System Support for Elastic ML Model Training

Chakshu Ahuja
cahuja2@wisc.edu

Sushma Kudlur Nirvanappa
kudlurnirvan@wisc.edu

Vibhor Goel
vgoel5@wisc.edu

Abstract

Efficient distributed ML training needs flexible control over its execution. A distributed environment with parallel execution and a fixed number of workers is a caveat for the efficient use of resources. Through this project, we aim at providing elastic system support for training machine learning / deep learning model in a distributed environment. To achieve this,

- We have worked on adding support to Submarine to be able to scale up and scale down the number of workers needed instead of the fixed number of workers specified by the user. Submarine currently provides the functionality to run distributed ML jobs with frameworks like Tensorflow, MXNet in an in-elastic manner.
- Bridge the Machine Learning (ML) and Deep Learning (DL) frameworks available and provide communication means to expose their application metrics to be used to decide the workers needed for the specific job.
- Explore different policies based on resource usage statistics to decide the scale up/down factor (dynamically).

Source code

The source code for the same is available at https://github.com/goel96vibhor/CS744-project

Introduction

Motivation

The current services available for distributed ML model training frameworks such as Tensorflow, Pytorch are inelastic, in a sense that they do not provide scope for re-distribution of the number of workers for jobs (which are decided by the user itself at the initialization of job). This creates several problems such as abuse of resources by the running jobs. Sometimes the users do not have a good estimate of the resources which are required for a job, and they may specify a large number of workers to be safe. It also leads to the blocking of resources, as they become unavailable for a new training job, until an existing running job releases resources required by the new job's workers. Third, this inelasticity does not provide scope for requirements based scaling (both up and down) and optimized use of resources.

Focusing on these shortfalls, we aim to expand Submarine to further provide support for changing the number of workers without affecting the actual job. As our first step, we worked on leveraging the iterative nature of ML training to check the quality of the model and resource utilization (system overhead) with a different number of scale up/down factor. This benchmarking helps us to decide on various factors required for elasticity like the number of workers that improves performance, between iterations (fixed/random steps) to call for scale up/down the workers. Our next step is directed towards creating link of communication between the ML frameworks and Submarine, for exchange of application level metrics. These metrics are then used to model the number of workers required for the job. We then integrate this worker flexibility with the existing Submarine support.
As a next step, we further explored different strategies for deciding the resources required dynamically, using approaches based on change in loss, number of iterations, and running time.

Related Work

The current state ML/DL frameworks do provide support for distributed ML but are rigid in their setup and need everything configured by the user (4), (6), (7). Submarine provides the basic boilerplate setup for all the frameworks. Hence all configurations like clusterSpec, distribution of tasks, TrainServer, TestServer are handled by Submarine.

We take ideas from a current Submarine proposal (1) to overcome the current issue of having to mention the fixed number of workers before running the ML/DL job. The idea is to be able to divide the job into smaller tasks and spawn the tasks based on the available workers rather than waiting for a fixed number of workers before beginning the job and releasing the workers unless needed. This dynamic allocation would let a worker be assigned a new task as it completes its current task. We use the system provided by **themis** which provides support over submarine to scale up and down the workers using events submission mechanism. In each event, we can basically specify a new request for a running ML job through themis, only specifying the new container configuration.

We will also be taking the ideas from other elastic frameworks that are exploiting the same aspects of adding and removing workers from the cluster. Most of these ideas are centred around approximation and staleness of gradients on the parameter server.

Integration of frameworks with Submarine : We plan to add support for the well known available frameworks for training Machine Learning or Deep Learning models and integrate with Submarine namely (a) Tensorflow (b) Caffe (c) Torch. However, for the purpose of this report and experimentation, we have currently only worked on TensorFlow framework.

