Scaling Deep Learning Without Increasing Batchsize

Alexander Heye
Data Analytics and Machine Learning
Cray, Inc
Seattle, Wa, USA
aheye@cray.com

***Abstract*—Deep learning has proven itself to be a difficult problem in the HPC space. Though the algorithm can scale very efficiently with a sufficiently large batchsize, the efficacy of training tends to decreases as the batchsize grows. Scaling the training of a single model may be effective in narrow fields such as image classification, but more generalizable options can be achieved when considering alternate methods of parallelism and the larger workflow surrounding neural network training. Hyperparameter optimization, dataset segmentation, hierarchical fine-tuning and model parallelism can all provide significant scaling capacity without increasing batchsize and can be paired with a traditional, single-model scaling approach for a multiplicative scaling improvement. This paper intends to further define and examine these scaling techniques in how they perform individually and how combining them can provide significant improvements in overall training times.**

***Keywords*-Neural Networks; Deep Learning; High Performance Computing; Data Analytics**

I. Introduction

Deep learning has transformed the approach to tackling some of the most difficult problems of the modern age. This has been focused largely in the realms of computer vision, automatic speech recognition and natural language processing, however this has been expanding rapidly to other fields traditionally dominated by statistical and simulation based approaches.

As very large datasets grow and become more accessible, deeper and more complex neural network models will be used to gain further insights and drive large gains in predictive power. Computational resources become a limiting factor in being able to fully explore the possible applications of neural networks. The ability to utilize HPC systems can be a solution, but for most modern techniques for scaling to this level, the requirements on network topology and batchsize will prevent applicability to a more broad range of problems.

Most efforts to bring Deep Learning to HPC scale have focused on the training of a single neural network on a large system[1]. There is a lot of value in this, however in day-to-day development, more can be gained by combining alternate approaches and focusing on scaling portions of the workflow. By doing this a multiplicative increase in parallel computation can be achieved, largely without any concessions on network topology or batchsize. In this paper I intend to dive deeper on some of these techniques and illustrate the benefits through an example deep-learning application.

II. HPC and Neural Networks

The best-known approaches to scaling neural network training can be broken down into two ways of distributing the training process. Either the dataset can be split among compute units batch by batch or the model topology itself can be stored and accessed across compute units. These are referred to as data parallel and model parallel training respectively[2]. These techniques tend to be used for different purposes as that each has its own limitations and strengths which will be detailed in the following sections.

A. Data Parallelism

Data parallelism is by far the most common approach used when scaling neural network training. It relies on replicated model parameters being deployed on each compute node, or worker, and each of these workers running the forward and backward pass on different parts its own minibatch in parallel. The gradients are then aggregated and applied at some cadence after each worker completes it's gradient calculation. Depending on the method chosen, the master set of parameters may be stored on one or more parameter servers.

Generally, this gradient update is done synchronously. This means that all workers must complete a forward pass, backward pass and gradient update with the aggregated parameters before a new batch is computed. This avoids gradient calculations computed based on stale weights which can hinder the overall training process. Here, stale weights refers to a set of model parameters that are no longer in sync with the master set.

This approach can, however, lead to poor utilization of resources as the fastest worker must wait for the slowest worker to complete it's gradient update before moving forward. Asynchronous updates can increase resource utilization, however the likelihood of gradients calculated on stale weights increases with increased asynchronicity, therefore potentially slowing the converge rate of the final accuracy.

Data parallel training serves to scale the overall throughput (individual samples per second) very effectively, however, with neural network training, an increase in throughput

does not guarantee faster training. Counterintuitively, when training to a threshold loss rather than a set amount of epochs or updates, an increase in throughput through data parallelism can correlate to longer training times if executed improperly. This is caused by the scaling of global batchsize with the number of workers, decreasing the convergence rate as each training step utilizes more data but shifts the weights by the same amount.

Increasing the step size, or learning rate, may appear to be a simple solution, as that more data per batch should allow for more confidence in updating the weights. This is rarely the case however. With a large batchsize, the training process becomes less stochastic and therefore the ability to explore the error surface for a deeper local error minimun begins to diminish. This decreases the ability of the network to generalize to problems outside of the training dataset and thus validation testing results will suffer.

B. Model Parallelism

Model parallel training refers to dividing the neural network model in some way and locating each part on different workers. There are many ways to do this, and how to go about it depends on the end goal and the specific network topology. If memory is a limiting factor for the size of the neural network, the network parameters can be divided among two or more workers. In this case, splitting the network vertically by layers could suffice, and is generally a popular tactic as it is intuitive and simple to implement with no implications on model performance. The downside is that in order to train synchronously, many of the workers will sit idle much of the time while waiting for results from lower layers.

