Distributed Synchronous SGD in Spark

Nelson Antunes, Manuel Cherep, Olivier Cloux, Cosmin-Ionuţ Rusu
Systems for Data Science (CS-449), EPFL, Switzerland

***Abstract*—Stochastic Gradient Descent (SGD) is a widely used algorithm that achieves state-of-the-art performance on many machine learning tasks. Different algorithms have been proposed to parallelize SGD both synchronously and asynchronously. This work aims to implement a distributed synchronous SGD using Spark on Kubernetes, and compare the results with previous implementations. We demonstrate experimentally that the asynchronous SGD is faster than both synchronous SGD, and that our alternative in Spark scales better than the previous synchronous implementation.**

I. Introduction

With the rise of Big Data, machine learning has come to the center of attention of many disciplines. However, if the amount and rate of collected data never ceases to grow, our machines reach a phase of "stalling". The everlasting exponential growing of computing power has dramatically slowed down, and all our attempts to overcome physical limitations have failed. To counter this problem, data science has turned towards systems composed of multiple machines.

Stochastic Gradient Descent (SGD) [1] is one of the most popular methods to optimize machine learning algorithms and has proved to be robust and simple simultaneously. The algorithm *Hogwild!*[2] proposes a way to split the computation asynchronously on multiple threads or machines while conserving correctness for sparse problems.

Our paper focuses on implementing a distributed SGD synchronously on multiple machines using the popular distributed framework Spark. This ensures the scalability of the algorithm for enormous datasets while conserving correctness. Then, we focus on comparing our Spark implementation with a synchronous and asynchronous implementations [3] in order to find the best-quality solution.

The mathematical guarantees of Hogwild! are outside of the scope of this paper, but can be seen in detail in the original paper. We focus on solving a classification task over the Reuters (RCV1) dataset (See Section III-A), optimizing the Support Vector Machine (SVM) algorithm with SGD in Spark across multiple workers in a Kubernetes cluster.

II. Distributed SGD

We now present two approaches to implement a high scalable distributed Stochastic Gradient Descent. The motivation to build such a system comes from the fact that Gradient Descent is a very powerful, and widely used algorithm that achieves state of the art results in many classic and modern machine learning applications.

A. Synchronous

In a synchronous fashion, we can design a system with one coordinator and multiple workers. The data is equally divided among every worker and they all compute the gradients for their subset of the data and send the updated weights to the coordinator. Afterwards, the coordinator is responsible for updating the weights vector to be optimized and for computing the loss. Meanwhile, the workers wait until the coordinator finishes updating the weights with all the feedback from each worker. As soon as the coordinator finishes, it tells the workers to repeat the process. Each such iteration is called a *synchronous epoch*. The main issue with this idea is that workers are idle waiting for the slowest worker to finish. The number of message passes scales linear with the number of workers on each synchronous epoch.

B. Asynchronous (Hogwild!)

In contrast, in an asynchronous algorithm the workers can independently proceed to the next epoch without waiting for the other workers. Hogwild! [2] shows that in sparse problems overrides in the memory rarely happen because they do not overlap, and when they do it barely affects the convergence.

In the Hogwild Python implementation [3], every node communicates with every node. The workers now send their updates of the weight vector to all the other workers as well as to the coordinator. Therefore, the coordinator receives the weight update messages, updates its view of the vector accordingly and computes the validation loss. The number of messages scales quadratically with the number of nodes. However, since the workers can independently advance in their computation, we expect this to behave better in practice in terms of running time.

III. Implementation of Distributed SGD in Spark

A. Reuters RCV1 Dataset

Reuters Corpus Volume I (RCV1) [4] contains over 800,000 manually categorized newswire stories in English made available by *Reuters, Ltd.* for research purposes. There are 103 topics and we focus on the topic "CCAT" (Corporate/Industrial) for the task of classification. A row can have up to 47,237 columns, nevertheless on average only 77 of them are non zero. Therefore, the problem is very sparse and ideal for using the approach in Hogwild.

