Machine Learning Programs Performances Comparison between Parameter Server, Tensorflow and Spark

Xiaodong Wang (xwang322@wisc.edu)
Minghao Dai (mdai9@wisc.edu)

I. Problem Statement

Solving large scale machine learning problem has become crucial due to the growth of data and resulting model complexity. Implementing an efficient distributed algorithm has become possible with occurrence of parameter server and Tensorflow. The key points in these systems are computational workloads and data communication which demands very careful system design. To get best performances within the balanced point of efficiency and accuracy, distributed optimization and how machines communicate becomes crucial.

Industrial level machine learning training data is huge, usually between 1TB and 1PB. This comes with feature vectors between 10^9 to 10^{12} for a model. Every feature vector needs to be shared globally across all workers. As is common for evaluation of big data systems, we care about two major concerns in these systems. The first one is network bandwidth. Due to the large volume of training data and model parameters, data transfer and model synchronization becomes regular work, which requires enormous amount of network bandwidth. Another concern needs to be taken care is fault tolerance. As the system runs in cloud environment, loss of power or inhomogeneous network could lead to graceful degradation of performances. Besides what has been mentioned above, to train large machine learning model. We also care about latency for updating model. We will discuss this with detailed description in Part C.

While Spark has been a major active project in Hadoop ecosystem, it leverages distributed memory for accessing a cluster's most computational resources. This leads to an important emerging application that enables reusing intermediate results across multiple computation. In machine learning and graph algorithms, data reuse is very common and it would decrease performances hugely by reading and writing to disk for each iteration. Spark solves this question by introducing resilient distributed datasets (RDDs). A lot of work has been done to improve the performances of Spark system from fault tolerances, communication between master and worker and machine learning API. Using Spark to analyzing big data has become very popular.

It is interesting to set benchmark comparison between parameter server (PS), Tensorflow (TF) and Spark as they are suitable for iterative machine learning algorithms. We can get a clear view of the system performances by comparing the completion time and accuracy from each method. By killing the worker machine at some setting time, we can have a rough idea about their fault tolerance ability. It would be interesting to setup a throughout comparison between PS, TF and Spark. We have made most efforts to finish what we plan to do and there is still a lot of work to finalize this complete comparison.

II. System Setup

Parameter Server has been developed by Mu Li *et al.* since 2010. We are using third generation. It has factored out the common required components of machine learning systems. Because of that, we also install MXNet on top of that. MXNet is developed by Tianqi Chen *et al.* for deep learning framework which allows users to self-define a neural network on multiple devices which provides scalable and fast model training. MXNet has mixed machine learning algorithms with graph optimization layer that makes symbolic execution fast and memory efficient. To install parameter server and MXNet, we need to execute following commands respectively on Ubuntu >= 14.04.

For Parameter Server:

```
sudo apt-get update && sudo apt-get install -y build-essential git
git clone https://github.com/dmlc/ps-lite
cd ps-lite && make -j4
```

For MXNet, a lists of quick installation commands:

```
sudo apt-get update
sudo apt-get -y install git
git clone https://github.com/dmlc/mxnet.git ~/mxnet --recursive
cd ~/mxnet/setup-utils
bash install-mxnet-ubuntu-python.sh
source ~/.bashrc
```

To make synchronous and asynchronous mode running across the cluster, we have changed following setting, in mxnet/make/config.mk, change USE_DIST_KVSTORE = 1. After that, get into the python lib for mxnet and reinstall the python library and recompile the codes:

```
cd /home/ubuntu/mxnet/python
sudo python setup.py install
cd ../
make clean_all
make
```

We implement above codes for all five VMs.

A simple way to test whether MXNet can run normally is getting into its example directory and run the script training a simple MNIST dataset by generating a multiple-layer processing network.

```
cd /home/ubuntu/mxnet/example/image-classification/
python train-mnist.py --network mlp
```

The default will take a few minutes with finally achieving more than 98% accuracy. To test the system able to run in synchronous and asynchronous mode, we create a file called hosts, inside which is private ips of 5 VMs, e.g.

```
$ cat hosts
10.254.0.254
10.254.0.197
...
```

By running the following command, we can verify that we have set everything correctly:

```
../../tools/launch.py -n 2 --launcher ssh -H hosts python
train_mnist.py --network lenet --kv-store dist_sync
```

By setting the kv-store value to dist_sync and dist_async respectively, we can run either synchronous or asynchronous mode.

