Distributed Machine Learning: Tensorflow on Docker

Xinghui Zhao, Ph.D., Aaron Goin, Ronald Cotton
Distributed Systems Research Lab, Washington State University, Vancouver Campus
{*x.zhao, aaron.goin, ronald.cotton*} *@wsu.edu*

ABSTRACT

Tensorflow is a robust machine learning API incorporating several key technologies to optimize performance, allowing for rapid deployment on their artificial neural networks.[1] Tensorflow handles many types of low to high-level Neural Networks utilizing a state data flow model thereby calculating results into multidimensional arrays known as Tensors. Distributed Tensorflow utilizes ClusterSpec which offers In-Graph Replication, Between Graph Replication, Asynchronous Training, and Synchronous distributed training.[9]

Docker[11], utilizes several resource isolation features to provide independent isolated containers to run in a single container with a minimal loss of performance. Isolation includes using the host kernel to isolate CPU and memory resources along with namespaces. Docker also provides networking communications between containers.

This paper demonstrates an implementation which improves Tensorflow performance by segmenting distributed Tensorflow and Docker on the same multicore server versus using Tensorflow natively. Open-Sourced Tensorflow as well as Google's former project DistBelief exploited model and data parallellism via Distributed training,[8] implementing Docker and ClusterSpec and further advantage for non-GPU learning clusters.[6] This Docker Tensorflow technique displays a measured improvement versus native Tensorflow and differs from other methods of distributing like Intel's YARN[22], Yahoo's Spark[21], TensorFrames[7] and Tensorflow on Kubernetes.[17] A similiar paper, *Performance Evaluation of Deep Learning Tools in Docker Containers* confirms our positive findings.[20]

General Terms

Distributed Deep Learning

Keywords

TensorFlow; Docker; DNN; CNN; Optimization

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2018 DSR Labs.

1. INTRODUCTION

Two programs, docker-cpu-affinity and docker-multiple were created based upon the Distributed Tensorflow[3] between-graph replication mnist_replica.py[4]. Each Worker instance duplicates Tensorflow's dataflow model. A Parameter Server (PS) stores the required values for the model. Worker 0 referred to as the chief worker, coordinates communications and monitors faults between all available workers. Distributed Tensorflow maintains a fault tolerant system by building checkpoints during training and mitigates data between all currently available workers. Finally, the chief worker averages the all the gradients provided by the replica workers, thereby making the model asynchronous.

SysBench 0.4.12[14] was used for benchmarking of the systems because of it's ease of use, portability between operating systems, the benchmarks are based on parallel processes (aka threads) and didn't require SQL dependencies for measuring IO. This data was used to contrast and compare native host machine operation, docker container operation, and a laptop's operation versus a server cluster.

The dataset used was the MNIST dataset[15], the model was a simple Dense Neural Network model using 784 inputs, 100 hidden nodes, and 10 outputs. Optimization was based on AdamOptimizer, data was collected for every 10 local steps. Total data analyzed was 10,000 global steps.

2. SYSTEM ARCHITECTURE

Testing was conducted on two platforms. The first platform was an laptop was configured with an *Intel i7-6700HQ* processor *(4 cores)* with *16GB DDR4-2133 RAM* with CAS Latency of *15-15-15-36-2T*. This machine's hard drive was a *HGST Travelstar 7K1000 2.5 1TB 7200 RPM SATA-III* with *32MB cache* and *4K sector size.*

The second testing platform was two cluster computers on WSU Vancouver's Network. Both mx01 and mx02 are *dual Intel(R) Xeon(R) CPU E5-2650 v4* processors *(2x12 cores)* socketed onto a *Supermicro X10DRW-iT* motherboard with *512 DDR4* installed.

