Distributed SGD

Sylvain Beaud, Gregoire Clement, Maxime Delisle
Systems for Data Science, EPFL

I. INTRODUCTION

Nowadays, robust and reliable systems are a core component of a respectable setup, for this reason, we focused more particularly on this side of the problem.

One of the main highlights of our implementation is the possibility to add and remove workers at will and at any time. Indeed, in the synchronous implementation, the coordinator monitors the number of workers and if a worker crashes, the computation can continue without him. On the contrary, if the user want to add more workers to the system, the new workers will connect to the coordinator or other workers and the computation will continue with these additional workers. In the asynchronous version, a new worker arrives, it will retrieve the list of workers from another worker and broadcast its updates to them and receive their computations; this is the only phase where a locking mechanism is used. When a worker encounters an error, it broadcasts an error message to the other workers and they will stop to communicate with the faulty node.

Another interesting feature of our implementation is the fact that once the computations are finished, the logs and statistics are uploaded and stored on transfer.sh and can be downloaded for a later use. We have also put options to adjust the level of verbosity of the logs.

II. DESIGN AND IMPLEMENTATION

For the basics, we have append the figures 5 and 6 at the end of this document. They schematically represent some of the design choices we made for both implementations.

For the milestone 2, we adapted the version of our worker so that each worker communicates with the other workers and a coordinator is no longer needed. To do so we adapted our GRPC communication protocol so that when a new worker arrives, it sends a Hello message sharing its details to another worker in order to obtain the current list of workers. As soon as the list is retrieved, it starts broadcasting its updates to the other workers. The other main difference with the synchronous communication protocol is that the workers broadcast their computations and do not wait on a response from their fellow workers. The workers broadcast only the updates of the weights, so it is a sparse vector that consists of the index and the delta that must be applied to the weights. Another important point is that each worker is responsible for a part of the weight set. For example, when two workers communicates together e.g. worker 1 and worker 2, they only send the updates that concern these two workers. It limits the size of the communications and avoid sending multiple times the updates of the other workers to worker 2 through worker 1. And since every worker communicates with all the others, the system will eventually be consistent.

Since workers can be added and removed, it is difficult to partition the data evenly between the workers because at some point, all the workers except one could be removed and the computations would continue only on a fraction of the data. So in a attempt to make the system as robust as possible, all the workers read the full dataset but they load only a subset of the data in memory. Moreover, we used lazy evaluation to quickly load the dataset and launch the computation as soon as possible.

We also noticed that a limiting factor in the computations was the computation of the loss term. So we decided to compute the loss only on the first pod i.e. hogwild-pod-0 and the others workers can compute the gradient updates at full speed.

Concerning the stopping criteria, the system stops when a minimum loss is reached. However, the system can stop earlier if there are no improvements for a chosen number of consecutive computations. We set up the ”early-stopping” policy such that only the best weights are kept.

III. EXPERIMENTS

For the choice of the hyper-parameters such as $\lambda = 0.0001$, we tested several settings and selected the best one based on a grid search. Also using grid search, we found our best results with the value 0.35 for our step size.

The training loss and the testing loss are displayed on the figure 4. We can see that both losses decreases quickly and tend to a limit as expected.

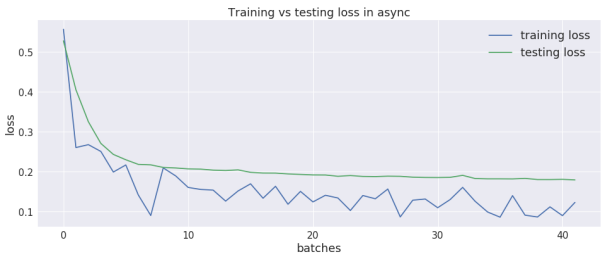

Fig. 1: Plot of the training and testing loss over all the iterations

To show the results of our experiments, we decided to plot the speedup of the computation of the loss as the number of workers grows just like it is done in the Hogwild! paper [1]. So we plotted the convergence speed and the speedup phenomenon for the asynchronous version. The results are displayed on figures 2 and 3 respectively.

We can see that the parallelization of the computation by adding more workers speed up the convergence but up to a certain limit. Indeed, with too many workers, the overall performance decreases as the number of communication increases.