For Tensorflow (TF), we use what is suggested in assignment 3 installation part commands across all the VMs:

```
sudo apt-get install --assume-yes python-pip python-dev
export
TF_BINARY_URL=https://storage.googleapis.com/tens
orflow/linux/cpu/tensorflow-0.11.0rc2-cp27-none-
linux_x86_64.whl
sudo pip install --upgrade $TF_BINARY_URL
```

For Spark:

Since the implementation of Spark part of this project relies on Spark's machine learning library, MLlib and Spark's Python API, which are already included in Spark package that we have set up in assignment 2 and assignment 3 for our cluster. There are no extra steps necessary for our project's purpose. But we did tune the SparkContext of memory for driver and executors based on our experiments and garbage collection option.

```
.config("spark.executor.memory","14g")
.config("spark.drive.memory","8g")
.config("spark.executor.extraJavaOptions","-
XX:+UseG1GC")
```

III. Implementation

After configuration work is done, we plan to run a bunch of tests based on system configuration. As our project has some overlapping with assignment 3, we decide to use TF reader class to read binary data from assignment 3. The original data has been one-hot encoded so that by calling tf.TFRecordReader() is the easiest way to read from queue. Our dataset has been split into 23 different parts and we have put them in HDFS and local machines. We plan to apply MXNet functions processing them through logistic regression, SVM and other machine learning methods. This combining performances test from assignment 3 and Spark will constitute our main work for this report.

One key thing in our implementation is that comparing the effects of using sparse matrix and dense tensor. Our data is about clicking ads whose original data has 39 features, 13 of which are numerical and the rest of 26 are categorical. After one-hot encoding (OHE), the data has grown to 33762578 feature vectors. The computation work has grown huge but necessary. To construct a matrix with 33762578 features is not an easy job and it does not show up in ordinary machine learning examples. One advantage about TF in assignment 3 is its ability to handle sparse matrix. By calling tf.gather() and tf.sparse_add(), we are able to minimize the bottleneck of workload most of which is network. However, in MXNet, currently there is no easy way to do this. MXNet limits the input as NDArray so converting from sparse values after reading each example by tf.TFRecordReader() to dense vector is a prerequisite. We expect to see this induces performance differences during our testing stage.

Spark nicely supports sparse vector representation by introducing SparseVector data type since Spark 2.0. To have a consistent logic with MXNet, we tried implement the experiment for spark in the way of: 1. read the entire sparse data set as an RDDs; 2. map each sparse element to dense for learning model; 3. let the model do online training in the unit of single feature example. But the plan fails for reasons discussed in Part D. Instead, we use the same data set provided by Kaggle but without encoded in

OHE fashion and carry out the conversion of categorical features to numerical within our codes. This solution somehow solves issues but still have glitches.

Another interesting work is to compare the results among different machine learning algorithms. This is relatively easy work as popular machine learning algorithms have been packaged into most machine-learning based systems. We plan to compare some algorithms which are suitable to classify binary labels, like logistic regression, SVM and others for their performances.

As we mention aforehead, fault tolerance is another important criterion to justify a big-data system. TF and MXNet has relatively simple fault tolerance mechanism (checkpoint-based), while Spark has a rather robust one to prevent machines from breaking down. Whenever some tasks are lost, the recovery system re-read corresponding input data and reconstruct RDDs via lineage. This fault recovery mechanism consumes less network and storage resources compared to a checkpoint-based way.

We use the scripts we have developed in the assignment 3 to parse the dataset. After communicating with Mihir Shete and Felipe Gutierrez, we also implement their methods. Technically those two methods are very similar, while the only difference is Felipe and Mihir have redefined a new class special for parsing TF data while we are embedding our MXNet machine learning methods together with codes to read TF data. Two methods show almost the same speed for processing the examples (1.41 sample/sec versus 1.33 sample/sec), so we decide to run all the bench tests using our codes.

The last thing possible to compare is between batch processing and online processing. While our machine learning is based stochastic gradient study, batching a size of examples versus single example would be expected to report something different.

IV. Experiment

After determining the interesting aspects for our project, we have done a bunch of plan-based tests. For TF and MXNet, the most difficulty comes with dealing data. As the MXNet community has not built on dealing with binary dataset, our data must be parsed by TF specific class. This part has been stated in last section and we eventually fixed it. To train the model after successfully parsing the data is relatively easy. One point might worth some attention here is that MXNet is designed for deep learning. In some machine learning topics such as image classification and speech recognition, it is hard to feature out some useful information so reading in the whole image or speech audio is unavoidable. However, we have already well-defined data so it is not necessary to build a complex network before training period. We only define one layer from input data to single output node. Our single output node is used to tell the label of numerical classification.

