This CVPR paper is the Open Access version, provided by the Computer Vision Foundation.
Except for this watermark, it is identical to the version available on IEEE Xplore.

FireCaffe: near-linear acceleration of deep neural network training on compute clusters

Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Kurt Keutzer
DeepScale* and UC Berkeley
{forresti, moskewcz, kashraf, keutzer}@eecs.berkeley.edu

Abstract

Long training times for high-accuracy deep neural networks (DNNs) impede research into new DNN architectures and slow the development of high-accuracy DNNs. In this paper we present FireCaffe, which successfully scales deep neural network training across a cluster of GPUs. We also present a number of best practices to aid in comparing advancements in methods for scaling and accelerating the training of deep neural networks. The speed and scalability of distributed algorithms is almost always limited by the overhead of communicating between servers; DNN training is not an exception to this rule. Therefore, the key consideration here is to reduce communication overhead wherever possible, while not degrading the accuracy of the DNN models that we train. Our approach has three key pillars. First, we select network hardware that achieves high bandwidth between GPU servers – Infiniband or Cray interconnects are ideal for this. Second, we consider a number of communication algorithms, and we find that reduction trees are more efficient and scalable than the traditional parameter server approach. Third, we optionally increase the batch size to reduce the total quantity of communication during DNN training, and we identify hyperparameters that allow us to reproduce the small-batch accuracy while training with large batch sizes. When training GoogLeNet and Network-in-Network on ImageNet, we achieve a 47x and 39x speedup, respectively, when training on a cluster of 128 GPUs.

1. Introduction and Motivation

Since the publication of AlexNet [17], a variety of new deep neural network (DNN) architectures such as GoogleNet [26], Network-in-Network [20], and VGG [24] have been developed at a rapid pace. This is natural, because with the training and testing dataset fixed (e.g. ImageNet-1k [9]), it is the DNN architecture that is primarily responsible for improvements in accuracy. In other words, the race for improvements in accuracy in image classification and other contemporary problems of computer science, has become a race in the development of new DNN architectures. So, what is the bottleneck in the development of new architectures?

In the development of new DNN architectures, as in any human research endeavor, creativity is a key element. However, the impact of architectural variations in DNNs – such as number of layers, filter dimensions, and so forth – can be hard to predict, and experimentation is required to assess their impact. A high-accuracy deep neural network (DNN) model such as GoogLeNet [26] can take weeks to train on a modern GPU. This is true even when leveraging deep neural network primitives like cuDNN [5], maxDNN [18], or fbfft [28] – all of which operate near the theoretical peak computation per second achievable on GPUs. Thus, training time is a key challenge at the root of the development of new DNN architectures.

This sentiment was voiced by Jeffrey Dean of Google in his recent keynote address [7]. The four key points that Dean makes are: (1) DNN researchers and users want results of experiments quickly. (2) There is a "patience threshold": No one wants to wait more than a few days or a week for a result. (3) This significantly affects scale of problems that can be tackled. (4) We sometimes optimize for experiment turnaround time, rather than absolute minimal system resources for performing the experiment.

As a particular example of where long training times are limiting the pace of DNN research and productization, consider the following. ImageNet-1k has 1.2 million training images, distributed across 1000 different category labels. From first-hand conversations with engineers and executives, we know that several internet companies have internal databases containing billions of images with hundreds of thousands of different category labels. Due to long training times, these companies are facing serious delays in bringing DNN-based solutions to market. Accelerated DNN training solutions would solve a major pain point for these companies.

So far, we have argued how accelerating DNN training

*http://deepscale.ai

would benefit applications where DNNs are in use today. Now, we consider ways in which accelerating DNN training would allow DNN-based techniques to be applied in entirely new ways. There are a number of situations where it is crucial to incorporate new data into a DNN model in real time. For example, reinforcement learning (RL) enables robots to learn things themselves with minimal supervision. A recent study by Levine *et al*. applied state-of-the-art DNN-based RL techniques to enable a robot to teach itself how to build lego structures and screw on bottle caps [19]. This technique is effective, and the robot does indeed learn to screw on bottle caps. However, it takes 3-4 hours for the robot to learn to screw on bottle caps, and the majority of this time is spent on DNN training. Faster DNN training would enable this and other RL applications to move toward real-time.

In our work, we focus directly on the problem of DNN training. Since single-GPU efficiency has reached the hard limits of the hardware, the next frontier for accelerating DNN training is to scale it across a compute cluster. In this paper, we present FireCaffe, which scales DNN training across a cluster of 128 GPUs with speedups of more than 40x compared to a single GPU. Our strategy for scaling up DNN training is to focus on reducing communication overhead, and we make a number of design choices toward this goal. For example, we use fast interconnects such as Infiniband or Cray Gemini to accelerate communication among the GPUs. We also show that reduction trees are a faster method for communication than using parameter servers. We map our parallelization strategy to high-accuracy DNN architectures.

