AI-Systems
Distributed Training

Joseph E. Gonzalez
Co-director of the RISE Lab
jegonzal@cs.berkeley.edu
What is? & Why? Distributed Training

- **Distributed Training** ~ Training across multiple devices
  - Different local and remote memory speeds / network

- Why do we need distributed training?
  - Faster training by leveraging **parallel computation**
  - **Additional memory** (memory bandwidth) for larger model
    - “Need” to store weights + activations
  - Reduce or eliminate **data movement**
    - Privacy → Federated Learning
    - Limited bandwidth to edge devices
  - Need to process all the data?

*Very simplified definition.*
On Dataset Size and Learning

- Data is a resource! (e.g., like processors and memory)
  - Is having lots of processors a problem?

- You don’t have to use all the data!
  - Though using more data can often help

- More data often* dominates models and algorithms

*More data also supports more sophisticated models and algorithms.
What are the Metrics of Success?

- **Marketing Team**: Maximize number of GPUs/CPUs used
  - A bad metric ... why?

- **Machine Learning**: Minimize passes through the training data
  - Easy to measure, but not complete ... why?

- **Systems**: minimize time to complete a pass through the training data
  - Easy to measure, but not complete ... why?
Ideal Metric of Success

\[
\left( \frac{\text{“Learning”}}{\text{Second}} \right) = \left( \frac{\text{“Learning”}}{\text{Record}} \right) \times \left( \frac{\text{Record}}{\text{Second}} \right)
\]

- Convergence Property
- Machine Learning Property
- Throughput System Property
Metrics of Success

- Minimize training time to “best model”
  - Best model measured in terms of test error

- Other Concerns?
  - **Complexity**: Does the approach introduce additional training complexity (e.g., hyper-parameters)?
  - **Stability**: How consistently does the system train the model?
  - **Cost**: Will obtaining a faster solution cost more money (power)?
Map-Reduce for Distributed Training

Learning by Distributed Aggregation
Learning from Statistics (Aggregation)*

• Chu et al., *Map-Reduce for Machine Learning on Multicore*. NIPS’06.

*next set of slides are old!
Can we compute

\[ \hat{\theta} = (X^T X)^{-1} X^T Y \]

using the statistical query pattern in map-reduce?
Can we compute using the statistical query pattern in map-reduce?

\[
\hat{\theta} = (X^T X)^{-1} X^T Y
\]

Break computation into two queries

\[\Sigma = \bigoplus_{r \in \text{Data}} f_\theta(r)\]
Cost Analysis

\[ \hat{\theta} = (X^T X)^{-1} X^T Y \]

Computation

<table>
<thead>
<tr>
<th>Expression</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A = X^T X)</td>
<td>(O(np^2))</td>
</tr>
<tr>
<td>(C = X^T Y)</td>
<td>(O(np))</td>
</tr>
</tbody>
</table>

When \(n >> p\) we want to distribute this computation

<table>
<thead>
<tr>
<th>Expression</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B = A^{-1})</td>
<td>(O(p^3))</td>
</tr>
<tr>
<td>(BC)</td>
<td>(O(p^2))</td>
</tr>
</tbody>
</table>
What about Logistic Regression using Gradient Descent?
Logistic Regression in Map-Reduce

Gradient descent:

$$f_w(x, y) = \nabla \log L(y, h_w(x))$$

Learning Algorithm

Update Model:

$$w \leftarrow w - \eta_t g$$

Query: $f_w$

$$g = \frac{1}{n} \sum_{i=1}^{n} f_w(x_i, y_i)$$

System

Data
Map-Reduce is not optimized for iteration and multi-stage computation.

