Benchmarking Distributed Deep Learning Using HDFS and the Hadoop Ecosystem

Laura Graesser
New York University
New York, USA
lhg256@nyu.edu

Taha Raslan
New York University
New York, USA
tr1223@nyu.edu

Abstract—

This project benchmarks the average training time per epoch for a neural network on the Hadoop/Spark ecosystem vs. a CPU and GPU. It is motivated by a desire to learn how to train neural network models in a distributed fashion, on CPUs, using the Hadoop ecosystem, and to understand if this offers a viable alternative to training neural networks in a more traditional fashion using GPUs. We found that using Spark, training time was 3.5 – 10x slower when compared to a single CPU. GPUs were significantly faster still. We conclude that currently it is preferable to use CPUs or GPUs to train neural networks even though Hadoop has a better model for storing and accessing very large models and datasets, however this may change in the future as the Hadoop/Spark ecosystem develops.

Keywords—analytics, deep learning, neural networks, distributed training of neural networks, Hadoop, CIFAR, MNIST

I. INTRODUCTION

Over the past five years, neural network algorithms have achieved state of the art results across many machine learning problems. This includes but is not limited to computer vision problems (object recognition [5], detection and bounding, character and text recognition [4] for example), speech recognition, and natural language understanding (machine translation, part of speech tagging, and parsing for example). These algorithms excel at approximating highly complex, non linear functions and a major benefit is that they learn features relevant to the problem at hand, significantly reducing the time needed to engineer features. In order to do this, neural network algorithms typically have millions of parameters that need training.

One of the challenges of this approach is that these algorithms require a lot of data and computational resources to train effectively.

Today, the industry standard is to use clusters of GPUs to train these models. However, this is very costly, with a single state of the art GPU costing over $1000 and requires specialist knowledge of GPU programming.

Over a slightly longer period of time, the last twelve years, technologies for processing extremely large quantities of data in a distributed fashion have developed significantly. Starting with Google in 2004 with the MapReduce programming model and a distributed file system [2,3], the development of open source versions of both of these technologies followed, created by Apache Hadoop [1]. One of the major strengths of these technologies is that they are typically built using commodity hardware and computations scale almost linearly with the amount of data.

Consqeuently, Hadoop and its ecosystem offers a cheaper, and perhaps more accessible, alternative to training neural networks models on multiple GPUs.

Many organizations already have Hadoop clusters, which could be used to train neural network models in a distributed fashion, so no additional hardware would be required. If some of the machines have large RAM to support Spark, this is an added bonus. Furthermore, to make use of neural network models, new users would only need to learn about these algorithms and would not need to spend time learning how to program GPUs.

There are two aspects of neural network's training that can be distributed [6]. The first is the model itself. The parameters of a network can be divided across multiple nodes containing GPUs or CPUs. Communication across these nodes can easily become a bottleneck, so layers with local connections are better suited to partitioning across nodes as is argued in [6] and demonstrated in [5]. However [9] showed that it is possible to distribute fully connected layers. Convolutional layers contain fewer weights and can typically be copied to all nodes. This was the approach taken in [9].

The second aspect that can be distributed is the data. Single models or partitioned models are replicated multiple times and batches are divided across the partitions and the forward pass processed in parallel. The challenging part is how to coordinate the gradient update to the parameters across all of the replicas. There are two families of solutions to this problem. Synchronous and asynchronous updates, discussed in more detail in section III.

This paper demonstrates that distributed deep learning using the Hadoop ecosystem is possible and provides some simple benchmarks comparing the time to train for comparable models on a single CPU, GPU, and Hadoop cluster. We use the classic MNIST and CIFAR 10 datasets.

II. MOTIVATION

This project is motivated by a desire to learn how to train neural network models in a distributed fashion, on CPUs, using the Hadoop ecosystem.

Whilst distributed deep learning is not original, we think our benchmarks are useful in two ways. First we provide a benchmark for distributed deep learning across commodity, open source software and hardware on CPUs. Google in **[6]** provided a benchmark on their own machines using their implementation of asynchronous stochastic gradient descent called DistBelief. Whilst this is a useful benchmark, it is not possible to reproduce. Second, we provide a benchmark comparing distributed CPU and GPU training. Most other benchmarks focus on cross GPU or multi-GPU comparisons. Understanding the trade off between a purchasing a single GPU vs. using a Hadoop cluster, may be beneficial for those new to deep learning.

III. Related Work

There is a broad consensus over how to partition models and data across multiple machines [5, 6, 7, 9]. However, there are a number of different approaches to the gradient update.