The rest of the paper is organized as follows. In Section 2, we describe our choice of hardware for evaluating scalable DNN training, and Section 3 introduces key factors that we will use for analyzing communication among GPU workers. We describe tradeoffs between DNN parallelism strategies in Section 4, and Section 5 explains why certain high-accuracy DNN architectures are particularly amenable to parallelism. In Section 6, we describe our approach to efficiently implementing distributed DNN training. In Section 7, we describe good practices that facilitate the comparison of scalable DNN training techniques and we present our speedups for training the NiN and GoogLeNet architectures on ImageNet. Finally, we conclude in Section 8.

2. Hardware for scalable DNN training

It is both useful and possible to experiment with the scalability of DNN computations using theoretical or scale models. However, demonstration and verification of the correctness and real-world scalability of the proposed FireCaffe system requires using concrete hardware platforms. The speed at which data can be sent between nodes is a key consideration in selecting a hardware platform for scalable DNN training. This is because, the faster the interconnect between nodes is, the more scale we can achieve without being dominated by communication overhead. Hardware manufacturers such as Cray and Mellanox address this by developing high-bandwidth, low-latency interconnects that are substantially faster than typical Ethernet connections.

For example, the Titan supercomputer at Oak Ridge Leadership Computing Facility (OLCF) has a high bandwidth, low latency Cray Gemini interconnect for communication among servers. The Titan supercomputer has a total of 18000 servers, with one NVIDIA Kepler-based K20x GPU per server [2, 23]. With this in mind, we choose the OLCF Titan supercomputer for tuning and evaluating FireCaffe.

In this research, we use relatively small slices of the overall capacity of Titan for each training run. The additional computational capacity (~27 PetaFLOPS/s in total) enables us to conduct multiple training runs concurrently, where each training run utilizes 32 to 128 GPUs. When considering 32-node slices of Titan, we found that the interconnect speed (at least for the applications of this work) is similar to that provided by having all nodes in the slice connected to a single Infiniband-class switch.

3. Preliminaries and terminology

Deep neural network training is comprised of iterating between two phases: forward and backward propagation. In the forward phase, a batch of data items (e.g. images) is taken from the training set, and the DNN attempts to classify them. Then comes the backward phase, which consists of computing gradients with respect to the weights (∇W) and gradients with respect to the data (∇D). The weight gradients are used to update the model's weights. Then, an other forward phase is performed, and so on. We train models using batched stochastic gradient descent (SGD), which is the standard choice for popular DNN models such as GoogLeNet [26].

We now present a few preliminaries, which we will use later in the paper for reasoning about data volume to communicate in distributed DNN training. In Equation 1, we show how to calculate the total size (in bytes) of the weights in all convolutional and fully-connected layers, combined.

$$|W| = \sum_{L=1}^{\#layers} ch_L * numFilt_L * filterW_L * filterH_L * 4 \quad (1)$$

where ch is the number of channels, $numFilt$ is the number of filters, $filterH$ is the filter height, and $filterW$ is the filter width. Next, Equation 2 expresses the size of activations produced by all layers, combined.

$$|D| = \sum_{L=1}^{\#layers} ch_L * numFilt_L * dataW_L * dataH_L * batch * 4 \quad (2)$$

where $dataH$ is the activation map height, $dataW$ is the activation width, and $batch$ is the batch size. Note the $*4$ in

Equations 1 and 2 – this is because a floating-point number is 4 bytes.

To minimize confusion, we now define some terminology. In our terminology, the each following sets of words are synonyms: (weights = parameters = filters = W); (nodes = workers = GPU servers). We also sometimes use the terms "activations" and "data" (D) interchangeably. *Fully-connected layers* are a special case of convolutional layers where $filterH = dataH$ and $filterW = dataW$. We define an "epoch" as one pass through the training data. Finally, the word "performance" can be ambiguous, so we write in terms of specific metrics such as "accuracy" and "training time."

4. Parallelism strategies

There are two commonly-used methods for parallelizing neural network training across multiple servers: model parallelism (e.g. [29]) and data parallelism (e.g. [34]).

For batched SGD training of DNNs, we define *data parallelism* as the case where each worker (e.g. GPU) gets a subset of the batch, and then the workers communicate by exchanging weight gradient updates ∇W. We define *model parallelism* as the case where each worker gets a subset of the model parameters, and the workers communicate by exchanging data gradients ∇D and exchanging activations D. Note that $|W| = |\nabla W|$ and $|D| = |\nabla D|$; in other words, the weights and weight gradients are the same size; and the data and data gradients are the same size.

Now, to maximize DNN training scalability, our goal is to select a parallelism strategy that requires the lowest possible quantity of communication between servers. The choice of whether it is ideal to use data parallelism, model parallelism, or both depends strongly on the DNN's architectural characteristics. In computer vision, some of the most popular and accurate DNN models (e.g. GoogLeNet [26]) consist primarily of convolution layers, where the spatial resolution of the filters is smaller than the resolution of the activations. For these convolutional models, data parallelism is typically preferable because it requires less communication – that is, $|\nabla W|$ is much smaller than $|\nabla D|$ at typical batch sizes. Notice that the computer vision DNNs in Table 1 all have this property. In FireCaffe, we enable data parallelism across a cluster of GPUs, and we find that it produces ample speedups for training popular deep convolutional neural network architectures. We illustrate our data parallel approach in Figure 1. Each GPU contains a complete copy of the DNN model parameters. Each worker (GPU) gets a subset of each batch. The GPUs compute their share of the weight gradients. Once the gradients are calculated locally, they are added together using either a parameter server or a reduction tree communication (described in Section 6.2).