Approach

We are using Themis Client which assists in bringing elasticity for ML model training in a distributed environment. Themis client acts as a wrapper on top of Hadoop-Yarn-Submarine platform. It uses FLEX module for dispatching events with the new configuration specified. The configuration specifies various scale up/down factors like the number of parameter servers and workers, docker image to run etc. Additionally, locality preferences of the workers can also be given. The module Flex encompasses many stages starting from receiving the scaling request until jobs/tasks have been started on the respective set of containers.

We worked with one TensorFlow model (Cifar10) using the Docker image(hadoopsubmarine/tf-1.8.0-cpu:0.0.1) as we mainly focused on checking the progress of the model with a different number of workers. In Hadoop-Yarn-Submarine platform, running a Tensorflow model launches Docker containers for Parameter Server and Workers (master node included). We are using the notion of Docker Containers as a means of running different jobs/tasks in a distributed environment. For accomplishing elastic model training, We started bench-marking progress of model and the system overhead (which is caused due to resource scale up/down request) for the different number of workers. We conducted experiments to observe resource utilization as we scale up the workers for training the same TensorFlow model and progressing from the previous checkpoint.

Additionally, we conducted experiments that exploit application metrics like average loss, number of iterations done to decide on the scaling factors (time to scale and how many workers to further use for model training that achieve the best performance). For the purpose of the same, we provide means for communication between the running ML job and themis application to gather the job metrics.

Experiment Setup

Hadoop Submarine: Hadoop Submarine project makes distributed deep learning/machine learning applications be easily launched, managed and monitored. It was initiated to provide the service support capabilities of deep learning algorithms for data (data acquisition, data processing, data cleaning), algorithms (interactive, visual programming and tuning), resource scheduling, algorithm model publishing, and job scheduling.

The project was designed to provide improvements such as Docker container support, container-DNS support, scheduling improvements, etc. These improvements make distributed deep learning/machine learning applications running on Apache Hadoop YARN as simple as running it locally, which can let machine-learning engineers focus on algorithms instead of worrying about the underlying infrastructure.

Installation: Submarine project has two parts,

- Submarine computation engine: It submits customized deep learning applications (like Tensorflow, Pytorch, etc.) to YARN from command line.
- Submarine ecosystem integration plugins/tools: Tools like Submarine-Zeppelin integration, Submarine-Azkaban integration and Submarine-installer are available options. We used Submarine-installer to install submarine and YARN on our distributed environment.

It is very difficult and time-consuming to properly deploy the Hadoop Submarine runtime environment. Since the distributed deep learning framework needs to run in multiple Docker containers and needs to be able to coordinate the various services running in the container, complete the services of model training and model publishing for distributed machine learning. Involving multiple system engineering problems such as DNS, Docker, Network, graphics card, operating system kernel modification, etc. The installation tool submarine installer for the Hadoop submarine runtime environment makes it easy to have Hadoop-Submarine setup in a distributed environment.

submarine-installer currently only supports operating systems based on CentOS(centos-release-7-3.1611.el7.centos.x86-64 and above). The steps that we followed to setup the cluster is as follow:

- We started with a cluster of three CentOS machines and we ensured that machines are able to communicate with each other (able to do ssh).
- Configurations: We took care of various configuration items like log directories: application logs, Zookeeper logs and Hadoop NameNode and DataNode directories etc and other Hadoop/Yarn/Docker specific configurations.
- Installations: We installed Hadoop-Yarn and Zookeeper from the source along with the installation of the required version of Java and Python.
- Submarine Installer: We used Submarine Installer to install other dependencies required to run ML jobs in Hadoop-Yarn-Submarine platform. Other dependencies include:

  - **ETCD:** It supports the overlay network through BGP routing and support tunnel mode when deployed across the equipment room.
  - **Docker:** Installs Docker, configures DOCKER-REGISTRY and takes care image management repository used to store the image files of the various ML/deep learning frameworks.
  - **Calico:** Provides network security solution for containers and virtual machines.
  - **Yarn container-executor:** Yarn and Docker Specific configurations (binaries) required to launch image, run tasks are mentioned here.