In order to leverage model parallelism for an improvement in training time, the network can be split horizontally rather than vertically. In order to do this, the network would need to be designed to allow layers to split into multiple concurrent flows of execution, an example of which is provided in Figure 1. In this case, execution for specified layers will be executed on multiple workers decreasing the walltime of each forward and backward pass. This does limit communication between layers, and doing so regularity will come at a cost to the scaling efficiency.

This technique has the benefit of not affecting the batchsize, however load-balancing issues and increased communication costs limit the applicability. For large parallel systems, this technique limits the amount of scaling by the number of ways in which the model can be divided. As that most networks are well below 100 layers in size, and splitting the network horizontally too many times can have detrimental effects on performance, generally each instance of model-parallel training can only scaling to a dozen or so nodes at most.

III. Workflow scaling

In order to optimize the scalability of deep learning, the overall workflow should be taken into consideration. Though distributing the training of a single network is very useful, it is rarely the largest contributor to the overall time investment for a developer. Datasets are regularly altered, varying subsets are evaluated separately and many individual models are trained to tune to the ideal configuration. By planning ahead and parallelizing as much of the workflow as possible, the overall time from initial investigation to an optimized, well-tuned model can be brought down dramatically with the aid of large HPC systems.

A. Hyperparameter Optimization

Designing a neural network leaves the developer or data scientist with a lot of decisions when constructing the network architecture and setting up the training process. Many of these decisions come down to values known as hyperparameters, an example set can be seen in Table I. Hyperparameters can define the network itself, i.e number of layers, activations functions or convolutional filter size, as well as define the training process, i.e. learning rate, weight decay or loss function. As the network and training options grow in size, the space of possible hyperparameter combinations grows exponentially. In order to explore enough of this space to optimize the model, a form of hyperparameter optimization (HPO) must be used.

HPO can come in many forms, from a naive hyperparameter sweep to a more directed optimization technique. A properly executed HPO process has the ability to produce large gains in accuracy as that an improperly tuned network can have a significant detrimental effect on its efficacy. Because of this, much of the time spent in developing a deep learning application will be spent here. Since this involves not only training a single model, but hundreds or thousands, an optimized, scaled approach to HPO can provide more benefit than even a well implemented large scale data parallel training process.

Simple techniques for hyperparameter sweeps include random and grid search. In a grid search approach, a matrix of possible hyperparameter combinations are selected where the developer will select a few options for each and run all the possibilities. With the example set of hyperparameter in Figure I, assuming only 3 options per this would still require the training of 3^{10} networks, or just shy of 60,000 training runs to explore the entire space. Increase this to 5 per and that number is up to 9.8M training runs. Obviously, this is not a reasonable expectation and chances are a sub-optimal network will be chosen as compute resources limit the amount of this space that can be explored. Random sweeps take a similar approach, but rather than defining a matrix of possibilities ahead of time, hyperparameters are chosen at random for each run. This may increase the chances of stumbling upon a good configuration, but again a lot of

Table I
DEFAULT HYPERPARAMETERS

Parameter	Default	Range
Optimizer	Adam	Adam, Ada, SGD, Mom.
Activation Function	ReLU	ReLU, ELU, tanh, sigmoid
Batchsize (BS)	100	Held Constant
Dropout Rate	0.4	[0.1, 0.9]
Learning Rate	1e-4	[1e-1, 1e-6]
Conv1 Filter Count	32	[2, 240]
Conv1 Filter Size	5	[1, 10]
Conv2 Filter Counts	256	[1, 256]
Conv2 Filter Sizes	5	[1, 10]
Fully Conn. Sizes	2048	[1, 2048]

compute would be wasted on very similar configurations that are known to be low quality.

Other search techniques use optimization algorithms to direct the selection of subsequent configuration choices, therefore accelerating the search for an ideal architecture and training approach. Bayesian techniques, such as those applied by the tool Spearmint[3], utilize bayesian modeling in the selection of future generations of models. Evolutionary techniques apply genetic algorithms to achieve a similar result. These algorithms are iterative in nature, requiring results from one generation to aid in the creation of a new generation, therefore they cannot reach the same level of parallelism as the random and grid sweep approaches, however the total amount of resources monopolized by the HPO process will decrease as that a desirable configuration will be achieved more rapidly by applying these guided techniques.

