creating things such as a global replay memory. Of course, we could initialize a large area of memory up front, and read and write from that, but we would have to handle what happens if or when we eventually run out of space, either adding an additional area or overwriting older entries. Or we could create a collection of shared memory objects. But this is difficult because we would have to create a shared list of the memory objects that have been created thus far, again adding to the issue of resizing. Issues such as these made working with shared memory a pain and caused results to suffer when shared memory accesses were frequent.

C. Implementing Parallel DQN

First, we implemented DQN in Python so that it could successfully learn to play Atari games. Then, we extended it to a concurrent implementation, guided by the Gorila framework discussed previously and the HogWild! algorithm. We implemented multiple actors that interact with their own instance of the Atari game. To synchronize network weights, we first implemented the naive approach of creating a mutual exclusion lock which that each actor acquired when updating the global network weights. This helped to ensure the soundness of the approach, and that the global network weights were synchronized correctly. We found that while it took less episodes to complete training in this manner, the time for each episode increased substantially due to contention for the lock and the amount of computation that needed to be performed within the lock.

We next tried implementing the parameter server outlined by the Gorila framework. In this implementation, actors sent asynchronously (via a queue) the local network gradients to each of the parameter server shards (run as separate processes). In the shard processes, gradients were removed from the queue and used to update the currently held parameters using SGD optimization. We also tried averaging several gradient updates removed from the queue before running SGD, so the shards' processing would never lag behind as multiple actors sent in their updates. Also. at the start of each run, actors copied the global parameters to their local networks.

Dealing with synchronizing the gradient updates from individual actors to and from the parameter server was a challenge. If the parameter server did not place the gradients in memory specifically allocated as shared memory, those updates would not be visible to other threads. So, when sending in the gradients to the parameter server, the gradients had to first be copied into a shared memory tensor. When copying the global parameters back to their local networks, actors had to again copy the gradients out of shared memory back to a local memory tensor, so that updates would be made to a local copy (versus continuing to operate on a shared memory location). Despite our best efforts to get this working, we found that performance did not improve with multiple threads. While we were able to train in less episodes overall, we think the overhead of operating on shared memory and copying tensors in and out of shared memory may have caused too much performance degradation, as further described in Section IV-B. We tried to adhere to Torch's recommendation of multiprocessing best practices [20] as far as how to share memory, but were still unhappy with results.

So, we decided instead to take the approach suggested by the HogWild! algorithm, which shows how SGD optimization can be run without locking to each of the gradients in the parameter vector [13]. This worked much better. Despite the linear speedup advertised by the paper, we found it important to limit the frequency in which agents operated on shared memory to apply updates to the global parameters as we increased the number of processes. Frequent updates caused each episode to take much longer, despite the lock-free implementation. Again, we hypothesize this may have something to do with Torch's shared memory implementation.

To implement the replay buffer, we decided to create local replay buffers for each actor, rather than use a global buffer. This helped us to avoid the overheads of dealing with shared memory and avoid contention when adding experiences to the global replay buffer. Furthermore, data in shared memory can not be resized, which causes additional problems in how to implement a global replay buffer, as described in Section IV-B. We considered using both a local and global replay buffer, in which a local buffer cache a number experiences, and only update the global buffer after the local buffer filled. This way, processes could still obtain the benefits of learning from shared memory, while reducing the number of times different threads try to add new experiences. However, we did not have time to try this technique, and thus leave this for future work.

D. Implementing Asynchronous Advantage Actor Critic

To implement A3C, we took an approach similar to parallel DQN. However, rather than use just one global network, we created two global networks in which to synchronize with the agent-local actor and critic networks. Having implemented parallel DQN, the implementation of this algorithm was fairly straightforward. A3C algorithm maintains a policy and an estimate value function, both approximated by a neural network. The actor-learners compute gradients in parallel and shared parameters are updated in a HogWild! fashion. We also considered splitting each actor into two processes, allowing one to perform computation on the actor network, while the other performs computation on the critic network. However, we leave this for future work.

E. Implementing Parallel Deep Dyna-Q

After completing the first two implementations, we decided to implement a parallel version of Deep Dyna-Q, in order to learn more about model-based approaches. For the purposes of playing Atari games, running multiple instances of the environment is not an issue and the environment is deterministic. However, for other applications, we wanted to see how it might be possible to build an RL algorithm where either a model of the environment is helpful (due to the changing nature of the real environment), or where we are unable to have multiple instances of the real environment. We first implemented a deep Dyna-A algorithm for a single process, and then extended this to work for multiple processes. We used just one process to act as an agent on the real environment. This agent works the same as in DQN, interacting with the real environment and updating the Q-network. The agent then sends its experiences to a shared Queue.

