Parallelizing Stochastic Gradient Descent in Spark
Milestone I: Synchronous Version

Adrian Guerra Ayyoub El Amrani Amaury Combes Julien Belguise
CS-449 Systems for Data Science 2019 EPF Lausanne

***Abstract*—The goal of this project is to design, implement and experiment a synchronous version in Spark of a distributed stochastic gradient descent (SGD) used in Support Vector Machines (SVMs) by comparing it with previous synchronous and asynchronous implementations in Python.**

The main reference for this project is the Hogwild! paper[1]. The Hogwild! paper is an important paper in the Machine Learning and Parallel Computing community that shows that SGD can be implemented without any locking when the associated optimization problem is sparse. hogwild-python is a synchronous and asynchronous implementation in Python of the Hogwild! algorithm by EPFL students. This project is part of the "CS-449 Systems for Data Science" course taught at EPFL in the Spring semester of 2019.

I. Introduction

Stochastic gradient descent (SGD) is an iterative method widely used in Machine Learning that randomly samples data to compute a gradient and optimize parameters of a model by performing parameter updates in each iteration. The algorithm only requires a single training data point to be in memory, thereby significantly decreasing the cost of computation compared to gradient descent [2]. Its inherent sequential nature, however, makes it hard to parallelize. The Hogwild! paper showed that when data access is sparse, meaning that individual SGD steps only modify a small part of the decision variable, memory overwrites are rare and introduce barely any error into the computation when they do occur. This key insight, in turn, allowed Hogwild! to show that all methods that propose memory locking are significantly slower than their respective lock-free counterparts on a variety of machine learning applications.

Students of the EPFL CS-449 Systems for Data Science 2018 course successfully implemented synchronous and asynchronous versions of a distributed SGD in Python based on the Hogwild! algorithm. As a follow-up, our task was to design, implement and experiment a synchronous version in Spark by comparing it with the previous implementations in Python.

Spark is a general-purpose distributed data processing framework available in Scala, Java, Python and R. Its architectural foundation resides in the resilient distributed dataset (RDD), a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way. [3]

Scala has a major advantage over Python in that it is the language that Apache Spark is written in. Moreover, Scala is statically typed and compiles in a known way to the JVM thus making it faster in most scenarios. [4] To make use of said advantages, our project is entirely implemented in Scala.

Experiments are performed by comparing the three parallelized versions of stochastic gradient descent (SGD) used for learning support vector machines (SVM) based on the Hogwild! algorithm: the synchronous version in Python, the asynchronous version in Python and our synchronous version in Spark.

This paper first explains the design of our algorithm, the methodology for the experiments and finally discusses the results of the experiments performed.

II. Design

The overall design of our project is very similar to the synchronous version of hogwild-python: synchronous workers sub-sample the data to compute a gradient and a synchronous coordinator receives the gradients and performs the weights updates. We first tried to partition the data across all the workers, but found better results when the data was present in all of them.

A. Synchronous Worker

Each worker randomly sub-samples the data, computes the gradient on the sub-sampled data and communicates it to the coordinator.

B. Synchronous Coordinator

The coordinator waits for the computed gradients of all workers Once he has received them, he performs the weight updates, calculates the validation loss and sends the updates of the current epoch to the workers.

III. Implementation

A. Reuters RCV1 Dataset

The dataset used to evaluate our algorithm is the *Reuters RCV1 Dataset*[5] set on the binary text classification task *CCAT*. The same dataset is used in Hogwild! and in hogwild-python to test the sparse SVM implementations. There are 804,414 examples split into 23,149 training and 781,265 test examples, and there are 47,236 features.