Both Intel architectures offer Hyper-Threading thereby increasing the number of independent instructions in the pipeline allowing concurrent scheduling of two processes per core.[12]

3. SOFTWARE INSTALLED

During testing, the laptop's was setup with Ubuntu 18.04 LTS bionic with Docker version 18.06.0-ce-dev build 3f80548. At the time of testing, no official build of Docker was made for Ubuntu 18.04 LTS, a nightly build was installed with the noted version above. The clusters installed OS was Ubuntu 16.04.4 LTS Xenial with the recommended build Docker version 17.05.0-ce.

3.1 Benchmarking Methodology

SysBench 0.4.12[14] is cross-platform and multi-threaded benchmark tool for evaluating system performance written in C. Used in unison with a python script called bench.py, it automated benchmarking performance of the CPU, Memory, Threads and File IO between the two architectures.

A series of ten tests for each type of benchmark resulted in individual text files in a `/tmp` directory. These results were then averaged. Initial tests for File IO were disregarded as files are generated for future tests at this time. Further tests execute random seeks over the generated data files.

While the File IO test should not be disregarded, File IO tests should not be considered infallible as the OS may have cached results into secondary memory.

Sysbench CPU Test	Laptop Average 8 threads	Laptop Average 8 threads	mx01 Average 48 threads	mx02 Average 48 threads
	Native	Docker	Docker	Docker
total time (in secs)	3.390	3.433	0.843	0.833
total time taken	27.106	27.448	38.658	38.357
Per-Request Statistics:				
min (in ms)	2.501	2.526	2.995	2.990
ave (in ms)	2.711	2.743	3.864	3.836
max (in ms)	10.095	15.449	49.359	46.537
95th percentile (in ms)	2.730	2.731	4.633	4.664
Threads Fairness:				
events 1250 (stddev)	3.302	26.191	13.445	11.291
execution time (avg)	3.388	3.431	0.809	0.799
execution time (stddev)	0.000	0.000	0.036	0.034

Figure 1: Sysbench CPU Benchmark

Sysbench Memory Test	Laptop Average 8 threads	Laptop Average 8 threads	mx01 Average 48 threads	mx02 Average 48 threads
	Native	Docker	Docker	Docker
ops/sec	4231050.885(	4394536.782(	1810474.352(	1761360.586(
MB/sec	4131.8850	4291.5400	1768.0410	1720.0780
total time (in sec)	2.4784	2.3862	5.8113	5.9562
total time taken	9.9512	9.7534	32.3434	26.2404
Per-Request Statistics:				
max (in ms)	4.1360	7.9430	1.5500	1.2200
Threads Fairness:				
events (stddev)	6517.8450	12136.5520	1004.8400	1128.5630
execution time (avg)	1.2439	1.2192	0.6738	0.5467
execution time (stddev)	0.0020	0.0070	0.0020	0.0000

Figure 2: Sysbench Memory Benchmark

Comparing docker and the native laptops tests, the data demonstrates CPU and memory is only slightly affected by Docker's containerization. Most negative effects on containerization is with Threading and File IO. The benchmark File IO files accessed directly from a docker volume, which

Sysbench Threads Test	Laptop Average 8 threads	Laptop Average 8 threads	mx01 Average 48 threads	mx02 Average 48 threads
	Native	Docker	Docker	Docker
total time (in sec)	2.8328	3.4705	4.9056	4.6516
total time taken by event execution	359.7356	440.7246	621.4726	589.2125
Per-Request Statistics:				
ave (in ms)	35.9740	44.0720	62.1500	58.9230
max (in ms)	227.8380	248.6270	878.5180	871.4820
95th percentile (in ms)	90.3020	104.1290	319.3510	303.2620
Threads Fairness:				
events (stddev)	6.1680	5.5900	16.7070	16.8980
execution time (avg)	2.8105	3.4432	4.8554	4.6032