For the part of Spark ML, our initial plan was to read entire data set containing sparse vectors into distributed storage system as RDDs and let the learning model train the sparse vector. But this plan always results in out of memory error for heap space when the number of features becomes very large. By testing the model to train a single line of feature example, the largest number of features Spark can handle is around 10 million, which is far less than the number of features in the Kaggle sparse data set. Even though aforementioned method works, Spark MLlib training models do not support to train the data in a loop by iterating every single feature example. We have also come up an idea to further partition the sparse data set by features such that each partition can fit in the capacity we have explored. But this idea involves manually updating of training models that we select. Unfortunately, we have not found a way of updating models.

After doing more research online, we find out the Pipeline feature provided by Spark ML API. In Spark MLlib, it represents the workflow of processing input data, converting categorical features into a numerical feature vector, and learning a prediction model using feature vectors and labels as Pipeline, which contains a sequence of PipelineStages to be run in a specific order. Then in our implementation, we chain various feature transformers and learning algorithms into our Pipeline and then train the pipeline over training data. To convert categorical features to numerical, we use StringIndexer and OneHotEncoder from API. StringIndexer encodes a string column of labels to column of label indices, which can then be passed onto the OneHotEncoder that encodes them into a binary vector. Thus, the combination of these two transforms each categorical feature with m possible values into m possible binary features with only one active.

V. Results

To illustrate some comparisons between TF, MXNet and Spark, we will use figures and values. Some of the results are from assignment 3. Our training set and test set keep a ratio 1:1. We first plot the accuracy from Logistic Regression by TF, MXNet and Spark. We have run 500, 1000 and 5000 training samples. The result is shown in fig 1. We find all three systems tend to converge after

1000 iterations. We have seen this in assignment 3. We thought the convergence would not come so fast and we have far more data than what we can run. The overall data number is around 10M. By only using 5000 of them, we already get converged curve. Another point is this comparison is between synchronous mode, as Spark default is synchronous mode, running asynchronous mode in Spark needs more time study which unfortunately we do not have in our project. There is some doubt about accuracy in TF, when we were working on assignment 3, we were also confused about our low accuracy as it was expected above 70%. We discuss our codes with Raajay Viswanathan a couple of times but still achieving this result. We care more about convergence rate in this case, not accuracy value itself so much here. It clearly shows that all three systems converge roughly at the same time.

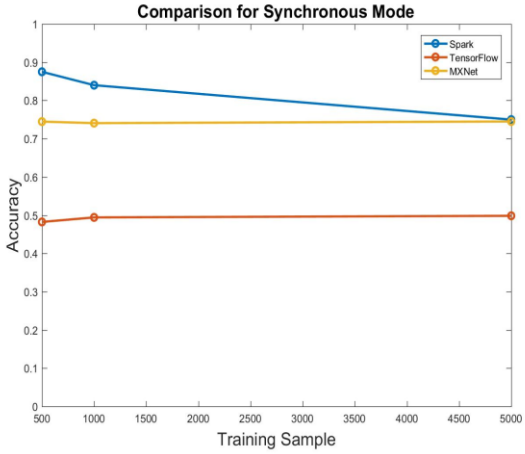

Fig.1. Different Systems Synchronous Mode Comparison

Next Comparison is between synchronous mode and asynchronous mode. We have not implemented asynchronous mode in Spark, so this comparison is between TF and MXNet. From fig.2, it shows the same issue as before, TF returns a lower accuracy rate. This has been discussed before and we are not digging more here. One interesting thing is synchronous mode gives a little higher accuracy than asynchronous mode, in both MXNet and TF. This matches our assumption as synchronous mode performs better in accuracy while resource utilization efficiency is lower than asynchronous mode.

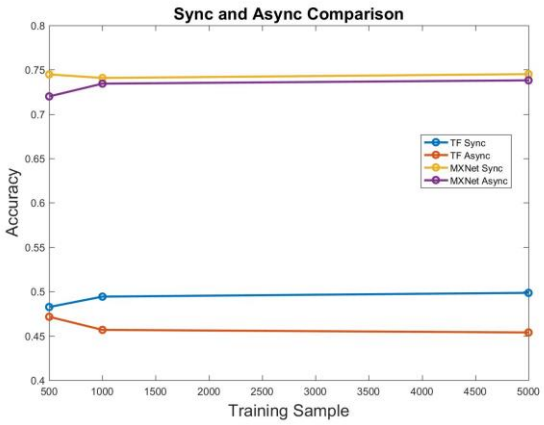

Fig.2. Synchronous and Async Mode Comparison

The third comparison is among different machine learning algorithms. We have limited this comparison on MXNet. We have compared three algorithms, softmax, SVM and logistic regression. The conclusion is that they give very close classify results, between 71% to 79%. SVM give slightly better than the other two. We think as the current data is feature extracted done, there would be a upper limit for the most accuracy it achieves for whatever machine learning algorithms. We believe by more feature engineering, accuracy can be improved.

