How to scale distributed deep learning?

Peter H. Jin, Qiaochu Yuan, Forrest Iandola, and Kurt Keutzer
University of California, Berkeley
{phj,qyuan,forresti,keutzer}@berkeley.edu

Abstract

Training time on large datasets for deep neural networks is the principal bottleneck in several important applications of deep learning, such as object classification and detection in autonomous vehicles. To minimize training time, training must be scaled across multiple processing nodes using a distributed optimization method, such as synchronous or asynchronous stochastic gradient descent (SGD). While synchronous SGD currently shows the greatest throughput at large scale, synchronous scaling suffers from the need to synchronize all nodes at each gradient step. In asynchronous approaches using parameter servers, training is slowed by contention to the parameter server. In this paper we compare the training time of synchronous and asynchronous approaches to SGD for training a modern ResNet architecture on the ImageNet classification problem. In an attempt to address deficiencies of both synchronous and asynchronous methods, we propose an asynchronous method, gossiping SGD, that aims to retain the positive features of both methods. We find that asynchronous SGD, including both elastic averaging and gossiping, converges faster at fewer nodes (up to about 32 nodes) and at larger step sizes, whereas synchronous SGD scales better at 32 nodes up to 128 nodes. Getting good scaling behavior above 128 nodes continues to be an open problem.

1 Introduction

Estimates of the data gathered by a self-driving car start from at least 750 MB/s. With proper annotation or through an unsupervised learning scheme, all of this data can become useful for training the object detection system or grid-occupancy system of a self-driving car. The resulting training set can lead to weeks or more of training time on a single CPU/GPU system. Therefore, for such applications training time defines the most time consuming element of the workflow, and reduced training time is highly desirable.

To achieve significant reductions in training time, the training must be distributed across multiple CPUs/GPUs with the goal of strong scaling: as more nodes (i.e. compute servers) are thrown at the problem, the training time should ideally decrease proportionally. There are two primary approaches to distributed stochastic gradient descent (SGD) for training deep neural networks: (i) synchronous all-reduce SGD based on a fast all-reduce collective communication operation [1, 2, 3, 4], and (ii) asynchronous SGD using a parameter server [5, 6].

Both approaches (i) and (ii) have weaknesses at scale. Synchronous SGD is penalized by straggling processors, underutilizes compute resources, and is not robust in the face of failing processors or nodes. On the other hand, asynchronous approaches using parameter servers create a communication bottleneck and underutilize the available network resources, slowing convergence.

Individual researchers also have numbers of nodes at their disposal, including compute and network resources. So determining the best approach for a given number of nodes, as well as the approach that scales to the most number of nodes, is of interest to practitioners with finite resources.

1 http://www.kurzweilai.net/googles-self-driving-car-gathers-nearly-1-gb/sec
We are concerned with the challenges of achieving the shortest time for training a deep neural net. Practically this translates into the problem of scaling the training to many processors. We are primarily interested in the overall convergence time; however, the speed of convergence varies over the training run, and also depends on the number of nodes. Thus we investigated the following two questions:

(a) How fast do asynchronous and synchronous SGD algorithms converge at both the beginning of training (large step sizes) and at the end of training (small step sizes)?

(b) How does the convergence of SGD vary with the number of nodes?

To compare the strengths and weaknesses of asynchronous and synchronous SGD algorithms, we train a modern ResNet convolutional network \[7\] on ImageNet \[8\] using various distributed SGD methods. We primarily compare synchronous all-reduce SGD, the recently proposed asynchronous elastic averaging SGD \[9\], as well as our own method, asynchronous gossiping SGD, based on an algorithm originally developed in a different problem setting \[10\]. Gossiping SGD is an asynchronous method that does not use a centralized parameter server; gossiping can be thought of as a decentralized version of elastic averaging. We find that asynchronous SGD, including both elastic averaging and gossiping, exhibits the best scaling at larger step sizes and, perhaps counterintuitively in the case of gossiping, at smaller scales (up to around 32 nodes). For smaller step sizes and at larger scales, all-reduce consistently converges to the most accurate solution faster than the asynchronous methods.

