DS222: MLLD 2018
Assignment 2

Rishi Hazra
Systems Engineering
14542
rishixtreme@gmail.com

Abstract

In this assignment, you will implement and evaluate a This assignment is based on implementation of L2-regularized logistic regression (in the standalone and distributed setting using Parameter Server) on DBPedia articles. We then, compare the time taken to complete the prediction process.

1 Introduction

1.1 Dataset

We use DBPedia [1] for out prediction task. This dataset is available in three types, namely *very small, small & full*. The *very small* dataset has 10,497 documents in train file, 2997 in validation file and 1497 in test file. It has 49 labels. The *full* dataset has 2,14,997 train documents, 61,497 validation documents and 29,997 test documents. It has 50 labels.

1.2 Problem Statement

In this assignment, we will train a Logistic Regression classifier in both local and distributed mode.

1. In this part of assignment, we need to implement a L2-regularized logistic regression and evaluate it in a local setting.
    - Local Logistic Regression: We are required to use SGD for learning and report train test accuracy, training time and test time.I use python for my implementation.
    - We need to re-implement the same setup as above but with the following three learning rate strategies: constant, increasing, and decreasing.

2. In this part of the assignment, we need to parallelize the classifier developed above. Based on our analysis above, we freeze the learning rate and hyperparameters for the experiments.
    - Train the classifier in the BSP SGD setting and report train test accuracy, training time and test time. Also, plot training loss against epochs.
    - Train the classifier in the Asynchronous SGD setting and report train test accuracy, training time and test time. Also, plot training loss against epochs.
    - Train the classifier in Bounded Asynchronous (SSP) SGD and report train test accuracy, training time and test time. Also, plot training loss against epochs.

2 Execution in distributed setup

2.1 Parameter servers and worker nodes

The reason we use parameter servers is that, using parameter server can give us better network utilization, and lets us scale our models to more machines. Let's say for instance,we have 250M parameters and it takes 1 second to compute gradient on each worker and there are 10 workers. This means that each worker has to send/receive 1 GB of data to 9 other workers every second, which needs *72 Gbps* full duplex network capacity on each worker, which is not practical. Please see Figure 1.

More realistically we could have *10 Gbps* network capacity per worker. We can prevent network bottlenecks by using parameter server split over 8 machines. Each worker machine communicates with each parameter machine for $\frac{1}{8}th$ of parameters. Please see Figure 2.

2.2 Asynchronous Execution

In asynchronous training there is no synchronization of weights among the workers. The weights

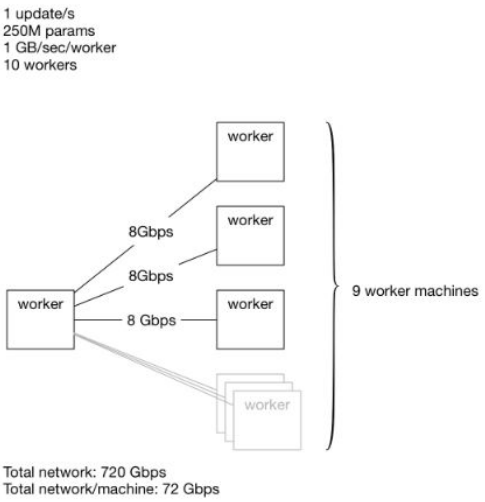

Figure 1: workers with workers

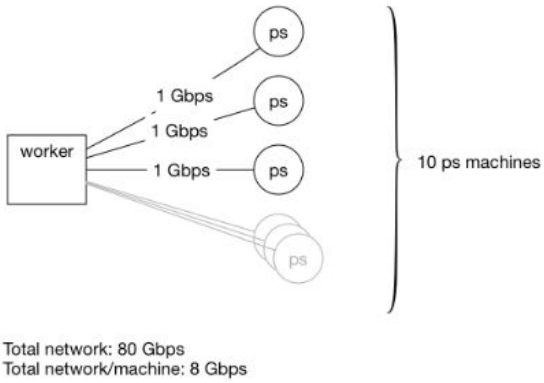

Figure 2: workers with parameter servers

are stored on the parameter server. Each worker loads and changes the shared weights independently from each other. This way if one worker finished an iteration faster than the other workers, it proceeds with the next iteration without waiting. The workers only interact with the shared parameter server and don't interact with each other.

Overall it can (depending on the task) speedup the computation significantly. Asynchronous execution lacks theoretical guarantee as distributed environment can have arbitrary delays.

2.3 Synchronous Execution

In contrast to asynchronous execution, synchronous execution will ensure that all of the workers read the same, up-to-date values for each model parameter; and that all of the updates for a synchronous step are aggregated before they are applied to the underlying variables. To do this, the workers are synchronized by a barrier, which they enter after sending their gradient update, and leave after the aggregated update has been applied to all variables.

