Report on

PERFORMANCE EVALUATION OF NLP APPLICATION USING DISTRIBUTED TENSORFLOW ON SPARK

By

SRUJAN BANDARKAR

SRIDEVI DIVYA KRISHNA DEVISETTY

Abstract

The increasing complexity of Neural Networks has made it challenging to exploit existing large-scale data processing pipelines for handling massive data and parameters involved in Deep Neural Networks (DNN) training. In this paper, we explore a model called TensorFlow on DeepSpark that simultaneously exploits Deep Spark on commodity clusters by implementing TensorFlow on Spark framework and evaluate its performance by comparing it to existing TensorFlow models executed in standalone mode. Our model is motivated from yahoo's TensorFlow on spark implementation to facilitate distributed execution of TensorFlow models. To support parallel operations, our model automatically distributes workloads and parameters to TensorFlow-running nodes using DeepSpark, and iteratively aggregates training results by a lock-free asynchronous variant of the popular elastic averaging stochastic gradient descent-based update scheme, effectively complementing the synchronized processing capabilities of Spark.

1] Introduction

With massive amounts of computational power, machines can now recognize objects and translate speech in real time. Artificial intelligence is finally getting smart and Deep neural networks (DNNs) have played a significant role in the field of Artificial Intelligence. For instance, automatically adding sounds to silent movies, this uses application of both convolutional neural networks and Long short-term memory (LSTM) recurrent neural networks (RNN). DNN's have many other applications that have significantly contributed in the advancements made in the field of Artificial Intelligence.

DNNs deliver a sophisticated modeling capability underpinned by multiple hidden layers. The input layer receives input data. The input layer passes the inputs to the first hidden layer. The hidden layers perform mathematical computations on our inputs. One of the challenges in creating neural networks is deciding the number of hidden layers, as well as the number of neurons for each layer. The "Deep" in Deep Learning refers to having more than one hidden layer. The more neurons and layers that need to be trained, and the more data available for training, the longer it takes. The output layer returns the output data.

Each connection between neurons is associated with a weight. This weight dictates the importance of the input value. The initial weights are set randomly. The neuron which affects the prediction the most will have a big weight. Each neuron has an Activation Function. These functions are hard to understand without mathematical reasoning. Simply put, one of its purposes is to "standardize" the output from the neuron. Once a set of input data has passed through all the layers of the neural network, it returns the output data through the output layer.

1.1 Main Problem and Related works

Although having multiple hidden layers allows DNNs to have powerful non-linear modeling capability, training such DNNs generally requires a large volume of data and a huge amount of computational resources. This leads to long training time ranging from several hours to days even.

Though there are multiple solutions for DNN training, each of these existing frameworks has at least one distinguishing characteristic. Caffe's strength is convolutional DNNs for image recognition. Cognitive Toolkit has a separate evaluation library for deploying

prediction models that works on ASP.Net websites. MXNet has excellent scalability for training on multi-GPU and multimachine configurations. Scikit-learn has a wide selection of robust machine learning methods and is easy to learn and use. Spark MLlib integrates with Hadoop and has excellent scalability for machine learning. TensorFlow has a unique diagnostic facility for its network graphs, TensorBoard. But the training speed of all these deep learning frameworks on GPUs is nearly identical.

1.2 Limitations that current solutions carry

Various approaches have been proposed to improve the efficiency of deep learning training. Highly optimized GPU implementations have significantly shortened the time spent on training DNNs, often showing 10–100× speed-up. However, accelerating DNN training on a single machine has limitations because of the limited resources such as GPU memory or the host machine's main memory. Scaling out methods in distributed environments have been suggested to overcome such issues.

1.3 Why is the problem important?

Many real-world datasets used for DNN training (such as raw images or speech signals) need to be converted into a format required by deep learning platforms and often require preprocessing to improve robustness. Such datasets are typically huge in scale; thus, the preprocessing procedure demands a considerable amount of time and resources and requires carefully designed software to process. However, these implementations seldom achieve speed-up for the cluster built on commodity hardware infrastructure. Speed is always a factor which is taken into consideration while training and if the time required to training is very long then using that model practically is not feasible. Just think about it who would want to wait weeks to several days just to train a model? And training models which huge datasets which is the case in Neural Nets may take weeks and days for training it. So, we propose TensorFlow on Deep spark model, which drastically reduces the training time of the model.