B. Support Vector Machine (SVM)

Support Vector Machine (SVM) [5] is the algorithm that we are optimizing using SGD. We use SVM to perform a supervised binary classification to detect whether a Reuters

news article belongs to the topic category "CCAT" or not. The SVM tries to find the hyperplane that divides both classes (i.e. belong or not belong) with the maximum margin. The associated loss is:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{n}[\sum_{i=1}^{n} max(0, 1 - y_i(\boldsymbol{w}\boldsymbol{x}_i - b))] + \lambda \left\|\boldsymbol{w}\right\|^2 \quad (1)$$

where $max(0, 1 - y_i(wx_i - b))$ is the *Hinge* loss function, y_i is the label of data point x_i, w is the vector of weights and b is the bias. The bias is removed from the equation and it is added as a column of ones to the data. We also introduce a regularization parameter λ which penalises large weights thus avoiding overfitting.

1) Data handling: Each worker receives an equal portion of the data, $\frac{1}{k}$ where k is the number of workers, used to sample and calculate the SGD.

2) Validation Loss and Accuracy: The coordinator (i.e. Spark driver as seen in Section III-D) is in charge of calculating the validation loss after each synchronous epoch. The validation set is *10%* of the given train set. The coordinator also calculates the train and test accuracies at the end of all synchronous epochs.

3) Learning rate: The update rule for the weights is

$$\boldsymbol{w} := \boldsymbol{w} - \eta \nabla \mathcal{L}(\boldsymbol{w}) \quad (2)$$

where w is the vector of weights, η is the learning rate and $\nabla \mathcal{L}(w)$ is the gradient of the loss function. Our learning rate is

$$\eta = \frac{0.03}{|k|} \frac{100}{|batch|} \quad (3)$$

where $|k|$ is the number of workers and $|batch|$ is the batch size. We scale inversely with the number of workers and the batch size to avoid divergence.

4) Stopping criterion: We implement a stopping criterion known as Early Stopping [6]. This is a form of regularization that is used to avoid overfitting. In this case we keep a window of the last *15* validation losses, thereby if the loss in the last iteration does not decrease with respect to the minimum loss in the window, then we stop. Otherwise we run for a total of 1000 synchronous epochs.

C. Kubernetes

Kubernetes [7] is a container orchestration platform widely used nowadays to achieve scalable software solutions. The highly adoption of the microservices architecture contributed to the growing popularity of Kubernetes. Spark [8] provides a Docker image that can be used to submit Spark jobs to a Kubernetes cluster.

The Spark driver lives inside a Kubernetes pod (i.e. the smallest deployable unit of computing). The driver spins, manages the worker pods and sets up the communication between them, since Kubernetes pods are by default isolated.

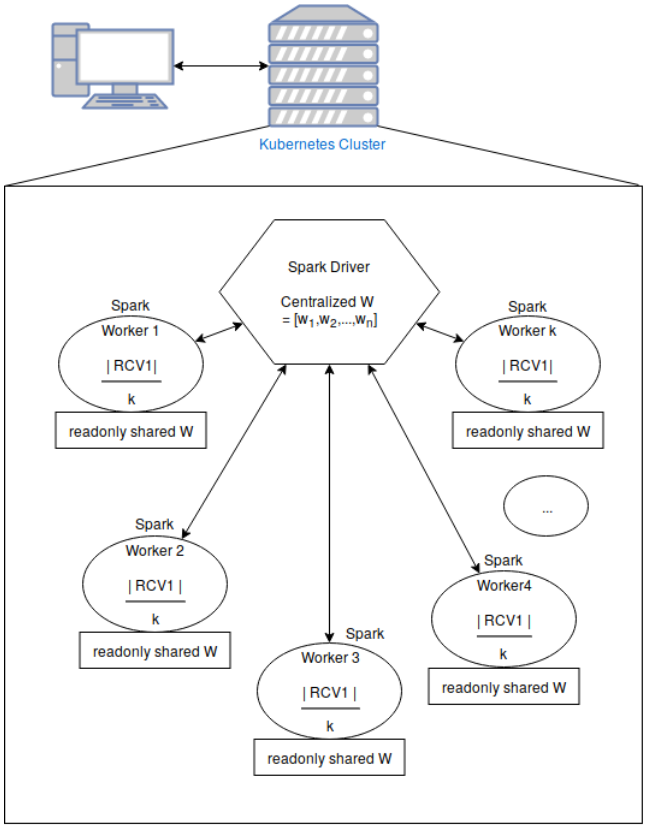

Fig. 1. Architecture design with K workers and W vector of weights

D. Spark Architecture

The Spark architecture implemented is shown in Figure 1. It is integrated by the following elements:

