Distributed Deep Q-Learning algorithm based on MPI

Javier Grandio
Computer Science Department, Virginia Commonwealth University, Richmond, USA
grandiogj@vcu.edu

***Abstract*—This project presents a distributed implementation of the Deep Q-Learning algorithm based on Message Passing Interface (MPI). In this implementation, one process computes the training of the Deep Neural Network, while the rest of the processes run simulations to generate more data. Usually, the computation of simulations require to solve complex and computationally expensive problems. With this approach we are alleviating this issue and improving the runtimes to train controllers. Without loss of generality, the case of study corresponds to three different problems to control, increasing their complexity until having a high complex simulation.**

I. INTRODUCTION

In recent years, the development of Reinforcement Learning techniques has been growing exponentially. Big tech companies such as Google or OpenAI are pushing for research in this field. In particular, among all the available algorithms, Deep Q-Learning and others based on this one had been implemented for diverse control tasks.

Due to this growth, some surprising results have been achieved recently, reaching even a level where the controllers generated are able to perform complex plans to achieve a goal. However, as its name says, Reinforcement Learning is based in a learning algorithm of trial and error, and in order to learn to perform a task, a lot of training data has to be generated. As an example, OpenAI has developed controllers to play a hide and seek game, and although they got really positive results, they needed billions of episodes of training to achieve that performance.

So, the main problems with Deep Q-Learning are related to the long time needed for trial and error learning.

In case of wanting to control a real system, the option of training the controller in the real environment is not viable. Mainly, this is due to security issues. The negative consequences of controller errors (e.g., in autonomous driving an error may lead to an accident) due to the learning process itself could cause dangerous situations.

The alternative to the real training is to train the controller using a simulation in a virtual environment. The fact of simulating the system, in most cases involves a computationally expensive simulation (e.g., simulating any physic system that involves collisions between objects). To alleviate the time requirements of the simulation, the two straightforward solutions would be either to use a more powerful computer or to try to parallelize the simulations in a cluster.

To address these problems, we explore the option of implementing a distributed training system based on MPI. In this distributed system, one process is the one that takes care of the training of the network, while the rest available processes run simulations in parallel to generate training data.

The proposed approach is tested in three different environments:

- Cartpole
- Lunar Lander
- Object Slider

The runtimes gotten using the traditional approach and proposed are compared, studying how the complexity of the problem affects the proposed approaches.

The reminder of the project is structured as follows. Section II explains the Deep Q-Learning Algorithm, with an emphasis in its challenges and benefits for parallelization. Section III presents all the different approaches implemented. Section IV presents and discusses the results obtained for each one of the different environments. Finally, the conclusions of the work are summarized in section V.

II. DEEP Q-LEARNING ALGORITHM

Deep Q-Learning is a model-free deterministic algorithm. The fact of being model-free allows it to control a system without any need of previous knowledge about the environment.

The algorithm itself is based in a value function. This function maps for a state the reward (defined by the designer) that could be achieved performing each one of the possible actions. This value function is approximated using a Deep Neural Network and this one is trained performing regression with the data generated during the training. In consequence, the simulation process can be represented by the two interconnected blocks as depicted in Fig. 1.

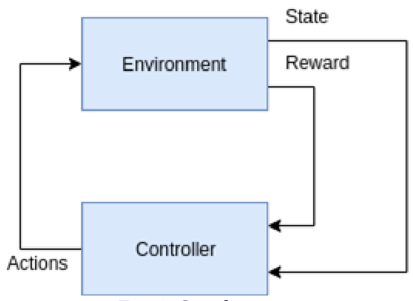

Fig. 1. Simulation process

The Environment block represents the system to be controlled. The Controller block in Fig. 1 is responsible for planning future actions based on the successive states of the system. When using Deep Q-Learning, the architecture of the controller can be represented as shown in Fig. 2.