In recent releases, TensorFlow has been enhanced for distributed learning and HDFS access. But the need for dedicated clusters for TensorFlow applications is still an unfulfilled requirement. While the existing approaches of implementing TensorFlow on spark clusters are a step in the right direction, they are limited to support synchronous distributed learning only, and don't allow TensorFlow servers to communicate with each other directly. Thus the need for effective distributed environment for implementing TensorFlow programs is still a unresolved issue.

1.4 Our solution

We propose a model, a deep learning Tensor Flow model on Deep Spark framework, to accelerate DNN training, and address the issues encountered in large-scale data handling. Specifically, our contributions include the following:

1. Integration of scalable data with deep learning: our model uses deep learning framework interface and the parameter exchanger on Apache Spark, which provides a straightforward but effective data parallelism layer.

2. Overcoming high communication overhead using asynchrony: We propose an asynchronous stochastic gradient descent (SGD) for better DNN training in Spark.

3. Flexibility: Our TensorFlow on Deep Spark Model, a deep learning framework for accelerating deep network training can be implemented both for Images and Text based Neural Network Applications based on DNN.

2] Deep Learning

Training Deep Neural Network (DNN) consists of two steps: feed-forward and backpropagation. In the feedforward step, an input pattern is applied to the input layer and its effect propagates, layer by layer, through the network until an output is produced. The network's actual output value is then compared to the expected output, and an error signal is computed for each of the output nodes. Since all the hidden nodes have to some degree, contributed to the errors evident in the output layer, the output error signals are transmitted backwards from the output layer to each node in the hidden layer that immediately contributed to the output layer. This process is then repeated, layer by layer, until each node in the network has received an error signal that describes its relative contribution to the overall error. Once the error signal for each node has been determined, the errors are then used by the nodes to update the values for each connection weights until the network converges to a state that allows all the training patterns to be encoded. The Backpropagation algorithm looks for the minimum value of the error function in weight space using a technique called the delta rule or gradient descent. The weights that minimize the error function is then considered to be a solution to the learning problem. Parameter optimization (e.g., stochastic gradient descent, RELU) is executed during the backpropagation step to better fit the target function. Although the complex neural network model successfully approximates input data distribution, it inherently leads to a large amount of parameters to learn. This situation demands a huge amount of time and computational resources, which are two of the major concerns in deep learning research. Some neural networks models are so large they cannot fit in memory of a single device (GPU). Google's Neural Machine Translation system is an example of such a network. Such models need to be split over many devices (workers in the TensorFlow documentation), carrying out the training in parallel on the devices. This leads way to distributed deep learning.

2.1 Distributed Deep Learning

The training procedure of splitting the training model over many devices is commonly known as model parallelism or in-graph replication in the TensorFlow documentation.

In data parallelism or between-graph replication in the TensorFlow documentation, you use the same model for every device, but train the model in each device using different training samples. This contrasts with model parallelism, which uses the same data for every device but partitions the model among the devices. Each device will independently compute the errors between its predictions for its training samples and the labeled outputs which are correct values for those training samples. Because each device trains on different samples, it computes different changes to the model known as the gradients. However, the algorithm depends on using the combined results of all processing for each new iteration, just as if the algorithm ran on a single processor. Therefore, each device has to send all of its changes to all of the models at all the other devices. Parameter

Optimizations are done using Stochastic Gradient Descent(SGD).

2.2 Parallelizing SGD

Stochastic gradient descent (SGD) is an iterative algorithm for finding optimal values and is one of the most popular algorithms for training in AI. It involves multiple rounds of training, where the results of each round are incorporated into the model in preparation for the next round. The rounds can be run on multiple devices either synchronously or asynchronously. A naive parallelization SGD can be implemented by splitting batch calculations over multiple worker nodes. The global parameters are initialized and broadcasted from master to each worker, and the workers will then derive gradients from their local data. Considering that the throughput of each node can differ, two strategies can be utilized to parallelize gradient update: synchronized or non-synchronized.

In synchronous training, all of the devices train their local model using different parts of data from a single large mini-batch. They then communicate their locally calculated gradients directly or indirectly to all devices. Only after all devices have successfully computed and sent their gradients is the model updated. The updated model is then sent to all nodes along with splits from the next mini-batch. That is, devices train on non-overlapping splits (subset) of the mini-batch. Synchronous SGD waits for every worker node to finish its computation and reach the barrier. Once all worker nodes have completed their tasks, the master node collects all the gradients, averages them, and applies them to the center parameter. Worker nodes then pull the updated parameters from the master. Although parallelism has the potential for greatly speeding up training, it naturally introduces overhead. A large model and/or slow network will increase the training time. Although synchronous SGD is the most straightforward form of parallelizing SGD, its performance is highly degraded by the slowest worker. It also suffers from network overhead while aggregating and broadcasting parameters through the network.