1) Spark Driver: The master node (i.e. Spark driver) is the node that distributes the workers. The training data is first partitioned into k equal subsets that are sent to each of the k workers. The mapping is not exactly known, as the Spark scheduler does this automatically. However, it is guaranteed that every element from a partition goes to the same worker. The driver is also in charge of updating the weight vector and broadcast it (i.e. read-only copy) to the workers for each synchronous epoch.

2) Spark Workers: On each synchronous epoch, each worker calculates the gradients for a *batch* of samples, updating its weight vector (available through broadcasting). Then it sends the result to the driver which updates the weight vector with the feedback from all the workers. The calculation is done in batches, which is not the same as Minibatch Gradient Descent, to avoid communication overload for each epoch.

The inherent design of Spark makes it a synchronous algorithm, where the driver waits for each worker in every epoch, and all the workers wait for the slowest one to finish its batch. Moreover, Spark is resilient to worker failures so that we do not have to deal with them and we trust that Spark will handle them gracefully.

IV. Experiments

In order to analyze and compare our Spark implementation with the previous synchronous and asynchronous implementations [3], we focus on three metrics:

- Running Time: The timing performance for the training, excluding reading data and calculating the test accuracy.
- Test Accuracy: The accuracy over the test dataset once the training has finished.
- Validation Loss Convergence: The convergence of the validation loss over time and its rate.

Our goal is to find the implementation that produces the best-quality result in the terms described above. In order to do so we run experiments exploring a different number of workers and different batch sizes.

All these experiments were run under the same conditions in the IC cluster. The coordinator allocated *20Gi* of memory and each worker allocated *4Gi*. The coordinator needs more memory to calculate the accuracy over the *14GB* of test set. Moreover, as seen in Section III-B we implement the same stopping criterion and the same learning rate. All the results shown below are the result of averaging multiple runs to reduce the bias.

A. Number of Workers

Experiments were run for different number of workers $\{1, 2, 3, 4, 5, 10\}$ in all three different implementations. Figure 2 shows that the execution time in both synchronous implementations increases exponentially. This is due to the fact that for each epoch (i.e. after each worker computes its batch) all workers have to wait for the slowest one to finish. However, we can observe that our Spark implementation scales better with respect to the number of workers. In the asynchronous implementation the execution time remains nearly constant independently from the number of workers.

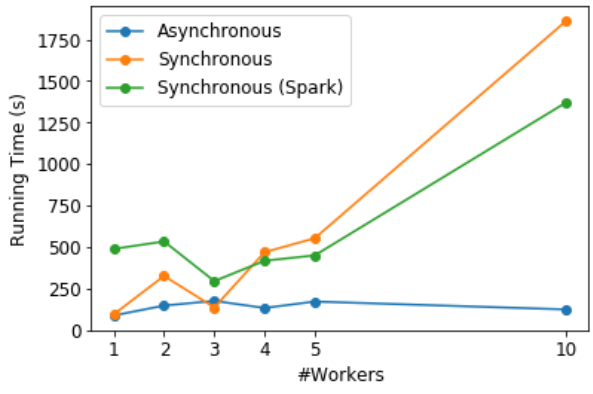

Fig. 2. Running time vs number of workers

It is important to remark that in the asynchronous case each worker sends its weights updates to all other workers. This means that the number of messages over the network is quadratically higher than in the synchronous case. However, this does not affect the performance of the asynchronous implementation, although it could have a higher impact with a higher network latency.

Figure 3 shows the test accuracy for different number of workers and implementations. The synchronous implementation in Spark is stable and achieves always a similar test accuracy. Contrarily, the accuracy for the other two implementations fluctuates with a standard deviation of ± 0.01. Nonetheless, we cannot conclude that either synchronous or asynchronous yields better results because it depends on multiple hyper-parameters that make it non deterministic.