### 2 Background

We will use the following naming convention for SGD: \(\theta\) are the parameters over which the objective is minimized, \(\hat{\theta}\) is the center parameter (if applicable), \(\alpha\) is the step size, \(\mu\) is the momentum, subscript \(i\) refers to the \(i\)-th node out of \(p\) total nodes, and subscript \(t\) refers to the \(t\)-th (minibatch) iteration. Additionally, \(b\) refers to the per-node minibatch size, and \(m\) refers to the aggregate minibatch size summed across all nodes.

#### 2.1 Synchronous All-Reduce SGD

In synchronous all-reduce SGD, two phases alternate in lock-step: (1) each node computes its local parameter gradients, and (2) all nodes collectively communicate to calculate the aggregate gradient, as if they all formed a large distributed minibatch. The second phase of aggregating gradients forms a barrier and is the communication-intensive phase, usually implemented by an eponymous all-reduce operation. The communication overhead of all-reduce is due to a synchronization barrier: each node must wait for all other nodes for the all-reduce to complete before proceeding to the next minibatch iteration. This leads to a straggler effect where the slowest nodes will prevent the rest of the nodes from making progress. Examples of large-scale synchronous data parallel SGD for distributed deep learning are given in \[1\], \[12\], \[2\], \[3\], and \[4\]. We provide pseudocode for synchronous data-parallel SGD in Algorithm 1.

#### 2.2 Asynchronous Parameter-Server SGD

A different approach to SGD consists of each node asynchronously performing its own gradient updates and occasionally synchronizing its parameters with a central parameter store. This form of asynchronous SGD was popularized by “Hogwild” SGD \[13\], which considered solving sparse problems on single machine shared memory systems. “Downpour” SGD \[5\] then generalized the approach to distributed SGD where nodes communicate their gradients with a central parameter server. The main weakness of the asynchronous parameter-server approach to SGD is that the workers communicate all-to-one with a central server, and the communication throughput is limited.
by the finite link reception bandwidth at the server. One approach for alleviating the communication bottleneck is introducing a delay between rounds of communication, but increasing the delay greatly decreases the rate of convergence \cite{9}. Large scale asynchronous SGD for deep learning was pioneered in the Google DistBelief \cite{5} and Microsoft Adam \cite{6} systems. Large scale parameter server systems in the non-deep learning setting have also been demonstrated in \cite{14} and \cite{15}.

2.3 Elastic Averaging SGD

Elastic averaging SGD \cite{9} is a recently proposed algorithm belonging to the family of asynchronous parameter-server methods which introduces a modification to the usual stochastic gradient objective to achieve faster convergence. Elastic averaging seeks to maximize the consensus between the center parameter $\tilde{\theta}$ and the local parameters $\theta_i$ in addition to the loss:

$$F_{\text{consensus}}(\theta_1, \ldots, \theta_p, \tilde{\theta}) = \sum_{i=1}^{p} \left[ f(\theta_i; X_i) + \frac{\rho}{2} \|\theta_i - \tilde{\theta}\|^2 \right].$$

(1)

The elastic averaging algorithm is given in Algorithm 2, where $X_i$ is the data at node $i$ and $\rho$ is a hyperparameter for the consensus term. The consensus objective of elastic averaging is closely related to the augmented Lagrangian of ADMM, and the gradient update derived from the consensus objective was experimentally shown by \cite{9} to converge significantly faster than standard parameter-server async SGD. However, as elastic averaging is a member of the family of asynchronous parameter-server approaches, it is still subject to a communication bottleneck between the central server and the client workers.

Because recent published results indicate that elastic averaging dominates previous asynchronous parameter-server methods \cite{9}, we will only consider elastic averaging from this point on.

Algorithm 2 Elastic averaging SGD. (The hyperparameter $\beta$ is the moving rate.)