In asynchronous training, no device waits for updates to the model from any other device. The devices can run independently and share results as peers or communicate through one or more central servers known as parameter servers. In the peer architecture, each device runs a loop that reads data, computes the gradients, sends them (directly or indirectly) to all devices, and updates the model to the latest version. In the more centralized architecture, the devices send their output in the form of gradients to the parameter servers. These servers collect and aggregate the gradients. In asynchronous training, parameter servers send gradients to devices that locally compute the new model. Asynchronous SGD has been suggested to resolve the inefficiency caused by the synchronous barrier locking. In the lock-free asynchronous SGD, each worker node independently communicates with the central parameter server without waiting for other nodes to finish. This strategy seems to be at risk for stale gradients, however, asynchronous SGD has been theoretically and empirically investigated to converge faster than the SGD on a single machine.

However, many GPUs can also be used for parallelizing hyperparameter optimization. That is, when we want to establish an appropriate learning rate or mini-batch size, we can run many experiments in parallel using different combinations of the hyperparameters.

2.2.1 Parameter Server

The notion of a parameter server is a framework that aims for large-scale machine learning training and inference. Its master-slave architecture distributes data and tasks over workers, and server nodes manage the global parameters. Parameter Server provides two advantages. First, by factoring out commonly required components of machine learning systems, it enables application-specific code to remain concise. At the same time, as a shared platform to target for system-level optimizations, it provides a robust, versatile and high-performance implementation capable of handling a diverse array of algorithms from sparse logistic regression to topic models and distributed sketching. In previous works, a parameter server has been successfully used in training various machine learning algorithms. Efficient communication, flexible consistency, elastic scalability, fault tolerance and durability and ease of use are the key features of parameter server.

2.3 Reducing Communication Overhead

The distributed training of DNNs consists of two steps: exploration and exploitation. The former is to explore the parameter space to identify the optimal parameters by gradient descent, and the latter is to update center parameters using local workers training results and proceed to the next step. Given that the parameter exchanging causes network overhead, a performance tradeoff exists between workers exploration and a master's exploitation.

Some approaches proposed reducing the communication dominance instead of accepting the penalty of the discrepancy. SparkNet presented the iteration hyperparameter τ, which is the number of processed minibatches before the next communication. The distributed training system can benefit from the large value of τ under the high-cost communication scenario by reducing communication overhead. However, large τ may end up requiring more iterations to attain convergence, which slows down the learning process.

The Elastic Averaging Stochastic Gradient Descent(EASGD) strategy can be used to maximize the benefit of exploring by regulating the discrepancy between a master and workers.

SGD, where gradients of local workers are shipped to the master, and updated center parameters are sent back to workers at every update. When updating parameters, worker nodes compute the elastic difference between them and apply the difference on both master and worker parameters. To compute the elastic force, moving rate $\alpha \in (0,1)$ is involved. At every communication, each worker and the master node update their parameters as follows:

$$w_{worker} = w_{worker} - \alpha(w_{worker} - w_{master})$$

$$w_{master} = w_{master} + \alpha(w_{worker} - w_{master})$$

EASGD shows a faster convergence of the training even with large value of τ, with which downpour SGD shows a slow convergence rate or even cannot converge.

2.4 Apache Hadoop, Spark and TensorFlow on Spark

Apache Hadoop YARN and Spark are cluster frameworks that allows large scale data processing on commodity hardware. In Hadoop, data sets are split into multiple blocks in Hadoop Distributed File System (HDFS). HDFS provides the overall control of these blocks and maintains fault tolerance.

Hadoop YARN, which is the framework for resource management and job monitoring, is responsible for containers working in parallel. Spark is the cluster computing engine running on YARN, Apache Mesos or EC2. The core of Spark is the in-memory distributed processing using resilient distributed dataset (RDD). RDD is the read-only collection of data partitioned across a set of machines. On RDD, parallel actions are performed to produce actual outcome or transformations can be applied in order to convert RDD into other type of RDD. For further reuse, RDD can be cached in cluster memory, which prevents unnecessary storage I/O and thus accelerates data processing. Hadoop and Spark are originally built for batch synchronized analysis on large data. They are, however, less suited for jobs that requires asynchronous actions on parallel workers. TensorFlow on Spark model uses the apache spark implementation which in-turn does not support the synchronized analysis of large data.