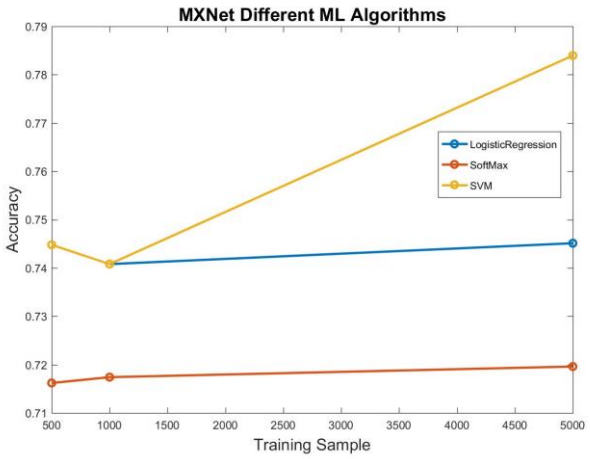

Fig.3. MXNet Machine Learning Comparison

The fourth comparison is about completion time. We use table here to illustrate our point. Between TF and MXNet, we have compared running the same dataset (1) locally (2) synchronous and (3) asynchronous. To run locally, we have put the dataset on one single machine and we expect this would be much faster than fetching from HDFS as network traffic is not causing any issue.

	Local	Synchronous	Asynchronous
TF	20/sec	1.2/sec	2/sec
MXNet	1.49/sec	1.37/sec	1.39/sec

Table.1. Completion Rate Comparison

From what we have observed, running either mode on MXNet returns almost same speed. TF sync mode runs a little faster than asynchronous mode. But except for local mode in TF, they all fall into a same range from one sample per second to two samples per second. However, in TF, locally running the program finishes much faster. The reason is the ability that TF can handle sparse matrix algebra. It takes in a weight vector and gathers with sparse matrix from one example, using index to update only very tiny part of the overall weight (at most 39 out of 33762578). With no traffic in network, locally updating weight saves a lot of time. While in TF sync/asynchronous mode, communication between machines is still a bottleneck. So is MXNet. What is worse in MXNet is its lacking ability to process sparse matrix. The bottleneck in MXNet is both CPU and network.

Since in our implementation, Logistic regression model runs much faster on Spark among same data set, we want to run 500,000 samples to achieve the convergence on accuracy. We randomly split the data set into training and testing with a ratio of 8 to 2. From the plot, the model is overfitted and yields a high accuracy with small number of total samples. But we still achieve a convergence by increasing the data set size. Unfortunately, we must constrain the input data set size under 10 million samples, otherwise an out of memory error will be raised from our Spark cluster. We have tried to tune our Spark configurations but have not found a solution to this problem yet. We guess the issue may be caused by our implementations of the machine learning framework, or it's simply because of limitations of the Spark framework.

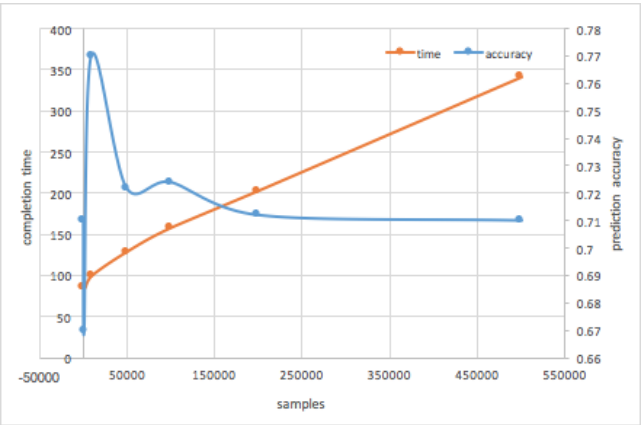

Fig.5.Spark Machine Learning run time and accuracy

The last comparison is about fault tolerance. Spark programming model provides nice functions of fault tolerance. We have the chance to test it during the experiment of running Spark Logistic Regression among 100000 samples. We killed one of workers at 30%, 50%, and 80% of run time of an execution. From the plot, we can conclude that the fault tolerance function helps to prevent degrading the performance in term of accuracy, with a penalty in execution time.

