Distributed Synchronous and Asynchronous SGD

Henry Declety
henry.declety@epfl.ch

Lucas Gauchoux
lucas.gauchoux@epfl.ch

Oussama Abouzaid
oussama.abouzaid@epfl.ch

***Abstract*—In this paper, we discuss our approach towards building a distributed SGD —in both synchronous and asynchronous fashions— which we use to train an SVM. For both implementations, we shall describe the architecture we opted for, the tools and libraries we used and finally, we report the performances of both architectures including the training/validation loss curves, the time until convergence and some hyperparameter search.**

I. Introduction

For convex optimization problems, SGD remains one of the broadly used algorithm for large-scale machine learning problems due to its simplicity, as well as the easiness to compute a gradient on a convex target function. The Hogwild! paper[1] suggests that in a highly sparse problem (and hence the weight vector), lock-free updates on the same components of the weight vector are unlikely to happen, and even in the case it does happen, the effect of these overwrites can be neglected since SGD performs updates in a stochastic way.

II. Dataset and preprocessing

The dataset RCV1 (Reuters Corpus Volume 1) is a collection of newswire stories recently made available by Reuters Ltd. It contains 781'265 samples (indexed by a unique id), and 47'236 features (float). The data is gathered from 4 files. Topic codes were assigned to each sample to capture the major subjects of a story (e.g. CCAT, ECAT, GCAT). Since the dataset is highly sparse (around 0.16%), we store our samples along with their features in a sparse form using a list of dictionaries, that we call X.

We extracted all the topics that each sample belongs to. For a fixed topic, we set the corresponding label y_n to 1 or -1 depending on whether the sample belongs to this topic or not, respectively. This allowed us to reduce our problem into a binary classification. We perform a 80%-20% split of the dataset for the training-test sets to avoid overfitting. For testing, we run your implementation on the binary text classification task CCAT.

III. Support Vector Machine

As a recall, Support Vector Machine (SVM) is a model that optimizes the following cost:

$$\min_{w} \sum_{n=1}^{N} [1 - y_n x_n^T w]_+ + \frac{\lambda}{2}\|w^2\| \tag{1}$$

where N is the number of samples, x_n a datapoint n, w is the weight vector, λ the regulizer term. The first term is the Hinge Loss, and is defined as $[z]_+ = \max\{0, z\}$.

Since this function is convex, we can use SGD to optimize the weight w:

$$w^{(t+1)} = w^{(t)} - \gamma \nabla w^{(t)} \tag{2}$$

where $\nabla(.)$ denotes the gradient, and γ is the learning rate.

The aim of our both implementations is to parallelize the computation of the gradient across the workers, hoping for a faster convergence, without affecting the accuracy.

The Hogwild! algorithm utilizes lock-free gradient updates, i.e. the weight vector w is updated by multiple worker nodes at the same time with the possibility of overwriting each other. It is crucial to note that the correctness of this algorithm lies under the condition that w is sparse, which is indeed our case. Collisions are therefore unlikely to happen.

IV. Communication

For the communication, we used gRPC and Google Protocol Buffers, which automatically generate idiomatic client and server stubs for our service.

A. Asynchronous Model

In the asynchronous model, we set up N workers nodes that will communicate between each other. Since there is no coordinator per se, we save one extra-hop whenever a node has to communicate with another node. In this setup, the data is stored at the worker nodes. All the nodes start with the same weight vector w. Each one of them computes a loss over a random sample, updates the vector w and send the updated weight to the $N-1$ other nodes. At the end of each gradient step, the worker update it's weight vector with the updates from the other nodes that were received during its own gradient computation. Unlike the synchronous setup, a node proceeds to the next random sample right after it has completed computing one gradient. Nodes no longer wait for their peers to proceed to the next iteration. Figure 1 is a simplified sketch of the asynchronous architecture.

In short, we changed our previous code by removing the coordinator, and making the workers able to directly communicate between each other: A worker computes the gradient, updates the weight and send it to his peers, then pops a new received weight from the queue to repeat the same process, and so on so forth.

For both setups, the convergence is attained when 2 successive losses computed by a worker is less below 100. After running over for a large number of times, we estimated that this amount is sufficient for the convergence as both the losses and accuracies barely change beyond that point.