Query: $f_w$

$$g = \frac{1}{n} \sum_{i=1}^{n} f_w(x_i, y_i)$$
Iteration in Map-Reduce

Initial Model

Training Data

Map

Reduce

Learned Model

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$

$w^{(0)}$

$w^{(1)}$

$w^{(2)}$

$w^{(3)}$
Cost of Iteration in Map-Reduce

1. Initial Model
2. Training Data
3. Read 1
4. Read 2
5. Read 3
6. Map
7. Reduce
8. Learned Model

Repeatedly
load same data
Cost of Iteration in Map-Reduce

Redundantly save output between stages
Iteration and Multi-stage computation

In-Memory Dataflow System


Dataflow View

Training Data (HDFS) → Map → Reduce → 

Map → Reduce → 

Map → Reduce → 

19
Memory Opt. Dataflow

10-100× faster than network and disk
Memory Opt. Dataflow View

Efficiently move data between stages
Statistical Inference in Large Latent Variable Models

- Large topic models associated variables with each word and document

Not a good fit for BSP model
Bulk Synchronous Parallel (BSP) Execution
Asynchronous Execution

Enable more frequent coordination on parameter values
Asynchronous Execution

Enable more frequent coordination on parameter values
Asynchronous Execution

Parameter Server (Logical)

Machine 1

Iteration  Iteration  Iteration  Iteration

Machine 2

Iteration  Iteration  Iteration

Machine 3

Iteration  Iteration  Iteration  Iteration  Iteration
AlexNet
ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky, Illya Sutskever, Geoffrey E. Hinton

TL;DR; This paper describes the deep convolutional architecture, training techniques, and system innovations that resulted in the winning entry for the ILSVRC-2012 Benchmark. This model substantially outperformed the next best model that year.
The AlexNet* Architecture

*Posthumously Named
The \textbf{Actual AlexNet} Architecture

from the paper

---

4 Reducing Overfitting

Our neural network architecture has 60 million parameters. Although the 1000 classes of ILSVRC make each training example impose 10 bits of constraint on the mapping from image to label, this turns out to be insufficient to learn so many parameters without considerable overfitting. Below, we describe the two primary ways in which we combat overfitting.

4.1 Data Augmentation

The easiest and most common method to reduce overfitting on image data is to artificially enlarge the dataset using label-preserving transformations (e.g., \cite{25,4,5}). We employ two distinct forms of data augmentation, both of which allow transformed images to be produced from the original images with very little computation, so the transformed images do not need to be stored on disk. In our implementation, the transformed images are generated in Python code on the CPU while the GPU is training on the previous batch of images. So these data augmentation schemes are, in effect, computationally free.

The first form of data augmentation consists of generating image translations and horizontal reflections. We do this by extracting random $224 \times 224$ patches (and their horizontal reflections) from the $256 \times 256$ images and training our network on these extracted patches. This increases the size of our training set by a factor of 2048, though the resulting training examples are, of course, highly interdependent. Without this scheme, our network suffers from substantial overfitting, which would have forced us to use much smaller networks. At test time, the network makes a prediction by extracting five $224 \times 224$ patches (the four corner patches and the center patch) as well as their horizontal reflections (hence ten patches in all), and averaging the predictions made by the network's softmax layer on the ten patches.

The second form of data augmentation consists of altering the intensities of the RGB channels in training images. Specifically, we perform PCA on the set of RGB pixel values throughout the ImageNet training set. To each training image, we add multiples of the found principal components.

---

*Posthumously Named*
Training on Multiple GPUs

- Limited by GPU memory using Nvidia GTX 580 (3GB RAM)
  - 60M Parameters ~ **240 MB**
  - Need to cache activation maps for backpropagation
    - Batch size = 128
    - \(128 \times (227 \times 227 \times 3 + 55 \times 55 \times 96 \times 2 + 96 \times 27 \times 27 \times 2 + 256 \times 27 \times 27 \times 2 + 256 \times 13 \times 13 \times 2 + 13 \times 13 \times 384 \times 2 + 256 \times 13 \times 13 + 6 \times 6 \times 256 + 4096 + 4096 + 1000) \times 4 \text{ Bytes} \approx 782 \text{MB Activations}
  - That is assuming no overhead and single precision values