Figure 3: Sysbench Threads Benchmark

Sysbench File IO Test	Laptop Average 8 threads	Laptop Average 8 threads	mx01 Average 48 threads	mx02 Average 48 threads
	Native	Docker	Docker	Docker
Mb/sec	4.5291	4.0778	379.7189	379.7189
Req/sec	289.8633	260.9756	24302.0467	24302.0467
total time (in sec)	34.9741	39.5905	0.4515	0.4515
total # of events	10107.7778	10063.6667	10971.3333	10971.3333
total time (event)	13.2057	41.8168	2.3459	2.3459
Per-Request Statistics:				
avg (in ms)	1.3078	4.1644	0.2144	0.2144
max (in ms)	123.0689	182.5878	8.3444	8.3444
95th percentile (in ms)	3.9878	27.9500	0.9978	0.9978
Threads Fairness:				
events (ave)	1263.4722	1257.9583	228.5694	228.5694
events (stddev)	106.1610	111.3600	21.0222	21.0222
exec time (avg)	1.6507	5.2271	0.0446	0.0446
exec time (stddev)	0.2000	0.2178	0.0100	0.0100

Figure 4: Sysbench File IO Benchmark

reduced IO access dramatically. File IO may be improved by storing that data on non-volumes.[13] Unexpectedly, the standard deviation increases for CPU and Memory tests, yet appear unchanged for Threads and File IO tests.

While mx01 and mx02 share the same architecture, benchmarks were generated to contrast differences in CPU tolerances, thermal throttling, and system configuration. Differences between these two sets of data are slight, as expected.

4. DOCKER TESTS

Figure 5 through *figure 7* illustrates differing Tensorflow tests as mentioned below while *Figure 8* through *Figure 10* demonstrates the time improvement over these tests. All machine learning models share the same model as mentioned in the *Introduction*. **Docker Single** executes a non-distributed Tensorflow model without Docker network and is treated as the baseline case. In contrast, **Docker Multiple** and **Docker CPU Affinity** both employ ClusterSpec Distribution and the Docker network on an individual host. While average time delta increased from single versus distributed models, time delta is measuring *time per worker* producing an overall improved *total time run*. Also note that worker 0 is utilizing `tf.train.SyncReplicasOptimizer`[5] which averages the gradients to various workers, producing decreased accuracy and increased loss in the model.

4.1 Docker Single

This executes a simple non-distributed Tensorflow instance within a docker container and its primary purpose for comparing this *baseline* versus the two methods below implementing distributed Tensorflow. Three executions of Docker Single were averaged producing a baseline for each Architecture.

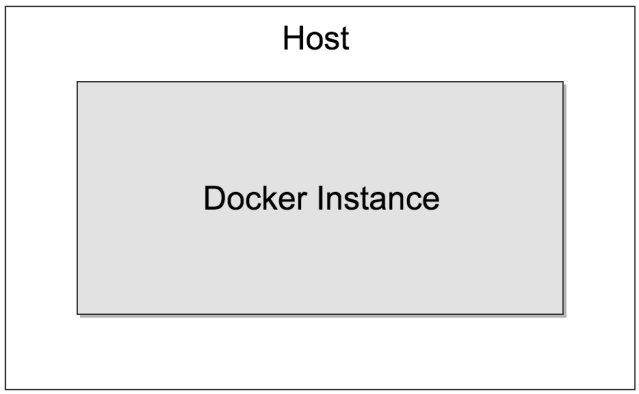

Figure 5: Single non-distributed docker instance; *baseline testcase*

4.2 Docker Multiple

Docker multiple encapsulates a docker container for every instance of a worker and parameter server. Each instance provides a minimal Ubuntu enviornment, necessary libraries, and the executed the distributed test. The test suite nested loops over 1 to 3 parameter servers, and 1 to 24 workers. The suite was implemented to find the optimal number of workers and parameter servers based on current running model and archetecture. This optimization was assessed by comparing runtime, accuracy and time delta between each run.