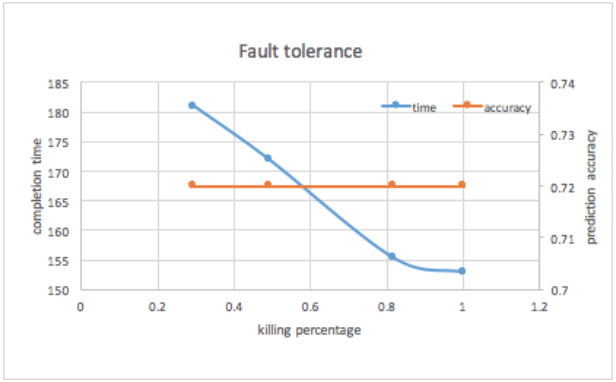

Fig.6.Spark fault tolerance test on 100000 samples

We have also done similar tests on MXNet for fault tolerance. What is different from Spark is the sample size. As MXNet completion rate is much slower than Spark, we would run a 2000 sample size for our experiments. We killed the same worker at the same ratio of overall completion time. The results show that the earlier it was killed, the longer it would take to recover. This has the same trend with Spark. What is different is that the accuracy is much more stable than Spark. We have observed before that the convergence achieved before 1000 samples are trained. This matches our expectation before doing the test.

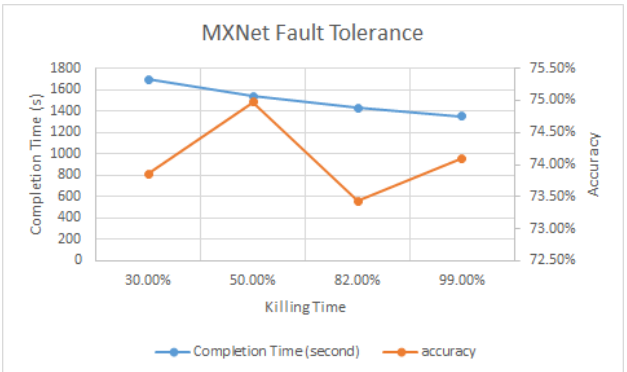

Fig.7.MXNet fault tolerance test on 2000 samples

VI. New ideas

Before giving some of our thoughts about this project possible ideas, we want to summary about what we have not got a chance to run. It is obvious that the correctness of our TF codes is a possible problem. As it gave a much lower accuracy than MXNet and Spark, we have some doubts on its accuracy. TF is the only platform where no well-defined API for machine learning algorithms. It seems like somewhere in our codes there are some bugs.

Another idea we have not done is comparison between batch and online machine learning. This is due to a couple of reasons. First, Spark does not support online machine learning. The way we made it working is only valid for batching a few examples at the same time. We know from references papers that batch processing gives a faster convergence rate. However, convergence is not a big problem in our test as all three platforms show a fast convergence. Secondly, running batch processing on MXNet is hard. We did trial on MXNet writing the parsed data examples to a 2D NDArray. However, using mx.NDArray() to save and load data is taking much more space than the cluster can handle. If we want to write a file containing 100 records, it takes more than 10GB. To write it to a file is one thing, while to load it back to memory is another. This is why we write a class specially for parsing TF record as converting it to a dense tensor by mx.NDArray() is taking too much space we can handle. We do not think this is a reasonable way to do even with much more cluster hardware configurations. If we can implement this in MXNet, we would expect something similar happens in MXNet. The batch training will show a faster convergence rate.

Some interesting thoughts about this project: First, the way how feature is extracted would determine the upper limit of prediction accuracy. We tried three ML algorithms and they respectively predicted an accuracy below 80%. We think this is mostly due to feature selection. For logistic regression, some extra possible work is to tune the learning rate and momentum value. Our current setting is learning rate is 0.01 and momentum value is 0. Changing this would somehow change what we observed and probably provide some optimizations. Second, throughout the Spark ML part of our project we noticed some limitations of Spark ML framework and graces brought by parameter server model. We think a mixture of these two frameworks, an implementation of "Parameter Server on Spark" could further facilitate the way of people doing large-scale data analysis. Fortunately, some developers from Spark community has already proposed similar idea and there's a plenty of room to further study.

Acknowledgment

We would like to appreciate Mihir Shete and Felipe Gutierrez Barragan for sharing codes and discussions for evaluation details.