Parallel SGD in Spark

Timoté Vaucher, Patrik Wagner, Thevie Mortiniera

***Abstract*—This project relies on a previous synchronous and asynchronous implementation of Hogwild-Python [1] during the course CS-449 "Systems for Data sciences" where the main reference is the original HOGWILD ! paper [2]. Our aim is to experiment and compare our synchronous implementation in Spark with the Python synchronous and asynchronous versions of stochastic gradient descent (SGD), used for learning support vector machines (SVM). We managed to obtain similar results in similar time as the synchronous version of Hogwild-Python.**

I. Introduction

Stochastic Gradient Descent (referred to as SGD in this report) is one of the most popular training algorithm used in Machine Learning. However, its iterative nature makes it inherently serial. One way to overcome this obstacle is through parallelization over mini-batches. Because the size of the mini-batches is crucial for the performance accuracy and is also limited by computer memory, the need to find a fast and stable solution for parallelizing training over multiple independent nodes (computers) to achieve higher speedups has become more and more important. One such solution would shrink the development cost and would help resolve scalability issues. During the past years, multiple solutions have been proposed to tackle this challenge [3].

In this paper, we focused on the python implementation [1] of the original **HOGWILD !** paper [2], and proposed an implementation in Spark. We were asked to run different experiments and determine which of the three implementations, among our synchronous implementation in Spark and their synchronous and asynchronous implementations in Python, produces the best-quality result in the shortest amount of time and under which conditions. As in the Python implementation, we were given the Reuters RCV1 Dataset, which contains pre-classified English news. The training dataset was quite balanced since $9'653$ are labeled as "CCAT" and $11'154$ don't have a "CCAT" label, which gives us a $46/54\%$ ratio. It was composed of $23'149$ samples for training, which split between a training set and a validation set representing 10% of the total set, and $781'265$ samples for the test set. Only 77 columns out of the $47'237$ features are filled in average, turning the problem into a very sparse one [1].

II. Synchronous Design and Spark Implementation

A. Synchronous Design

In this parallel synchronous implementation, we want to have a coordinator that manages the application, sends tasks to the workers, collects the gradients and some executors that do all the computations : prepare the data, compute the gradients and do the predictions. In **Hogwild-python** this as to be taken care of by the programmer, using dedicated libraries like $gRPC$.

In Spark, this communication layer is hidden and we take advantage of the RDD (*Resilient Distributed Datasets*). We use them to apply transformations and actions on our data in order to compute SGD in a distributed manner. Each executor receives a partition of the RDD and apply map functions to it to return its part of the gradient. This is also more efficient, as we distribute the load when preparing the data and making predictions. A serious disadvantage of this approach is that we are limited by the different synchronization barriers imposed by spark and that we have to wait for every worker to send its gradient back to the coordinator at every epoch.

B. Spark implementation

Because of the important sparsity of the problem, they used dictionaries as a way to efficiently store the feature vectors, while we chose to use Scipy sparse matrices [4]. Furthermore, when reading the **Howgwild-python** implementation, the report mentioned the regular hinge loss function, i.e,

$$\mathcal{L}(w) = \frac{1}{n}\sum_{i=1}^{n} max(0, 1 - y_i w^T \hat{x_i}) + \lambda \left\|w\right\|^2$$

while they implemented in their code (with a missing minus sign in *SVM.__regularization_gradient* in order to remain consistant with *SVM.__gradient*) the loss function stated in the original **HOGWILD !** paper, i.e,

$$\mathcal{L}(w) = \sum_{i=1}^{n} \left(max(0, 1 - y_i w^T \hat{x_i}) + \lambda \sum_{u \in x_i} \frac{w_u^2}{d_i} \right)$$

Where u are the indices non-zero features of a given point x_i and d_i is the number of non-zero features of x_i.

Hence, we adapted and followed the loss function implemented in their code, hence, the one in the original paper, to be in experimental conditions as fair and comparable as possible. For the interested reader, one might be interested in comparing results with the original loss function as it's implemented (but commented) in the source code which yields similar results on a synchronous architecture.

III. Running on the Cluster

The idea behind running the spark version on the cluster is very similar to the one implemented in **Hogwild-python**, except that spark is bundled with some useful utilitaries and ressources :

- *spark-submit* : that handles most of the bulk work of pulling the Docker image, creating the containers, coordinator and executors and running commands on Kubernetes
- *docker-image-tool.sh* : a customisable file, that facilitates the creation and deployment of spark images to Docker
- *Dockerfile* : a base Dockerfile, that we modified to suit our needs, including extra python libraries, like *numpy* or *scipy* needed for our computations.

Thus, the overall steps to run on the Kubernetes cluster are :

1) Build and push the image from the Dockerfile using *docker-image-tool.sh*
2) Remove the old infrastructure
3) Run the app using *Spark-submit*
4) Once the job is done, the coordinator writes a log in its container
5) Copy the logs on our machine

In order to observe the server load, we ran the same tasks on a random day during the week prior to the deadline and on the last Sunday and observed up to a two-time slowdown as probably many colleagues were trying to run their setup at that time.