3] PROPOSED TensorFlow on Deep Spark Framework

3.1 Motivations

Apache Spark is an attractive platform for data-processing pipelines such as database query processing. However, the frequent demand for communication operates to the disadvantage of the synchronous SGD process on the commodity environment. Furthermore, Spark RDD provides limited asynchronous operations between the master and the workers.

To address the disadvantages of Spark, we implemented an asynchronous SGD solver with a custom parameter exchanger on the Spark environment. Our implementation is motivated from yahoo's TensorFlowonSpark and DeepSpark frameworks.

Our Model consists of three main parts: namely, Spark Driver, a parameter exchanger for asynchronous SGD TensorFlow model, and the GPU-supported computing engine. Apache Spark manages workers and available resources assigned by a resource manager.

3.2 Distributed Setup for TensorFlow on Spark for Deep Learning

How our model works and why is it better than other models

In this section, we explain our model's distributed workflow from the data preparation to asynchronous SGD. Given that the model is running on top of the Spark framework, it needs to load and transform raw data in the form of Spark RDD.

The first step in model training is to create RDD for training and inference. We defined a container class, which stores label information and corresponding data for each data sample. The data-specific loader then creates the RDD of this data container class, which is followed by the preprocessing phase. In the preprocessing phase, data containers would be connected to the preprocessing pipeline such as filtering, mapping, transforming, or shuffling. The processed RDD repartitioning is then performed to match the number of partitions to the number of worker executors.

TensorFlow on Spark, the actual backend computing engine, however, cannot directly access RDD. In our model, the entire dataset is distributed across all workers, and each worker can cache or convert its own parts of the dataset into the LMDB (Lightning Memory Mapped Database) file format if the data are relatively larger than the memory size. For a relatively small dataset, the RDD for each Partition action is executed, where every data partition is loaded in the local worker's memory as a form of the List. These data then become available to neural network model of

the backend using a tensor object for the TensorFlow. In this case, we should set the dimension of data, such as batch size, the number of channels, the sentence width, and height.

For a large dataset that is difficult to hold in the physical memory, the RDD for each operation is performed, and each data partition is converted to the LMDB (Lightning Memory Mapped Database) file format and stored in the temporary local repository of the node it belongs to. Once the LMDB (Lightning Memory Mapped Database) files are created, TensorFlow On Spark automatically computes data dimension and finally completes the neural network model parameters. We used LMDB (Lightning Memory Mapped Database) JNI to manipulate LMDB on Spark.

Distributed TensorFlow consists of a cluster containing one or more parameter servers and workers. Since workers calculate gradients during training, they are typically placed on a GPU. Parameter servers only need to aggregate gradients and broadcast updates, so they are typically placed on CPUs, not GPUs. One of the workers, the chief worker, coordinates model training, initializes the model, counts the number of training steps completed, monitors the session, saves logs for TensorBoard, and saves and restores model checkpoints to recover from failures. The chief worker also manages failures, ensuring fault tolerance if a worker or parameter server fails. If the chief worker itself dies, training will need to be restarted from the most recent checkpoint.

One limitation of Distributed TensorFlow, part of core TensorFlow, is that we have to manage the starting and stopping of servers explicitly. This means keeping track of the IP addresses and ports of all the TensorFlow servers in our program and starting and stopping those servers manually. Hence, we used YARN as resource manager. It is also recommended to use Kubernets or Mesos as resource managers alternate to YARN in order to reduce the complexity of configuring and managing TensorFlow applications.

On the other hand, TensorFlow on Spark (TFoS) framework was designed to run on existing Spark and Hadoop clusters, and use existing Spark libraries like SparkSQL or Spark's MLlib machine learning libraries. In TFoS Parallel instance of TensorFlow can communicate directly. Data can be ingested from TensorFlow's native facilities or reading from HDFS or through spark. As the native TensorFlow does not support remote direct memory access, TFoS has its own RDMA support. Current TensorFlow releases, however, only support distributed learning using gRPC over Ethernet. However, RDMA is faster compared to gRPC. RDMA manager is created to ensure tensors are written directly into the memory of remote servers. To minimize the tensor buffer creation: Tensor buffers are allocated once at the beginning, and then reused across all training steps of a TensorFlow job.