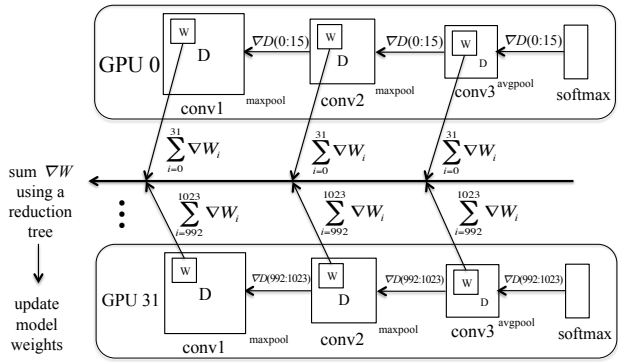


Figure 1. **Data parallel DNN training in FireCaffe:** Each worker (GPU) gets a subset of each batch.

Table 1. Volumes of data and computation for four widely-used DNN architectures. The batch size impacts all numbers in this table except for $|W|$, and we use a batch size of 1024 in this table.

DNN architecture	data_size $\|D\|$	weight_size $\|W\|$	data/weight ratio
NiN [20]	5800MB	30MB	195
AlexNet [16]	1680MB	249MB	10.2
GoogLeNet [26]	19100MB	54MB	358
VGG-19 [24]	42700MB	575MB	71.7

5. Choosing DNN architectures to accelerate

Of the popular deep convolutional neural network architectures for computer vision, some are more amenable to data parallel training than others. One might naïvely assume that DNN models with more parameters would produce higher classification accuracy. To evaluate this assumption, consider Figure 2, where we plot the total size of all parameters in bytes versus top-5 ImageNet accuracy for several popular DNN architectures. Observe that Nerwork-in-Network (NiN) [20] and AlexNet [17] have similar accuracy, while NiN has 8x fewer parameters than AlexNet. Likewise, GoogLeNet [26] and VGG [24] have similar accuracy, yet GoogLeNet has 10x fewer parameters. In data parallel training, $|\nabla W|$ is the quantity of data sent by each GPU worker, so DNN architectures with fewer parameters require less communication and are more amenable to training at large scale.

You may wonder, what are the architectural choices that led to NiN and GoogLeNet having 8-10x fewer parameters than AlexNet and VGG? The answer is twofold. First, GoogLeNet and NiN are more judicious in their use of filters with spatial resolution: many of the filters in GoogLeNet and NiN have a resolution of 1x1 instead of 3x3 or larger. Second, while VGG and AlexNet each have more than 150MB of fully-connected layer parameters, GoogLeNet has smaller fully-connected layers, and NiN does not have fully-connected layers.

In summary, models with fewer parameters are more amenable to scalability in data parallel training, while still delivering high accuracy. Therefore, for the rest of the pa-

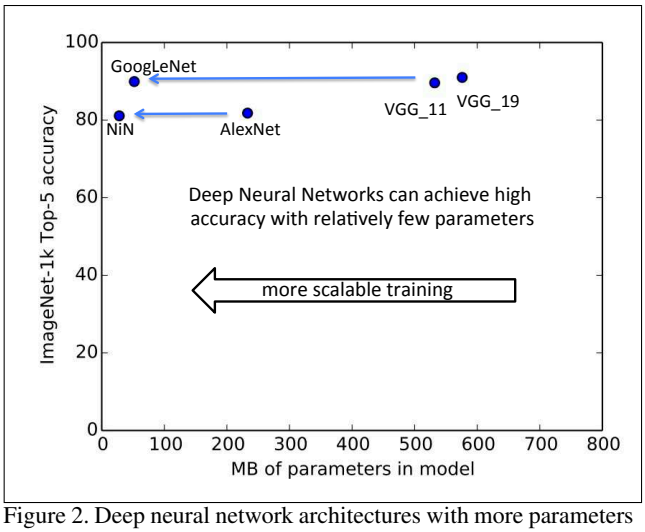


Figure 2. Deep neural network architectures with more parameters do not necessarily deliver higher accuracy.

per, we focus our efforts on accelerating the training of models with fewer parameters (e.g. NiN and GoogLeNet) while maintaining high accuracy.

6. Implementing efficient data parallel training

Our data-parallel distributed training strategy requires no communication among GPU workers in the forward pass. In the backward pass, a traditional single-GPU implementation (e.g. single-GPU Caffe [15]) sums the weight gradients over all images in the batch and then uses the weight gradient sum to update the model.[1] When we distribute the backward pass over a compute cluster, each GPU worker computes a sum of the weight gradients ($\sum \nabla W$) for its subset of the batch. Then, we sum the weight gradients across GPUs. This gradient aggregation scheme produces identical numerical results as you would find on a single GPU.

Now, our task is to find an efficient way to sum up ∇W among GPUs in a compute cluster. We consider two strategies for implementing this gradient aggregation: parameter servers, and reduction trees.