- Tuned splitting across GPUs to balance communication and computation

The heuristic which we followed was to divide the learning rate by 10 when the validation error stopped improving with the current learning rate. The learning rate was initialized at 0.01 and weight decay of 0.0005. We found that this small amount of weight decay was important for the model to learn. In other words, weight decay here is not merely a regularizer: it reduces the model’s training error. The update rule for weight decay is:

\[ \frac{v}{2} = \frac{v}{2} - \epsilon \cdot \frac{\partial L}{\partial w} \]

where \( \epsilon \) is the momentum variable, \( v \) is the average over the weight \( w \), and \( L \) is the loss. We initialize the neuron biases in the second, fourth, and fifth convolutional layers, as well as in the fully-connected hidden layers, with the constant 0. This initialization accelerates the early stages of learning by providing the ReLUs with positive inputs.

We initialized the weights in each layer from a zero-mean Gaussian distribution with standard deviation 0.1. Therefore to each RGB image pixel \((i, j, k)\), we add the following quantity:

\[ w_i \cdot xy \]

where \( w_i \) is the \( i \)-th eigenvector and eigenvalue of the covariance matrix of RGB pixel values. At test time, we use all the neurons but multiply their outputs by 0.5, which is a reasonable approximation to taking the geometric mean of the predictive distributions produced by the exponentially-many dropout networks.

**4.2 Dropout**

We use dropout in the first two fully-connected layers of Figure 2. Without dropout, our network exhibits substantial overfitting. Dropout roughly doubles the number of iterations required to converge. Since a neuron cannot rely on the presence of particular other neurons, it is, therefore, forced to learn more general features. The mathematical details are as follows.

At training time, a neuron is randomly set to zero with probability \( p \) for each training input. At test time, we use all the neurons but multiply their outputs by \( 0.5 \).

Dropout can be thought of as a way to randomly perturb the weights and biases of the network, thereby reducing the effective number of weights and biases that the network learns. The effect of dropout is to force the network to learn more general features, which can lead to improved generalization.

Combining the predictions of many different models is a very successful way to reduce test errors. One such technique is to randomly drop out a subset of the neurons during training. This has been shown to improve the performance of deep neural networks.

Dropout is similar to other regularization methods, such as weight decay and early stopping, but it has some important differences. For example, dropout reduces the effective number of parameters in the network, which can help to prevent overfitting. In contrast, weight decay reduces the magnitude of the weights, which can also help to prevent overfitting.

The recently-introduced technique, called “dropout” [10], consists of setting to zero the output of each hidden neuron with probability \( 0.5 \). The neurons which are “dropped out” in this way do not contribute to the forward pass and do not participate in back-propagation. So every time an input is presented, the neural network samples a different architecture, namely, that object identity is invariant to changes in the intensity and color of the illumination. This point it is re-drawn. This scheme approximately captures an important property of natural images, where

\[ I \]

is the input image, \( p \) is the probability of keeping a neuron, and \( E \) is a function that aggregates the outputs of the neurons. The neurons which are “dropped out” are kept for the forward pass and back-propagation.

Figure 3: 96 convolutional kernels of size \( 11 \times 11 \times 3 \) learned by the first convolutional layer on the \( 224 \times 224 \times 3 \) input images. The top 48 kernels were learned on GPU 1 while the bottom 48 kernels were learned on GPU 2. See Section 6.1 for details.
Put into historical context

ILSVRC top-5 error on ImageNet

- 2010: 22.5%
- 2011: 30%
- 2012: AlexNet
- 2013: 17.5%
- 2014: 15%
- Human: 7.5%
- ArXiv 2015: 5%
Good Embeddings ... 
This will later be the foundation of many papers

Query

Images with largest dot product with query

Embedding Layer

Our model

- Max-pooling layers follow first, second, and fifth convolutional layers
- The number of neurons in each layer is given by 253440, 186624, 64896, 64896, 43264, 4096, 4096, 1000

This will later be the foundation of many papers
DistBelief
Large Scale Distributed Deep Networks