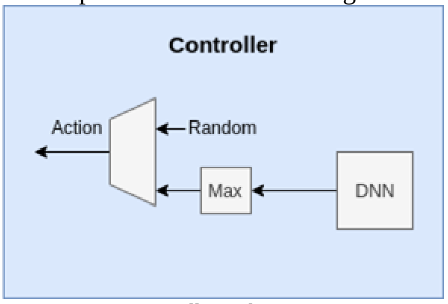

Fig. 2. Controller architecture

The policy of the controller is deterministic because it always chooses the action with the highest reward value. However, during the training of the network, a factor of randomness is inserted in the system so there is a chance of exploring new actions.

For the training of the network, it is needed to generate training data. Also, that data should be generated using the most recent trained network, because in that way the states explored during the training are better and better with time, and the neural network learns to approximate the value function for those improved states. However, it is also convenient to keep training using old data, so the network does not overfit the function for the new improved states and "forgets" what action it should take in an adversarial state. To solve this problem, it is very convenient the way to update the Q-function, that follows equation 1. The previously mentioned equation takes into account the policy that the controller is following at that exact moment of training, and not the policy that it was following when the data was generated. This fact makes Q-Learning algorithm an offline method, allowing to train the network using old data instead of having to train in each step of the simulation. This is a key factor for parallelization, because there is no need to be continuously training the network when the data is generated. The usual approach to take advantage of this property in the standard implementation is to create a buffer of data with limited size but a big capacity. What is done with this buffer is to keep filling it with new data and take random samples to train the network.

Taking into account the training process explained previously, it could be divided into two different processes (simulating and training) that need to communicate to each other using the data buffer and sending the new trained network to generate more training data. This architecture is depicted in Fig. 3.

$$Q_\theta(S_t, A_t) = Q_\theta(S_t, A_t) + \alpha(R_{t+1} + \gamma\, Q_{\theta'}, (S_{t+1}, \arg max_a Q_\theta(S_{t+1}, a)) - Q(S_t, A_t)) \qquad (1)$$

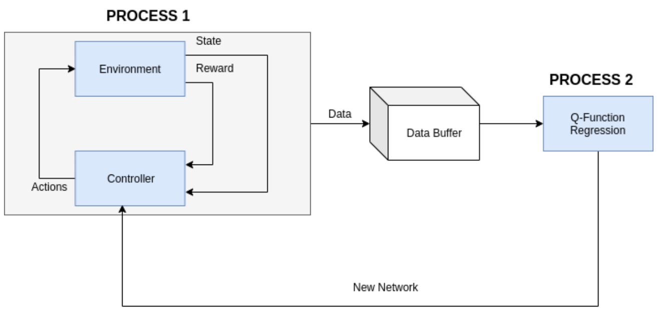

Fig. 3. Training architecture

This need of communication between processes makes more difficult to parallelize the whole process than in the trivial case where they do not need to communicate to each other.

The naïve approach to perform the training and deal with the communication between the two processes is to face it as a sequential problem. In this sequential problem, one episode of simulation is performed, and then the network is trained with the data available.

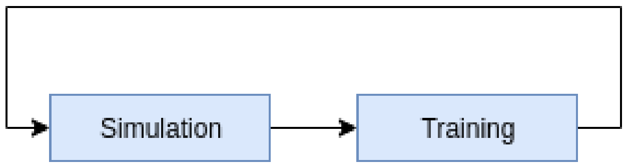

Fig. 4. Sequential approach.

As a conclusion, although the use of Deep Q-Learning has potential to keep growing, the huge amount of data needed to train makes the training too long and there is a strong need of parallelization to improve its performance.

III. Proposed Solutions

In this project, diverse approaches have been proposed to compute the Deep Q-Learning algorithm in a more efficient way.

A. Sequential Approach

The first implementation coded is the traditional sequential approach. Here, as explained previously, the simulation and training are computed one after the other nested in a for loop. The pseudocode for this implementation is showed in Algorithm 1.