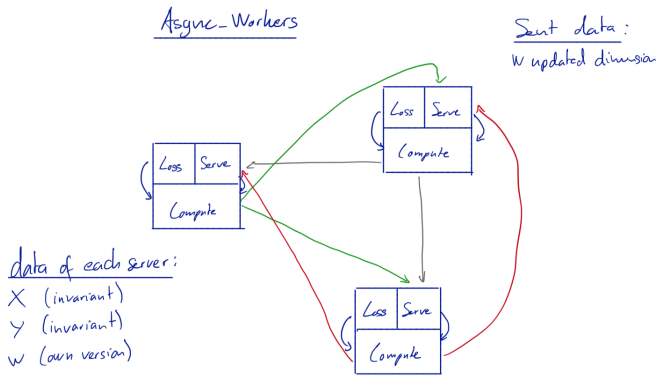

Fig. 1. Asynchronous model architecture with $N = 3$ worker nodes communicating between each other

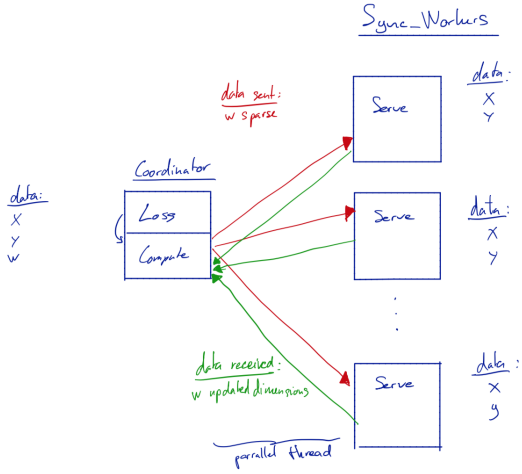

Fig. 2. Synchronous model architecture, 1 coordinator and $N = 4$ worker nodes

B. Synchronous Model

In the synchronous model, we set up a coordinator and N server stubs as workers. The data is stored at the N workers, and each worker reads a file of the data. The coordinator sends the weight vector w to each of the worker nodes, who pick a random sample from their data, compute the stochastic gradient of the Hinge loss[2] on their sample, and send the result back to the coordinator, which updates the weight vector. The coordinator waits for a response from all the N workers before sending back the new weights to the workers. Figure 2 shows a simplified sketch of the architecture.

V. Docker and Kubernetes

We built a single docker image that contains all of our python code exposing port 50051 and we pushed it to Docker Hub. The docker image installs some libraries but does not run any code automatically.

Then we built a Kubernetes StatefulSet that instantiates N replicas containers of that image (exposing port 50051) and mounts the data-sets in a folder inside of our container. We also have Headless Service to make every pod accessible to one another.

To deploy our implementation, we launch N pods with the statefulset and then in we run in each pod a python script (worker or coordinator) passing as argument all the necessary information for the communication with other pods. Whenever the job is done, we delete the statefulset.

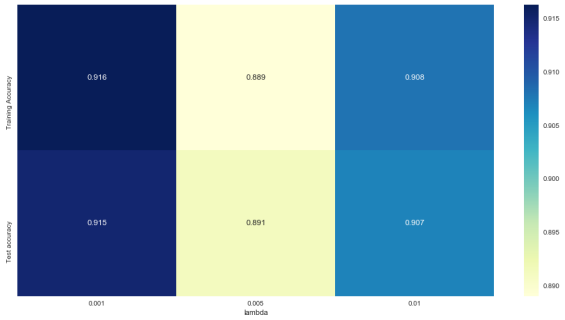

Fig. 3. Training and test accuracies for $N = 4$ workers, with various λs for the Asynchronous setup

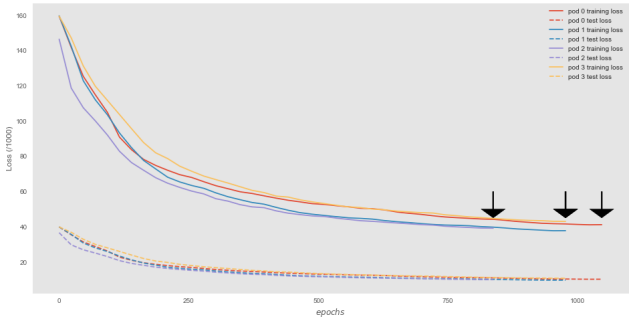

Fig. 4. Training and test losses across the $N = 4$ workers. The arrows indicate the convergence for a pod (worker).

VI. Results

A. Asynchronous Model

For the asynchronous setup, we first run the algorithm using $N = 4$ workers. We then vary the regularizer term λ for the values $0.001, 0.005, 0.01$. We report the accuracies obtained in Figure 3

We can see that $\lambda = 0.001$ performs the best, giving an accuracy of **0.915** on the test set, after **908** iterations, and **2.07 minutes** of running time. We plot the corresponding loss curve for the 4 workers in Figure 4.