B. Ensemble Training

Ensemble neural networks provide further confidence in the predictions made by neural networks. An ensemble of neural networks will be trained independently while varying hyperparameters, dataset, and/or initializations. The final prediction is determined by a method of ensemble averaging where each network may be assigned a weight toward the final solution based on performance. When making a prediction from such a system, not only will a prediction be made, but a higher level of confidence can be prescribed by analyzing the confidence of each individual network and the proportion of networks that made a similar prediction. It is common for an ensemble system to perform better on average than any individual model would perform on its own.

Because multiple networks will be training entirely independently with only minimal communication cost at the end, this method can employed to boost resource utilization while also boosting the over performance of the system. This can easily be applied as part of the HPO process, in the end choosing not only the best configuration, but the top N configurations as your ensemble. The genetic HPO algorithm can aid this further by provided N distinct populations where the best is chosen from each. As each population is optimized from a distinct starting point, there would be more variability in the top performers and thus more confidence can be provided by the ensemble result.

C. Transfer Learning and Subproblem Modeling

Transfer learning[4] and model fine-tuning can also provide a method for both boosting performance as well as scaling ability. In transfer learning, a neural network can be pre-trained with a dataset in a similar format. By designing a hierarchy of transfer learning we aim to boost performance while allowing multiple distinct networks to be training in parallel.

An example of this can come from an application of deep learning in precipitation forecasting. If the goal is to predict the evolution of a storm from local radar images, general image recognition can provide a great baseline for understanding low level features of a radar image. Because of this, a network pre-trained on imagenet may train faster on the radar dataset as much of the underlying functionality is already there leaving only the high level radar understanding to be trained at the final layers.

This can broken down further into subproblems. In this case, each location will have it's own subset of radar images and local features that need to be learned. A network can be trained to predict precipitation based on the overarching dataset to gain fundamental knowledge of precipitation trends creating a global model. From this, each local area can fine-tune a model specific to that region, again only needing to learn the high level specific features to that new training set. Because it was able to inherit the learning from the global dataset, it is likely that this will generalize better to situations not seen in that location, but are present in the larger dataset.

IV. COMBINED NEURAL NETWORK SCALING

These scaling techniques can be very useful independently, however each will hit a limiting factor requiring either more work or concessions in the model design. Individually, the limited factor of data parallelism tends to a lack of convergence with large batchsizes. Model parallelism is limited by the number of ways in which the model can be split, especially in ways that allow truly parallel computation. As we will see later, an intelligent HPO algorithm can only train so many models in parallel before the benefits of the smart optimization begin to diminish. Brute force hyperparameter sweeps are likely to waste a significant amount of compute resources as poorly performing configurations can be evaluated repetitively.

If these techniques can be used in tandem, each at a level in which the limiting factors have yet to take a toll, we can achieve a higher level of efficiency when considering

Table II
BASELINE RUNS

Method	Random Search
Total Training Runs	5000
Runs that hit thres.	5
Best overal run	98.63%
Mean runs to thres.	500
Mean Time/Run	765s

the broader development process and the total time to a well-trained model. In the following sections, I will describe how I intend to illustrate this point through some basic experiments and analysis in order to provide some insight into how best these techniques can be implemented.

V. EXPERIMENTS

To prove this point, I've developed a set of experiments with a simple dataset to demonstrate the potential cumulative benefits of multiple parallel techniques in training neural networks. In this case, I will provide data on the effect of using data parallelism, model parallelism and hyperparameter optimization compared to a baseline of random search and no training parallelism.

A set of baselines will be computed based on speedup seen of a single training run for data and model parallelism vs using neither on a single compute node. The speedups seen here will be applied in tandem to determine the combined speedup of implementing multiple tactics at once.

The primary focus of these tests is to determine the advantage of parallelizing the full training process, including not only a single training run but as many runs as it takes to reach a sufficiently well trained model within a set number of steps.

All tests noted here were performed on a Cray XC-30 system. Each node contained 2.7GHz Dual-Socket 24-Core Intel IvyBridge generation CPUs and 64GB of DDR3 Memory. The Cray Urika-XC Analytics and Artificial Intelligence Software Suite was used to run distributed Tensorflow and Dask jobs as needed.

A. Dataset and Training

The experiments will run with the well-known and well-tested MNIST dataset[5]. This dataset is comprised of 28x28 labeled images representing handwritten digits from zero to nine. The goal is to accurately classify which digit each image represents. An example set of these images can be seen in Figure 1 as the input layer to the network at the bottom.