6.1. Parameter server

One strategy for communicating gradients is to appoint one node as a *parameter server*. The remaining *worker* nodes are each assigned a subset of the batch on which to perform forward and backward-propagation. After each backward pass, all the workers send their gradient updates to the parameter server. Then, the parameter server computes the sum of the gradients. Finally, the parameter server sends the summed gradients to the workers, and the workers apply these gradient updates to their local copies of the model. We illustrate the parameter server communication pattern in Figure 3(a).

[1] However, the data gradients (∇D) are not summed up.

The logical question here is, *what is the communication overhead of a parameter server, and how does that overhead scale as we increase the number of GPU workers?* Recall from Section 4 that each GPU worker provides $|W| = |\nabla W|$ bytes of weight gradients (Equation 1), which need to be summed with gradients from all other GPU workers. Now, the bottleneck is is in sending and receiving all the gradients on one parameter server. If there are p GPU workers, the parameter server is responsible for sending and receiving $|\nabla W| * p$ bytes of data. If each node (GPU worker or parameter server) can send and receive data at a rate of BW bytes/s, then we can calculate the minimum communication time as follows:

$$param_server_communication_time = \frac{|\nabla W| * p}{BW} (sec) \quad (3)$$

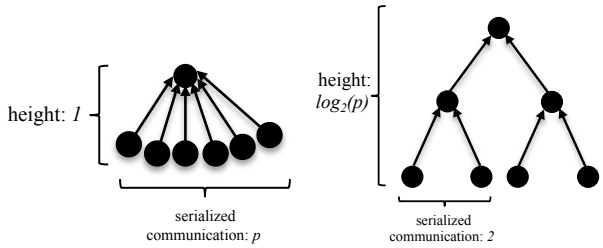
In other words, the parameter server's communication time scales linearly as we increase the number of GPU workers; doubling the number of workers leads to at least 2x more communication time per gradient update. We confirm this experimentally in Figure 4.

For the parameter server experiments in Figure 4, we have implemented a fully synchronous parameter server with the following characteristics. The parameter server is one arbitrarily-selected server in the cluster, while the other servers are workers; the parameter server and worker servers have identical hardware. After each batch, the workers send their weight gradients to the parameter server, the parameter server computes the sum, and then the parameter server sends the sum back to the workers.

There are a number of ways to augment the parameter server for greater scalability. For example, when having a single parameter server became a bottleneck, Microsoft Adam [6] and Google DistBelief [8] each defined a pool of nodes that collectively behave as a parameter server. The parameter server pool can be implemented hierarchically, where the servers aggregate gradients in a tree-like fashion [21]. In the next section we will take this idea to its logical conclusion, where every server is involved in parameter-gradient aggregation as well as computing DNN forward/backward passes.

6.2. Reduction tree

There are various common patterns of communication in parallel programs; among such common patterns, a frequently occurring one is *allreduce*. This pattern occurs when each worker produces one or more data values that must be globally reduced (generally with a commutative binary element-wise operator) to produce a single result value, and then this single value must be broadcast to all workers before they can continue. In this work, each worker produces a single vector of length $|\nabla W|$ (the gradient updates for that worker), which must be reduced using element-wise vector addition (to sum the per-worker gradient updates for each parameter). Since this computation



(a) parameter server (b) reduction tree

Figure 3. Illustrating how parameter servers and reduction trees communicate weight gradients. In this figure, we only show the summing-up of weight gradients. We distribute the weight gradient sums by going back down the tree.

exactly fits the *allreduce* communication pattern it is convenient to use existing library support for such operations. While there are many possible implementations of *allreduce*, most share the key property that the time taken to perform the operation scales as the log of the number of workers (at least for large numbers of workers). Intuitively, this is because *allreduce* algorithms use binomial reduction tree and/or butterfly communication patterns internally [27]. Out of the possible allreduce implementation strategies, we find that the binomial reduction tree is particularly easy to reason about on a theoretical level. So, for the rest of this section, we focus on allreduce communication implemented with a reduction tree.

In Figures 3(a) and 3(b), we present the intuition on how parameter servers and reduction trees differ. We might think of a parameter server as a reduction tree with a height of 1 and a branching factor of p. However, many cluster computers and supercomputers have several dimensions of network fabric among nodes (e.g. an *N-D Torus*), which enable nodes to talk to each other via many different paths. With this in mind, we can sum gradients using a taller reduction tree, where nodes collaboratively sum the gradients. For example, consider a binary communication tree with a branching factor of 2 and a depth of $log_2(p)$. In this case, the serialized communication is $2log_2(p)$; the outer 2 represents the fact that each node receives data from 2 children, and the $log_2(p)$ is the height of the tree. Therefore, unlike the parameter server model, the reduction tree's communication time is:

$$reduction_tree_communication_time = \frac{|\nabla W| * 2log_2(p)}{BW}(sec) \quad (4)$$

In practice, the base of $log(p)$ depends on the branching factor in the reduction tree, but the basic idea here is straightforward: While the parameter server communication overhead scales *linearly* with p, reduction tree communication is much more efficient because it scales *logarithmically* as $O(log(p))$. We confirm that reduction trees scale more efficiently than parameter servers in Figure 4.

