Assignment 3 Part A
Group -11

Abhanshu Gupta (agupta89@wisc.edu)
Parikshit Sharma (psharma43@wisc.edu)
Vishnu Lokhande (lokhande@wisc.edu)

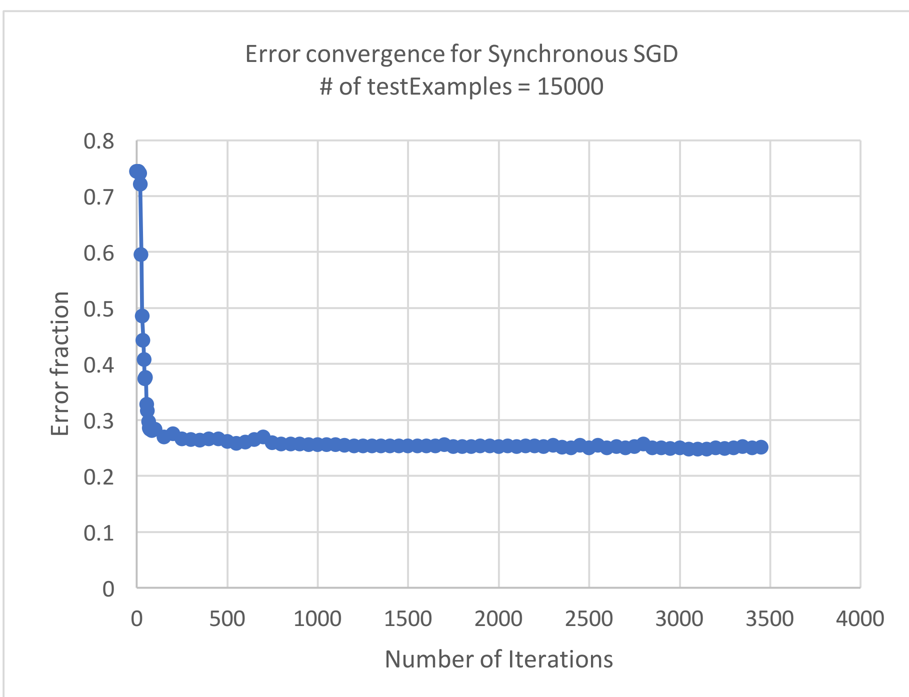
Machine Learning in Tensor Flow versus Spark

Apache Spark is a general-use data processing engine with libraries that make it easy to do machine learning, batch processing, streaming, and graph computation. Tensorflow is a library used primarily for more scalable and easier machine learning.

- Tensor Flow code can be executed on a variety of devices from clusters to mobile devices, this makes it very extensible.
- TF provides more flexibility in defining on which device to place an operation. It can make use of device heterogeneity e.g. GPUs/TPUs attached to a machine, whereas Spark assumes homogenous cluster.
- Spark processes batches of data in the form of RDDs (useful for batch processing algos). TensorFLow can, in addition, efficiently handle one example flowing through the graph in one iteration.

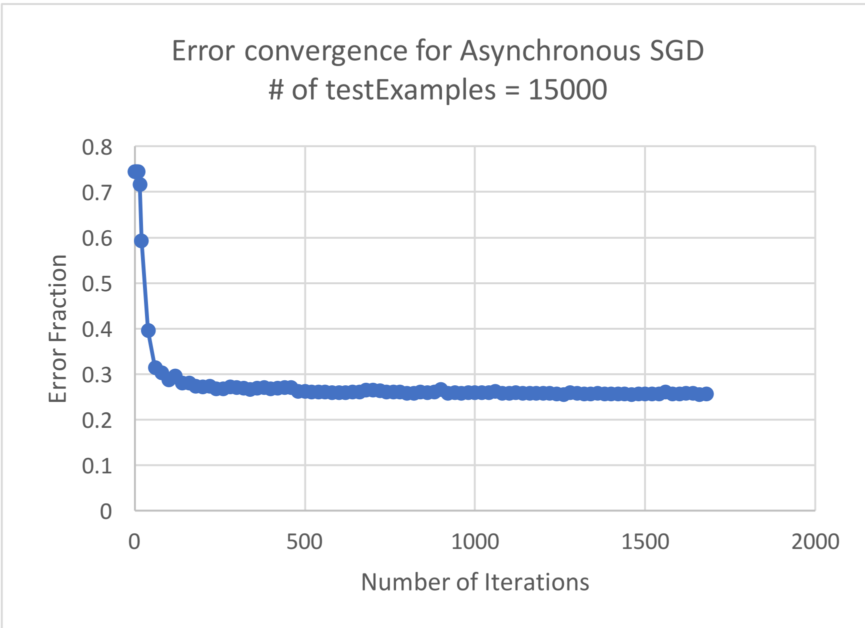
Variation in test error for Synchronous SGD