# client code
initialize $\theta_{0,i} \leftarrow \theta_0$
for $t \in \{0, \ldots, T\}$ do
  if $t > 0$ and $t \equiv 0 \mod \tau$ then
    $\theta \leftarrow$ receive-server-param()
    send-param-update($+\beta(\theta_{t,i} - \tilde{\theta})$)
    $\theta_{t,i} \leftarrow \theta_{t,i} - \beta(\theta_{t,i} - \tilde{\theta})$
  end if
  $\Delta \theta_{t,i} \leftarrow -\alpha_t \nabla f_i(\theta_{t,i}; X_{t,i}) + \mu \Delta \theta_{t-1,i}$
  $\theta_{t+1,i} \leftarrow \theta_{t,i} + \Delta \theta_{t,i}$
end for

# server code
initialize $\tilde{\theta}_0 \leftarrow \theta_0$
for $t \in \{0, \ldots, T\}$ do
  send-server-param($\tilde{\theta}_t$)
  $\Delta \theta \leftarrow$ receive-param-update()
  $\tilde{\theta}_{t+1} \leftarrow \tilde{\theta}_t + \Delta \theta$
end for

3 Gossiping SGD

In a nutshell, the synchronous all-reduce algorithm consists of two repeating phases: (1) calculation of the local gradients at each node, and (2) exact aggregation of the local gradients via all-reduce. To derive gossiping SGD, we would like to replace the synchronous all-reduce operation with a more asynchronous-friendly communication pattern. The fundamental building block we use is a gossip aggregation algorithm \cite{16,17}, which combined with SGD leads to the gossiping SGD algorithm. Asynchronous gossiping SGD was introduced in \cite{10} for the general case of a sparse communication graph between nodes (e.g. wireless sensor networks). The original problem setting of gossiping also typically involved synchronous rounds of communication, whereas we are most interested in asynchronous gossip.

We can also derive the mathematical formulation of the gossiping SGD update by conceptually linking gossiping to elastic averaging. We introduce a distributed version of the global consensus objective in which the center parameter is replaced with the average of the local parameters:

$$F_{\text{dist-consensus}}(\theta_1, \ldots, \theta_p) = \sum_{i=1}^{p} \left[ f(\theta_i; X_i) + \frac{\rho}{2} \left\| \theta_i - \frac{1}{p} \sum_{j=1}^{p} \theta_j \right\|^2 \right].$$

(2)
If we replace the distributed mean $\frac{1}{p} \sum_{j=1}^{p} \theta_{t,j}$ with the unbiased one-node estimator $\theta_{t,j_{t,i}}$, such that $j_{t,i} \sim \text{Uniform}(\{1, \ldots, p\})$ and $\mathbb{E}[\theta_{t,j_{t,i}}] = \frac{1}{p} \sum_{j=1}^{p} \theta_{t,j}$, then we derive the gossiping SGD update:

$$\theta_{t,i} = \theta_{t,i} - \alpha \nabla f(\theta_{t,i}; X_i)$$

$$\theta_{t+1,i} = \theta_{t+1,i} - \beta (\theta_{t+1,i} - \theta_{t_{j_{t,i}},i})$$

$$= (1 - \beta) \theta_{t+1,i} + \beta \theta_{t,j_{t,i},i}.$$  

If $j_{t,i}$ is chosen uniformly as above, then the algorithm is equivalent to “pull-gossip,” i.e. each node pulls or receives $\theta_j$ from one and only one other random node per iteration. On the other hand, if we replace the “one-node estimator” with querying $\theta_j$ from multiple nodes, with the constraint that each $j$ is represented only once per iteration, then the algorithm becomes “push-gossip,” i.e. each node pushes or sends its own $\theta_t$ to one and only one other random node, while receiving from between zero and multiple other nodes. Push-gossiping SGD can be interpreted as an interleaving of a gradient step and a simplified push-sum gossip step [16]. Algorithms 3 and 4 describe pull-gossiping and push-gossiping SGD respectively.