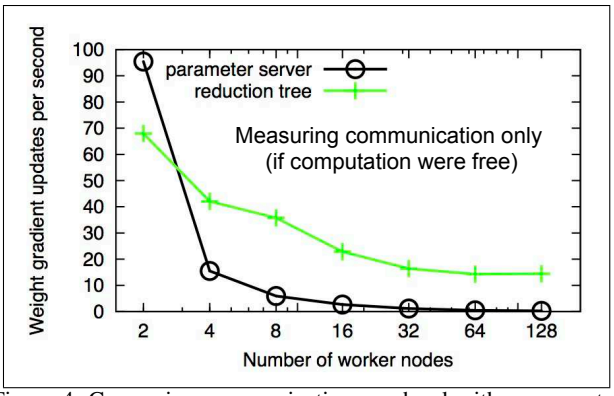


Figure 4. Comparing communication overhead with a parameter server vs. a reduction tree. This is for the Network-in-Network DNN architecture, so each GPU worker contributes 30MB of gradient updates.

7. Evaluation of FireCaffe-accelerated training on ImageNet

In this section, we evaluate how FireCaffe can accelerate DNN training on a cluster of GPUs. We train GoogLeNet [26] and Network-in-Network [20] on up to 128 GPU servers in the Titan supercomputer (described in Section 2), leveraging FireCaffe's reduction tree data parallelism (Section 6.2). We begin by describing our evaluation methodology, and then we analyze the results.

7.1. Evaluation Methodology

We now describe a few practices that aid in comparing advancements in accelerating the training of deep neural networks.

1. Evaluate the speed and accuracy of DNN training on a publicly-available dataset.

In a recent study, Azizpour *et al*. applied DNNs to more than 10 different visual recognition challenge datasets, including human attribute prediction, fine-grained flower classification, and indoor scene recognition [3]. The accuracy obtained by Azizpour *et al*. ranged from 56% on scene recognition to 91% on human attribute prediction. As you can see, the accuracy of DNNs and other machine learning algorithms depends highly on the specifics of the application and dataset to which they are applied. Thus, when researchers report improvements in training speed or accuracy on proprietary datasets, there is no clear way to compare the improvements with the related literature. For example, Amazon [25] and Baidu [30][2] each reported their training speedups on a proprietary dataset, so it's not clear how to compare these results with the related literature. In contrast, we conduct our evaluation on a publicly-available

[2]Baidu evaluated their training times using a proprietary dataset [30]. Baidu also did some ImageNet experiments, but Baidu did not report the training time on ImageNet.

dataset, ImageNet-1k [9], which contains more than 1 million training images.ImageNet-1k is a widely-studied dataset, so we can easily compare our accuracy, training speed, and scalability results with other studies that use this data.

2. Report hyperparameter settings such as weight initialization, momentum, batch size, and learning rate.
Glorot *et al*. [12], Breuel [4], and Xu *et al*. [32] have each shown that seemingly-subtle hyperparameter settings such as weight initialization can have a big impact on the speed and accuracy produced in DNN training. When training Network-in-Network (NiN) [20], we initialize the weights with a gaussian distribution centered at 0, and we set the standard deviation (std) to 0.01 for 1x1 convolution layers, and we use std=0.05 for other layers. For NiN, we initialize the bias terms to a constant value of 0, we set the weight decay to 0.0005, and we set momentum to 0.9. These settings are consistent with the Caffe configuration files released by the NiN authors [20].

Frustratingly, in Google's technical reports on GoogLeNet [26, 14], training details such as batch size, momentum, and learning rate are not disclosed. Fortunately, Wu *et al*. [31] and Guadarrama [13] each reproduced GoogLeNet and released all the details of their training protocols. As in [13], we train GoogLeNet with momentum=0.9 and weight decay=0.0002, we use xavier [12] weight initialization, and we initialize the bias terms to a constant value of 0.2. We will address learning rate and batch size settings in the following sections.

Given a DNN architecture, there are a number of strategies that can further increase accuracy, albeit at a substantial computational cost. One such strategy is to train multiple independent copies of a DNN architecture (e.g. GoogLeNet), each with a different random number generator seed for initializing the parameters. At test time, these DNNs can be used as an *ensemble* – that is, all DNNs are run on the test data, and for each test data item, the DNN's classification activations are averaged. For example, using an ensemble of 7 GoogLeNet DNNs, Szegedy *et al*. achieved a 2 percentage-point accuracy improvement on ImageNet, compared to a single GoogLeNet baseline [26]. An other such technique is to augment the data by adding deformations or color variations during training and/or testing [30]. Our focus in this paper is to show speedup on training single models and compare with reported baselines. Hence we avoid using exotic data augmentation or ensembles of multiple DNNs. In our experiments, we resize images to 256x256; at training time we use a 224x224 crop with a randomized offset, and at test time we classify the 224x224 crop in the center of the image; these settings are also commonly used in the AlexNet [17] and Network-in-Network [20] DNN architectures.