Implementation

Elasticity

Having run all the services required, namely, Hadoop (NameNode and DataNode), YARN (NodeManager and ResourceManager), JobHistoryServer, ZooKeeper, ApplicationHistoryServer, we experimented and benchmarked different metrics when elasticity is performed on running model. We use Themis client to trigger the first *launchEvent* to start the

tensorflow model (on Cifar-10 data) and it starts running. After sleep of around 2 minutes, we launch *changeEvent* to change the number of workers. We experimented with changing the worker count from 3 workers to 6, 9 and 12 workers.

To be more precise, in our case when we say experiment the change in worker count from 3 to 6, it changes the configuration from 1 Parameter Server, 1 Master, 1 Worker to 1 Parameter Server, 1 Master and 5 Workers. However, the configuration is customizable in the Themis client and we can set any number of Parameter Server or the number of workers.

Dynamic Worker Control

Building on top of the event based worker scaling provided by the Themis system, we add the functionality of dynamic worker control to scale workers up and down based on user criteria. The implementation of this requires to build a link of communication between the running containers and the Themis application. Basically, the task of Themis application is launching the user program in different containers, where each container may act as a parameter server, master or a worker. These containers run the user's program in a distributed setting and coordinate with each other for different aspects.

Now, for dynamically controlling the number of workers, we focus on using the metrics generated by these containers to track the job's progress. These metrics are stored in the logs for different containers. We currently read these aggregated logs to get the corresponding metrics. For this purpose, have added functionality to the log aggregator class LogCLI in yarn applications. Note, that this worker control is obtained through event-based mechanisms. We can, in fact, submit these events through a different running Themis process. This is possible due to that fact that containers are completely separate from yarn and Themis. Since yarn maintains a separate application id and name for each running job, we can submit these change events using the job name.

We now describe how we used the metrics for deciding how to scale workers. As a first and simple test, we use the number of iterations as our criteria to decide upon scaling. As a second criterion, we use the change in the loss for scaling our workers.

Empirical Evaluation

Timing Stages in Elasticity Process

Serving elasticity over a running ML model encompasses many stages. We have benchmarked the timing of different stages involved in the process, starting from the point of scaling request until tasks start running on the workers. We started running the model in a cluster setup and had scheduled the scaling request after 60 seconds of the model run. The figure **??** shows three major stages engaged after the request has been made:

- **Flex Down Time:** It measures the time from the point request is been made until all the running containers (workers) are brought down. This stage involves stopping the running model, saving the state of individual workers and deleting the containers. The highlight of this stage is the state save which we consider to be the **check-point time**.
- **Flex Up Time:** It measures the time required to bring up the new set of the container and initialize them to be ready to accept tasks.
- **Ready State:** This is the stage where the worker starts executing the task. The major focus here is that the containers have to read the old state and then proceed further. This is greatly taken care by the TensorFlow workflow module, we did minor changes to improve the performance at this stage by eliminating unnecessary validations.

We can observe from the figure that significant time is elapsed in Flex Down State, as all the workers have to write their state to shared hdfs storage. In this experiment, we scaled the number of workers from 3 to 6.