In 2012 Google pioneered large scale distributed training of neural networks across over 100 machines with 20 cores per machine [6], partitioning both the model and the data to speed up training. Google's DistBelief framework updates the gradients in an asynchronous manner. A master copy of the model's parameters are held on a parameter server, which may be sharded across multiple machines. Before processing each mini-batch, a model replica requests an updated copy of the model's parameters. Since both the model and parameters are distributed across multiple machines, each model node need only communicate with the parameter shard that contains the parameters it is responsible for. However, it is possible that this communication becomes as bottleneck [7] if many replica nodes are trying to communicate an update to the same parameter shard. After updating its parameters locally, the replica processes a mini-batch, calculate the gradients and sends the gradient to the parameter server, which is applied to to the current value of the model parameters.

One advantage of the asynchronous approach is that no replica needs wait for other replicas to finish processing their mini-batch. Instead the replica simply uses whatever the most up to date version is available on the parameter server. This eliminates a common issue of processing being held up by lagging nodes, a phenomenon that is also observed in MapReduce applications and dealt with using a number of strategies. This approach is also robust to machine failures, significantly more so than synchronous SGD which fails if a single machine fails.

In contrast to Google, Baidu used a synchronous approach to the gradient update [9]. Synchronous models do not have a parameter server. Instead after each replica has processed a gradient update, the gradients from each of the replicas are aggregated and broadcast back out to each of the replicas, which use this to update their local parameters.

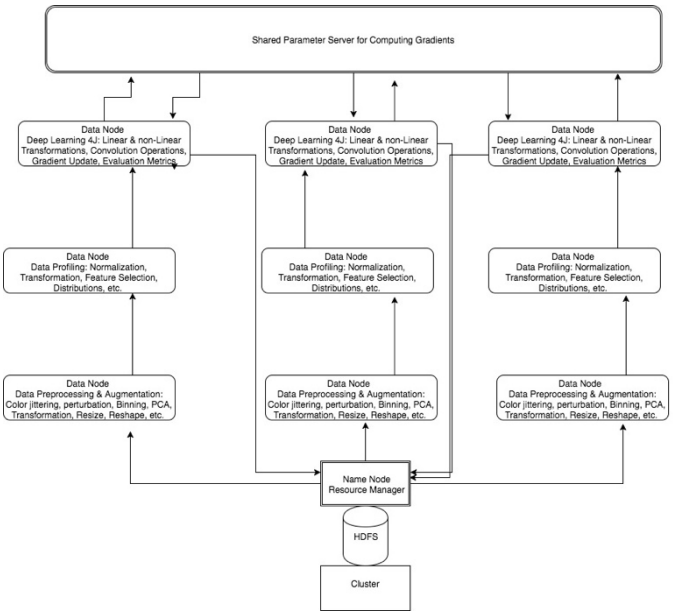
Synchronous approaches have gradients which are less noisy because the gradients are aggregated across the whole mini-batch and parameters across each of the replicas are always in synch and have all had the same number of updates. This is not necessarily the case for asynchronous methods. However, synchronous methods are held back by the slowest replica, which can significantly slow training.

There have been a number of extensions to asynchronous approaches which aim to improve convergence speed [7]. First, elastic averaging [11] which modifies the loss function to encourage consensus between the central parameter sever and local parameters. Alternatively, [9], propose a gossiping SGD which can be thought of as a decentralized version of elastic averaging. This provided further convergence speedups over elastic averaging when tested on a GPU cluster.

Both [6] and [9] provided benchmarks which demonstrated significant speed up using a distributed framework. [9] focused on distributing computation across GPUs and benchmarking the performance vs. a single GPU. Their best model using 32 GPUs obtained a 25x speedup vs. a single GPU, which interestingly is sublinear. In contrast [6] focused on distributing training across commodity CPUs and reported a speed up of up to 12x in training time for their largest model using 81 machines. [6] is also the closest to the approach taken in this paper.

Finally, our approach combines elements of both [6] and [9] for the benchmarks, since we compare training time for a model distributed across multiple CPUs vs. a single CPU and a single GPU.

IV. Design



V. Data

We used two classic computer vision datasets, MNIST, and CIFAR 10. The MNIST dataset consists of 50,000 grayscale 28 x 28 images of handwritten digits, with an additional 10,000 reserved as a held out dataset for testing. The CIFAR 10 dataset consists of 50,000 color 32 x 32 x 3 images of objects from 10 well known categories (airplanes, cats, trucks, etc.), with an additional 10,000 reserved as a held out dataset

for testing. The data was stored in a comma separated csv file on HDFS.