IV. Experiments

The chosen metrics to compare our implementation and **Hogwild-python**'s were the validation loss and the accuracy. In order to find the conditions under which the best performance was achieved in order to compare the models, we first needed to first determine the effect of the mini-batch size of SGD on the running time, the effect of the number of workers on the convergence time of the loss function, and the learning rate.

A. Effect of SGD mini-batch size on the running time

Because we are using SGD and not GD, we wanted to see how the batch-size affected the running time performance for 100 iterations.

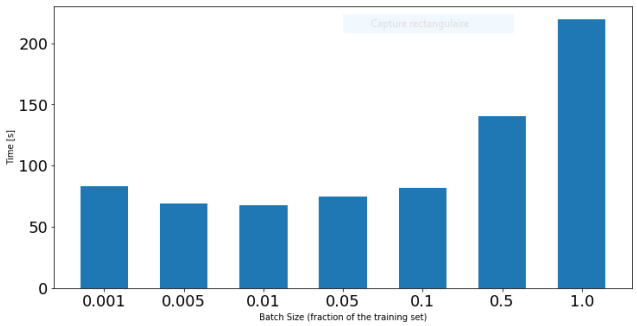

Fig. 1. Effect of batch size on time per iteration performance. The batch sizes here are in fact a fraction of the training set.

One can clearly see, if the batch size is too small (i.e. 0.1%), some performances are lost because too much time is spent in communications, pushing the gradients back to the coordinator and the weights to the executors. It's interesting to note that taking 1% of the training set or 100% only results in a 3-time increase of the execution time. Thus our application is effectively bounded by synchronization and communication time, inherent to spark and synchronous applications. For our implementation and in order to be able to compare with **Hogwild-python** we restrict ourselves to using using batch-size between 0.5% (roughly the same batch size as in **Hogwild-python**) to 2%.

B. Effect of the number of workers on the convergence time of the loss function

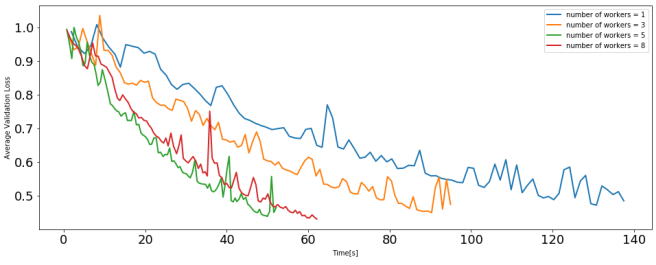

Fig. 2. Average validation loss as a function of time per number of workers

When looking at the plot above, we noticed that 5 workers was the perfect balance between the number of messages exchanged between the workers and the

important number of computations to do on a single machine. Indeed, at first the trend suggested that the higher the number of workers, the lower was the convergence time, but it was not true anymore when we moved to 8 workers, since it took more time than in the case with 5 workers. We also wanted to try with 16 workers, however, we did not succeed to do it, since Spark did not allow us to start 16 executors on Kubernetes.

C. Model selection

We ran a grid-search to tune our hyper-parameters and selected the model which returned the best performance on the validation set. We ran this three times because we use SGD, which is a highly stochastic approach and we wanted to get the most stable parameters. The hyper-parameters were searched among the following : $\epsilon \leftarrow$ [0.005, 0.01, 0.015, 0.02, 0.03, 0.04, 0.05], *batch fraction* $\leftarrow$ [0.005, 0.01, 0.02], $\lambda \leftarrow$ [1e-6, 1e-5, 1e-4, 1e-3]. The learning rate was chosen in this interval because with $\epsilon > 0.05$, the SGD algorithm doesn't converge on our set and with $\epsilon < 0.005$, the algorithm converges too slowly. The batch fraction interval comes from the previous part where we found the wished batch size range. The regularization factor was chosen to allow some margin in the hyper-plane in order to allow misclassification.

The table below illustrates the performance of the different models regarding their hyper-parameters settings.

	learning_rate	lambda_reg	frac	training_accuracy	validation_accuracy	validation_loss
0	0,04	1,00E-06	0,005	0,914322804	0,904255319	0,4226785
1	0,015	0,001	0,01	0,910332228	0,903404255	0,450029582
2	0,015	0,001	0,01	0,908120583	0,90212766	0,450528102
3	0,03	1,00E-06	0,005	0,910476465	0,901702128	0,474771945
4	0,04	1,00E-05	0,005	0,908986009	0,9	0,403138276
5	0,015	0,001	0,01	0,908505217	0,898297872	0,465293564
6	0,02	1,00E-06	0,005	0,902687629	0,897021277	0,564912674
7	0,015	1,00E-06	0,01	0,902110678	0,895319149	0,444288679
8	0,015	0,0001	0,01	0,907591711	0,894468085	0,468721154
9	0,02	1,00E-05	0,005	0,901004856	0,894042553	0,566230799
10	0,04	0,001	0,005	0,90263955	0,893191489	0,401993448

Fig. 3. Top 10 Grid Search results based on accuracy. The highlighted results are the best settings.