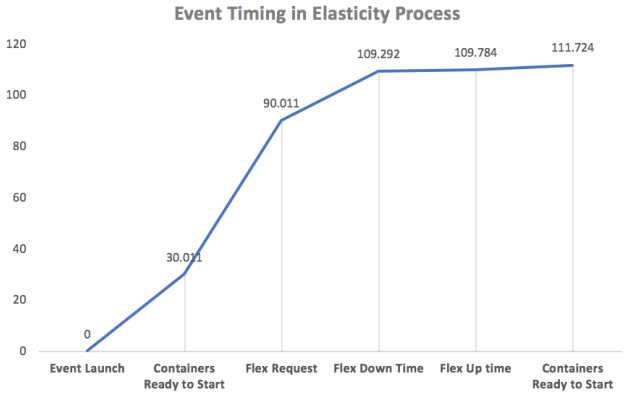

Figure 1: Event Timing in Elasticity Request Stages

Elasticity on single machine

System Analysis We exploited the notion of Docker Containers as a means of running different jobs/tasks depicting the distributed environment. Having all the workers on a single machine we tried to bench-marked the timing of stages(mentioned above) of elasticity. Starting with 3 workers we scaled to have 6 workers and then scaled from 6 to 9 workers. Between these two transitions, we tried to observe if there are any system level overhead at any stage. Figure **??** shows timing for the checkpoint, Flex uptime and total request serving period.

- **Check-pointing:** It is evident that adding more workers will increase the check-pointing time as capturing the state of workers on single shared storage has to be serialized.
- **Flex Up Time**: In our experiment, we are increasing worker from 3 to 6 and then from 6 to 9 i.e., adding 3 more workers at both the scaling phase. The Flex Up with Themis client which implements Flexing module has its current implementation to delete all the current workers and launch a new set of workers freshly. There are negligibly milliseconds of difference in launching a different number of workers as Flex up phase just includes launching workers and be ready to accept tasks which can happen in parallel.
- **Request Serving Time:** This stage measure time from the time of the request to the phase where workers start running assigned task/job. The figure shows the readings in kilo milliseconds and we can observe the slight increase in total time required to serve elasticity.

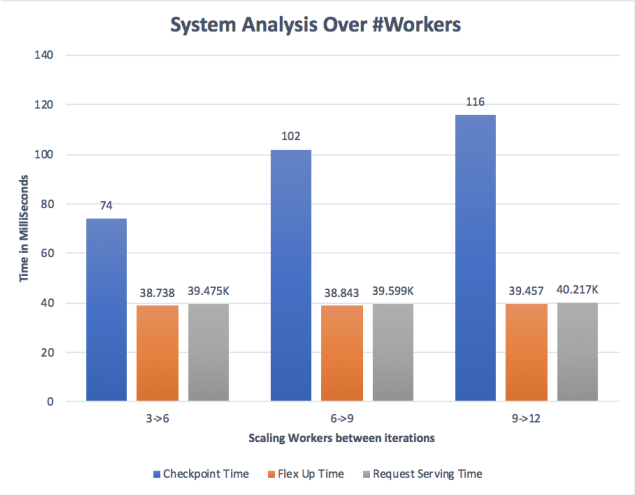

Figure 2: Event Timing in Elasticity Stages

Elasticity in Distributed Environment

In our cluster set up with 3 machines, we conducted experiments by multiplying the number of workers for every scaling period. Every scaling operation include multiple stages/events which is listed in the table 1. We measure the time consumption at each of these stages while we are varying number of workers.

Event Id	Explanation
E_1	Yarn Application submitted
E_2	Change Event from Themis Master sent
E_3	Component dispatcher receives FLEX DOWN request to scale down from $N_{initial} to 0 (bothpsandworkers)$.
E_4	*Deleted component instance* request received for every component instance currently running.
E_5	*Flex Request* now contains the new *TensorFlow* config with the desired changes in number of workers/ps. However, while creating the config, we can also set locality preferences to spawn the component instances at their orignal locations.
E_6	Component dispatcher receives FLEX UP request to scale up from 0 to $N_{new} (bothpsandworkers)$.
E_7	Task on worker re-start again after the state transitions from STARTED to READY on BECOME READY event on the component instance.

Table 1: Summary of Stages involved

,

Change in workers	Time (in milliseconds)
$3 \rightarrow 6$	119
$3 \rightarrow 9$	120
$3 \rightarrow 12$	115

Table 2: Performance analysis of check-pointing time of PS with varying change in number of workers

,

System Analysis

- **Check Pointing Time** In our case, we have assumed that the checkpointing time is the time between events E_3 and E_4. After experimenting with different worker change as in [Table **??**], we found out that the checkpointing time (in our definition) of PS in each remains almost the change. However this is expected since, in our experiment, the start point of workers is always 3 (1 PS), so checkpointing should take approximately the same amount of time irrespective of the new request yet to be made.