MNIST is comprised of 60,000 labeled training images and 10,000 labeled validation images. Each epoch of training consists of the gradient calculation and application for the equivalent of this entire training dataset. In the case of data parallelism, this may not lead to each individual image being

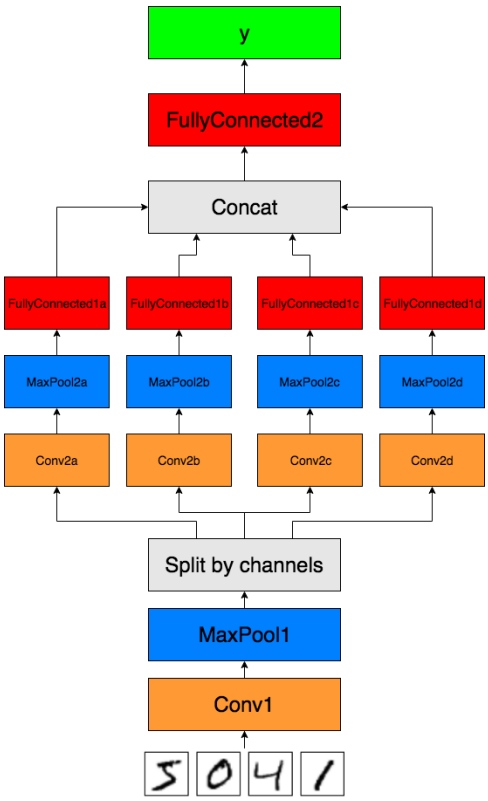

Figure 1. Model Diagram

seen by the model each epoch however since each sample is seen at random and the dataset is sufficiently large, this should not be a problem. For the sake of these runs, the training culminated at 5 epochs, keeping the amount of work consistent for each training run and providing a consistent cutoff point for determining whether or not the threshold accuracy has been hit.

In a real world deep neural network, this number would likely be much higher, on the order 100 epochs to a trained model. When this combines with a larger dataset, the training times can become quite large even with a lot of compute resources. This experiment is intended to provide an illustration of how much time can be saved which can be extrapolated to these larger cases.

To determine a sufficiently difficult threshold to hit within 5 epochs, random search with a wide range of possibilities for each hyper-parameter was applied 5000 times with 5000 different random hyperparameter configurations. The threshold was set to 98.6% which conveniently was achieved 0.1% of the time, or 1 in 1000 runs. These results are listed in Table II. The wide range in the possible configurations listed in Table I was chosen to simulate an environment in which a developer had no prior experience with this model as would be more likely with a larger dataset and more specialized neural network model.

Table III
SINGLE NETWORK PARALLEL DL METHODS

Method	Time/Iteration	Speedup	Nodes
Standard	1090s	1.0x	1
Model Parallel	718s	1.5x	4
Data Parallel	310s	3.5x	4

B. Model Architecture

In order to evaluate model parallelism, I've developed a variant of a fairly standard convolutional neural network (CNN) specifically designed to work well with model parallelism. This architecture is diagramed in Figure 1. In this model, the second convolutional layer and first fully connected layer are split into 4 distinct, independent sets of layers.

These layers are created by splitting the set of activation maps from the first convolutional layer into 4 equal size tensors. These are then concatenated just before the final fully-connected layer. This allows the bulk of the workload to happen in parallel creating the option for a horizontally model parallel network. The initial convolution and final fully connected layer by necessity will still be computed sequentially.

VI. RESULTS

A. Single Run Parallelism

In order to evaluate model and data parallelism in this example neural network, three tests with the default hyperparameters listed in Table I were run. The baseline ran on a single node with no parallelism applied.

The model parallel run utilized tensorflow's builtin gRPC distributed runtime[6] to split the execution of the second round of convolutional layers and the first round of full-connected layers on 4 separate nodes. Much of the process still requires sequential execution including data preparation as well as the initial and final layers, therefore it will not reach a high level of scaling efficiency, however in this test, even with a fairly basic network, we see a 1.5x improvement in time of execution.

The data parallel test was performed with an MPI communication model developed by engineers at Cray, again within the tensorflow framework. In this case, we run on 4 nodes to provide a clear comparison to the model parallel approach as well as to limit the necessary increase in batchsize. In this case, as that only 5 epochs worth of data was trained on, we see some deviation from ideal scaling due to startup costs of loading and prepping data, however it still attains a speedup of 3.5x that of the baseline. Allowing this approach to run longer will lead to near ideal scaling efficiency, however to keep things consistent and limit the run time of this experiment, the decision was made to stick to the 5 epoch equivalent.

