Assignment 3 - PartA

Group8: Xin Jin & Hongyi Wang & Yuzhe Ma

December 4, 2017

1 Part A: TensorFlow

1.1 Q-1: Comparison between Spark and TensorFlow

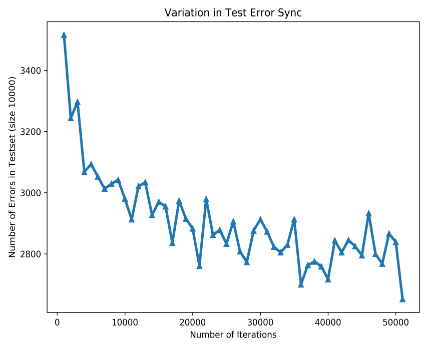
From our viewpoint,

- **TensorFlow** is a specifically designed framework to implement machine learning (especially deep learning). Thus, numerical computation and matrices operations are straightforward in TensorFlow and APIs and ops are similar to basic python operations. And TensorFlow offers enough flexibility for those deep learning experts to design their customized network model (e.g. expand CNN or more complex models) and implement new ML stuff e.g. define new loss functions. Moreover, TensorFlow provides full distributed supports also implements of some specific settings e.g. Parameter Server
  cons: distributed performance in TensorFlow (e.g. speedups for sync/async settings) is not scaled up very well in TensorFlow, which means there is a communication bottleneck in the TensorFlow distributed cluster. Moreover, for combinations of small model and dense data structure, performance (especially time cost) of TensorFlow can be worse than scikit-learn or MATLAB.
- **Spark** is a cluster computing framework, in-memory features will speedup the computation and data processing. In general, nearly all applications can be implemented in Spark and performance can be optimized by parallelization through formalize them into MapReduce. Also, good fault tolerance is achieved in Spark.
  cons: complex machine learning applications can not be efficiently implemented in Spark e.g. huge deep network plus large dataset. Moreover, there is no specific optimization strategy in Spark for sparse data structure, which makes it's performance fall behind TensorFlow.

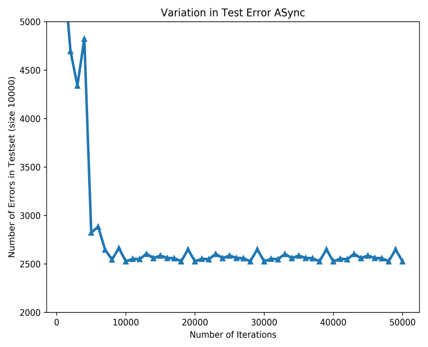
1.2 Sync/Async Testset Performance

After fully utilize the sparsity of the data structure, we can finally optimized the run time into around 6 sec per 1k iterations for synchronous setting and around

2.5 sec per 1k iterations for asynchronous setting. And wrt per iteration time cost, async setting is a bit faster than sync setting (on average). For the results provided on this report, we only run 50k steps for both sync and async settings considering time costs. During the training process, we evaluate the model for each 2k steps, during model evaluation, we used 10k data points from the test set. The variation of test error are given in Fig.1 and time cost statistics are provided in Table.1.
Please note: that results provided here are with model evaluation time considered.



(a) Testset Performance for Sync Setting



(b) Testset Performance for Async Setting

Figure 1: Testset performance for sync/async settings running for 50k iterations with test number at 10k

Table 1: Time Costs of Different Settings

Performance for sync/async Distributed Training		
Distributed Setting	Time Cost (min)	Accuracy Converged To
sync	23.75	72.48%
async	4.75	74.73%

1.3 Bottleneck

For bottleneck checking during these experiments, we just repeat both of the sync/async experiments. For synchronous settings, we run experiment for 10k iterations while run 30k iterations work asynchronous experiments and counts CPU utility, Network read/write, disk I/O as metrics to find out where the bottleneck happens.

Table 2: Metrics for Bottleneck Test

Performance for sync/async Distributed Training					
Distributed Setting	CPU utility (on avg)	Network Read	Network Write	Disk Read	Disk Write
sync	129%	92 M	92 M	1.3 M	2.1 M
async	58%	248 M	248 M	0.31 M	12.13 M

Based on the data provided in Table 3, we can tell that most of the bottleneck for sync settings probably comes from CPU utility, given this high CPU cost, there might be more straggler effect for slow workers (which do not have enough computational resources) and thus the whole training process will affected by stragglers. And most of the bottleneck in async settings comes from network read and write, which make sense since in async mode, communication between nodes are frequent (e.g. mode fetching and gradient sending among worker nodes) in this case communication overhead is relatively heavy.

2 Extra Credit

The implement for mini-batch SGD for this part is similar to the previous applications except for data preparation. Since we will need to implement batched input for this application, we use **tf.parse_example** instead of **tf.parse_single_example**. Also, we used **tf.train.shuffle_batch** to extract data batch at each iteration.

For experiment, we run 20k steps for sync settings and 100k steps for async settings. And test the model performance at the end of the training process. For each setting, we tried batch size at 1 (which reduced to naive SGD, which processing one data point per step), 100, 1k, and 10k respectively, and got similar test set accuracy (fluctuate around 70%-75%).
Please note that, runtime we consider here is with respect to number of data points processed during the training process i.e. for batch size at 1, we run 20k steps, and for batch size at B, we only run $\frac{20k}{B}$ steps to make sure data points processed for each case are the same.
The runtime of mini-batched SGD is given below:

Table 3: Runtime for Different Batch Sizes

Performance for sync/async Distributed Training				
Distributed Setting	single	batch size 100	batch size 1000	batch size 10000
sync	135.7s	67.9s	63.4s	64.8s
async	145.5s	65.5s	62.3s	62.1s

From Table3, we can tell that, we when implement mini-batched SGD instead of SGD (one data point per iteration), by fixing the number of data points we processed in each training process, we can observe speedups. However, the speedups vary among different batch sizes. e.g. for sync setting, the highest speedup is achieved at batch size of 1000 while for async setting, batch size at 1000 and 10000 are both quite promising. Thus, we guess there should be a ”optimal” or ”suboptimal” batch size for different model/cluster setting. However, we need to delve deeper into this topic (e.g. running the experiments for more iterations and trying more batch sizes).