3. Measure *speedups* with respect to a single-server baseline.
In order to meaningfully measure how much we have accelerated DNN training by adding more GPUs, we must have a representative baseline, e.g. with a single GPU. When reporting results, we begin by considering time required to train a DNN on single GPU, and we report our multi-GPU speedups with respect to this single-GPU baseline. A recent study by Microsoft [6] reported training a custom DNN architecture (e.g. not GoogLeNet or NiN) on a cluster of CPU servers. This may sound impressive, but Microsoft did not report the time that the model would take to train on a single server. It could be that Microsoft achieved a 10x speedup by going from 1 server to 10 servers, or the speedup could be 2x – this isn't clear from the information provided in Microsoft's paper. This illustrates the importance of measuring the speed of scalable DNN training systems with respect to a single-server baseline.

4. Measure *accuracy* with respect to a single-server baseline.
In our experience, if hyperparameters such as learning rate and batch size are selected too aggressively, a DNN model may converge quickly, but fall short of the state-of-art accuracy. Therefore, in our experiments, we train multi-GPU models until they reach to the single-GPU accuracy baseline; this validates that we can accelerate DNN training without degrading accuracy. However, in cluster-scale multi-GPU training experiments by Baidu [30] and Flickr [22], the training is stopped prematurely before the DNNs converge. This leaves us wondering whether the Baidu and Flickr multi-GPU training experiments would have reproduced the accuracy produced on a single GPU. To avoid this type of confusion, we evaluate both the speed and accuracy of FireCaffe DNN training with respect to a single-server/single-GPU baseline.

7.2. Results: Midsized deep models

Using the settings described by Krizhevsky [17], we find that AlexNet achieves 58.9% top-1 ImageNet-1k accuracy after 100 epochs of training. After just 47 epochs of training, we find that NiN also converges to 58.9% top-1 accuracy. Each training iteration of NiN is more time-consuming than AlexNet, and AlexNet and NiN both take approximately 6 days to converge to this level of accuracy.

At Google, Krizhevsky developed a scheme for accelerating AlexNet training using multiple GPUs within a single server [16]. Krizhevsky's strategy uses data parallelism in convolutional layers and model parallelism in fully-connected layers. As we show in Table 2, Krizhevsky achieves near-linear acceleration up to 8 GPUs, but it has not been shown to scale beyond a single server. For reasons that we don't entirely understand, Krizhevsky's accu-

racy drops by 1.8 percentage points when doing multi-GPU training [16].

In FireCaffe, we scale NiN training to 32 GPUs, which is the scale at which we find communication time and computation are approximately equal[3]. We begin by using the learning rate and batch size settings that were reported in the Caffe configuration file released by the NiN authors [20]: For a batch size of 256, we use an initial learning rate of 0.01, and we reduce this by a factor of 10x twice during our training. Using this configuration, we reproduce the single-GPU NiN accuracy in 11 hours (13x speedup) when training on 32 GPUs.

For a fixed number of epochs, increasing the batch size reduces the number of times we need to communicate weight gradients, thus reducing the overall training time. With this in mind, we now train NiN with a batch size of 1024.[4] As in [16] when we increase the batch size, we increase the learning rate by an equal proportion. For example, when we use a batch size of 1024, we initialize the learning rate to 0.04. In this configuration, we train NiN in just 6 hours (23x speedup) on 32 GPUs. By increasing the batch size to 1024, we achieved a substantial speedup, but this came at the price of reducing the final accuracy by $\frac{3}{10}$ of a percentage point. We expect that this $\frac{3}{10}\%$ of accuracy could be regained at a batch size of 1024 – while retaining a substantial speed advantage – by training for a few more epochs. Finally, on 128 GPUs, we achieve a 39x speedup over single-GPU training.

So far, we have compared FireCaffe to the cuda-convnet2 framework from Google [16], which runs on a single-server/multi-GPU platform but not in a multi-server distributed platform. Google has also developed the TensorFlow framework [1] which reportedly supports distributed DNN training, but Google has not released speed benchmarks on TensorFlow. Twitter [11] has also experimented with scaling DNN training to 8 GPUs, but speed and accuracy results have not been released. Tencent [34], Theano [10], and Facebook [33] have published AlexNet single-server/multi-GPU training times that are slower than Google [16]. Other than FireCaffe, we have not seen literature on training AlexNet/NiN-scale models in a multi-server/multi-GPU setting. On 32 GPUs, FireCaffe is at least 3x faster to train AlexNet/NiN-scale models than all of the aforementioned results.