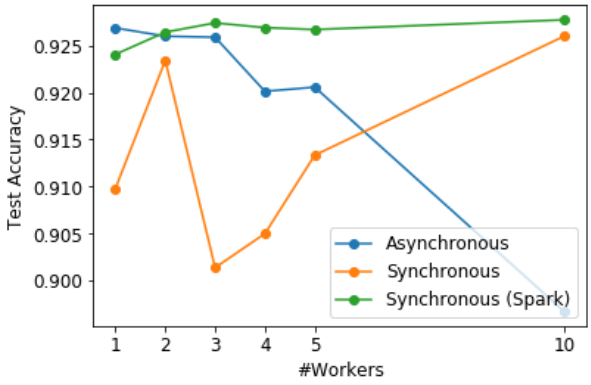

Fig. 3. Test accuracy vs number of workers

Finally, we analyze the convergence of the validation loss. Figure 4 shows the validation loss over time in all three implementations for different number of workers.

The asynchronous implementation is clearly faster converging, and tends to be faster as the number of workers increases. On the synchronous implementations, the one in Spark is faster converging when the number of workers increases and the validation loss fluctuates less.

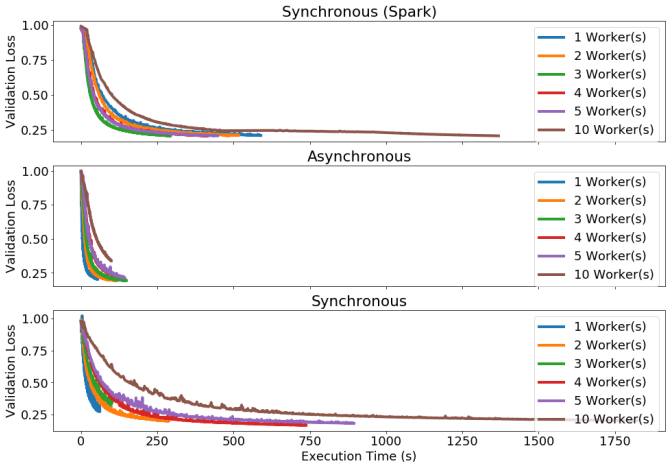

Fig. 4. Convergence of validation loss over time for different number of workers

B. Batch Size

Experiments were run for different batch sizes {*1, 10, 50, 100*} and *10* workers in all three different implementations. Workers calculate gradients in different batches, but the learning rate is adapted accordingly (See Section III-B) to prevent from loss divergence. Figure 5 shows that the execution time in both synchronous cases increases as the batch size increases, while in the asynchronous case remains constant.

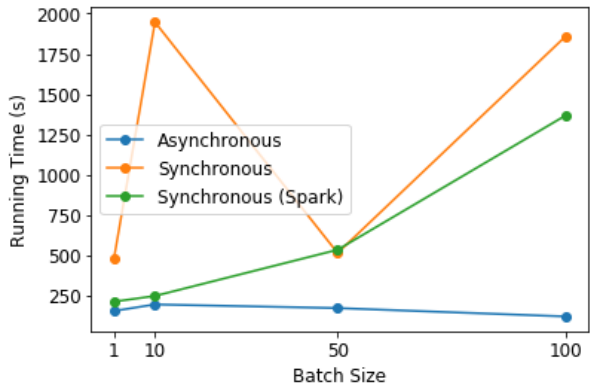

Fig. 5. Running time vs batch sizes

Figure 6 shows the test accuracy for different batch sizes and implementations. The synchronous implementation in Spark is stable and achieves always a similar test accuracy once the batch size is significant (i.e. *10*). Contrarily, the accuracy for the other synchronous implementations fluctuates with a standard deviation of ± 0.03. The test accuracy for the asynchronous case decreases when the batch size increases. Nonetheless, we cannot conclude that either synchronous or asynchronous yields better results because it depends on multiple hyper-parameters that make it non deterministic.