Change in workers	Time (in milliseconds)
$3 \rightarrow 6$	40492
$3 \rightarrow 9$	40762
$3 \rightarrow 12$	41847

Table 3: Performance analysis of total downtime time of training task with varying change in number of workers

,

- **Total Down Time** In our case, we have assumed that the total downtime time for which the training task was stalled is the time between events E_2 and E_7 till all the workers are again on the READY state. After experimenting with different worker change as in [Table 3], we found out that the total event time of workers increases as the change that needs to be brought in increases. This seems to be the situation as we imagined, since in each case, more workers need to be spawned, and we mark the end of the event till the last worker is back on the READY state.
- **Container Down Time** In addition, we tried to evaluate the downtime of the container (Time between E_3 and E_6), however, the change was negligible and around 100 milliseconds in each case.
- **Request Serving Time** In yet another experiment, we tried to find out the time it takes to serve the new request received. Serving new request is the time between E_5 and E_6. On average, it takes around 40 seconds on the 3-node cluster and constitutes a majority of the Total Downtime time of the training.

Resource Usage The most anticipated observation to make while we are exploiting elasticity in a distributed environment is resource usage. Resources like memory and network have to be efficiently utilized especially while running a distributed job. Here we tried to benchmark the variation in network and memory usage as we vary the number of workers.

Memory Usage: Figure 3 shows the peak memory usage for every set of workers in cluster. We can observe from the figure that the memory utilization on each worker increases initially up until a good number of workers was added to the cluster. The threshold is ¿9 workers after which adding any more worker has reduced the memory consumption on each worker. It is evident from this observation that the tasks have been well distributed across more workers with every worker getting less job.

Network Usage: Unlike the kind of behaviour we saw wrt memory usage, for the network we observed that it is consistently increasing as we increase the number of workers in the cluster. Figure 4 depicts the network usage in a distributed environment with a varying set of workers. This observation is expected as all workers have to communicate with one or more parameter server(s). With more number of workers, there is more demand for network bandwidth especially for iterative ML training jobs (more frequent gradients updates from workers to parameter servers). We can conclude that the network is a constraint resource in running distributed

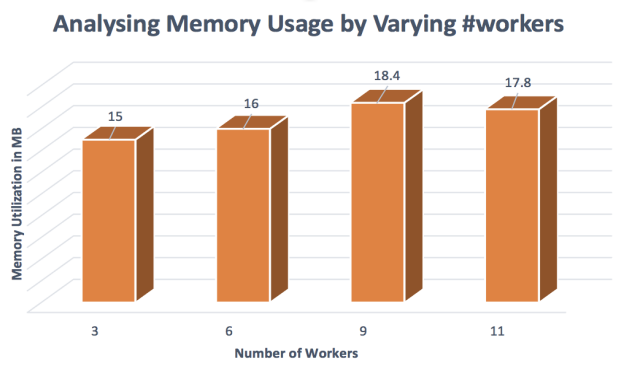

Figure 3: Analysing Memory Usage as we vary Number of Workers

ML/DL models.

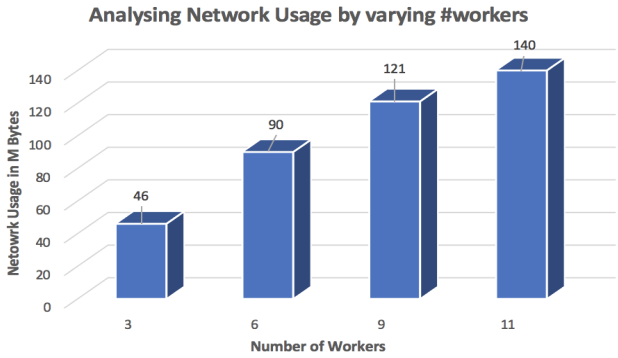

Figure 4: Analysing Network Usage as we vary Number of Workers

Scaling tests

In this section, we describe the tests which we conducted for scaling the system for different number of workers in a distributed setting based on the metrics logged in by the job. We conducted these tests for three machines with a varied number of workers.