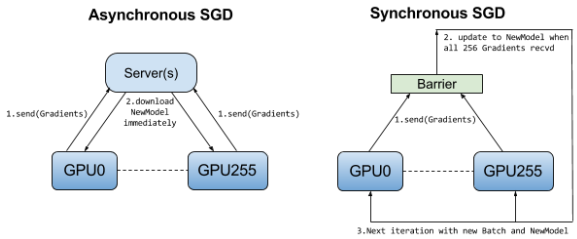

Figure 3: Execution types

2.4 Stale Synchronous Execution

The SSP consistency model guarantees that if a worker reads from the parameter server at iteration c, it is guaranteed to receive all updates from all workers computed at and before iteration $cs1$, where s is the staleness threshold. If this is impossible because some straggling worker is more than s iterations behind, the reader will stop until the straggler catches up and sends its updates. For stochastic gradient descent algorithms, SSP has very attractive theoretical properties

3 Approach

- **Data Preprocessing** : The first task is to preprocess the training data to split the documents into label and text. The text is further processed to remove html links, special symbols, digits, extra spaces and stop words. The words are converted to lower case which gives us a lower size of vocabulary to deal with. In order to reduce the vocabulary size, we consider two methods, *tfidf vectorizer* for rare words and high frequency words (with frequency higher than 99). It was observed from the test results that high frequency words act as a better classification feature as compared to tfidf or a SVD based tfidf.

  We create a document-term matrix with 1 replacing the word which has a count higher than 99 in that particular document. Similarly, a document-label matrix is assigned with a uniform probability over all labels of that document. We follow the same approach for the test documents.

- **Training** : For the local logistic regression task, we load the data matrices and divide

the training data into batches with size 4000. We run the training for 54 epochs a learning rate of 0.002 (determined empirically over a range of learning rates). We train it with three different settings, one with constant learning rate, second with increasing learning rate and third with decreasing learning rate.

For the distributed setting, we use one parameter server with two worker nodes for each of the settings. Regularized stochastic gradient descent (SGD) is used in the training process.

- **Testing :** : The model is used to predict a single class for the data. If that class matches any of the given labels of the test data, we say that the prediction is correct.

4 Experimental Results

Train data shape (*full*) : (214994, 8077)
Number of training classes (*full*) : (214994, 50)
Test data shape (*full*) : (29994, 8077)
Number of epochs : 54
Batch Size : 4000
learning rate : 0.02

4.1 Run Time

Table 1: Run Time combined (Train + Test) for 54 epochs

Method	Time
Local Log Reg	5 min
BSP (1 worker)	12 min 41 sec
BSP (2 workers)	15 min 43 sec
SSP (1 ps 2 workers)	9 min 34 sec
SSP (2 ps 2 workers)	9 min 41 sec
Asynchronous (2 workers)	9 min 02 sec
Asynchronous (3 workers)	9 min 18 sec

As we can see from Table 1, Asynchronous setting has the lowest execution time. Here we have taken the average of time taken by all the workers for training and testing. Also, we can observe from Table 3 that the accuracy of all the three settings are almost comparable. Prima Facie, Asynchronous setting looks ideal, however it lacks theoretical guarantee, as pointed earlier. BSP takes a lot of time to run as most of the time is utilized in the synchronizing process.

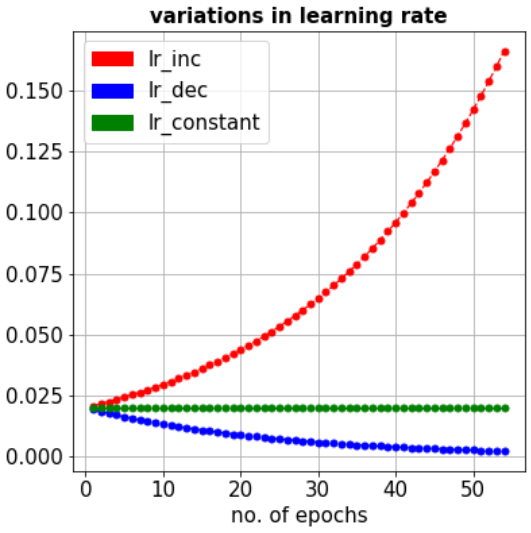

Figure 4: Variations of learning rate with epochs

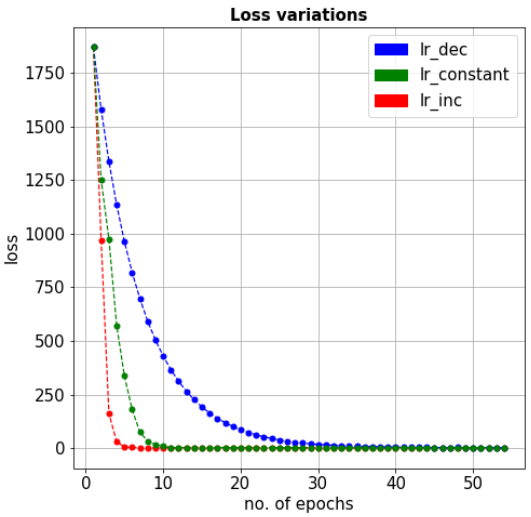

Figure 5: Variations of loss with epochs

4.2 Accuracy

Table 2 and Table 3 gives us a rough idea about the accuracy in different frameworks. We can observe that the worst accuracy is observed when we run SGD with increasing learning rate. If the learning rate is low, then training is more reliable, but optimization will take a lot of time because steps towards the minimum of the loss function are tiny. As can be observed from Figure 6, If the learning rate is high, then training may not converge or even diverge. Weight changes can be so big that the optimizer overshoots the minimum and makes the loss worse.