3.3 Asynchronous EASGD Operation

Inherently, Spark does not support step-wise asynchronous operations for asynchronous SGD updates. We adopt the method that exploits Spark RDD operations to overcome the limitation of Spark.

Once the LMDB (Lightning Memory Mapped Database) local repository for each worker has been prepared, the dummy RDD is created and distributed across every worker. These dummy data have an important role in launching the distributed action (i.e., parallel model training is performed). Although the explicit dependency between spilling and training steps is nonexistent at the code level, each worker node would be guided to launch the training process with a spilled dataset by

the dummy RDD. This exploits the property of Spark that the Spark scheduler reuses the pre-existing worker node session. The size of the dummy RDD is explicitly set to the number of workers for full parallel operation, and the for each Partition action is executed on this dummy RDD. Inside the for each Partition process, each worker can use the local data repository that has been created in the previous job and starts the training step.

During the training process, the Spark driver program serves as a central parameter exchanger, which performs an asynchronous EASGD update. At the initial step, the driver node broadcasts its network address and neural network setup files to all workers. Workers then create their own models and start training using broadcasted data.

3.4 Parameter Exchanger

The parameter exchanger is essential for asynchronous update. In our model, the application driver node serves as the parameter exchanger to enable worker nodes to update their parameters asynchronously.

When multiple-parameter exchange requests from worker nodes exist, a thread pool is implemented to handle the requests at the same time. For each connection request, the thread pool allocates the pre-created threads that process the parameter-exchange requests. The size of the thread pool is fixed in the program, and we set this up to eight threads because of limited memory and network bandwidth. If the number of requests exceeds the size, the unallocated requests wait in a queue until the preceding requests are completed.

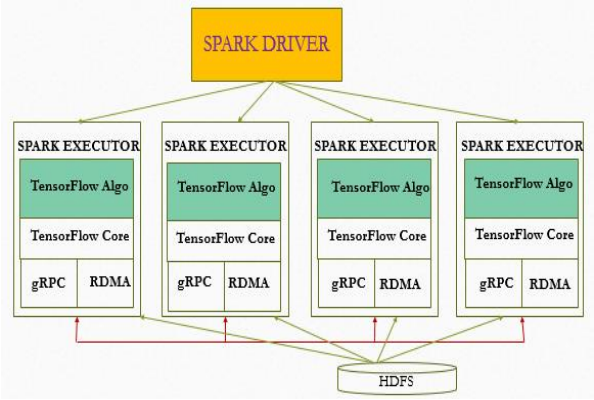
Exchange threads asynchronously access and update the neural net model in the parameter exchanger based on the EASGD algorithm. Given that it is a lock-free system, parameters can be overwritten by simultaneous updates. Nevertheless, training results are accumulated successfully. After the parameter exchange action, each worker returns to the SGD phase to explore the parameter space asynchronously.

3.5 Backend Engine and Adaptive Communication Period

Each worker node can use TensorFlow on Spark for the GPU accelerated backend engine. However, Spark application is written in Java, Scala, and Python, which cannot use the native backend engines directly in the source code level. We implemented our code with Python so that Spark executors can reference the native library TensorFlow. We can run a distributed TensorFlow session on the wrapper with the operational graph which was obtained from model description written in Python.

Because we observed that the testing loss decreased faster at the early learning stage, we guessed that the early learning stage communication affected more the model training than the communication of later iteration. Thus, we designed the heuristics to increase the communication period as the model trained. At the initial setup stage, we set the threshold to communicate. During the training process, each executor cumulates its own training loss. At the time the cumulated loss arrives at the threshold, the executor performs parameter exchanging and then resets the cumulated loss. In a general situation, the communication period will increase as the training loss drops. The grown period aids in relaxing communication overhead.

Below is the prototype model of tensorflow on deep spark .



4] Experiments

4.1 Experimental setup and Feasibility

We conducted our experiments in several ways. One of them involved running the TensorFlow model in stand-alone mode and another experiment involved executing the TensorFlow model in distributed mode.

We prepared a single-cluster environment and also a distributed cluster environment. The distributed cluster was composed of 4 identical virtual machines. Each with 4GB RAM and 4 virtual cores. We have installed spark, pyspark, anaconda, python, TensorFlow, TQDM on each of these clusters.

One of these clusters was used for conducting the experiments for stand-alone execution of our TensorFlow model. Our model is executed on Ubuntu 16.04 virtual machine with 4GB memory and 4 virtual processor cores. We have used standalone mode of spark to execute this model.