Iteration based Scaling In our first set, we change the number of workers based on the number of iterations. We aim to analyse the impact of changing the workers too quickly and slowly on the application runtime and loss metrics. We test this using asynchronous as well as synchronous learning.

We further state here that, our algorithm waits for 1 min in between reading logs, to avoid continuous CPU wastage and locking of logs files. This is due to the fact that we are currently reading all the aggregated logs and it starts to take significant time in the later stages due to the regex match we are using. We can modify our method to read only the extra logs added within the interval period and also optimize the regex match. Further, since usually such frequent changing of workers does not happen in the usual scenario our assumption is valid.

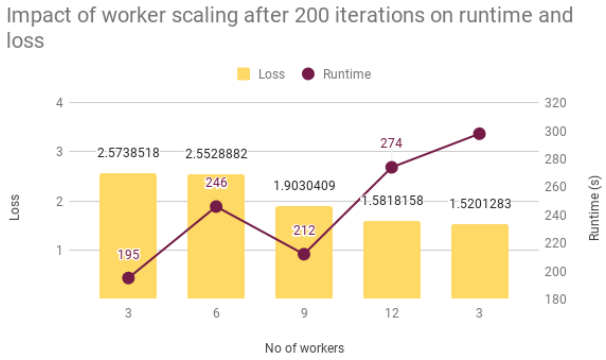

Figure 5: Impact of worker scaling after 200 iterations on runtime and loss

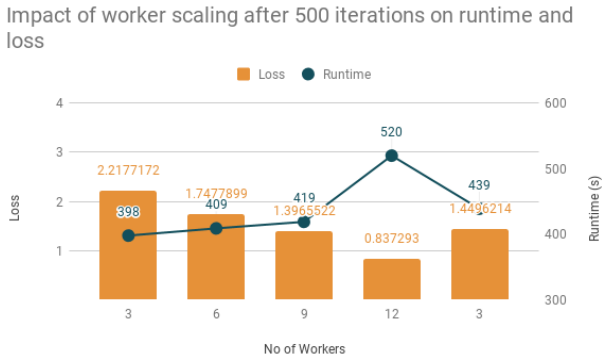

Figure 6: Impact of worker scaling after 500 iterations on runtime and loss

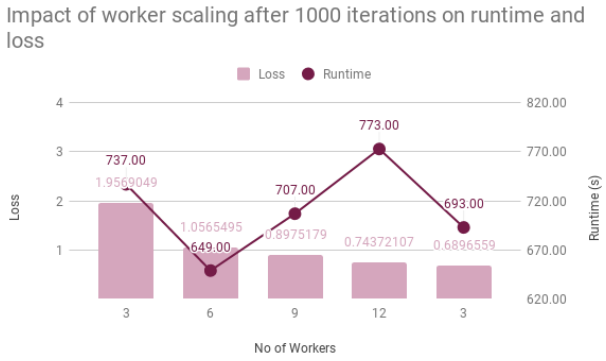

Figure 7: Impact of worker scaling after 1000 iterations on runtime and loss

The three plots [Fig. 5], [Fig. 6], [Fig. 7] show the impact of changing the number of workers after a specified number of iterations (200, 500 and 1000 iterations respectively) on loss and the runtime using synchronous training. We chose the different number of iterations to show the impact of restarting the workers from the previous checkpoint on

runtimes. In our implementation, we set the checkpointing period to be after every 10 iterations. We had further set the logging interval as well to be after every 10 iterations, so as to facilitate faster detection of iteration completion. These, in reality, depending on how frequent scaling is required and how much each iteration takes for the ML job.