We now fix $\lambda = 0.001$ for the rest of the analysis. Let us now have a look at how the accuracy and the convergence time vary as a function of the number of workers. We run the algorithm again for $N = 4, 10, 15$ workers. The runnning times results are shown in Figure 5, and the corresponding training/test accuraccies in Figure 6

For $N = 10$ workers, the asynchronous setup gives the best accuracy, yet with a more significant running time (**6.75 minutes** $>>$ 2.07 minutes for $N = 4$ workers).

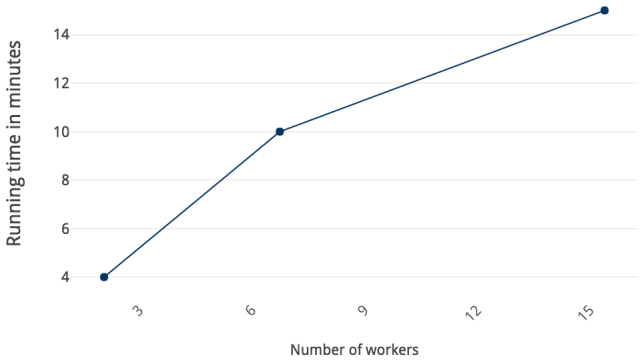

Fig. 5. Convergence time as a function of the number of workers

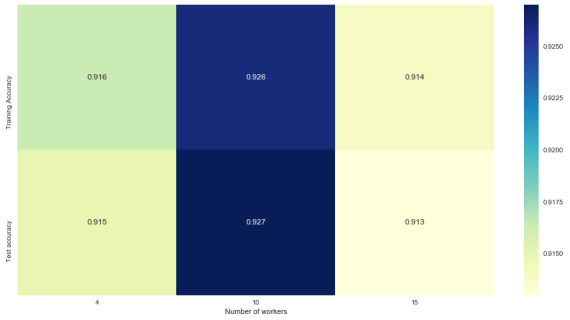

Fig. 6. Accuracies as a function of the number of workers

B. Synchronous Model

For the synchronous model, we run the algorithm for $N = 4, 10$ workers. Let us first plot the loss curves over the epochs, still for $\lambda = 0.001$ (best overall performance).

An accuracy of **93.05%** is attained on the test set, after 3313 iterations and **19.54 minutes** of running time.

VII. Synchronous vs Asynchronous

The results from Table I and II show that the asynchronous implemenation of SVM converges significantly faster than than the synchronous one, however, with a (slightly) lower accuracy.

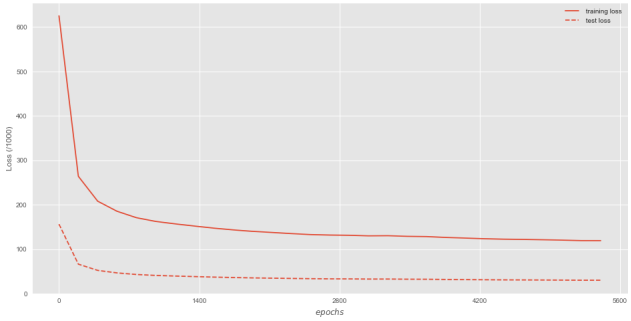

Fig. 7. Synchronous model architecture, 1 coordinator and $N = 4$ worker nodes

TABLE I
Synchronous setup

Nb workers	Acc.	Runtime (mn)	Avg # epochs
4	93.12%	19.74	3300
10	93.56%	35.19	1200
15	92.92%	19.89	941

TABLE II
Asynchronous setup

Nb workers	Acc.	Runtime (mn)	Avg # epochs
4	91.58%	**2.07**	**908**
10	92.76%	6.75	950
15	91.32%	15.4	930

Notice that in the asynchronous version, SGD exhibits a significantly fewer number of epochs, and this is due to the fact that some pods (workers) run faster than others, and this explains also the low running times; the pods run concurrently and update the weights without weighting for the 'slower' pods. There is therefore no bottleneck node.

VIII. Conclusion

After several testing on different conditions, we could not conclude that having more workers necessarily speeds up the iterations. However, opting for an asynchronous implementation is worth being used due to its very efficient running time. This work was based on the Hogwild! paper where the implementation was based on a multiprocessing asynchronous approach on shared memory. In those conditions an improvement of the optimization was observed. We can therefore interpret those result to conclude that as in many distributed implementations, communications between machines are the bottleneck of such systems.

References

[1] Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent, Feng Niu, Benjamin Recht, Christopher Re and Stephen J. Wright. 2011
[2] Hinge Loss, https://en.wikipedia.org/wiki/Hinge_loss
[3] gRPC Remote Procedure Calls, https://grpc.io/