The following result is plotted for a test set size of 15,000. The test error decreases to 28% in the first 100 iterations and then the decrease is very slow with some minor oscillations. It is around 25% at the 3000^{th} iteration.



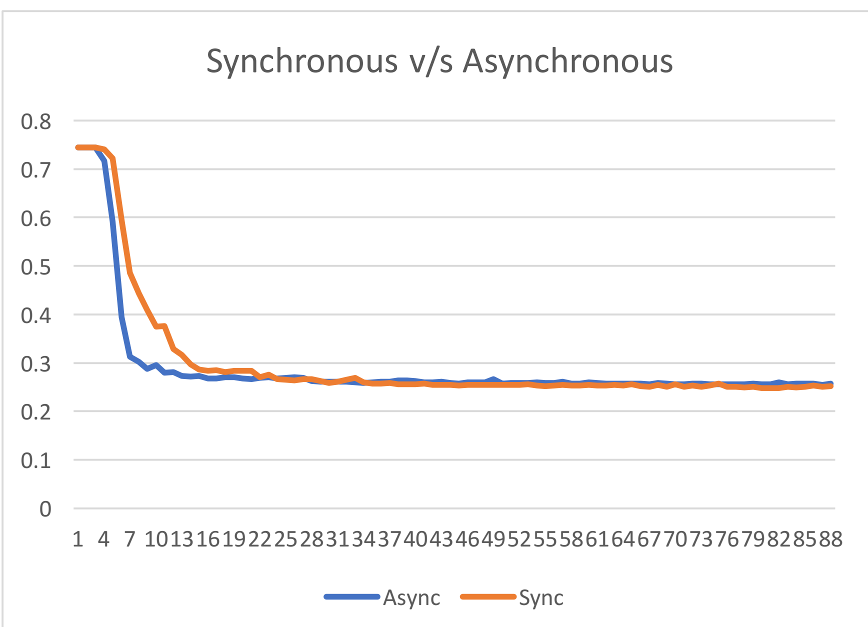
Variation in test error for Asynchronous SGD

The convergence graph is like synchronous SGD, with minor variations shown in the next section.