From the figures, we can see that the runtime almost remains constant for the iterations when we choose the frequency of scaling (determined through the number of iteration) wisely. Further, we can see due to synchronous training, as we increase the number of workers from 3-¿6-9-¿12, we see increasingly larger drops in the loss. When we downscale the again back to 3 workers, the drop in loss reduces. Comparing for the runtime between the three figures, we see that the total runtimes increase as the number of iteration increases and similarly we see larger drops in a loss upon increasing the number of iterations.

However, a significant point to note is that if we keep very high scaling frequency, we begin to see impact of various factors such as reading aggregated metrics from files, repeat of iterations due to restoring model from old checkpoint etc.. Thus, our experiments **no longer remain controlled** and we begin to see randomness in the iteration count where rescaling occurs. Since these times now become on a significant scale as the job running time, it becomes necessary to avoid very high scaling frequency to avoid resource wastage. We observer the same thing when we tried to run our rescaling after every 50 iterations.

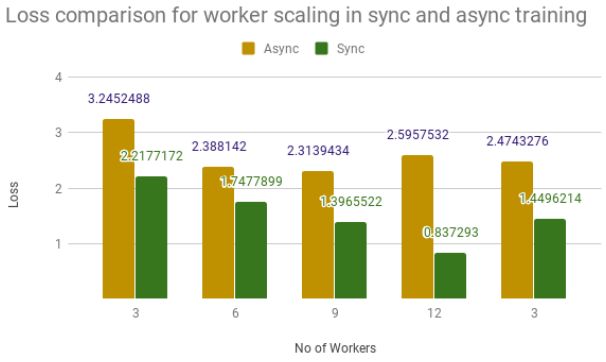

Figure 8: Loss comparison for worker scaling in synch and async training

Asynchronous vs Synchronous training comparison for scaling We further tried to analyse the impact of scaling on asynchronous training. Since in asynchronous training each worker increases its step independently, the time per step reduces inversely as the number of workers increases in asynchronous training. This also leads to smaller checkpointing and scaling interval as they are dependent on the number of steps in our algorithm. We show the comparison between the reduction in loss and runtime between asynchronous and synchronous training in figures 8 and 9 respectively. We ran this experiment scaling after 500 iterations each time.

From figure 8, we can notice that as the number of workers increases we see much larger drops in the loss in case of synchronous training. This is expected as for asynchronous training, each step has accumulated gradients from all the workers. While asynchronous training has gradients from individual workers. These individual gradients may, in fact, move the convergence to the opposite direction as we can observe in the case of 12 workers.

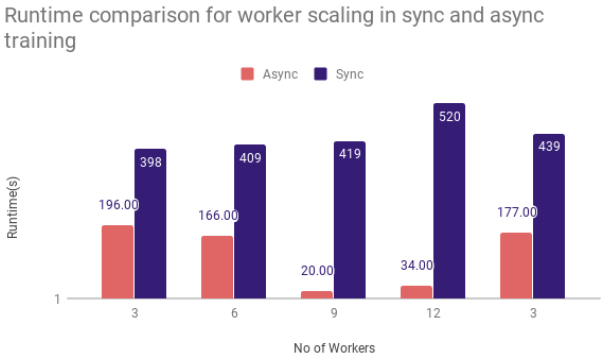

Figure 9: Runtime comparison for worker scaling in synch and async traning

The figure 9 shows the runtime comparison between asynchronous training and synchronous training with scaling after every 500 iterations. We can again see that the runtimes remain almost the same for synchronous training with a slight increase since each worker operates on complete data which is aggregated. However, for asynchronous training, the workers get to work on partial data as the number of worker increases reducing the runtime.

From the above figures, we can see the **rescaling does affect** when we try to scale to 12 workers. Here again, the iteration interval is so small that time for reading metrics and model from checkpoints start becoming significant.

