ECE 901: Large-scale Machine Learning and Optimization **Fall 2016**

Lecture 24 — 12/13

Lecturer: Dimitris Papailiopoulos *Scribe: Yanyao Yi & Jinman Zhao*

Note: These lecture notes are still rough, and have only have been mildly proofread.

24.1 Introduction

- Advances and challenges of distributed machine learning
- Lots of not-well-understandings
- Depression of performance gain

24.2 Parallel vs. Distributed

Stochastic Gradient Descent(SGD) is an Uber-Algorithm whose main goal to minimize losses

$$\min_{x \in R^d} \sum_{i=1}^{n} f_i(x)$$

where $f_i(x)$ is loss for data point i. And when used to minimize the above loss function, a standard gradient descent method will preform the following iterations:

$$x_{k+1} = x_k - r_k \cdot \bigtriangledown f_{s_k}(x_k)$$

SGD has been around for a while due to some good reasons, like:

- Robust to noise
- Simple to implement
- Near-optimal learning performance
- Small computational foot-print

But when the training set is enormous and no simple formulas exist, evaluating the sums of gradients becomes very expensive, because evaluating the gradient requires evaluating all the summand functions' gradients. **SGD can take 100s hrs on large data sets** [Dean et al, Google Brain]. So to speed up machine learning is also our goal. One simple way to economize on the computational cost at every iteration is that stochastic gradient descent samples a subset of summand functions at every step. This is effective in the case of large-scale machine learning problems.

There are two settings to perform the idea of training at scale.
- Parallel(CPU):

- Single machine with multicores(usually up to 100s)
- Shared memory (all cores have access to RAM)

- Distributed:

- Many machines(usually up to 100s) connected via network
- Shared-nothing architecture(each node has its own resources)

Both methods have their own advantages and disadvantages, single machine with multicores or large memory is very expensive but its communication to RAM is cheap, while many machines connected via network has non-negligible communication costs. So ideally what we want is:

- Cores $\longrightarrow \infty$
- RAM $\longrightarrow \infty$
- Comm. Cost $\longrightarrow 0$
- Cost to build $\longrightarrow 0$

There are two feasible solutions:

Scaling Up: (getting the largest machine possible, with maxed out RAM)
There is no network requirement for scaling up such that the RAM communication cost is cheap. And scaling up has less implementation overheads and less power (smaller environmental footprint). However, a single machine with 100 cores is very expensive, and scaling up has smaller fault tolerance with limited upgradability.

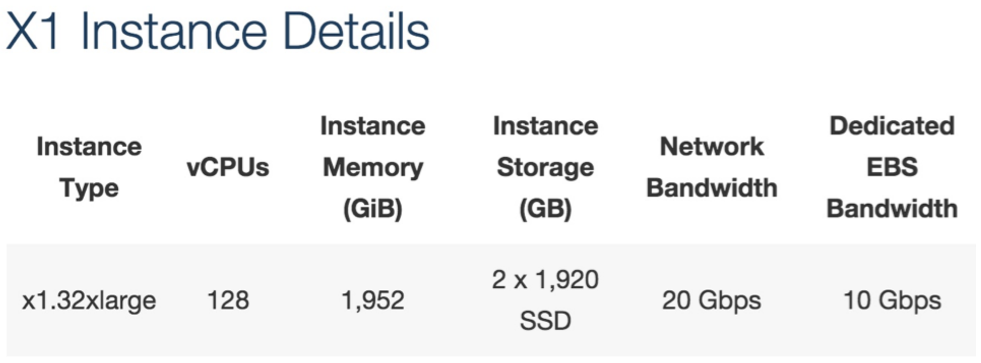
Scaling Out: (getting a bunch of machines, and linked them together)
The RAM communication cost for scaling out is much cheaper, and it can replace faulty parts which means it has better fault tolerance. However, it has network bound, major implementation overheads and large environmental footprint.

Choice of hardware and algorithm helps us better select scaling up or scaling out.

24.3 Choice of Hardware

Here we provide price comparisons for different choices:

- Single machine multi-core (CPU): for 128-threads Intel Xeon Phi + 1TB RAM, it is more than \$ 50,000. (Figure 24.1)
- Rent Instances(EC2, Google Cloud Platform, Microsoft Azure): EC2 price for scaling-up to 128 cores as the above X1 instance is \$1.04/hr and 1-year with nonstop experiments is approximate \$15,000.
- EC2 price for scaling-out to 128 cores is \$ 8×0.11/hr and 1-year nonstop experiments is about \$ 7,500. But Network BW can be 100x slower than single machine Business BW.