7.3. Results: Ultra-deep models

Ultra-deep models such as GoogLeNet can produce higher accuracy, but they present an even bigger challenge in terms of training time. Internally, Google has trained GoogLeNet on a cluster of CPU servers, but they have not reported the time required to complete this training [26, 14]. Fortunately, Guadarrama reproduced GoogLeNet in Caffe, and he released his GoogLeNet Caffe configuration files [13]. Guadarrama trained for 64 epochs using a batch size of 32 and an initial learning rate of 0.01, and we use these settings in our single-GPU GoogLeNet training experiments. Instead of occasionally reducing the learning rate by 10x, Guadarrama used a polynomial learning rate – that is, the learning rate is gradually reduced after every iteration of training. More specifically, at a given iteration of training, the learning rate is calculated as $initialLearningRate(1 - \frac{iter}{max_iter})^{power}$, and we set $power$ to 0.5 in all of our GoogLeNet training runs. Running this in Caffe on a single-GPU, GoogLeNet takes 21 days to train on ImageNet-1k, producing 68.3% top-1 accuracy and 88.7% top-5 accuracy. This is slightly lower than the 89.9% top-5 single-model accuracy reported by Google [26], and it will be interesting to see whether the open-source Caffe community will eventually be able reproduce or surpass Google's GoogLeNet accuracy. Here, we use the single-GPU Caffe GoogLeNet accuracy (88.7% top-5 accuracy) as a baseline, and we aim to reproduce this rapidly on a cluster of GPUs.

Now, we consider how to accelerate GoogLeNet training using FireCaffe. We initially tried to run GoogLeNet with a batch size of 32 on a GPU cluster, but there just wasn't enough work per batch to keep a GPU cluster saturated. As we learned earlier in the paper, larger batch sizes lead to less frequent communication and therefore enable more scalability in a distributed setting. When modifying the batch size, Breuel [4] and Krizhevsky [16] found that the choice of learning rate is crucial in order to preserve high accuracy. We trained five separate versions of GoogLeNet, each with a different initial learning rate (LR): {0.02, 0.04, 0.08, 0.16, and 0.32}, and all with a batch size of 1024. With LR=0.16 and LR=0.32, GoogLeNet failed to learn beyond random-chance accuracy on the test set. Using LR=0.02 produced 66.1% top-1 ImageNet-1k accuracy, and LR=0.04 produced 67.2%. Finally, we declare victory with LR=0.08, where we achieved 68.3% accuracy (again, with a batch size of 1024), which matches the accuracy of the baseline that used a batch size of 32. With a batch size of 1024 and a fixed number of epochs, we find that FireCaffe on 32 GPUs can train GoogLeNet 23x faster than a single GPU. When we move from a batch size of 32 with LR=0.01 to a batch size of 1024 with LR=0.08, we find that GoogLeNet takes a few more epochs to converge (72 epochs instead of 64 epochs), so the absolute training speedup is 20x; we show these results in Table 3. In other words, FireCaffe can train GoogLeNet in 23.4 hours on 32 GPUs, compared to 21 days on a single GPU. Finally, on 128 GPUs, we achieve a 47x speedup over single-GPU GoogLeNet training, while matching the single-GPU accuracy.

[3] at a batch size of 1024

[4] While keeping a fixed number of epochs. In other words, with a batch size of 1024, we perform 4x fewer training iterations than with a batch size of 256.

Table 2. Accelerating the training of midsized deep models on ImageNet-1k.

	Hardware	Net	Epochs	Batch size	Initial Learning Rate	Train time	Speedup	Top-1 Accuracy
Caffe [15]	1 NVIDIA K20	AlexNet [17]	100	256	0.01	6.0 days	1x	58.9%
Caffe	1 NVIDIA K20	NiN [20]	47	256	0.01	5.8 days	1x	58.9%
Google cuda-convnet2 [16]	8 NVIDIA K20s (1 node)	AlexNet	100	varies	0.02	16 hours	7.7x	57.1%
FireCaffe (ours)	32 NVIDIA K20s (Titan supercomputer)	NiN	47	256	0.01	11 hours	13x	58.9%
FireCaffe-batch1024 (ours)	32 NVIDIA K20s (Titan supercomputer)	NiN	47	1024	0.04	6 hours	23x	58.6%
FireCaffe-batch1024 (ours)	128 NVIDIA K20s (Titan supercomputer)	NiN	47	1024	0.04	3.6 hours	**39x**	58.6%

Table 3. Accelerating the training of ultra-deep, computationally intensive models on ImageNet-1k.

	Hardware	Net	Epochs	Batch size	Initial Learning Rate	Train time	Speedup	Top-1 Accuracy	Top-5 Accuracy
Caffe	1 NVIDIA K20	GoogLeNet [26]	64	32	0.01	21 days	1x	68.3%	88.7%
FireCaffe (ours)	32 NVIDIA K20s (Titan supercomputer)	GoogLeNet	72	1024	0.08	23.4 hours	20x	68.3%	88.7%
FireCaffe (ours)	128 NVIDIA K20s (Titan supercomputer)	GoogLeNet	72	1024	0.08	10.5 hours	**47x**	68.3%	88.7%

8. Conclusions

Long training times impose a severe limitation on progress in deep neural network research and productization. Accelerating DNN training has several benefits. First, faster DNN training enables models to be trained on ever-increasing dataset sizes in a tractable amount of time. Accelerating DNN training also enables product teams to bring DNN-based products to market more rapidly. Finally, there are a number of compelling use-cases for real-time DNN training, such as robot self-learning. These and other compelling applications led us to focus on the problem of accelerating DNN training, and our work has culminated in the FireCaffe distributed DNN training system.