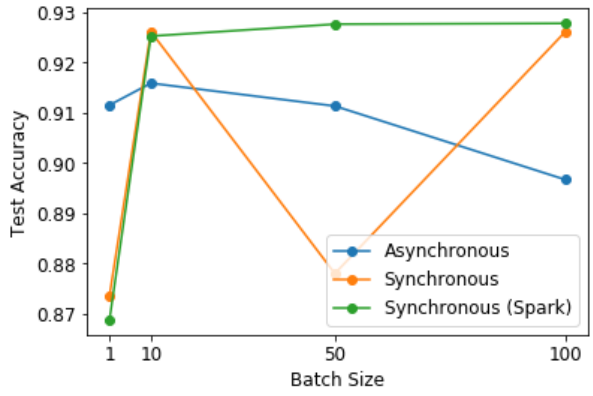

Fig. 6. Test accuracy vs batch sizes

Finally, we analyze the convergence of the validation loss. Figure 7 shows the validation loss over time in all three implementations for different batch sizes.

The asynchronous implementation is clearly faster converging. However, the reached validation loss is higher than in the synchronous cases. On the synchronous implementations, the one in Spark is faster converging when the batch size increases but the validation loss fluctuates more.

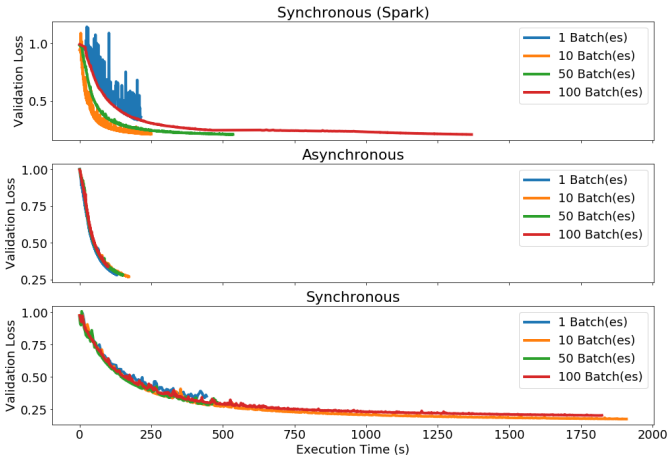

Fig. 7. Convergence of validation loss over time for different batch sizes

V. Discussion

The results shown in Section IV are averaging at least three different trials in order to find relevant experimental results. However, in order to draw significant statistical conclusions it would be necessary to run more experiments. We also ran experiments at night to avoid the overload of the cluster.

We do not have results with a higher number of workers due to time constraints and an overloaded cluster. For instance, the synchronous implementations take over six hours to execute with *20* workers. We also do not have results for larger batch sizes because the learning rate decreased to almost zero and therefore not update the weights.

The problems detailed above show a discrepancy between our experimental results and the ones seen in the report for the previous implementations. They probably adapt the learning rate for larger batch sizes (i.e. 500 and 1000) in a way that is not detailed in the report.

We believe that using Numpy [9] instead of dictionaries would speed up the calculations considerably. Furthermore, exploiting GPUs would be another approach to faster operations.

VI. Conclusion

The results shown in Section IV emphasize the importance of an efficient implementation of SGD. The distributed asynchronous SGD is faster than the synchronous implementations when the number of workers or the batch size increases. However, it is important to remark that the asynchronous solution is suitable for sparse problems, and therefore it is not a silver bullet.

Our Spark implementation of synchronous SGD scales better than the compared synchronous implementation, and it achieves a high accuracy more systematically. This is likely due to a more efficient partition and communication in Spark.

REFERENCES

[1] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[2] F. R. Benjamin; Re Christopher; Wright Stephen J. Niu, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. eprint arxiv:1106.5730," 2011.

[3] M. A. Liamarcia Bifano, Roman Bachmann, "Distributed asynchronous sgd," https://github.com/liabifano/hogwild-python, 2018.

[4] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, "Rcv1: A new benchmark collection for text categorization research," *Journal of machine learning research*, vol. 5, no. Apr, pp. 361–397, 2004.

[5] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[6] L. Prechelt, "Early stopping-but when?" in *Neural Networks: Tricks of the trade*. Springer, 1998, pp. 55–69.

[7] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," 2016.

[8] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[9] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, p. 22, 2011.