In distributed TensorFlow training one of the clusters formed as a chief worker and coordinated model training, initialized the model saved the logs on tensor board. In this case model periodically saves checkpoints to recover in case of failures. The chief worker also manages failures, ensuring fault tolerance if a worker or parameter server fails. If the master node itself fails we had to start from recent checkpoint.

Another experimental setup involved using TensorFlow on Spark framework. We have used python 3.5 on each of our cluster nodes and installed TensorFlow on all of them.

We then installed TensorFlow on Spark Framework on each of these cluster nodes. As our previous setup to run the model in distributed environment involved manually configuring and managing servers. TensorFlow on spark framework saves us the time in manually specifying cluster configuration. It also provides APIs for feeding data from Spark RDDs to TensorFlow Programs. In this environment we used TFCluster object of TensorFlow on Spark framework to start the cluster and train the model. We used TensorFlow input mode while starting the cluster as it is more efficient in case of multi-threaded input queue from a distributed filesystem, such as HDFS.

As our hardware resources are limited we have limited our dataset to 30K sentences in which 5k sentences are used for validation and another 5k sentences are used for Testing purposes. Our model is feasible to run on huge dataset. We suggest using Amazon EC2 clusters in order to train the same model on huge datasets.

4.2 Performance metrics and Experimental Results

Our performance metrics to compare the model is time. As we performed the model training in the same dataset it yielded us same accuracy. Hence accuracy could not be considered as performance metrics.

We observed that for the same data set and for the same TensorFlow model the model

executed on stand-alone cluster took 9 seconds to train. Whereas the same model and for the same data set executed in distributed environment using Spark streaming took only 2ms yielding the same model accuracy as standalone model.

Below is the tabular representation of the experimental results.

Model	Time
Stand-alone TensorFlow model	9 seconds
Distributed TensorFlow model	2 milliseconds

As we can see the distributed TensorFlow model converges much faster than the stand-alone model.

5] Discussion

5.1 Comparison between our model and standalone TensorFlow model

Our model achieved a high speed-up compared with the stand alone TensorFlow model. Using TensorFlow on Deep spark helps in alleviating communication overhead. Communication overhead plays a crucial role in the slowdown of models. The time our model spent to aggregate, and broadcast was in milliseconds compared to the time stand-alone TensorFlow model spent.

5.2 Speed-up Analysis

For the dataset that is too large to hold in memory, the distributed environment was helpful in reducing the disk-operation load. In the distributed system, Data Ingestion functionality of TensorFlow on Spark mitigates the time taken in loading and transferring the data. Thus the speed-up for the distributed TensorFlow model was much better than the standalone TensorFlow model.

5.3 Limitations:

The identified limitations so far are that setting up environment for execution of the proposed model. Due to lack of sufficient resources, we could only test for sample data and arrived at the conclusions.

6] Conclusion

We built a TensorFlow model and executed it in standalone mode and distributed mode with TensorFlow on spark framework. Our results conclude that the distributed tensor flow model with spark framework performs faster than the standalone TensorFlow models. Our future works involve improving the model and conducting the similar experiments with proposed deep spark framework for TensorFlow with added asynchronous features.

7] References

[1] K. Simonyan et al. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.

[2] C. Szegedy, et al. Going deeper with convolutions. In CVPR, pages 1–9, 2015.

[3] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. arXiv preprint arXiv:1404.5997, 2014.

[4] M. Li, et al. Scaling distributed machine learning with the parameter server. In OSDI, pages 583–598, 2014.

[5] J. Dean, et al. Large scale distributed deep networks. In NIPS, 2012.

[6] Q. Ho, et al. More effective distributed ml via a stale synchronous parallel parameter server. In
NIPS, pages 1223–1231, 2013.

[7] B. C. Ooi, et al. Singa: A distributed deep learning platform. In Proceedings of the ACM International Conference on Multimedia, pages 685–688. ACM, 2015.

[8] P. Moritz, et al. Sparknet: Training deep networks in spark. arXiv preprint arXiv:1511.06051,
2015.

[9] M. Zaharia, et al. Spark: cluster computing with working sets. In Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, volume 10, page 10, 2010.

[10] M. Zinkevich, et al. Parallelized stochastic gradient descent. In NIPS, pages 2595–2603, 2010.
3:993–1022, 2003.

[11] B. Recht, et al. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In
NIPS, pages 693–701, 2011.

12]https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create-deploy-python-common-steps.html

NOTE (The Screenshots of our results are attached in the Presentation slides)