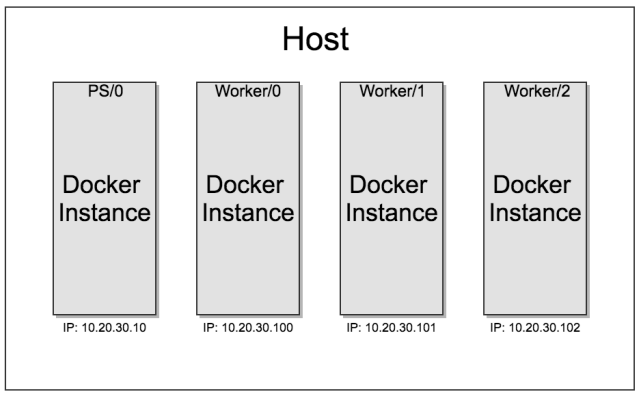

Figure 6: Multiple docker instances; unique IP addresses; *thread overhead*

While effectively simulating a distributed network, the number of workers plus the number of parameter servers increases the number of threads on the host machine linearly. This overhead quickly increases to thousands of threads as multiple operating systems exist for this architecture.

4.3 Docker CPU Affinity

To reduce the overhead of multiple container instances, we utilized one container instance instead implementing *CPU Affinity* to distribute the workload amongst multiple cores. The role of chief is crucial to a ClusterSpec distributed model, therefore, any remaining cores were devoted to Worker 0. When compared to **Docker Multiple**, the number of threads in operation no longer produces linear growth for the host, ultimately reducing the total run time between tests.

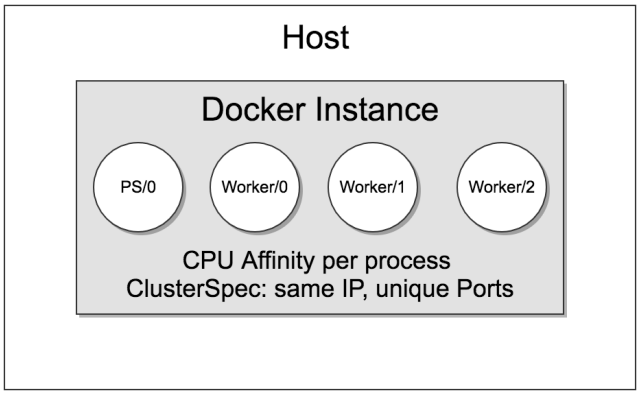

Figure 7: Single docker instance executing multiple processes; *split over multiple cores*

	Average Run Time	MAX Accuracy	STDEV Accuracy	Average Time Delta
Laptop	00:22.52	0.976267	0.009828	0.022544
mx01	00:46.50	0.977533	0.009653	0.046549
mx02	00:46.74	0.977867	0.010091	0.046791

Figure 8: Docker Single; *baseline execution*

		PS	Workers	Total Run Time	Average Time Delta (per Worker)*	MAX Accuracy
Best Total Run Times	**Laptop**	1	12	00:17.12	0.19998	0.94280
	mx01	1	7	00:10.11	0.06857	0.96840
	mx02	1	7	00:09.69	0.06628	0.96520
Worst Total Run Times	**Laptop**	3	1	02:58.17	0.17817	0.97460
	mx01	3	22	01:36.00	2.10633	0.84100
	mx02	3	24	01:37.39	2.28055	0.88100

Figure 9: Docker Multiple; *Selection of best & worst total run times*

		PS	Workers	Total Run Time	Average Time Delta (per Worker)*	MAX Accuracy
Best Total Run Times	**Laptop**	1	7	00:18.46	0.12735	0.97140
	mx01	1	7	00:07.07	0.04907	0.91480
	mx02	1	7	00:07.17	0.04973	0.91200
Worst Total Run Times	**Laptop**	3	1	00:54.29	0.05429	0.97460
	mx01	2	1	01:28.50	0.08850	0.97460
	mx02	2	1	01:28.88	0.08888	0.97460

Figure 10: Docker CPU Affinity; *Selection of best & worst total run times*

4.4 Analysis of Docker Results

Both **Docker Multiple** and **Docker CPU Affinity** offers a large performance boost in comparison to **Docker Single.** Container Instances improve in comparison to non-distributed Tensorflow, as Tensorflow primarily requires Processing and Memory resources. *Average Time Delta* increases dramatically for the distributed model. This is possibly caused from from networking overhead and/or the async nature of the model.