We also created multiple agents which would train to learn the world model (a model of the Atari game itself in this case). The model, represented by a neural network, contains two input layers (one for state, one for one-hot encoded action). There are then two hidden layers and 3 output layers. Of these output layers, one outputs the next state, one outputs the expected reward, and one outputs whether or not the game has ended upon taking that action. So, these multiple processes obtain data from the shared Queue, and use the data to train a global world model. Network parameters of the world model were synchronized using the HogWild! algorithm, as in our DQN implementation. After training with batches of experiences, the agents then interact with the world model, taking random actions so that the environment state space can be explored more quickly. The agents then use these simulated experiences to update the Q-network, in order to train the Q-network more quickly.

F. Evaluation

We evaluated the total amount of time it took to train the algorithms, as well as the total number of episodes to complete training. Overall, for DQN and A3C, we found that it took fewer episodes to train to completion with an increasing number of processes. This is displayed in Figure 1. However, adding additional processes resulted in each episode taking longer to complete, due to the overhead of sharing memory and synchronizing network weights. The average time per episode is shown in Figure 2. In looking at the total amount of time

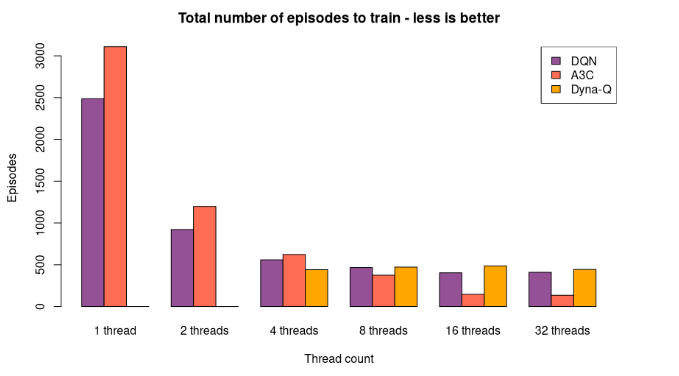

Fig. 1. Total number of episodes to train

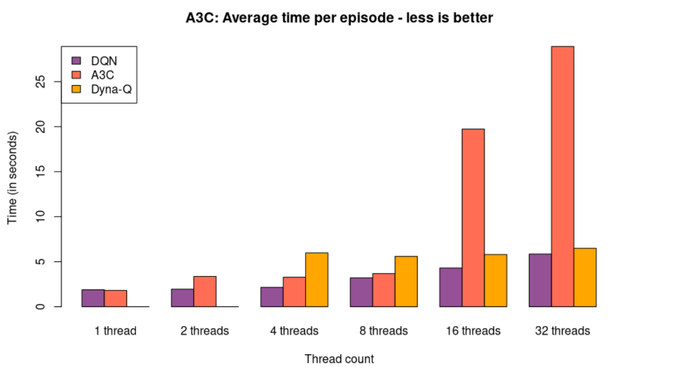

Fig. 3. Time to train (in minutes)

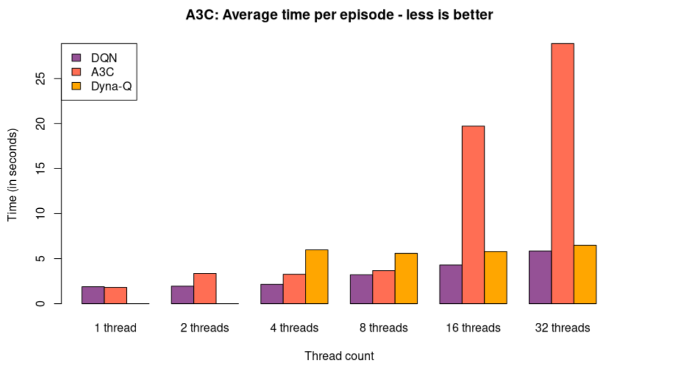

Fig. 2. Time per episode (in seconds)

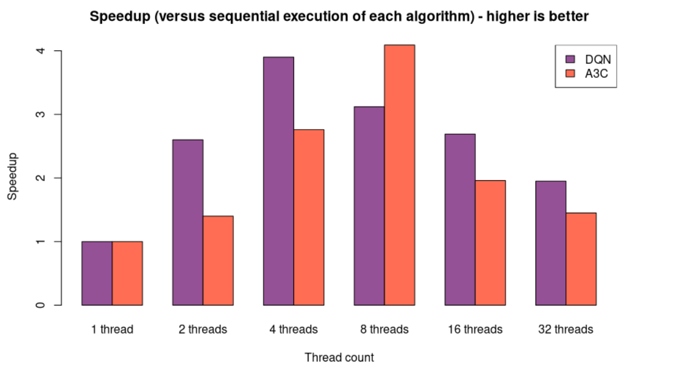

Fig. 4. Speedup

to train, displayed in Figure 3, we can see that using 4 or 8 threads generally gave optimal results. Adding additional processors beyond this resulted in increased training time, despite taking less episodes overall. Nevertheless, using any number of processors gave an increased speedup compared to a sequential execution with one processor. Speedup is shown in Figure 4.