B. Hyperparameter Optimization

Hyperparameter optimization has the ability to dramatically decrease the number of training runs necessary to produce a sufficiently well-trained model. In order to evaluate the ability of such an algorithm to do so, a genetic HPO algorithm developed by engineers at Cray was used on this neural network model to hit the threshold specified in the Dataset and Training section. The range of each hyperparameter evaluated matched that of the random search (Table I) and the defaults where set to initially low values and basic techniques (standard Stochastic Gradient Descent (SGD) and sigmoid activation functions) such as to simulate a network in which the developer has no prior knowledge of which configuration would be most effective.

This algorithm defines a population of unique hyperparameter configurations. At random this population will be mutated (random adjustments to the hyperparameters), mated (combined with other high-performing configurations) and evaluated generation by generation. This process will allow some amount of random search of the space of hyperparameters while on average also moving toward configurations that are known to perform best.

To gain insight into how scaling this HPO algorithm can speedup the process of tuning hyperparameters, tests were run from 1-64 nodes by powers of 2. As the node count increased, so did the population size so as to ensure full utilization during each generation. Table IV shows the results of this test. The first column lists the node counts. The second shows the number of generations necessary to reach the threshold (98.6% as before). This decreases with increasing node counts as each subsequent generation will have more information to determine a new set of test configurations. The total number of networks trained by the threshold are listed in column 3, also increasing as that larger populations allow more unique individual runs per generation. Column 4 shows total wall-time to the threshold accuracy in seconds. Column 5 shows the speedup achieved relative to random search. This is calculated based on both the decreased number of total runs as well as the increase in runs per second compared to a single node. Column 6 shows the final error after hitting convergence.

These data come from a single run each. As that the genetic algorithm depends greatly on randomness, there are some clear deviations from the trends in this data. This can be seen clearly in the increase of generations between 4 nodes and 8 nodes. In this case, it can be assumed that random mutations and initialization in the 4 node run quickly put it on the path to an optimal configuration.

The final error at convergence for the majority of these HPO runs was in fact higher than we were able to achieve with random search. This is despite random search running dramatically more total training runs (5000). This may indicate the value of an optimized approach in that hyperparam-

Table IV
HPO SCALING

Nodes	Gens	Runs	Time(s)	Speedup	Acc.
1	50	80	13,496	6.3x	98.60%
2	34	101	11,258	7.5x	98.67%
4	31	207	10,312	8.2x	98.69%
8	22	272	7,860	10.7x	98.66%
16	16	443	5,529	15.3x	98.69%
32	14	761	4,961	17.0x	98.90%
64	11	1187	3,855	21.9x	99.15%

Table V
COMBINED SPEEDUP

Methods	Runs	Nodes	BS	Pot. Speedup
Random Search	500	1	100	1x
MP and DP	500	8	400	5x
HPO alone	272	8	100	11x
HPO and MP	272	32	100	16x
HPO and DP	272	32	400	38x
HPO, MP and DP	272	128	400	56x

eter configurations known to result in lower accuracy will be avoided in future generations leading to more exploration in both new areas of the hyperparameter space and within areas known to show promise.

C. Combining Parallel Training Techniques

Based on the data compiled above, we can begin to understand the potential overall benefit to the training process in terms of time to an optimally trained model. This can be seen in Table V where the speedup of Hyperparameter Optimization (HPO), Data Parallelism (DP) and Model Parallelism (MP) are provided. The potential speedup is calculated based on a number of assumptions, and it is important to note that further work is necessary to validate some of these assumptions.

The random search baseline assumes running on a single node repetitively until the threshold is passed by a training run. Assuming only 1 per 1000 runs on average reaches the threshold as noted in Table II, then when starting from scratch the assumption can be made that on average it will take 500 runs to reach the threshold accuracy. This may be generous in that random search will likely need to be run many more times to provide enough confidence in that result.

The HPO alone speedup takes into account not only the decrease in number of total runs necessary, but also the number of nodes on which the runs each generation of the genetic algorithm will be performed. The 8 node run from Table IV was used for this evaluation as that it performed well without testing an overly wide range of configuration. Additionally, beyond 8 nodes, scaling efficiency based on this metric begins to decrease below 100%. The utilization rate never quite reaches 100% in this case due to the fact that there are a variable sized number of unique configurations per genration rather than a generation size being set by the available workers. Therefore, the total speedup from running on 8 nodes was only 5.8x that of running on a single node. This was determined by calculating the number seconds per run and determining the ratio of those numbers between 1 node and 8 nodes.