B. Support Vector Machines

As is explained in last year's project report, the machine learning models optimized using SGD are Support Vector Machines. More precisely, SVMs are used in a supervised binary classification task of detecting whether or not a given Reuters news article belongs to the *CCAT* category. The SVM tries to find the maximum-margin hyper-plane dividing the group of points labelled -1 and 1. The associated soft-margin loss is the Hinge loss with an additional regularization parameter λ.

$$L(w) = \frac{1}{N}\sum_{i=1}^{N} max(0,\ 1 - y_i(wx_i + b)) + \lambda||w||^2$$

where:

- y_i : the target
- w : the weight vector
- x_i : the row i of our input
- b : bias
- λ : regularizer

C. Algorithm

The algorithm implemented is basically as follows:

```
1:  procedure SGD EXPERIMENT
2:      load data                                   ▷ Take time t_0
3:      broadcast data
4:      for all epoch i do                          ▷ Take time t_1
5:          broadcast weights
6:          for all partitions p do
7:              take random batch of data
8:              compute gradients
9:              compute loss
10:         end for
11:         merge gradients computed at each partition
12:         update weights
13:         average losses
14:     end for                                     ▷ Take time t_2
15: end procedure
```

IV. Methodology

The central question is which of the three implementations produces the best-quality results in the shortest amount of time. All experiments are run on the faculty's cluster, set up with the same amount of memory for each of the containers to to make the trials comparable.

Each worker is allocated a maximum of 4 GB and 2 cores of CPU, while the driver is allocated 20GB and 10 cores. The conditions of our experiments are as fair as possible and compatible with both synchronous and asynchronous versions of SGD.

A. Metrics

The time taken for our algorithm to reach a validation loss of 0.3 is our main metric. Time is measured at times t_0, t_1 and t_2. An epoch's time is the difference $t_2 - t_1$ and the data load time is the difference $t_1 - t_0$. The epoch times are accumulated in a vector which is then retrieved at the end of the procedure. The summed epoch times are the ones compared across different implementations. Each experiment is performed several times and time is averaged due to the stochastic nature of SGD.

B. Parameters

Times are measured for different number of workers $|W| \in \{3, 6, 12, 24\}$ in each of the three implementations.

The other parameters are kept identical to hogwild-python to make comparable experiments:

- Regularizer is set to 10^{-5}
- Learning rate is set to $\frac{0.03*100}{b*|W|}$ where b is the batch size and $|W|$ is the number of workers.
- 90% of the data is used for training and 10% for testing.

V. Results

The results' tables show the training duration for all three codebases. Additionally, we present the number of epochs for both synchronous implementations. We see that the python asynchronous implementation consistently beats the other implementations while the python synchronous implementation performs better compared to our spark implementation. However, we notice that our spark implementation seems to be less penalized by the number of workers.