In evaluating Dyna-Q, we did not evaluate using one thread, as we split the code for the agent acting on the real environment and the agents acting on the world model to different processes. We also found that using two or three threads did not work well, as the world model did not train quickly enough to provide useful simulated experiences. Also, using too many threads added too many simulated experiences (vs. real experiences) to the Q-network training. Therefore, we limited the number of updates of simulated experiences. In the future, we could try improving this ratio. We could also try using Prioritized Sweeping [22] to allow world agents to strike a better balance between exploitation and exploration. This way, agents adding simulated experiences can focus on exploring more interesting or unknown areas of the state space to update the Q-network, rather than just randomly or greedily exploring.

G. *Conclusions and Future Work*

Several areas could be improved upon in the future to improve results. For overheads due to using shared memory, as discussed in Section IV-B, we did not achieve a linear speedup when using the HogWild! implementation. Perhaps, it would be worthwhile to investigate other ways to share memory in Python, or look using Torch's C++ library, if accessing shared memory in C++ happens to be more efficient. We could also consider implementations in other languages as well. This could also help us to implement a global replay memory, which could further help improve results.

We could also investigate better concurrent optimization algorithms. This could help to reduce the total amount of time needed to synchronize global network weights, resulting in less training time overall. Additionally, many ideas in this area have been proposed, such as ”Asynchronous Decentralized parallel SGD” [23] or ”A Linearly Convergent Proximal Gradient Algorithm for Decentralized Optimization” [24]. Optimizing the hyperparameters we selected to use, such as the learning rate, which would likely further improve results. Running these algorithms on a GPU would also likely enable faster training. Finally, applying these RL algorithms to other problems or evaluating different RL algorithms could be an interesting future direction.

References

[1] Y. Li, “Reinforcement learning applications,” 2019.

[2] A. V. Clemente, H. N. Castejón, and A. Chandra, “Efficient parallel methods for deep reinforcement learning,” 2017.

[3] A. Stooke and P. Abbeel, “Accelerated methods for deep reinforcement learning,” *ArXiv*, vol. abs/1803.02811, 2018.

[4] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons, and J. Kautz, “Reinforcement learning through asynchronous advantage actor-critic on a gpu,” 2016.

[5] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. D. Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver, “Massively parallel methods for deep reinforcement learning,” 2015.

[6] C. Watkins, “Learning from delayed rewards,” 01 1989.

[7] C. J. C. H. Watkins and P. Dayan, “Q-learning,” in *Machine Learning*, 1992, pp. 279–292.

[8] H. van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” 2015.

[9] V. Konda, “Actor-critic algorithms,” Ph.D. dissertation, Cambridge, MA, USA, 2002, aAI0804543.

[10] I. Grondman, L. Busoniu, G. A. D. Lopes, and R. Babuska, “A survey of actor-critic reinforcement learning: Standard and natural policy gradients,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 6, pp. 1291–1307, Nov 2012.

[11] L. Kuvayev and R. S. Sutton, “Model-based reinforcement learning with an approximate, learned model,” in *in Proceedings of the Ninth Yale Workshop on Adaptive and Learning Systems*, 1996, pp. 101–105.

[12] B. Peng, X. Li, J. Gao, J. Liu, K.-F. Wong, and S.-Y. Su, “Deep dyna-q: Integrating planning for task-completion dialogue policy learning,” 2018.

[13] B. Recht, C. Re, S. Wright, and F. Niu, “Hogwild: A lock-free approach to parallelizing stochastic gradient descent,” in *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2011, pp. 693–701. [Online]. Available: http://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf

[14] H. Zhang, C.-J. Hsieh, and V. Akella, “Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent,” *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pp. 629–638, 2016.

[15] C. Noel and S. Osindero, “Dogwild!-distributed hogwild for cpu gpu,” 2014.

[16] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” 2016.

[17] Takeshi Tateyama, Seiichi Kawata, and Toshiki Shimomura, “Parallel reinforcement learning systems using exploration agents and dyna-q algorithm,” in *SICE Annual Conference 2007*, Sep. 2007, pp. 2774–2778.

[18] R. Collobert, S. Bengio, and J. Marithoz, “Torch: A modular machine learning software library,” 2002.

[19] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” 2016.

[20] “Multiprocessing best practices¶.” [Online]. Available: https://pytorch.org/docs/stable/notes/multiprocessing.html

[21] Pytorch, “pytorch/pytorch.” [Online]. Available: https://github.com/pytorch/pytorch/wiki/Multiprocessing-Technical-Notes

[22] A. Moore and C. G. Atkeson, “Prioritized sweeping: Reinforcement learning with less data and less real time,” *Machine Learning*, vol. 13, October 1993.

[23] X. Lian, W. Zhang, C. Zhang, and J. Liu, “Asynchronous decentralized parallel stochastic gradient descent,” 2017.

[24] S. A. Alghunaim, K. Yuan, and A. H. Sayed, “A linearly convergent proximal gradient algorithm for decentralized optimization,” 2019.