We examined the maximum and minimum values for each of the different pixels in the CIFAR images using a simple Map Reduce program. In each case we found that the pixels covered the whole possible range from 0 to 255. This indicated that we could normalize each pixel to be in a (0,1) range by dividing by 255 whilst ensuring that the normalized pixels shared the same scales.

VI. Results

To benchmark the Hadoop/Spark platform we used Spark's mllib to build a simple multilayer perceptron classifier with three layers. The second and third layers had the same size for the models trained both datasets, (1024 and 10 dimensions respectively), whereas necessarily the size of the first layer differed due the the difference in input image size. The MNIST model had a 1st layer of size 28 * 28 = 784, whereas the CIFAR 10 model had a 1st layer of size 32 * 32 * 3 = 3072. We also implemented identical models in pytorch so that we could run them on a CPU and a GPU.

Figure 1: Benchmarking MNIST

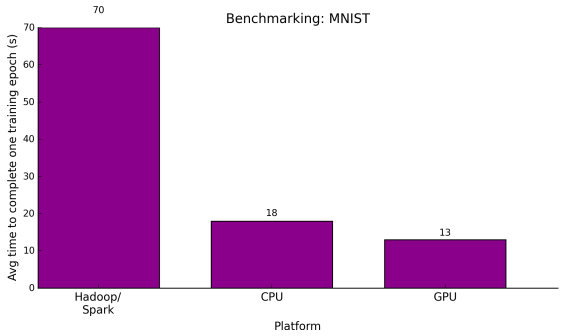
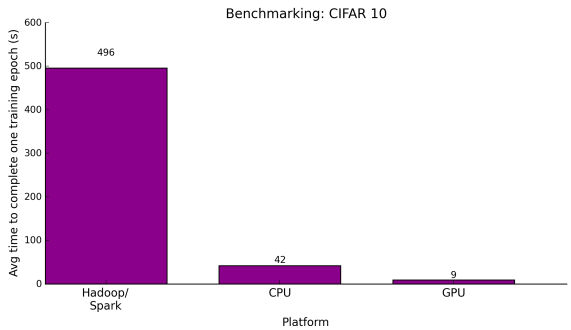


Figure 2: Benchmarking CIFAR 10



To compare the different platforms we ran each model for a number of epochs (one epoch = one full cycle through the training dataset) on each platform, and computed the average training time per epoch in seconds.

We used NYU HPC's Dumbo, a 48-node Hadoop cluster, running Cloudera CDH 5.9.0 (Hadoop 2.6.0 with Yarn) as our Hadoop/Spark ecosystem, a 1.6 GHz Intel Core i5 processor with 2 cores as our CPU, and a NVIDIA K-80 240 GB and a NVIDIA GeForce GTX 1080 as our GPUs.

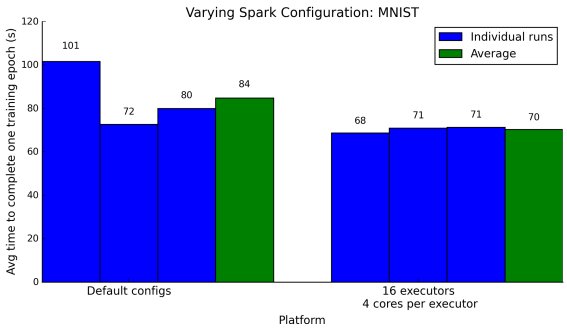
The Hadoop/Spark ecosystem was the slowest of the three platforms we tested, taking on average over 3.5x as long compared to a single CPU on the MNIST dataset and over 10x as long compared to a single CPU on the CIFAR 10 dataset (see figures 1 and 2 for details). As expected the GPU was the fastest platform of the three tested.[1]

We think that the communication between nodes is the limiting factor for the Hadoop ecosystem and is the main cause of the slower performance of the algorithm on this platform. Additionally, we think that the neural network models which are available through Spark's mllib are not as optimized as the highly specialist libraries such as pytorch which are available to train multilayer perceptron's on CPUs and GPUs.

To test this hypothesis, we partially implemented a distributed version of a gradient update and it ran faster than the implementation available through Spark.

We also experimented with increasing the number of executors and cores per executor to try and improve the performance in Hadoop/Spark. Whilst this did appear to benefit the model trained on the MNIST dataset (see Figure 3), it did not help the model trained on CIFAR 10 (see Figure 4), which is somewhat surprising. A potential explanation for this is that since the CIFAR 10 dataset is 5x larger than MNIST, the time needed to transfer additional data across the network when the number of executors is increased cancels out the benefit from increasing the parallelism of the execution.