Our approach to accelerating DNN training at scale has three key pillars. First, we select network hardware that achieves high bandwidth between GPU servers – Infiniband or Cray interconnects are ideal for this. Second, when selecting a communication algorithm, we find that reduction trees are more efficient and scalable than the traditional parameter server approach. Third, we optionally increase the batch size to reduce the total quantity of communication during DNN training, and we identify hyperparameters that allow us to reproduce the small-batch accuracy while training with large batch sizes. These three pillars helped us to achieve a near-linear speedup for a number of leading deep neural network architectures. In particular, we have achieved 39x speedup on NiN training, and a 47x speedup on GoogLeNet training on a 128 GPU cluster.

Acknowledgments

Research partially supported by DARPA award HR0011-12-2-0016, DOE contract DE-AC05-00OR22725, NSF award 1251276, and industrial sponsors Intel, Google, Huawei, Nokia, NVIDIA, Oracle, and Samsung. Thanks to Bryan Catanzaro, Evan Shelhamer, Trevor Darrell, Sergio Guadarrama, Christian Szegedy, Thomas Breuel, Kostadin Ilov, and Aditya Devarakonda for helpful discussions.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. *Google Technical Report*, 2015. 7

[2] V. Anantharaj, F. Foertter, W. Joubert, and J. Wells. Approaching exascale: application requirements for olcf leadership computing. https://www.olcf.ornl.gov/wp-content/uploads/2013/01/OLCF_Requirements_TM_2013_Final1.pdf, 2013. p43–45. 2

[3] H. Azizpour, A. S. Razavian, J. Sullivan, A. Maki, and S. Carlsson. From generic to specific deep representations for visual recognition. In *CVPR Deep Vision Workshop*, 2015. 5

[4] T. M. Breuel. The effects of hyperparameters on SGD training of neural networks. *arXiv:1508.02788*, 2015. 6, 7

[5] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: efficient primitives for deep learning. *arXiv:1410.0759*, 2014. 1

[6] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: building an efficient and scalable deep learning training system. In *OSDI*, 2014. 4, 6

[7] J. Dean. Keynote: Large scale deep learning. In *ACM Conference on Information and Knowledge Management (CIKM)*, 2014. 1

[8] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012. 4

[9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *CVPR*, 2009. 1, 6

[10] W. Ding, R. Wang, F. Mao, and G. W. Taylor. Theano-based large-scale visual recognition with multiple gpus. *arXiv:1412.2302*, 2014. 7

[11] C. Farabet. Real-time, content-driven representations at twitter. In *GPU Technology Conference*, 2015. 7

[12] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *JMLR*, 2010. 6

[13] S. Guadarrama. BVLC googlenet. https://github.com/BVLC/caffe/tree/master/models/bvlc_googlenet, 2015. 6, 7

[14] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *JMLR*, 2015. 6, 7

[15] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv:1408.5093*, 2014. 4, 8

[16] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv:1404.5997*, 2014. 3, 6, 7, 8

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012. 1, 3, 6, 8

[18] A. Lavin. maxDNN: an efficient convolution kernel for deep learning with maxwell gpus. *arXiv:1501.06633*, 2015. 1

[19] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *arXiv:1504.00702*, 2015. 2

[20] M. Lin, Q. Chen, and S. Yan. Network in network. *arXiv:1312.4400*, 2013. 1, 3, 5, 6, 7, 8

[21] L. Mai, C. Hong, and P. Costa. Optimizing network performance in distributed machine learning. In *HotCloud*, 2015. 4

[22] C. Noel, J. Shi, and A. Feng. Large scale distributed deep learning on hadoop clusters. 2015. http://yahoohadoop.tumblr.com/post/129872361846. 6

[23] Oak Ridge Leadership Computing Facility (OLCF). Introducing titan: Advancing the era of accelerated computing. http://www.olcf.ornl.gov/titan, 2015. 2

[24] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2014. 1, 3

[25] N. Strom. Scalable distributed dnn training using commodity gpu cloud computing. In *INTERSPEECH*, 2015. 5

[26] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *arXiv:1409.4842*, 2014. 1, 2, 3, 5, 6, 7, 8

[27] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 2005. 5

[28] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun. Fast convolutional nets with fbfft: A gpu performance evaluation. *arXiv:1412.7580*, 2014. 1

[29] J. Wawrzynek, K. Asanovic, B. Kingsbury, D. Johnson, J. Beck, and N. Morgan. Spert-ii: A vector microprocessor system. *Computer*, 1996. 3

[30] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun. Deep image: Scaling up image recognition. *arXiv:1501.02876*, 2015. 5, 6

[31] Z. Wu, Y. Zhang, F. Yu, and J. Xiao. A gpu implementation of googlenet. http://vision.princeton.edu/pvt/GoogLeNet, 2014. 6

[32] B. Xu, N. Wang, T. Chen, and M. Li. Empirical evaluation of rectified activations in convolutional network. *arXiv:1505.00853*, 2015. 6

[33] O. Yadan, K. Adams, Y. Taigman, and M. Ranzato. Multi-gpu training of convnets. *arXiv:1312.5853*, 2013. 7

[34] Y. Zou, X. Jin, Y. Li, Z. Guo, E. Wang, and B. Xiao. Mariana: Tencent deep learning platform and its applications. In *VLDB*, 2014. 3, 7