Algorithm 1 Sequential Deep Q-Learning Algorithm

```
1: Initialize variables
2: Create environment
3: Initialize Neural Network
4: for e in episodes do
5:     Initialize environment
6:     while episode not done do
7:         Take action
8:         Save new data in buffer
9:     end while
10:    Reduce randomness
11:    Train Neural Network
12: end for
```

B. *Two Processes, Point to Point Communication*

Taking a quick look to the sequential algorithm, the first improvement that can be made is to parallelize the simulation and training into two different processes. In this case both of them would be running at the same time, and at the end of every single episode a communication between them would be created to communicate the new weights of the neural network and also the new data collected from the simulation. Fig. 5 illustrates the flow of the process for computing and communication.

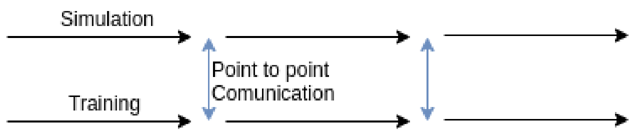

Fig. 5. Point to Point Communication flow

To improve performance by saving time in communication, in this case, instead of communicating at the end of every episode, the time-lapse to communicate is defined by the user in number of episodes. The perfect time-lapse can be different to solve different problems that generate different amount of data during training.

There are several changes that have to be done in the code of the first implementation to achieve this parallelization. However, in this case we are going to explain the following three major changes:

- **Creation of a third network**. Since now we are training and simulating at the same time, a new neural network has to be created for the simulation controller. Besides, although it was not mentioned earlier because of lack of importance, the Deep Q-Learning algorithm implemented uses two networks during the training. In consequence, the class that creates the networks has been modified to create a single new network for each simulation process.
- **Communication of the network weights**. As explained, every n episodes, the training and simulation processes have to communicate. The communication of the weights of the network is done using MPI functions send and recv.
- **Communication of the training data**. Is performed in the same way as the weights.

With this implementation the limit speedup is 2. This limit would be achieved in the ideal case of taking exactly the same time for the simulation and the training, and besides not having any communication overhead.

C. *Collective Communication*

After the successful implementation of the first level of parallelization, this approach adds another layer of parallelization by running simulation episodes in parallel using different processes.

The decision of parallelizing the simulation process is based in two different reasons:

1. Simulations for real world scenarios are complex in most cases, and the time to simulate a single episode may be equivalent to the training of several "episodes" for the network.
2. The training of the network can be run in a cluster using Tensorflow API in case of having a very complex neural network.

Once the reasons for this implementation are clear, we can explain how it is done.

The communication topology consists on a star configuration having as main process the one that is running the training of the neural network. This process gathers all the data generated by the simulation processes and it sends them the updated network every n episodes. This architecture is depicted in Fig. 6.

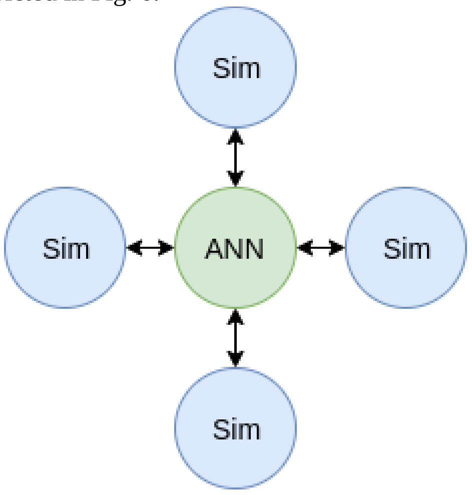

Fig. 6. Collective Communication architecture

Regarding the flow of the algorithm, the simulation processes run n episodes each one before the communication and the network perform the training for n*p "episodes", being p the number of processes that are running simulations during that time. With this, what we are doing is parametrizing the number of training steps that we perform to the network based on the number of processes available to run simulations. After all the processes have finished running, they have to synchronize to communicate as shown in Fig 7.

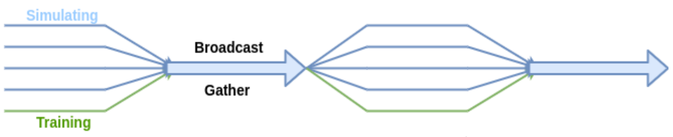

Fig. 7. Collective Communications flow

In this case again, there are several changes needed for the algorithm, but in here we are just going to explain those that are most interesting for this project.

