Challenges for Machine Learning on Distributed Platforms

Tom Goldstein[1]
University of Maryland, College Park, MD, USA
tomg@cs.umd.edu

Abstract

Deep neural networks are trained by solving huge optimization problems with large datasets and millions of variables. On the surface, it seems that the size of these problems makes them a natural target for distributed computing. Despite this, most deep learning research still takes place on a single compute node with a small number of GPUs, and only recently have researchers succeeded in unlocking the power of HPC. In this talk, we'll give a brief overview of how deep networks are trained, and use HPC tools to explore and explain deep network behaviors. Then, we'll explain the problems and challenges that arise when scaling deep nets over large system, and highlight reasons why naive distributed training methods fail. Finally, we'll discuss recent algorithmic innovations that have overcome these limitations, including "big batch" training for tightly coupled clusters and supercomputers, and "variance reduction" strategies to reduce communication in high latency settings.

2012 ACM Subject Classification Computing methodologies → Machine learning

Keywords and phrases Machine learning, distributed optimization

Digital Object Identifier 10.4230/LIPIcs.DISC.2018.2

Category Invited Talk

1 How do we train neural nets

Deep neural networks are one of the most flexible and powerful tools in machine learning. Neural networks are complex models that are "trained" by solving a large optimization that minimizes an objective function, called the "loss," that measures how well the neural net fits to training data. Computing this loss function is expensive because it requires summing over every element in a large training dataset. To avoid this expense, neural optimization problems are commonly solved using stochastic gradient descent (SGD). This algorithm works by randomly sampling a small batch of data on each iteration, forming an approximate loss function using only this small data sample, and then doing an approximate gradient descent step using this approximate loss function. This SGD algorithm was originally adopted because computers in the 1980s didn't have the computing power to evaluate the exact loss function (which requires the full dataset). SGD only uses a small batch of data on each iteration, and this makes each gradient decent update cheap, but "noisy" (inexact).

[1] Support for this work was provided by DARPA Lifelong Learning Machines (FA8650-18-2-7833), the US Office of Naval Research (N00014-17-1-2078), the US National Science Foundation (CCF-1535902), and the Sloan Foundation.

2 Why is training on HPC platforms hard?

It is commonly said that SGD is an "inherently serial" algorithm; the $(k+1)$th iteration of SGD uses the result if the kth iteration as a starting point, and so iterations need to take place one at a time. Furthermore, each iteration is cheap when a small batch size is used. This makes the algorithm hard to scale – when the batch size is small, iterations are cheap and don't require enough work to spread over a large number of workers. For example, with a minibatch size of 128 data samples (which is fairly standard for many imaging problems) and 128 workers, each worker would be processing only 1 image/sample per iteration, and the workers would have to communicate after each computation. In this case, the communication costs would far outweigh the compute costs, and training would be inefficient. This is largely what motivated the recent emergence of GPUs for machine learning. On a GPU, a single iteration of SGD can be split over 1000s of small cores in a shared memory architecture. In this case, we're still doing the same old serial SGD algorithm, but using lots of parallelism to get each serial step done faster. This only works on a GPU because all of the cores are synchronized and memory is shared, which means there is little or no communication overhead.

3 Can't we just use bigger batch sizes?

There's an obvious (but naive) solution to the scalability problem described about: increase the batch size. This gives us more accurate (i.e., less "noisy") gradient computations that should make the algorithm converge faster. With bigger batch sizes, there's lots more work to do per iteration, and this work can be spread over many workers. Furthermore, if convergence happens in fewer iterations, then this could speed things up and enable training with lower wall-clock time.

But big batch training poses a problem: the argument above assumes that more accurate big-batch gradients work better than less accurate small-batch gradients. Shockingly, this is the opposite of what happens in practice; larger batches and more accurate gradients are *worse* for neural optimization. You need the noise to find good minimizers. Big-batch algorithms sometimes get stuck in local minimizers of the non-convex loss functions, or else find global minimizers that perform poorly on new data samples that weren't used for training. In contrast, noisy methods "bounce out" of these local minimizer traps, and tend to find global minimizers that perform well on new data points that aren't used for training. This good behavior of small batch SGD is known as "implicit regularization"; while there are many minimizers to neural loss functions, small-batch SGD creates a bias towards minimizers that avoid "over-fitting," and perform well on test data. The difference between small batch and large batch optimization, and the cause of implicit regularization is still not well understood. The qualitative differences between large and small batch training were recently explored in [3].

4 So what can be done to scale up SGD in distributed environments?

There are three main approaches to scaling up SGD.

- *Find a way to use bigger batches without finding bad minimizers:* While using big batches in a naive way results in poor models, it is possible to use big batches in a more sophisticated way that still performs well. In one approach [2], we start with a small batch size at the early stages of optimization, and quickly expand the batch size to be

very large. We show that this approach helps mitigate the loss in performance that comes from starting with a big batch, while simultaneously making it easier to automate the training process.

- *Find a way to reduce communication overhead so that SGD can tolerate small batches:* This usually requires an algorithm that can do multiple iterations on a worker before communicating back to a central server. By using delayed asynchronous communication, algorithms avoid being communication bound because they keep working while they wait for communication to happen in a separate (often asynchronous) thread. Special variants of SGD can be developed in which workers share information that enables them to "stay on the same page" and search for similar solutions even when communication is infrequent. This direction was explored in [1].
- *Find problem domains where iterations are so expensive that HPC is needed, even for small batch sizes:* One such problem domain is the processing 3D datasets (as opposed to 2D images). Processing videos and 3D volumes requires a large amount of memory and far more FLOPS per byte than 2D processing. This is a new frontier domain where HPC is likely to be dominant.

References

1 Soham De and Tom Goldstein. Efficient distributed SGD with variance reduction. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 111–120. IEEE, 2016. doi:10.1109/ICDM.2016.0022.

2 Soham De, Abhay Yadav, David Jacobs, and Tom Goldstein. Automated inference with adaptive batches. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 1504–1513. PMLR, 2017. URL: http://proceedings.mlr.press/v54/de17a.html.

3 Hao Li, Zheng Xu, Gavin Taylor, and Tom Goldstein. Visualizing the loss landscape of neural nets, 2017. arXiv:1712.09913v1.