X1 Instance Details

Instance Type	vCPUs	Instance Memory (GiB)	Instance Storage (GB)	Network Bandwidth	Dedicated EBS Bandwidth
x1.32xlarge	128	1,952	2 x 1,920 SSD	20 Gbps	10 Gbps

Figure 24.1. X1 instance details

24.4 Choice of Algorithm: minibatch SGD

The master-worker setting for minibatch SGD consists of one master node (parameter server) to store all the parameters, many worker nodes to compute the gradient, and network through which all the nodes communicate (Figure 24.2). At each round, all workers read the parameters from the master and compute the gradient over their own minibatches in parallel. Then the gradients from each worker send back to the master and the master simply update the parameters using all the gradients in sum. This process continues until the error is small or other stopping criteria are satisfied.

24.5 Challenges with Performance Gains/Analysis

Many challenges are there when one wants to evaluate the performance of distributed machine learning algorithms. Surprisingly there is no clear answer to most of those questions including

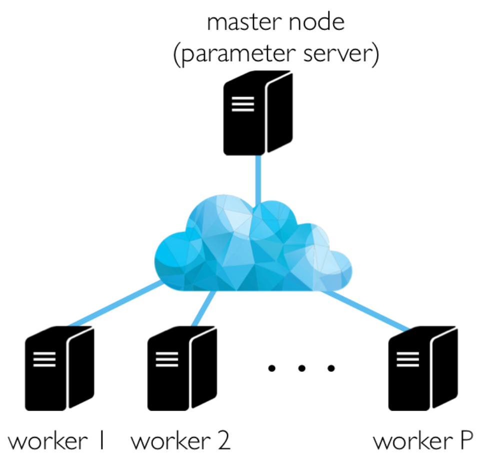


Figure 24.2. The master-worker setting for minibatch SGD.

- How fast does minibatch-SGD converge?
- How can we measure speedups?
- Communication is expensive, how often do we average?
- How do we choose the right model?
- What happens with delayed nodes?
- Does fault tolerance matter?

In the following we use minibatch SGD to give an example how they are complicated.

24.5.1 Convergence

How does minibatch SGD compare against serial SGD? Specifically, we consider the convergence after T gradient computations.

Answer is complicated and depends on problem. Generally a sharp result states that if batch size $B_0 > B(data, loss)$, some function depending on the data distribution and the loss function, minibatch SGD offers no speedup.

As a remark, minibatch SGD actually cannot guarantee convergence if the step size is fixed and the batch size goes large, which means in order to converge, one shall set smaller step size for big batch size, and the convergence rate is thus made slow.

24.5.2 Generalization

How do models trained with minibatch SGD generalize compared against those trained with serial SGD? Specifically, we consider the generalization after T gradient computations.

It turns out the situation here is similar to the convergence case: answer is complicated and depends on problem. Generally if batch size $B_0 > B(data, loss)$, some function depending on the data distribution and the loss function, minibatch SGD offers worse generalization.

It is still open whether this statement is sharp, e.g. if otherwise we can have better generalization.

24.5.3 Choice of right model

Some models are better than others, even from a systems perspective.

Some common considerations are

- Does it fit in a single machine?
- Is model architecture amenable to low communication?
- Is model easy to partition?
- Can we increase sparsity (less communication) without losing with accuracy?

An example is the model for deep neural networks trained using minibatch SGD over a number of machines [1], where their larger (and sparser) models achieve more speedups with increasing number of machines (Figure 24.3).

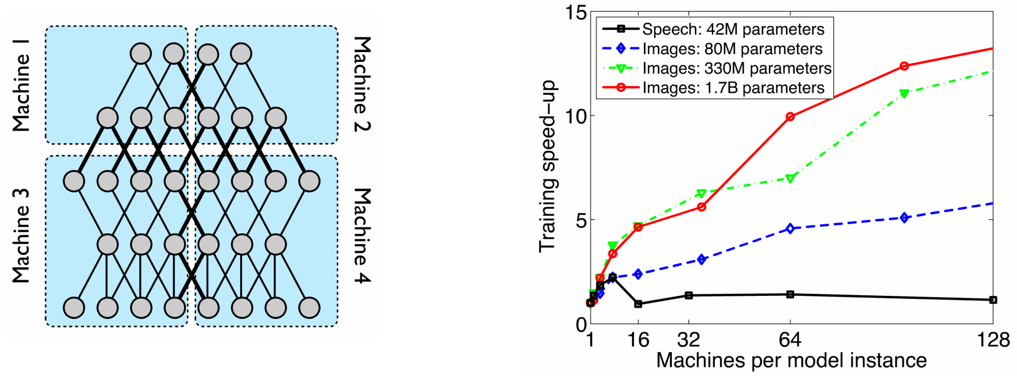


Figure 24.3. The model and the experiment results for training large deep neural networks in distributed setting. [1]