**Algorithm 3 Pull-gossiping SGD.**

initialize $\theta_{0,i} \leftarrow \theta_0$

for $t \in \{0, \ldots, T\}$ do

if $t > 0$ and $t \equiv 0 \mod \tau$ then

set $x_i \leftarrow \theta_{t,i}$

choose a target $j$

$\theta_{t,i} \leftarrow \text{average of } x_i, x_j$

end if

$\Delta \theta_{t,i} \leftarrow -\alpha t \nabla f(\theta_{t,i}; X_{t,i}) + \mu \Delta \theta_{t-1,i}$

$\theta_{t+1,i} \leftarrow \theta_{t+1,i} + \Delta \theta_{t,i}$

end for

**Algorithm 4 Push-gossiping SGD.**

initialize $\theta_{0,i} \leftarrow \theta_0$

for $t \in \{0, \ldots, T\}$ do

if $t > 0$ and $t \equiv 0 \mod \tau$ then

set $x_i \leftarrow \theta_{t,i}$

choose a target $j$

send $x_i$ to $i$ (ourselves) and to $j$

$\theta_{t,i} \leftarrow \text{average of received } x_i$’s

end if

$\Delta \theta_{t,i} \leftarrow -\alpha t \nabla f(\theta_{t,i}; X_{t,i}) + \mu \Delta \theta_{t-1,i}$

$\theta_{t+1,i} \leftarrow \theta_{t+1,i} + \Delta \theta_{t,i}$

end for

4 Experiments

Figure 1: Center-crop validation loss and top-1 error on ImageNet over training wall-clock time and epochs. Shown are: (left) $p = 8$ nodes with per-node minibatch size $b = 32$, and (right) $p = 16$ nodes with per-node minibatch size $b = 16$.

4.1 Implementation

We implement the communication systems of gossiping SGD and other algorithms using Message Passing Interface (MPI) [18]. Because we wanted to run our code in cluster computing environments.
Figure 2: Center-crop validation loss and top-1 error on ImageNet over training wall-clock time and epochs, with different numbers of nodes $p$ and per-node minibatch size $b = 16$. Shown are: (left) $p = 32$ nodes, (middle) $p = 64$ nodes, and (right) $p = 128$ nodes.

with Infiniband or more specialized interconnects, then targeting MPI was the easiest solution. We targeted our code to run on GPUs, using the Nvidia CUDA 7.0 driver and using the cuBLAS and cuDNNv4 libraries for the core computational kernels.

For our experiments up to $p = 16$ nodes, we used a local cluster of 16 machines, each one consisting of an Nvidia Kepler K80 dual GPU, an 8-core Intel Haswell E5-1680v2 CPU, and a Mellanox ConnectX-3 FDR 4× Infiniband (56 Gb/s) NIC. We utilize only one GPU per K80.

For our larger scale experiments up to $p = 128$ nodes, we used a GPU supercomputer with over 10,000 total nodes. Nodes consist of an Nvidia Kepler K20X GPU and an 8-core AMD Bulldozer Opteron 6274 CPU, and are connected by a Cray Gemini interconnect in a 3D torus configuration.

4.2 Methodology

We chose ResNets for our neural network architecture; specifically, we trained ResNet-18, which is small enough to train rapidly, but also possesses features relevant to modern networks, including depth, residual layers, and batch normalization. We ran on the image classification problem of ImageNet consisting of 1.28 million training images and 50,000 validation images divided into 1000 classes. Our data augmentation is as follows: we performed multi-scale training by scaling the shortest dimension of images to between 256 and 480 pixels, we took random $224 \times 224$ crops and horizontal flips, and we added pixelwise color noise. We evaluate validation loss and top-1 error on center crops of the validation set images with the shortest dimension scaled to 256 pixels.

Unless otherwise noted, we initialized the step size to $\alpha = 0.1$, then we annealed it twice by a factor of 0.1 each. For our experiments with aggregate minibatch size $m = pb = 256$, we annealed at exactly 150k and 300k iterations into training (see section 4.3 for details on batch size settings). For our experiments with larger aggregate minibatch sizes, we decreased the number of iterations at which the step size was annealed. We used Nesterov momentum of $\mu = 0.9$ and weight decay of $\lambda = 10^{-4}$. For all-reduce and gossiping, we used a communication interval of $\tau = 1$, i.e. communication occurred every iteration. For elastic averaging, we set $\beta = 0.8/p$ and used both $\tau = 1$ and $\tau = 10$ (the latter is recommended in [9]).