- **Gathering of the testing data** by the training process using the MPI instruction gather.
- **Broadcasting the network weights** from the training process to all the processes using the MPI instruction bcast.
- **Calculating the parameters** to define the number of simulations for each process.

With this approach, the improvement with respect to the one explained before is considerable for cases when the simulation process is heavy. However, the mayor drawback for this approach relating to runtime is the necessity of having to synchronize all the processes each time that the communication has to be done. Even so, this necessity of synchronization may be beneficial for the implementation, making easier the election of some training parameters.

D. *Asynchronous communication*

In order to speed up the process even more than using collective communications, I tried to implement an approach using asynchronous communication. However, the use of asynchronous communication produces an error when trying to send the weights of the network. I have tried the same code sending smaller arrays and the network weights, and while one of them was working the other is popping an error. This error is due to the fact of using Pickle communication by Python, which is the Package that is causing the error. Also, although not fully working, the implemented code is available in Github and here I will explain the logic of the approach.

In this case, just as before, we have one process to train and the rest are simulating. All the simulation processes are checking every n steps if the network process has sent a new network, if it did, they send the accumulated data to the network process and keep simulating. If they have not received anything, they keep simulating until the next check. On the side of the network process, the philosophy is the same, it keeps training and each n steps checks with the rest of the processes separately if they have sent new data, and if they did, it sends them the new neural network. The workflow of this approach would consist on having all the processes running in parallel and checking for data periodically as shown in the Fig. 7.

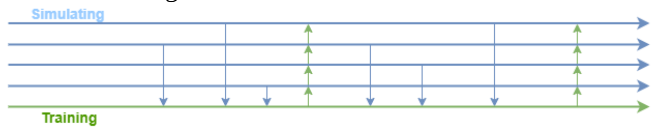

Fig. 8. Asynchronous flow

With respect to the algorithm, while this approach would improve its runtime, it would also add some issues. Since in this case everything is running asynchronous, is more complicated to define the reduction of randomness during the training. With the others approaches, since we are controlling how much the network train while the simulation is running, we can control that randomness reduction based on that. However, in this case, it could happen that the simulation during the first phase of training is very fast because it has not knowledge about the environment. In that case, the randomness would decrease too fast for the knowledge acquired. So probably it would be necessary to adjust dynamically the randomness decreasing parameter as the training advances.

IV. Runtime Study

A. *Cartpole Control*

The cartpole problem was the first one to test. This system consists on an inverted pendulum that has to be balanced by moving left or right. The rendering of the environment can be seen in Fig 9.

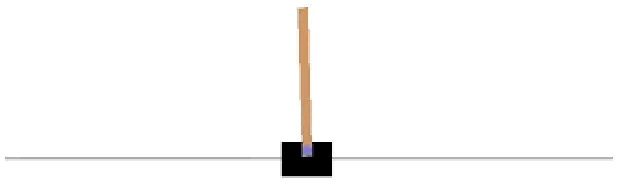

Fig. 9. Cartpole environment

The election of this problem is due to its simplicity. The network needed for controlling is relatively small and the simulation of the environment is also fairly simple.

In order to make the tests, the algorithms are run for 500 episodes of training in all the cases. The results for cartpole problem are shown in Table 2.

When running simulation and training in two separate processes with point to point communication, we get the same speedup than when we use multiple processes for simulation. This means that the time to run the simulations is faster than the training, so adding this new layer of parallelization is not needed.

B. *Lunar Lander*

The second problem to test consists on landing a spaceship, the control is done on the motors and the spaceship has to be properly landed in a determined place as shown in Fig. 10.