24.5.4 Parallelism Gains

Be careful with the distinction between strong and weak scaling when someone claims about their gains from parallelism. *Strong scaling* engages more computational power (e.g. number of machines) with the problem and the data fixed. Meanwhile, *weak scaling* scales the size of the problem and/or the data along with the computational power. It is usually easier for weak scaling to report better speedups with parallelism. There are already good results regarding weak scaling, but strong scaling is particularly nontrivial and remains a major open problem in the field. An interactive demo [3] shows how the two types of scaling differ in the speedups for some widely used hardware and software architecture in parallel and distributed computing.

24.6 Dealing with Straggler Nodes

As if the design landscape was not complicated enough, another issue is how can we mitigate delays and straggler nodes.

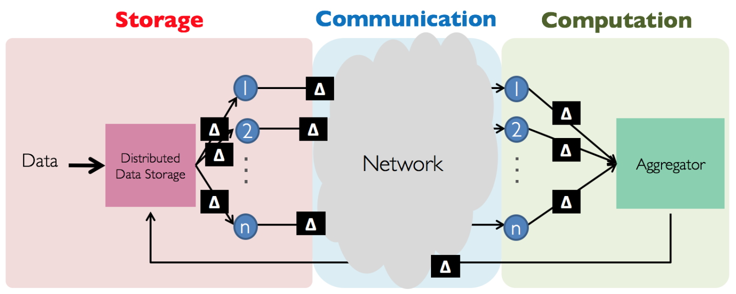


Figure 24.4. Three major functional component of a large-scale distributed machine learning system.

Large-scale distributed machine learning systems can be viewed as three functional components: storage, communication and computation. (Figure 24.4) Data is read from the storage layer, and transmitted through the network towards the computing nodes. The new results of computation are then stored again to complete the loop. Each of these steps incurs some delay and uncertainty. Some measurement on Amazon AWS shows $\tilde{5}$% of the iterations end up with double or even triple latency compared to the rest. (Figure 24.5)

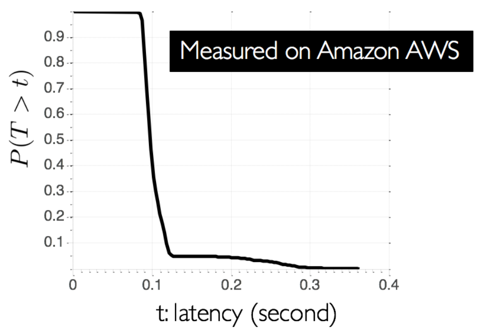


Figure 24.5. A measurement of latency in a distributed system.

Can we use "fault tolerance" solutions to robustify distributed ML speedups? The answer is yes.

A recent work [2] uses erasure codes for data partition and data shuffling.

- The idea for coded partitioning is to use code to assign redundant computation to each nodes so that one does not need to wait for all worker node (i.e. the slowest ones) to yield in order to obtain all the results. In other words, full results can be recovered from partial failure. With the assumption of linearity of the objective and that computation times are i.i.d. shifted exponential random variables, coded partition is proved a $\Theta(\log n)$ speedup for matrix multiplication task.
- The idea for coded shuffling is to use code to at each iteration combine the new data that is to be broadcast to each worker because the limited size of workers' cache, so that the total amount of bits transferred are minimized and each worker can still convert the message back to their needs. When α fraction of data can be cached at each worker, coded shuffling reduces the communication cost by a factor $\Theta(\alpha\gamma(n))$, where $\gamma(n)$ is the ratio of the cost of unicasting n messages to n users to broadcasting a common message to n users. Empirical results show that their coded solutions can achieve speedups up to 40%.

Bibliography

[1] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, and Quoc V Le. Large scale distributed deep networks. *Advances in neural information processing systems*, 12 2012.

[2] Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris S Papailiopoulos, and Kannan Ramchandran. Speeding up distributed machine learning using codes. pages 1143–1147, 2016.

[3] Hang Qi, Evan R. Sparks, and Ameet Talwalkar. An analytical model to estimate the scalability and performance of deep learning systems. `https://talwalkarlab.github.io/paleo/`, 2016.