Surprisingly, the *Average Run Time* for **Docker Single** for the server mx01 and mx02 lacks performance versus the laptop. When referring to Passmark benchmarks, the single threaded rating of 1801 for the *i7-6700HQ*[18] provides an edge over the *E5-2650 v4* rating of 1565.[19] Single threaded performance may be the cause of this discrepancy.

Depending on the test, we can see a marginal improvement with **Docker CPU Affinity** versus **Docker Multiple.** Server mx01 and mx02 improved for fastest *Total Run Time*, while the laptop's largest improvement was *Total Run Times* over all tests. Percent decrease for the top three times when comparing both distributed tests versus the baseline test was between 18% and 24% for the Laptop benchmarks and between 78% and 84% for mx01 & mx02.

4.5 Additional Parameter Servers

Tests above demonstrate a negative impact on using multiple instances of Parameter Servers for either **Docker Multiple** or **Docker CPU Affinity**. The degraded performance can be attributed to two possible reasons. Either the model or MNIST dataset density was minimal, unable to leverage the additional Parameter Server(s). The second possible cause is network throughput found on one parameter server caused no bottleneck to the number of workers tested, thereby needlessly separating parameters into two containers. Intel/NERSC Big Data Center along with researchers implemented a 512-core Cori Supercomputer utilizing a ResNet-50 model containing roughly 25.5 million parameters and demonstrated improvements when implementing multiple Parameter Servers.[16]

5. ADDITIONAL DEEP LEARNING APPLICATIONS

Docker-build is an automated Docker Container which builds an optimized python wheel by installing all requirements for Bazel. Docker-build opens the required ports for Tensorboard as well as providing an Anaconda environment for research. It allows for making a Python 2 or Python 3 specific environment, as well as building specific versions of Tensorflow.

A simple hyper-parameter optimizer known as opt-layers.py which ran multiple occurrences of dnn-json.py. This processed a DNN via Tensorflow and a higher-level library known as TFLearn. This reduced execution times by setting CPU Affinity to individual tests and became the basis for *Docker CPU Affinity.*

KerasSingle.py was an application began as a Tensorflow /Keras Hybrid to interface with Aaron Goin's Tensorflow.js application, *WebNN.*

6. CONCLUSIONS

When executing deep learning models on CPU only systems, optimizing the computation by any means necessary is a requirement - because deep learning is a time intensive process. Utilizing Docker with Distributed Tensorflow to decrease time intensive problems over multiple CPU cores is beneficial, allowing further research into deep learning models. Docker containerization has the added advantages of creating a reproducible deep learning research environment that's isolated from the host operating system.

7. FURTHER RESEARCH

Most deep learning research is gravitating towards GPU and TPU optimization for deep learning as the architecture for these processors are suited parallelization. Hyperparameter Optimization dramatically reduces learning time over a number of global steps. Model optimization via AutoML Bayesian Optimization reduced future processing time by building an optimal model for training.[10] Further research into Distributed ClusterSpec Tensorflow which allows mixed architecture of GPU/CPU/TPU assets combined with virtualization methods *(such as nvidia-docker)* can decreasing overall processing time.[20] Further research into *NVIDIA Container Runtime for Docker* and optimizing containers before deployment would be advantageous.[2]

8. REPOSITORY

All the research is available on GitHub and publicly on WSU Vancouver's GitLab.