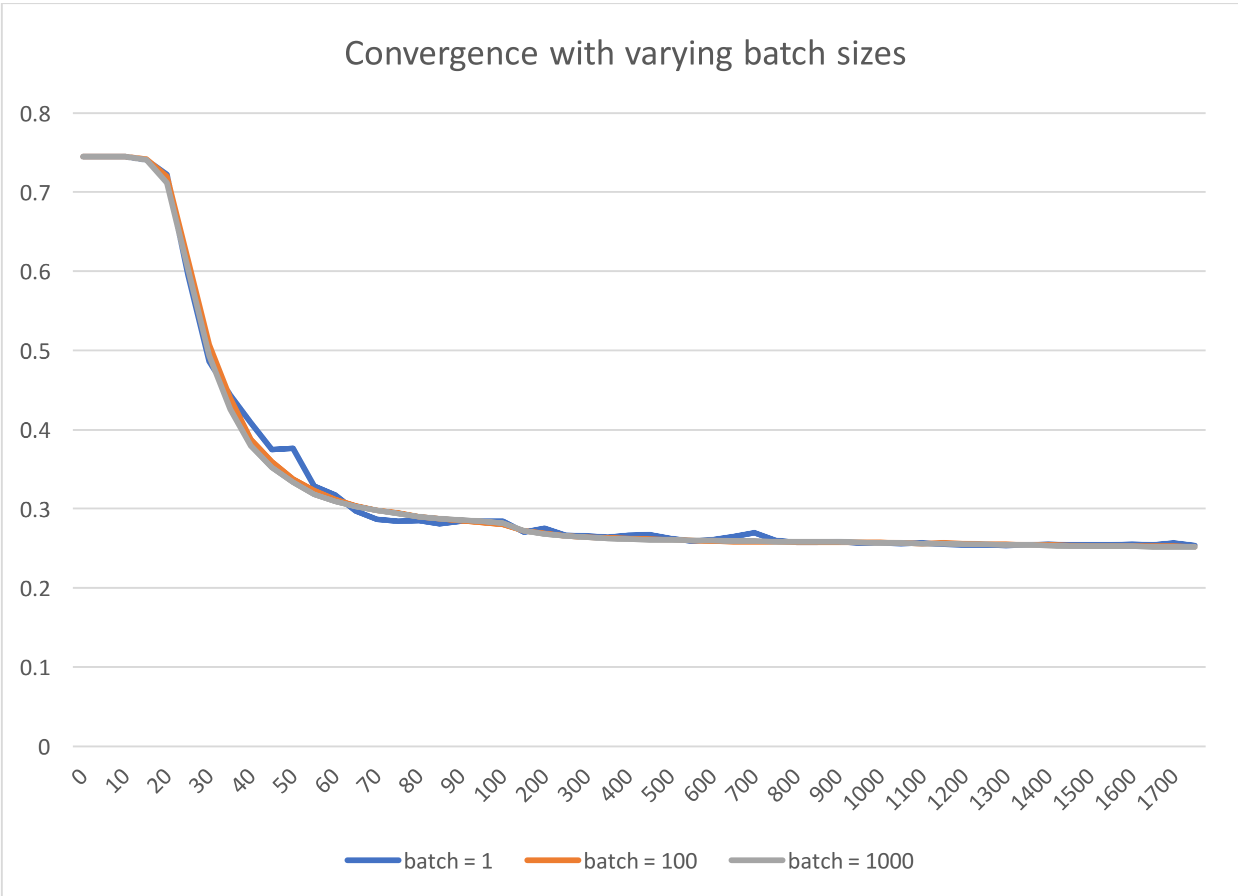
Difference between Asynchronous and Synchronous SGD

Asynchronous shows slightly faster convergence but has slightly more oscillations than synchronous in the later stages.



Batch SGD

The error convergence is very similar but Batch sizes of 100 and 1000 show a much smoother convergence graph than SGD.



Bottleneck for synchronous and asynchronous SGD

The bottleneck for both was found to be compute.

There are three major steps in SGD:

1) Read the data from disk
2) Compute local gradient
   - Gather the necessary indices from ‘w’ and compute dot product and local gradient
3) Update ‘w’

It takes ~10-100ms for steps up to 2 and then ~1-2s for step 3. So, adding a sparse gradient vector to a dense vector ‘w’ was the bottleneck. [Note: tf.gather(w, indices) was placed at node 0 to reduce network load]

‘Top’ shows almost 100% CPU utilization at node 0, which is responsible for updating w.

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
29002 ubuntu    20   0 4025736 3.748g  17636 R  99.7 19.1   4:44.81 python
23000 ubuntu    20   0 11.953g 481120   1208 S   5.6  2.3   7:14.50 java
22846 ubuntu    20   0 10.954g 569568   1340 S   0.7  2.8   0:41.17 java
24193 ubuntu    20   0 1850336 359372   1780 S   0.7  1.7   1:12.76 java
 9653 ubuntu    20   0 2607884 217108      0 S   0.3  1.1   3:02.02 java
10186 ubuntu    20   0 2607884 211364      0 S   0.3  1.0   3:03.29 java
12706 ubuntu    20   0 10.899g 135256      0 S   0.3  0.7  44:02.87 java
```

Disk and network utilizations were found to be very low. (Disk utilization could peak when check pointing kicks in but this would be rare and not observed with initial iterations)
For 20 iterations, the number of bytes read and written were 163840 bytes/1642496 bytes and the NW read/write were 787630 bytes/778095 bytes.
The following figure shows it for one instance of time.

