DS222-Assignment 2

Aakash Khochare [1]

Abstract

As a part of this project, Logistic Regression was trained in Tensorflow with two broad settings, local and distributed. The classifiers were trained using Stochastic Gradient Descent, which was also implemented in Tensorflow. The distributed learning, in itself, has three different implementation, namely, BSP, Asynchronous and Stale Synchronous.

1. Introduction

Logistic Regression is a discriminative model that learns one weight vector per class and each vector has dimension equal to the number of features. In case of document classification, the number of features is the size of the vocabulary. The DBPedia dataset has 50 classes. One major assumption that should be stated here is that one document in DBPedia can belong to multiple classes, however, in our implementation, I fix each document to belong to exactly one of the classes. I choose Tensorflow as the framework of choice over Hadoop and Spark because it has built in parameter server capabilities which Spark doesn't and it does't write to disk between every iteration like Hadoop. Spark would also not fit well for the completely asynchronous implementations.

2. Implementation and Analysis

2.1. Local

LR was implemented in the local setting with three different types of learning rates. For the constant learning rate setting, the rate was fixed to *0.05*. For the increasing learning rate, the function $0.05 + 0.01/epoch$ was used. This function was chosen with the rationale that it would move faster towards the minima, and as it approached the minima, the rate would almost seem like a constant since $0.01/epoch$ would be a very small value for high value of epoch. Similary, for the decreasing learning rate, the function $0.05 - 0.01/epoch$ was used, with the rationale that as it approached the minima, the learning rate would be small and almost constant and thus we would approach the minima without overshooting it. The training time on the full dataset on my local machine was as follows,

constant rate = 4423.77 seconds increasing rate = 4551.61 seconds decreasing rate = 4726.16 seconds

The test accuracy was 69 % for the increasing rate and it was similar for the two other models as well. And the time required for testing was 46 seconds. The training accuracies were slightly better at around 74%

However, from the plots it seems like they perform more or less the same.

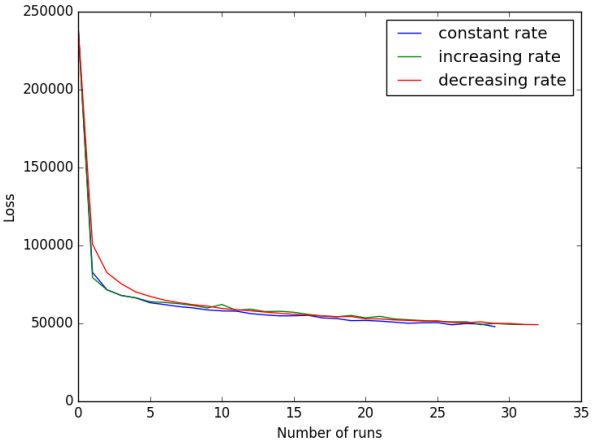

Figure 1. Gantt Chart of BSP on 4 nodes

2.2. Distributed

All the distributed training techniques follow the between-graph replication. This means that they are naturally asynchronous, so an effort had to be made to synchronize them.

2.2.1. BSP

The Barrier Synchronization has been implemented by using vector that contains one entry per worker. At the start of every iteration, each worker pulls this vector and decides if all the other workers are in the same iteration. If yes, it goes ahead with SGD, else it busy waits on this vector. This busy waiting involves reading the vector from the paramter server and hence is costly. The overall tuple processing time goes to around 4 seconds when using any of the parallel techniques and hence I do not report any convergence timings, since processing the entire dataset would be very

time consuming. The Gantt charts produced from the timings for each node, however, shows the functional correctness of the code. Ideally implementing mini batch gradient descent would benefit due to the increase in computation per iteration.

In Figure 2 we can see that after the first iteration, where all the nodes were started asynchronously, they all synchronize. The plot shows only the first 8 iterations on each node, but the point to notice here is that the iterations all finish at the same time. Note: Each color represents a different iteration.

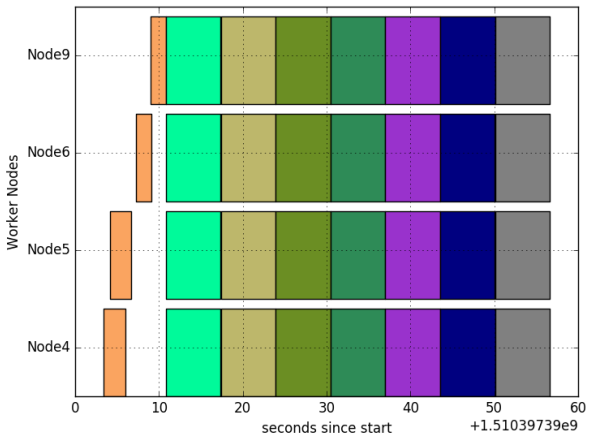

Figure 2. Gantt Chart of BSP on 4 nodes

2.2.2. Stale Synchronous

This implementation too has a vector maintained in the parameter server. The only difference here is that the check made here is that each node verifies if it's own iteration is off by the staleness factor from the minimum value in the vector. If yes, it blocks else it continues with the iteration.

In Figure 3, there are two points worth noticing, Node4 stops processing after 4 iterations. (4 was the staleness factor) and lets Node 5 complete. Similarly Node 6 also stops after it's 4th iteration to let Node 5 complete. Also node that the iteration don't end at the same time. In the logs it can be seen that each node continues further with it's iterations, here we just don't plot them.

2.2.3. Asynchronous

Asynchronous was the easiest of the lot since between graph replication is inherently asynchronous. No barriers were added. In Figure 4, we can see that all the workers complete their iterations at different times and there is no synchronization between them.

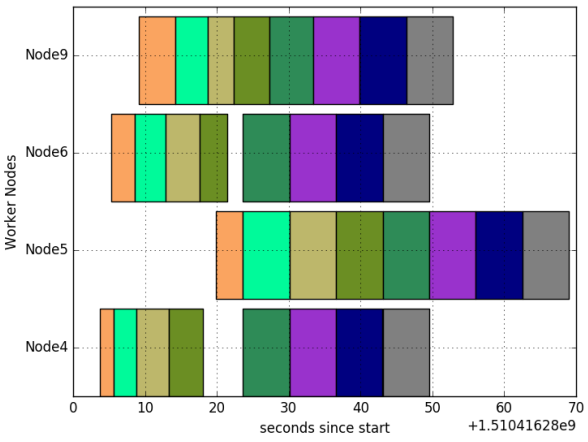

Figure 3. Gantt Chart of Stale Synchronous on 4 nodes

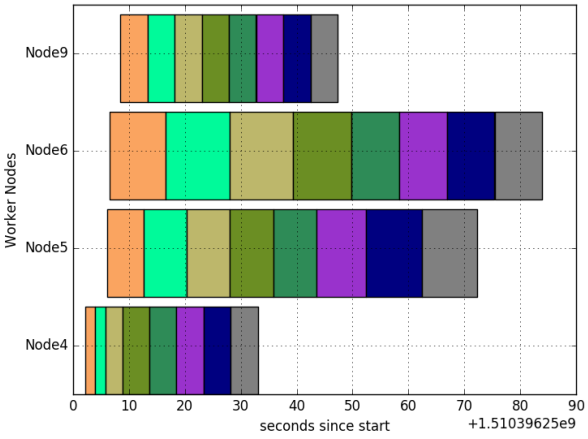

Figure 4. Gantt Chart of Asynchronous on 4 nodes