When looking at the results, we found that the best and most robust model was the one highlighted three times : $\epsilon = 0.015$, $\lambda = 0.001$ and frac = 0.01.

D. Models comparisons

We wanted to compare the performance of our best synchronous model against theirs. When looking at their code and report, it was not clearly stated whether they ran a grid-search for their best models selection. We assumed, as a rule of thumb, that their presented model was the best one.

1) Loss convergence versus time: As expected, we obtained similar results as the previous implementation in Python, the asynchronous architecture still being the best one.

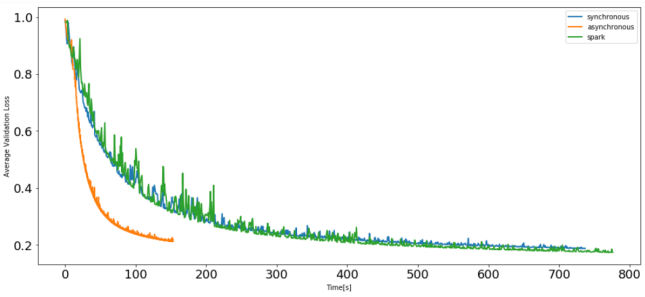

Fig. 4. Average loss of the three implementations on the validation set across time.

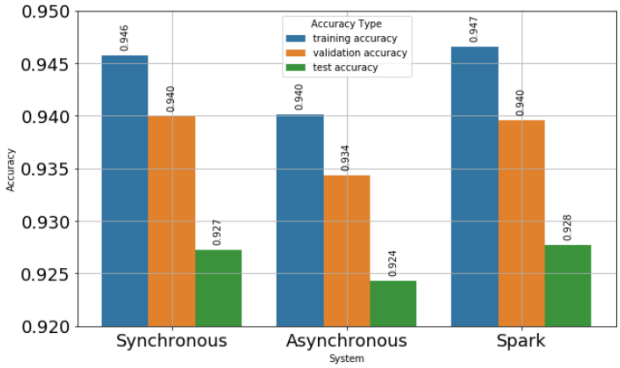

Fig. 5. Accuracy performance of the three implementations on train, validation and test set.

2) Accuracy performance: When observing Fig.5, we noticed that we obtained similar results than the **Hogwild-Python** for the training, validation and test accuracy.

V. Discussion

As mentioned previously, with a lack of clear information, we assumed that their presented model was the best one. We can argue that this assumption is too presumptuous and it might be a good idea to either have confirmation from the authors or modify their implementation to run a grid-search and run the experiments again to assess or deny our conclusions. Furthermore, our implementation mostly used Spark *map* transformations while we could have used the power of Spark *mapPartitions* to probably increase our

performance. Indeed, the latter may be faster than the former since it calls functions once per partition, not once per element reducing the overhead costs [5].

A way to make our results more robust in Fig.5 would be to use *Cross-Validation* instead of Train-Validation split. With this method, we can produce the mean accuracy and the standard deviation for each system. Then, we use statistical significance tests in order to strengthen our assumptions that a system is better than another one. In the state of the art, we cannot decide on which system is the best looking at Fig.5 since the results are very narrow.

Note on how to improve ***Hogwild-Python****:* It could be more advantageous to use optimized and more efficient libraries like *numpy* for the weights and indexing datasets and *scipy* to store the features as Sparse matrices using the native *dot* function to compute dot products.

VI. Conclusion

Throughout our project, we ran multiples experiments to assess models performance using validation loss and accuracy at train time as metrics. Our synchronous implementation in Spark obtained similar results as **Hogwild-Python** synchronous implementation with the asynchronous architecture still being the best one. However, using Spark framework, allow us a more efficient code and simplify deployment on the cluster. In a future work, provided more time at our disposal, it would have been interesting to investigate more advanced frameworks like GraphLab[6], which looked really promising.

References

[1] L. Bifano, R. Bachmann, and M. Allemann. Distributed asynchronous sgd. [Online]. Available: https://github.com/liabifano/hogwild-python

[2] B. Recht, C. Ré, S. J. Wright, and F. Niu, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent." *eprint arXiv:1106.5730, 2011.*

[3] B. Hollnder. Accelerating deep learning using distributed sgd - an overview. [Online]. Available: http://tiny.cc/distribSGD

[4] T. S. community. Sparse matrices. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/sparse.html

[5] R. Ghadiyaram. Apache spark: map vs mappartitions? [Online]. Available: https://stackoverflow.com/questions/21185092/apache-spark-map-vs-mappartitions

[6] Y. L. et. al. Graphlab: A new framework for parallel machine learning. [Online]. Available: https://www.cs.cmu.edu/~bickson/graphlab.pdf