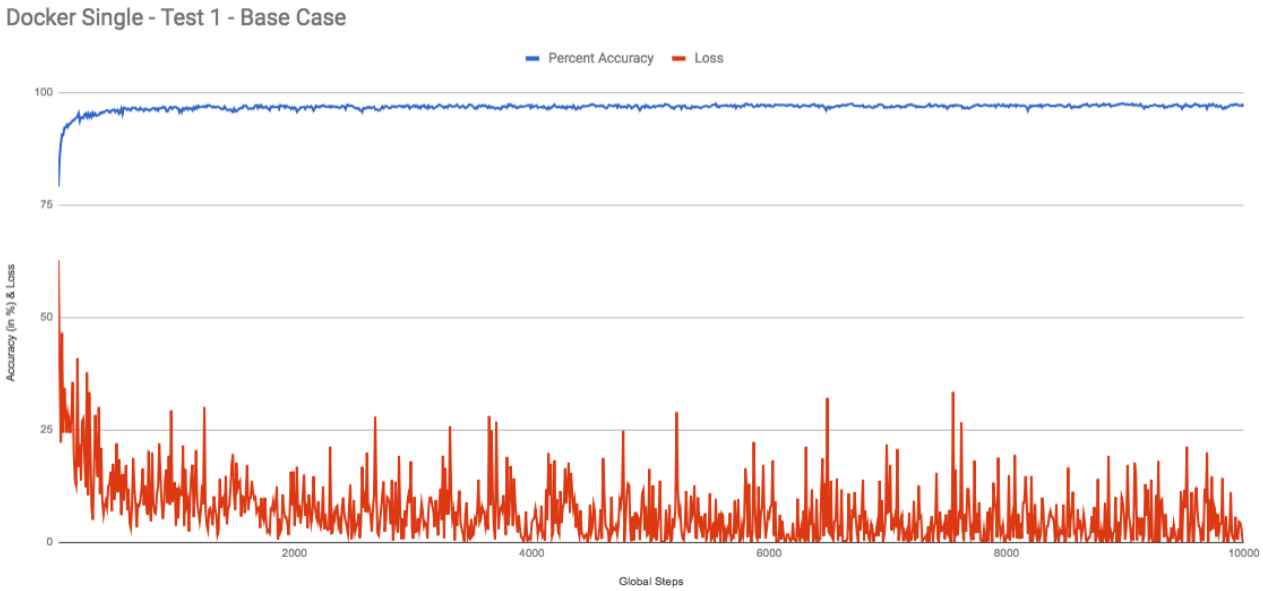

Figure 11: Typical Tensorflow Model running in CPU mode

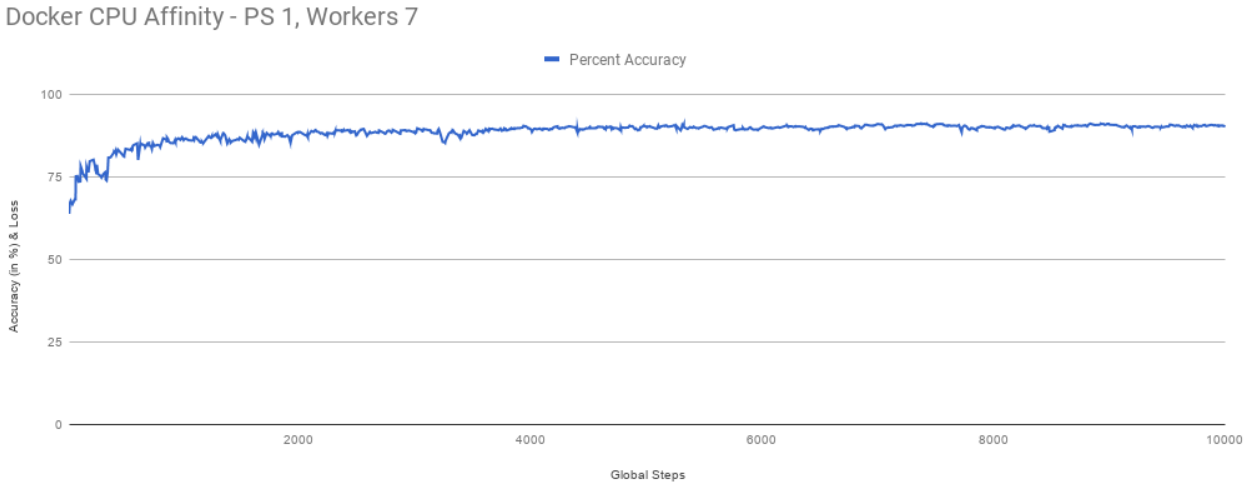

Figure 12: Distributed Tensorflow - CPU Affinity (Loss not graphed - value before average)

9. REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265 – 283, 2016.

[2] C. Arango and R. D. J. Sanabria. Performance evaluation of container-based virtualization for high performance computing environments. In *Unknown*, 2017.

[3] T. T. Authors. Distributed tensorflow. https://www.tensorflow.org/deploy/distributed. Accessed: 2018-05-26.

[4] T. T. Authors. mnist_replica.py. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/tools/dist_test/python/mnist_replica.py. Accessed: 2018-05-26.

[5] T. T. Authors. tf.train.syncreplicasoptimizer. https://www.tensorflow.org/api_docs/python/tf/train/SyncReplicasOptimizer. Accessed: 2018-05-27.

[6] B. Burns and D. Oppenheimer. Design patterns for container-based distributed systems. In *The 8th Usenix Workshop on Hot Topics in Cloud Computing (HotCloud '16)*, 2016.

[7] Databricks.com. Github:tensorframes. https://github.com/databricks/tensorframes. Accessed: 2018-05-26.

[8] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS (2012)*, pages 1–9, 2012.

[9] J. Dowling. Distributed tensorflow. https://www.oreilly.com/ideas/distributed-tensorflow. Accessed: 2018-05-27, Written: 2017-12-19.

[10] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter. Methods for improving bayesian optimization for automl. In *JMLR: Workshop and Conference Proceedings 2015*, 2015.

[11] W. Foundation. Docker (software). https://en.wikipedia.org/wiki/Docker_(software). Accessed: 2018-05-27.

[12] W. Foundation. Hyper-threading. https://en.wikipedia.org/wiki/Hyper-threading. Accessed: 2018-05-27.

[13] D. Inc. Runtime metrics. https://docs.docker.com/config/containers/runmetrics/. Accessed: 2018-05-26.

[14] A. Kopytov. Sysbench manual. http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf. Accessed: 2018-05-26.

[15] Y. LeCun and C. Cortes. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/, 2010.

[16] A. Mathuriya, T. Kurth, V. Rane, M. Mustafa, L. Shao, D. Bard, Prabhat, and V. W. Lee. Scaling grpc tensorflow on up to 512 nodes of cori supercomputer. In *NIPS 2017 Workshop: Deep Learning At Supercomputer Scale*, pages 1–7, 2017.

[17] D. Meng. Kubernetes: Distributed deep learning on heterogeneous gpu clusters. https://thenewstack.io/kubernetes-distributed-deep-learning-on-heterogeneous-gpu-clusters/. Accessed: 2018-05-26.

[18] PassMark. Intel core i7-6700hq @ 2.60ghz. https://www.cpubenchmark.net/cpu.php?cpu=Intel+Core+i7-6700HQ+%40+2.60GHz&id=2586. Accessed: 2018-05-27.

[19] PassMark. Intel xeon e5-2650 v4 @ 2.20ghz. https://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+E5-2650+v4+%40+2.20GHz&id=2797. Accessed: 2018-05-27.

[20] P. Xu, S. Shi, and X. Chu. Performance evaluation of deep learning tools in docker containers. In *Unknown*, 2017.

[21] L. W. Yang. Github:yahoo. https://github.com/yahoo/TensorFlowOnSpark. Accessed: 2018-05-26.

[22] M. Zhang. Github:intel-bigdata. https://github.com/Intel-bigdata/TensorFlowOnYARN. Accessed: 2018-05-26.