Parallelizing Stochastic Gradient Descent in Scala

Milestone II: Asynchronous 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 an asynchronous version in Scala of a distributed stochastic gradient descent (SGD) used in Support Vector Machines (SVMs). Our system exploits the multicore processing capability of modern servers by running the algorithm in multiple threads in parallel and uses unprotected shared memory to concurrently modify a common representation of the weights vector. 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. Additionally, our implementation is compared to the asynchronous version of last year's 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.

In Milestone I of our project, we successfully designed, implemented and experimented a synchronous version of Hogwild! in Spark. As a follow-up, we design implement, and experiment an asynchronous version in Scala by comparing it with a modified version that uses locks and last year's asynchronous version in Python.

As Scala is a statically typed language and thus faster than Python in most scenarios [3], our project is entirely implemented in Scala.

This paper first reminds us of the Support Vector Machines (SVM) formulation, then explains our implementation and explores the methodology for the experiments to finally discuss the results of the experiments performed.

II. Support Vector Machines

As explained in our previous project report, the machine learning models optimized using SGD are SVMs. The SVM's optimization 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

III. Implementation

As mentioned, the optimization procedure is distributed among many workers. Updates to the weight vector are performed concurrently through a lock-free shared collection.

A. Shared memory

All workers share a reference to a concurrent lock-free collection. We use the mutable **Map** implementation in Scala as it provides such features.
Writes are performed using the **update** method which allow to overwrite a single element in the map atomically. Reads are performed using the **apply** method.

B. Worker

Our program is composed of two kind of workers. Each worker runs in a distinct thread and are created by the main section of the program.

1) Optimizer: Optimizers are workers that perform gradient steps. They loop indefinitely until they receive a stop signal from the monitor. We implement two kind of optimizers. A lock-free version (See Algorithm III-B1) and another version that locks the weights when computing the gradient and updating the parameters to prevent conflicting write updates (See Algorithm III-B1).

```
procedure OPTIMIZER-ROUTINE-LOCK-FREE()
    while !stop do
        take a random datapoint
        read concerned weights
        compute gradients
        update weights with non-zero gradient
    end while
end procedure
```

```
procedure OPTIMIZER-ROUTINE-WITH-LOCK
    while !stop do
        take a random datapoint
        lock shared weights
            read concerned weights
            compute gradients
            update weights with non-zero gradient
        release lock
    end while
end procedure
```

2) Monitor: The monitor runs in its own thread. It periodically reads the state of the weights and evaluates the loss on the validation set. If the computed value is below a certain threshold, it sends a stop signal to all optimizers and stops itself.

```
procedure MONITOR-ROUTINE(threshold, optimizers)
    validation_loss = +∞
    while validation_loss ≥ threshold do
        evaluate current model on validation set
        set current validation_loss
        sleep for 1 second
    end while
    for all optimizer t do
        interrupt t
    end for
end procedure
```

C. Overall Algorithm

The algorithm implemented is basically as shown in Algorithm III-C.

IV. Experiment

A. Dataset

The dataset used to evaluate our algorithm is again the *Reuters RCV1 Dataset*[4] 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.

```
procedure EXPERIMENT(worker_count, validation_threshold)
    load data
    split data into train and test set
    register time t_0
    for i ← 1 to worker_count do
        run new thread with subroutine OPTIMIZER-ROUTINE()
    end for
    run new thread M with subroutine MONITOR-ROUTINE(validation_threshold, optimizers)
    wait on M
    register time t_1
    output training statistics
end procedure
```

B. Metrics

The only metric we track for our experiments is the time required to reach a validation loss of 0.2. In Algorithm III-C it corresponds to the absolute difference between t_0 and t_1.

C. Setup & Parameters

The experiments were done on the same computer with 4 I7-3520M CPUs at 2.90 GHz and 8 GB RAM.

Timings are measured for different number of workers $|W| \in \{4, 6, 8\}$ in each of the three implementations.

The parameters are adjusted as follows in all implementations:

- Regularizer is set to 10^{-5}
- Learning rate is set to 0.1
- 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. We see that the Scala implementations consistently beat the Python implementation, while the Scala lock-free implementation performs better than its locked counterpart. Aditionally, plots are provided at the end (see Figure 1 and 2).

Table I
RESULTS FOR ASYNCHRONOUS HOGWILD-PYTHON

# of workers	Training duration (s)
4	79
6	84
8	196

Table II
RESULTS FOR ASYNCHRONOUS LOCK-FREE HOGWILD-SCALA

# of workers	Training duration (s)
4	44
6	31
8	36

Table III
RESULTS FOR ASYNCHRONOUS HOGWILD-SPARK WITH LOCK

# of workers	Training duration (s)
4	55
6	81
8	56

VI. CONCLUSION

Regarding the results we obtain, we conjecture that Hogwild! implemented in Scala runs faster than in Python and the lock-free version runs faster than the locked version.

This was expected, as locks slow the computation down and the original Hogwild! paper proved that SGD could be implemented in a lock-free manner.

Concerning the comparison of Python and Scala, Python uses the multiprocessing library, which creates processes and not threads, unlike our implementation in Scala. Creation of a process is time-consuming and resource intensive, while the creation of a thread is economical in both time and resource [5]. Moreover, Scala threads have access to shared memory, while Python processes must communicate between themselves to perform the weight updates, thereby adding a communication bottleneck.

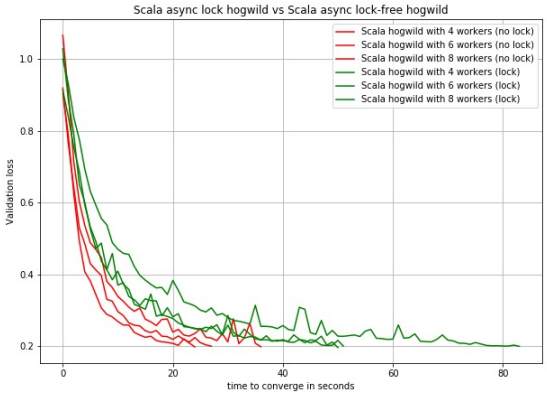

Figure 1. Plot comparing the lock-free and locked versions in Scala

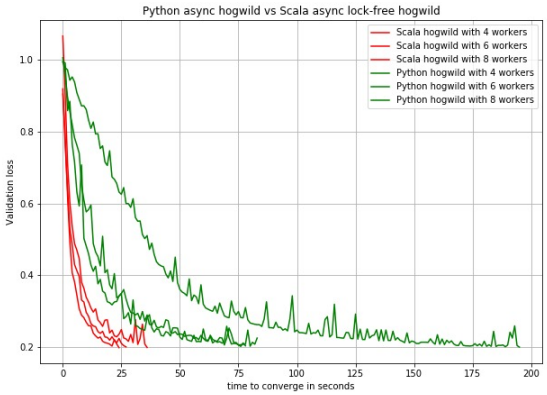

Figure 2. Plot comparing the Python and Scala implementations

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

[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] "Difference between multiprocessing and multithreading (with comparison chart)," Jul 2017. [Online]. Available: https://techdifferences.com/difference-between-multiprocessing-and-multithreading.html