4.3 Results

Our first set of experiments compare all-reduce, elastic averaging, and push-gossiping at $p = 8$ and $p = 16$ with an aggregate minibatch size $m = pb = 256$. The results are in Figure 1. For $p = 8$, elastic averaging with a communication delay $\tau = 10$ completes each iteration more quickly than the other methods. Interestingly, all-reduce has practically no synchronization overhead on the system.
at $p = 8$ and is as fast as gossiping. For $p = 16$, gossiping converges faster than elastic averaging with $\tau = 10$, and both come ahead of all-reduce. Additionally, elastic averaging with both $\tau = 1$ and $\tau = 10$ has trouble converging to the same validation loss as the other methods once the step size has been annealed to a small value ($\alpha = 0.001$ in this case).

We perform larger scale experiments at $p = 32$ nodes, $p = 64$ nodes, and $p = 128$ nodes on the GPU supercomputer. On the supercomputer environment, both elastic averaging and push-gossiping were not well matched to the remote-memory interface of MPI we used to implement the algorithms, so we only show results for synchronous SGD and pull-gossiping SGD. The results are in Figure 2. At this scale, we begin to see the scaling advantage of synchronous all-reduce SGD. One iteration of gossiping SGD is still faster than one iteration of all-reduce SGD, and gossiping works quickly at the initial step size. But gossiping SGD begins to converge much slower after the step size has annealed.

We note that the training time of SGD can be thought of as the product (wall-clock time per iteration) $\times$ (number of iterations). We made an observation consistent with [4]: letting synchronous all-reduce SGD run for many epochs, it will typically converge to a lower optimal validation loss than either elastic averaging or gossiping SGD. We found that letting all-reduce SGD run for over 1 million iterations with a minibatch size of 256 led to a peak top-1 validation accuracy of 68.7% (or 31.3% top-1 error). However, elastic averaging often had trouble breaking 67% accuracy using $p = 16$ nodes, as did gossiping using more than $p = 32$. In other words, at larger scales the asynchronous methods require more iterations to convergence despite lower wall-clock time per iteration.

5 Discussion

We are interested in how to attain the shortest time to train a deep neural net. In practice, this means scaling to multiple nodes. Because the speed of convergence varies with the number of iterations as well as the number of nodes, we posed the following questions:

(a) How fast do asynchronous and synchronous SGD algorithms converge at both the beginning of training (large step sizes) and at the end of training (small step sizes)?

Up to 32 nodes, asynchronous SGD can converge to a given accuracy level in fewer hours than all-reduce SGD when the step size is large. At small step size ($\alpha \lesssim 0.001$), gossiping can converge faster than elastic averaging, but all-reduce SGD converges most consistently.

(b) How does the convergence of SGD vary with the number of nodes?

Both elastic averaging and gossiping seem to converge faster than synchronous all-reduce SGD up to 16–32 nodes. Up to 128 nodes, all-reduce SGD can consistently converge to a high-accuracy solution, whereas asynchronous methods plateau at lower accuracy. In particular, the fact that gossiping SGD does not scale as well as does synchronous SGD with more nodes suggests that the asynchrony and the pattern of communication are responsible for the difference in convergence, rather than the amount of communication (both methods have low amounts of communication).

We note that scaling is complementary to other approaches to reducing the training time, such as designing new networks [25] and quantizing the gradients [12].

In our investigation we discovered that asynchronous methods were fastest on 8, 16, and 32 nodes. Above 32 nodes, synchronous SGD provided the fastest training. This is somewhat counterintuitive as one might naturally expect that the penalty of waiting on straggler nodes incurred during synchronous SGD would increase with the number of nodes, and that as a result synchronous SGD would not scale well. Finally, getting good scaling results above 128 nodes continues to be a challenge.

Acknowledgments

We thank Kostadin Ilov, Ed F. D’Azevedo, and Chris Fuson for helping make this work possible, as well as Josh Tobin for insightful discussions. We would also like to thank the anonymous reviewers for their constructive feedback. This research is partially funded by DARPA Award Number HR0011-12-2-0016 and by ASPIRE Lab industrial sponsors and affiliates Intel, Google, Hewlett-Packard, Huawei, LG, NVIDIA, Oracle, and Samsung. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
References