Table 1. Cartpole runtimes

	Cartpole					
	Standard	2 Processes	MP (2)	MP(3)	MP (4)	MP (5)
t (s)	110.5	73.38	78.54	80.75	70.63	77.5
Speedup	1	1.506	1.407	1.368	1.564	1.426

Table 2. Lunar Lander runtimes

	Lunar Lander					
	Standard	2 Processes	MP (2)	MP(3)	MP (4)	MP (5)
t (s)	432.63	319.52	369.43	179.68	130.31	87.61
Speedup	1.000	1.354	1.171	2.408	3.320	4.938

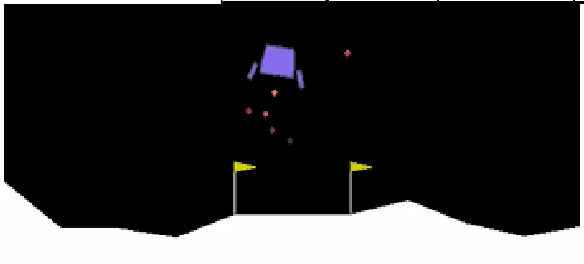
Fig. 10. Lunar Lander environment

The complexity of this problem is bigger than the first one regarding simulation, but the problem can be solved with a fairly small neural network.

Looking at the results shown in table 2, we can see that using point to point communication the speedup is relatively small. This means that one of the processes is the one that is taking the biggest amount of time during the sequential approach. This belief is corroborated with the results when using collective communication. The speedup in this case keeps increasing until reaching 4.94 when using five different processes. With this we can conclude that as expected, the simulation of the process was the most time consuming process, and in this case the extra layer of parallelization for the simulation is very convenient in terms of performance.

A. Box slider

This environment has been created using the software VRep. It consists on an Unmanned Ground Vehicle (UGV) that has to move a box to a given position. The information that the controller gets is the image from a camera that is placed over the robot. An image of the environment can be seen in Fig. 11.

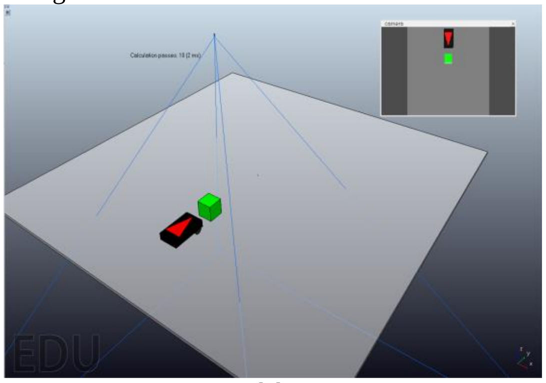
Fig. 11. Box Slider environment

The complexity of this problem is bigger than the ones explained previously. Regarding the training of the neural network, it is slower than in the other cases because the complexity of this network is not comparable to the others. Regarding the simulation of the environment, the complexity also grows, this simulation has to deal with collisions, and besides, there is also communication between the software and the scrip in every step, which makes the simulation even slower.

The results for this case have been only tested for the standard implementation and the one with collective communications with five processes. The main reason is that the runtime in this case is too long. The speedup gotten is 1.63. Obviously, this indicates that the training of the network is slower than the simulation in this case. However, in a real world scenario, the speedup could be greater. Here, we have only one computer available to run the experiment, and all the networks needed do not fit in the GPU. On the other hand, if we were working in a real cluster with multiple GPUs, the runtime for the training would be shorter, and in consequence, we would improve the speedup by eliminating that bottleneck.

Table 1. Box Slider runtimes

	Box Slider	
	Standard	MP (5)
t (s)	7780.38	4765.37
Speedup	1	1.633

V. Conclusions

In this project we have studied three different approaches to parallelize the Deep Q-Learning algorithm using a distributed system based on MPI. The first approach parallelizes simulation and training, and it reaches a speedup of around 1.5 in the best scenario. This improvement is significant for long time processing algorithms like this one. The second approach proposes several processes running simulations with collective communication. In cases when the training of the network is heavy, this approach does not improve the first one, but the speedup raises to almost 5 in one case where the simulation workload is larger. Finally, the last approach consists on asynchronous communication, testing if there is new data periodically. It has not been fully implemented due to issues with the format of the weights of the network and asynchronous communications. However, in case of implementing it, further research would be needed to make it work correctly in an asynchronous way.

As a conclusion, the results obtained satisfy the need of performance improvement for this kind of algorithms, and the speedup could be raised even more in the case of working in a real cluster with several CPUs and GPUs available.