```
ubuntu @ ~
 [9] → dstat -c --top-cpu -dn
----total-cpu-usage---- -most-expensive- -dsk/total- -net/total-
usr sys idl wai hiq siq|  cpu process   | read  writ| recv  send
  3   1  96   0   0   0|ksoftirqd/0  0.0|  32k   72k|   0     0
 23  11  63   0   0   2|startserver.p 28|   0     0 |1903B 1230B
 25  11  63   0   0   1|startserver.p 29|   0     0 |2576B 1296B
 25  11  63   0   0   1|startserver.p 28|   0     0 |6859B 5455B
 21  11  68   0   0   1|startserver.p 26|   0   60k|1837B  987B
 20  10  69   0   0   1|startserver.p 17|   0     0 |2708B 1346B
 19  13  68   0   0   0|asyncsgd.py   15|   0     0 |6719B 6277B
 20   9  70   0   0   1|startserver.p 26|   0     0 |2557B 3210B
 21   8  71   0   0   1|startserver.p 14|   0     0 |5142B 2300B
 21   9  69   0   0   1|startserver.p 16|   0   48k|7075B 5806B
```

Assignment 3 Part B
Group -11

Abhanshu Gupta (agupta89@wisc.edu)
Parikshit Sharma (psharma43@wisc.edu)
Vishnu Lokhande (lokhande@wisc.edu)

Application 1

Pagerank application was developed using graphX api's. Two key Api's from graphX which were used were (a) outerJoinVertices and (b) aggregateMessages.

This application has been compared with Assignment1-PartC-Question3 where just spark api's were used to run the pagerank algorithm.

Inorder to make a uniform comparison, the configurations parameters were set the same in both the applications. Specifically, these were the configuration parameters.

("spark.locality.wait", "0")
("spark.driver.memory", "1g")
("spark.executor.memory", "18g")
("spark.executor.instances", "5")
("spark.executor.cores", "4")
("spark.task.cpus", "1")
("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
("spark.logLineage", "false")

The output of the application is saved on hdfs at /pagerank_results. The application completion time is measured by the time taken by the application from the beginning of reading time to saving the results on hdfs. Here is summary.

	Pagerank (spark api's)	*Pagerank (graphX api's)*
Time Taken	29 min	5.4 min
Network Read	13242 MB	18344 MB
Network Write	13549 MB	19026 MB
Storage Read	1434 MB / 7 MBps	1368 MB / 9 MBps
Storage Write	24249 MB / 2 MBps	4807 MB / 2 MBps
Number of Stages	24	68
Number of Tasks	480	544

Comparison

Additional Benefits of graphX:

GraphX provides api's to work directly with the edge and vertex attributes of a graph. This makes developing a graph based application lot easier. One example of this is the "aggregateMessages" api, which passes messages from previous vertices to next vertices over the edges.

GraphX provided “TripletFields.Src” functionality, which helps populate only the necessary fields of an RDD/graph thereby further improving the efficiency.

GraphX when used for graph applications provides better visualizations in the form of triplet views, vertex views and edge views. This is not the case for spark as there is only RDD based view.

<u>graphX vs. Spark</u>
It is observed that pagerank using graphX takes lot less time compared to that of Spark. This is possibly because the ranks graph is being cached in every iteration in the graphX application unlike the spark application. Ranks are used in every iteration, hence, caching them seems to improve the performance a lot. We did not cache ranks in the spark application because we did not get much improvements then as we were dealing with small size datasets. Another possible reason might be that the outerJoinVertices api used in graphX might be more optimized than that of graphX. The number of tasks in spark are less than that of graphX. The number of tasks of an application usually dependent on the level of parallelism which is related to the number of partitions in the application. We used 20 paritions in spark app (because we used 20 paritions before) as compared to 8 parititions in graphX app (default for the datasize).

Application 2
Input: For this application input is the output file "Question2_words.txt" of PartB from Assignment2. We have written a python script "language_cleanup.py" to cleanup web addresses, hashtags and non-alphanumeric words and stored it in the new file "graph_input.txt". This file will serve as input for all sub-applications that are developed under Application2.