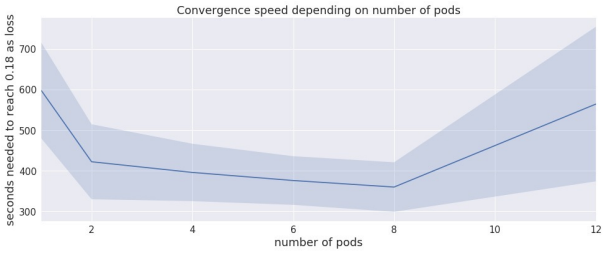

Fig. 2: Plot of the training and testing loss over all the iterations

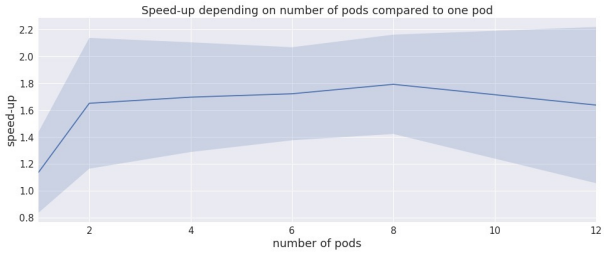

Fig. 3: Plot of the training and testing loss over all the iterations

About the comparison between the synchronous and asynchronous versions, we have seen that the asynchronous version is much faster than the synchronous one. It can be explained by the locking phase of our system. In fact, in the synchronous version, workers are only computing gradients when they are asked to. After they're done they wait for the coordinator for another batch. On the other hand, in the asynchronous version, we keep computing all the time, loss computations, weights broadcasting and weights receiving are all handled in parallel.

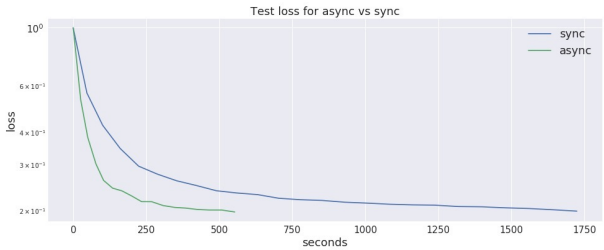

Fig. 4: Plot of the testing loss for 4 pods in async and sync

Note that our speed-up is a bit biased (shown on figure 3) because our implementation allows the addition of workers so the first workers start to compute as soon as their pod is ready. So with this in mind, the fact that the workers are added incrementally by Kubernetes in a Statefulset framework means that not all workers work for the full duration of the entire iteration. For example, during our tests, we got the results displayed in the following table I.

# of pods	1st loss	2nd loss	3rd loss	4th loss
3	1	2	3	3
4	1	1	3	4
8	3	6	8	8
12	1	7	12	12
18	5	5	8	8
28	4	11	20	20

TABLE I: Number of pods that compute the first, second, third and fourth loss term for several setups with the same hyper parameters except for the number of pods.

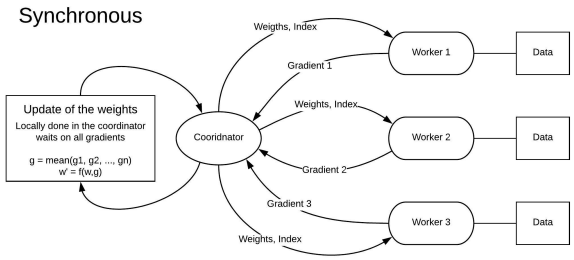

Fig. 5: Design of the Synchronous version : Each worker has its own copy of the data and receives the indexes and the weights from the coordinator.

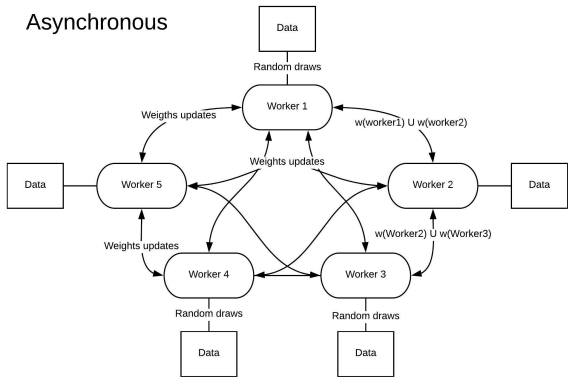

Fig. 6: Design of the Asynchronous version : All workers communicate with each other and send their weights updates. If you take a closer look at a weight update between two workers, you can see that we aggregate the weights of those two specific workers.

REFERENCES

[1] Recht, Benjamin and Re, Christopher and Wright, Stephen and Feng Niu, Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent, 2011.