from 2018-10-30 16-36-53.png

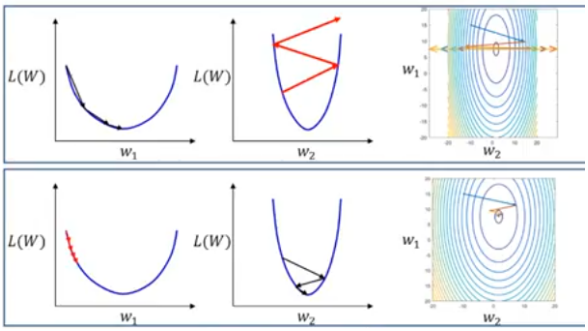

Figure 6: Effect of Learning Rate on convergence

Table 2: Accuracy with local settings

Method	Acc train(%)	Acc test(%)
Log Reg	72.32	71.50
Log Reg (inc lr)	53.68	63.87
Log Reg (dec lr)	72.09	73.89

Table 3: Accuracy with distributed settings (1 ps 2 workers)

Method	Acc train(%)	Acc test(%)
Bulk Synchronous	75.32	63.222
Asynchronous	75.14	64.86
Stale Synchronous	75.06	64.47

4.3 Other observations

- With a decreasing learning rate, the convergence is slow, as can be seen in Figure 5.
- With the increase in number of workers, the loss converges faster with the number of epochs, as is seen in Figure 7. This may be attributed to the fact that each worker updates the parameters independently of one another and hence ends up getting larger updates as compared to updates from one single worker. Irrespective of that, there is no significant change in accuracy as compared to other settings.
- The loss in case of SSP converges faster with the decrease in staleness. This can be attributed to the fact that with decrease in staleness, there is a reduction in error bound, [$R[X] \leq 4FL\sqrt{\tau T}$ where τ is the staleness] though it comes at the cost of increase in processing time [Figure 8]

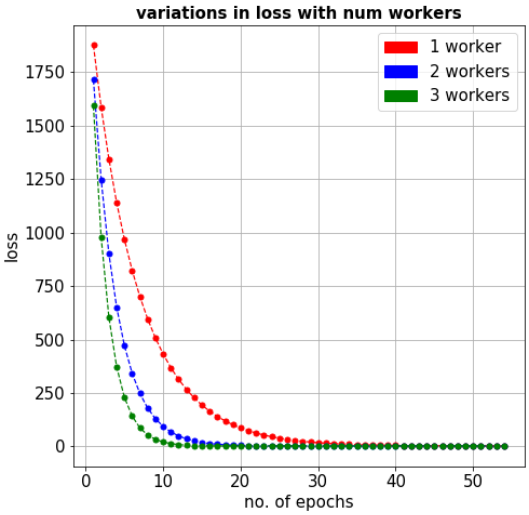

Figure 7: Asynchronous setting with different workers

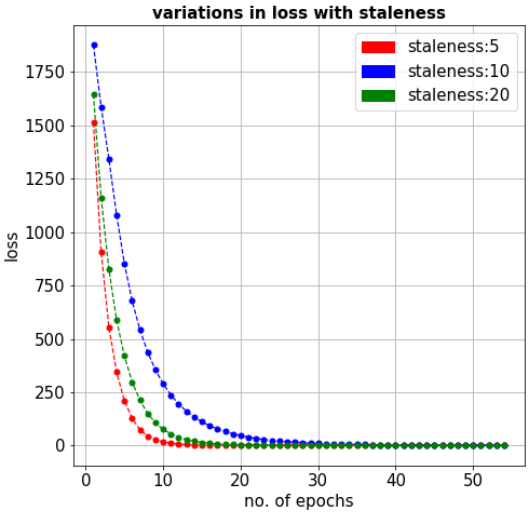

Figure 8: Variations of loss with staleness

5 Acknowledgements

I would like to thank my batchmates Souvik and GM Anand, for assisting me in understanding the specifics of distributed computing using tensorflow. I would also like to thank the whole DS-222 group for actively promoting detailed

discussions in the group.

Here is the link to github repository.

References

[1] https://www.tensorflow.org/apidocs/python/tf/train/SyncReplicasOptimizer

[2] https://github.com/ischlag/distributed-tensorflow-example

[3] https://www.tensorflow.org/deploy/distributed

[4] https://www.oreilly.com/ideas/distributed-tensorflow