Dynamic scaling based on loss We further tested our setup for dynamic scaling based on the training loss. For doing so, we check for training loss after every few intervals (500 iterations). We have tested dynamic scaling on asynchronous learning. For now, we have kept the simple criteria that if loss if decreasing, we keep on adding the workers and reduce the number of workers when we see increment in the loss.

It must be mentioned that this is a **very naive approach for scaling** and ideally there should be a curve fitting for the training loss, to see whether there was actually some benefit in adding the workers. However, since our current focus is not to devise this algorithm, we have currently not implemented this curve fitting. Moreover, this also requires knowledge of which training function the algorithm is using beforehand. Hence, for our testing, we are happy to keep only a naive method, as our aim is to only show **only the possibility of dynamic scaling**.

The figure 10 shows the number of change in the number of workers through our dynamic scaling and the training loss. As we can see, we keep on increasing the number of workers from 3 to 6,9 and then 12 till the time loss is decreasing. When we see a reduction in the loss, we reduce the worker count from 12 to 9.

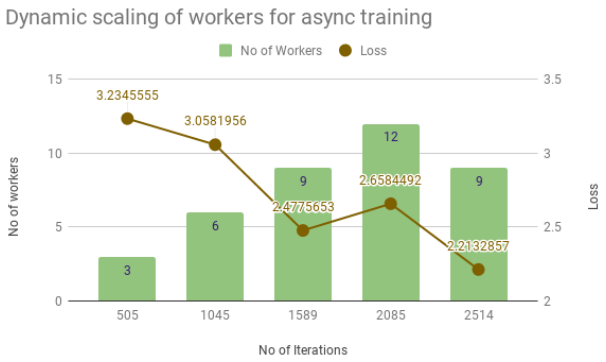

Figure 10: Dynamic scaling of workers based on loss for asynchronous training

This idea makes sense especially in the case of asynchronous training because increasing the number of workers also divides the data between them. Then a given worker may have its gradient moving in the opposite direction of convergence. We then wish to reduce the number of workers in this case, as this extra worker is **contributing negatively** to the overall learning.

Future Work

The integration of dynamic scaling for ML jobs with yarn and submarine opens a myriad of possibilities to look into for the future. Our work is a very small step in this direction. We have currently used event-based worker scaling implemented by Themis, which presently works by bringing all the workers down and back up from checkpoint. This can be definitely optimized by not bringing all the containers up and down and doing so only for the required ones.

Further, the way we have implemented reading the aggregated metrics for the running ML jobs is by reading the application logs. This takes more time as the number of logs grow and can be optimized by making inter-process communication link between the running ML job and the Themis application.

Further, we have presently implemented scaling based on loss. We can test on several metrics. Our code has provisions for the same as we can change the regex for matching the metrics to consider. However, more important criteria to consider is the extrapolation of loss and other metrics for testing whether the workers need to be scaled up or down by comparing with the current metrics and predicted metrics.

References

[1] Hadoop issue proposal
https://docs.google.com/document/d/199J4pB3blqgV9SCNvBbTqkEoQdjoyGMjESV4MktCo0k/edit#

[2] Submarine installer
https://github.com/hadoopsubmarine/hadoop-submarine-ecosystem/tree/master/submarine-installer

[3] Launch application and tensorboard
https://hadoop.apache.org/docs/r3.2.0/hadoop-yarn/hadoop-yarn-applications/hadoop-yarn-submarine/QuickStart.html

[4] Distributed Tensorflow
http://amid.fish/assets/DistributedTensorFlow-AGentleIntroduction.html

[5] Frameworks
http://www.pdl.cmu.edu/BigLearning/

[6] Setup Tensorflow on Google Cloud
https://cloud.google.com/solutions/running-distributed-tensorflow-on-compute-engine

[7] Distributed computing on Tensorflow by Databricks
https://databricks.com/tensorflow/distributed-computing-with-tensorflow

[8] Hadoop and Yarn installation from Source
https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/ClusterSetup.html