Question1
Implementation: PartBApplication2Question1.scala

Logic:
1. Read the "graph_input.txt" file and create a VertexRDD which is a RDD(VertexId,Array[String) using 'map' function.
2. Create EdgeRDD by doing cartesian product of vertexRDD with itself and then filter out the entries where there is no common word between two vertices
3. Create a graph using VertexRDD and EdgeRDD.
4. Get the triplet view of the graph and use filter on the condition that source word list should be larger than the destination word list.
5. Get the count of the filtered RDD as answer

Output: For my case of 262 vertex the total number of edges that satisfy the criteria: 33648

Question2
Implementation: PartBApplication2Question2.scala

Logic:
1. Read the "graph_input.txt" file and create a VertexRDD which is a RDD(VertexId,Array[String) using 'map' function.
2. Create EdgeRDD by doing cartesian product of vertexRDD with itself and then filter out the entries where there is no common word between two vertices.
3. Create a graph using VertexRDD and EdgeRDD.

4. Get the aggregated message of counter from each neighbor, thereby creating a neighbor RDD which stores the number of neighbors of each vertex.
5. Find the maximum degree in the created RDD.
6. Filter the RDD to only have entries corresponding to maximum degree.
7. If RDD has only one entry than the entry is the answer.
8. Else find the entry which has maximum words and that entry will be the answer.

Output: For my case of 262 vertex the vertex most popular corresponds to time interval: 53

Question3
Implementation: PartBApplication2Question3.scala

Logic:
1. Read the "graph_input.txt" file and create a VertexRDD which is a RDD(VertexId,Array[String) using 'map' function.
2. Create EdgeRDD by doing cartesian product of vertexRDD with itself and then filter out the entries where there is no common word between two vertices.
3. Create a graph using VertexRDD and EdgeRDD.
4. Get the aggregated message of word size from each neighbor and reducing it by adding, thereby creating a neighbor RDD which stores the total number of words in neighbors of each vertex.
5. Use map function to take average value on each entry of RDD.
6. Print the RDD values.

Output: The output is list of tuples of the form (VertexId,Average Words in neighbor), printed one vertex per line.

Question4
Implementation: PartBApplication2Question4.scala

Logic:
1. Read the "graph_input.txt" file and create a VertexRDD which is a RDD(VertexId,Array[String) using 'map' function.
2. Flatten the vertexRDD to get list of words and then reduce them on key to get words and their count.
3. Find the maximum count in the created RDD.
4. Print any entry randomly from the RDD that has its count equal to maximum count.

Output: For my case of 262 vertex the word most popular comes out to be: rt

Question5
Implementation: PartBApplication2Question5.scala

Logic:
1. Read the "graph_input.txt" file and create a VertexRDD which is a RDD(VertexId,Array[String) using 'map' function.
2. Create EdgeRDD by doing cartesian product of vertexRDD with itself and then filter out the entries where there is no common word between two vertices.
3. Create a graph using VertexRDD and EdgeRDD.
4. Call the connectedComponents() algorithm on this graph to get connected components.

5. Reduce the RDD based on cluster's min vertex and use a counter to keep track of nodes.
6. Print the RDD value that has the maximum count as the biggest cluster.

Output: For my case of 262 vertex the biggest size of subgraph comes to be: 262

Question6
Implementation: PartBApplication2Question6.scala

Logic:
1. Read the "graph_input.txt" file and create a VertexRDD which is a RDD(VertexId,Array[String) using 'map' function.
2. Flatten the vertexRDD to get list of words and then reduce them on key to get words and their count.
3. Find the maximum count in the created RDD.
4. Locate a word in the RDD that has maximum count.
5. Filter and print all the vertex in vertexRDD that have the popular word in its word list.

Output: For my case of 262 vertex the all the 262 vertex has the most popular word rt