Table I
RESULTS FOR SYNCHRONOUS HOGWILD-PYTHON

# of workers	# of epochs	Training duration (s)
3	231	72.2
6	236	168.7
12	230	334.4
24	220	685.1

Table II
RESULTS FOR ASYNCHRONOUS HOGWILD-PYTHON

# of workers	Training duration (s)
3	23.8
6	42.2
12	87.8
24	177

Table III
RESULTS FOR SYNCHRONOUS HOGWILD-SPARK

# of workers	# of epochs	Training duration (s)
3	234	474.4
6	230	900.8
12	243	1609.1
24	NA	NA

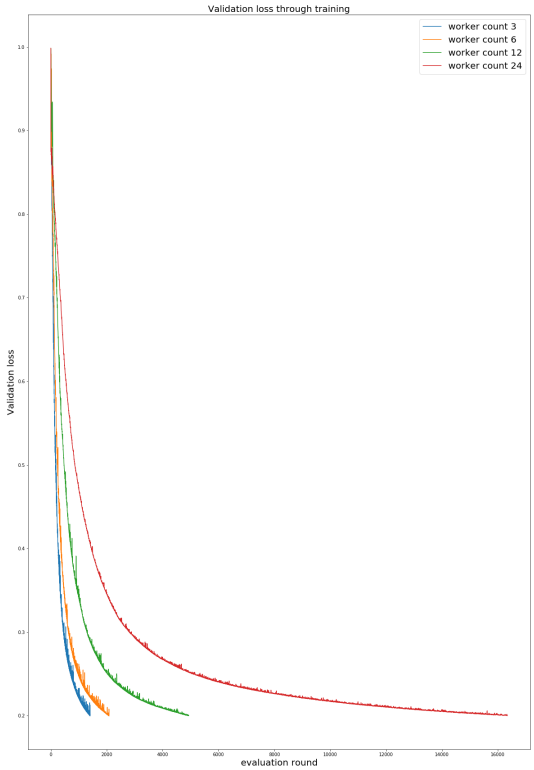

Figure 1. Validation losses in synchronous hogwild-spark

VI. CONCLUSION

Regarding the results we obtained, we conjecture that the Spark version implemented on Scala is slower than the Hogwild version. Indeed, the hogwild implementation is approximately 5/6 times faster. Besides the design we implemented in this report, we also implemented a design where the data was not broadcasted in all the nodes. However, we faced "data locality" issues that resulted in having only two workers running jobs instead of the allocated number (3, 6 or 12). Our current design proved to be most effective.

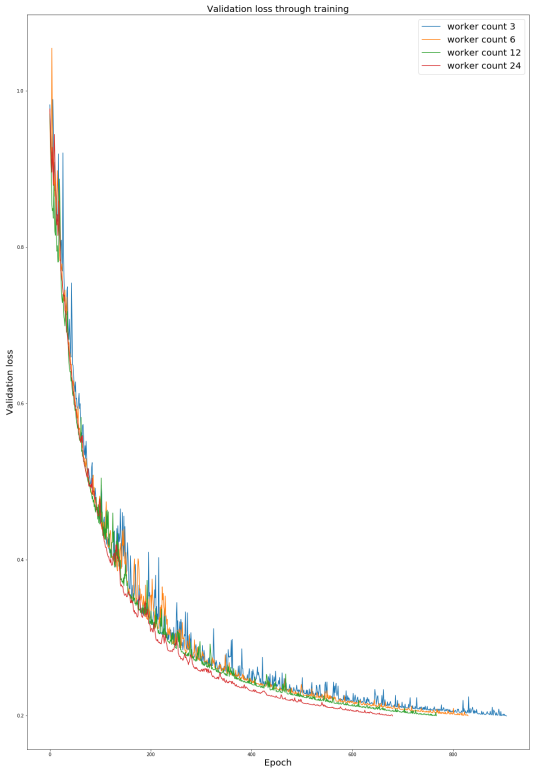

Figure 2. Validation losses in synchronous hogwild-python

REFERENCES

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

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

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[4] P. Ghandi, "Kdnuggets," May 2018. [Online]. Available: https://www.kdnuggets.com/2018/05/apache-spark-python-scala.html

[5] 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.

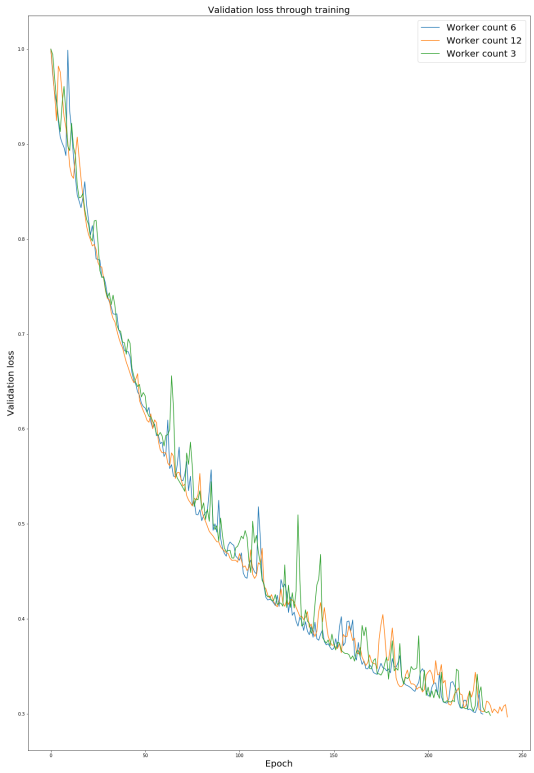

Figure 3. Validation losses in asynchronous hogwild-python