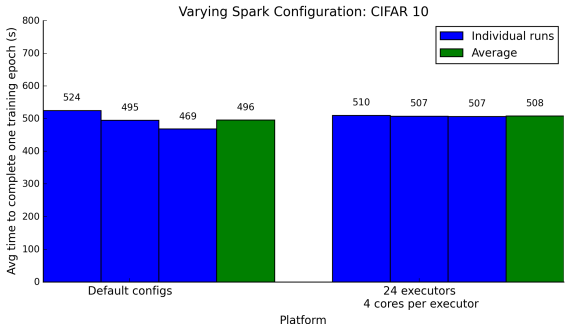
Figure 3: Varying Spark config: MNIST



[1] Please note: The GPUs used to benchmark the MNIST and CIFAR 10 models were different which explains the faster training time for the CIFAR 10 dataset, despite this model being the larger of the two.

One final observation is that we did not have the cluster to ourselves. Many jobs running on Dumbo may be slowing down the compute stages, making it difficult to make exact like for like comparisons. However, we do note that multiple users on a Hadoop cluster does mimic a real world situation since it is unlikely that one user would ever have a cluster to themselves.

Figure 4: Varying Spark config: CIFAR 10



VII. Future Work

There are two directions for future work that we see. The first is to explore different options for improving the efficiency of gradient descent, by pursuing one or more of the approaches below.

1. More efficient automatic differentiation tools to calculate derivatives of python error functions
2. Try second order optimization algorithms (I.e L-BFGS to get the exact location of the minimum.
3. Efficient asynchronous SGD: we run multiple replicas of a model in parallel on subsets of the training data models where we send the updates to a parameter server, which is split across many machines. Each machine is responsible for storing and updating a fraction of the model's parameters. However, as replicas don't communicate with each other e.g. by sharing weights or updates, their parameters are continuously at risk of diverging, hindering convergence.

The second avenue for further work would be to extend the benchmarking in one or more of the following ways.

1. Using different libraries for training neural networks on Hadoop. For example, Distributed Tensor Flow and Deep Learning 4J
2. Using Hadoop clusters of different sizes.
3. Comparing the performance of different iterative algorithms, such as the Perceptron algorithm.

VIII. Conclusion

The Hadoop ecosystem has an inherent advantage compared to training neural networks on single GPUs or CPUs since it supports storage for very large models, gradient parameter vectors, and datasets.

However, based on our benchmarking, the current Hadoop ecosystem appears to be significantly slower for training neural networks compared to single CPUs and GPUs.

This may not continue to be the case in the future, and we feel that the Hadoop eco system has the potential to speed up the training time for deep neural networks with the right configurations and settings. This is currently an active research area focusing on using more efficient distributed numeric computation, automatic differentiation, and optimization for gradient-based learning.

Acknowledgments

Thank you to Shenglong Wang and Santhosh Konda from NYU HPC who helped us install various libraries on the Hadoop cluster and was always willing to answer our questions. Thank you also to Professor Suzanne McIntosh for her guidance.

References

[1] T. White. Hadoop: The Definitive Guide. O'Reilly Media Inc., Sebastopol, CA, May 2012.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In proceedings of 6th Symposium on Operating Systems Design and Implemenation, 2004.

[3] S. Ghemawat, H. Gobioff, S. T. Leung. The Google File System. In Proceedings of the nineteenth ACM Symposium on Operating Systems Principles – SOSP '03, 2003.

[4] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner. Gradient-Based Learning Applied to Document Recognition. Proceedings of the IEEE, Nov 1998

[5] A. Krizhevsky, I. Sutskever, G. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. NIPS, 2012

[6] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, A. Ng. Large Scale Distributed Deep Networks, NIPS, 2012

[7] P. Jin, Q. Yuan, F. Iandola, K. Keuzter. How to Scale Distributed Deep Learning? Nov 2016

[8] K. Simonyan, A. Zisserman. Very Deep Convolutional Networks for Large Scale Image Recognition, ICLR 2015

[9] R. Wu, S. Yan, Y. Shan, Q. Dang, G. Sun. Deep Image: Scaling up Image Recognition, Feb 2015

[10] T. Chillimbi, Y. Sunzue, J. Apacible, K. Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. Microsoft Reasearch, Open access to the Proceedings of the 11th USENIX Symposium on Operating Systems, 2014.

[11] S. Zhang, A. Choromanska, Y. LeCun. Deep learning with Elastic Averaging SGD. In Advances in Neural Information Processing Systems 28, pages 685–693, 2015.