The addition of model and/or data parallelism should increase the speedup by the multiplier listed in Table III, as in the speedup for combined MP and DP is $3.5{*}1.5 = 5.25$x, rounded down to 5x in this table. This assumption limits us to the 4 nodes used in the single training run tests per MP and DP for the values listed here, however of course this can be increased in practice.

VII. ANALYSIS

The baseline runs with random search hyperparameter sweeps bring to light an important consequence of failing to properly guide the HPO process. Despite the high number of total training runs, the top performer still failed to outpace the majority of genetic HPO runs listed in Table IV.

Single model training will continue to be dominated by data parallelism due to its clear advantage in being able to scale overall throughput. Even with some concessions and adding development time, there is no argument that DP is the best-known way to scale deep learning. This is evident by the data gathered in Table III. Another takeaway, however, is that even with a small model where the ratio of parallel work to sequential work is limited, we are still able to see further reduction in training time, without any affect on batchsize. We can speculate that a much deeper network, say over 100 layers, with a higher ratio of parallel to sequential layers, may come much closer to ideal scaling efficiency.

From this data, HPO scaling clearly comes to the forefront in providing a true speedup in neural network training. Even beyond data parallelism, having an efficient HPO algorithm in your toolbox can not only allow efficient parallel training, but also decrease the amount of total work necessary to optimize a neural network. Table IV shows that although random variation can have an affect, in general more compute nodes will decrease the time to a well-trained model, and on top of that can likely lead to a better configuration that can be found through brute-force hyperparameter sweeps.

From Table V, the main takeaway is that we are able to scale the training and optimization of a deep neural network to 128 nodes, providing 56x improvement from the baseline, all while only increasing the global batchsize by a factor of 4. Because the increase in global batchsize is kept to a minimum, 4x the local batchsize as opposed to 128x the local batchsize, any effect on the convergence rate will be insignificant. This provides a developer with more flexibility in model and training design while still making full use of available compute resources.

VIII. Conclusion and Future Work

This initial look at combined deep learning scaling approaches leaves room for further work to verify these assumptions and determine how far this approach can generalize. A deeper model and larger dataset should show that the gains we see with MNIST on a basic CNN will be only more evident with a larger computational load. On top of this, a real-world dataset and unique model designed to take advantage of combined parallelism could be developed to evaluate HPO techniques when the model is less well-known and tested.

Currently, the limiting factor to combining scaling techniques is that they are not regularly supported in widely available toolkits. A tool that would allow model and data parallel training that can integrate with HPO algorithms as well as ensemble networks could allow further evaluation of the possible decrease in time to a well-trained model and more importantly, make this accesible to more deep learning practitioners.

When attempting to tackle the enormous problem of bringing Deep Learning to an HPC scale, the problem of exploding batchsizes cannot be ignored. Weve provided a number of techniques based on a standard DL workflow that can expand on traditional approaches to neural network scaling, most without affecting the batchsize. When considering the scaling of neural network training, it is important to always consider this holistic approach to the overall workflow.

References

[1] Thorsten Kurth, Jian Zhang, Nadathur Satish, Ioannis Mitliagkas, Evan Racah, Mostofa Ali Patwary, Tareq Malas, Narayanan Sundaram, Wahid Bhimji, Mikhail Smorkalov, Jack Deslippe, Mikhail Shiryaev, Srinivas Sridharan, Prabhat, Pradeep Dubey *Deep Learning at 15PF: Supervised and Semi-Supervised Classification for Scientific Data* Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2017.

[2] J. Dean, et. al., *Large Scale Distributed Deep Networks*, Neural Information Processing Systems (NIPS), 2012.

[3] Jaskper Snoek, et. al., *Practical Bayesian Optimization of Machine Learning Algorithms* Neural Information Processing Systems (NIPS), 2012.

[4] Sinno Jialin Pan, Qiang Yang, *A Survey of Transfer Learning* IEEE Transactions on Knowledge and Data Engineering.

[5] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. *Gradient-based learning applied to document recognition* Proceedings of the IEEE, 86, 22782324, 1998.

[6] Martn Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Man, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Vigas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems* Software available from tensorflow.org, 2015.