Described the system for the 2012 ICML Paper

Abstract

Recent work in unsupervised feature learning and deep learning has shown that being able to train large models can dramatically improve performance. In this paper, we consider the problem of training a deep network with billions of parameters using tens of thousands of CPU cores. We have developed a software framework called DistBelief that can utilize computing clusters with thousands of machines to train large models. Within this framework, we have developed two algorithms for large-scale distributed training: (i) Downpour SGD, an asynchronous stochastic gradient descent procedure supporting a large number of model replicas, and (ii) Sandblaster, a framework that supports a variety of distributed batch optimization procedures, including a distributed implementation of L-BFGS. Downpour SGD and Sandblaster L-BFGS both increase the scale and speed of deep network training. We have successfully used our system to train a deep network 10x larger than previously reported in the literature, and achieves state-of-the-art accuracy.

1 Introduction

Deep learning and unsupervised feature extraction methods have achieved great success in many practical applications. State-of-the-art performance in speech recognition [1, 2], visual object recognition [3], and natural language processing [4, 5, 6] have been achieved by deep learning.

It has also been observed that increasing the scale of deep learning, with respect to the number of training examples, the number of model parameters, or both, can drastically improve ultimate classification accuracy [3, 4, 7]. These results have led to a surge of interest in scaling up the training and inference algorithms used for these models [8] and in improving applicable optimization procedures [7, 9]. The use of GPUs [1, 2, 3, 8] is a significant advance in recent years that makes the training of very large deep networks practical. A known limitation of the GPU approach is the long time required for training large neural networks.

Recent work in unsupervised feature learning and deep learning has shown that being able to train large models can dramatically improve performance. In this paper, we consider the problem of training a deep network with billions of parameters using tens of thousands of CPU cores. We have developed a software framework called DistBelief that can utilize computing clusters with thousands of machines to train large models. Within this framework, we have developed two algorithms for large-scale distributed training: (i) Downpour SGD, an asynchronous stochastic gradient descent procedure supporting a large number of model replicas, and (ii) Sandblaster, a framework that supports a variety of distributed batch optimization procedures, including a distributed implementation of L-BFGS. Downpour SGD and Sandblaster L-BFGS both increase the scale and speed of deep network training. We have successfully used our system to train a deep network 10x larger than previously reported in the literature, and achieves state-of-the-art accuracy.

1 Introduction

Deep learning and unsupervised feature extraction methods have achieved great success in many practical applications. State-of-the-art performance in speech recognition [1, 2], visual object recognition [3], and natural language processing [4, 5, 6] have been achieved by deep learning.

It has also been observed that increasing the scale of deep learning, with respect to the number of training examples, the number of model parameters, or both, can drastically improve ultimate classification accuracy [3, 4, 7]. These results have led to a surge of interest in scaling up the training and inference algorithms used for these models [8] and in improving applicable optimization procedures [7, 9]. The use of GPUs [1, 2, 3, 8] is a significant advance in recent years that makes the training of very large deep networks practical. A known limitation of the GPU approach is the long time required for training large neural networks.

Recent work in unsupervised feature learning and deep learning has shown that being able to train large models can dramatically improve performance. In this paper, we consider the problem of training a deep network with billions of parameters using tens of thousands of CPU cores. We have developed a software framework called DistBelief that can utilize computing clusters with thousands of machines to train large models. Within this framework, we have developed two algorithms for large-scale distributed training: (i) Downpour SGD, an asynchronous stochastic gradient descent procedure supporting a large number of model replicas, and (ii) Sandblaster, a framework that supports a variety of distributed batch optimization procedures, including a distributed implementation of L-BFGS. Downpour SGD and Sandblaster L-BFGS both increase the scale and speed of deep network training. We have successfully used our system to train a deep network 10x larger than previously reported in the literature, and achieves state-of-the-art accuracy.

1 Introduction

Deep learning and unsupervised feature extraction methods have achieved great success in many practical applications. State-of-the-art performance in speech recognition [1, 2], visual object recognition [3], and natural language processing [4, 5, 6] have been achieved by deep learning.

It has also been observed that increasing the scale of deep learning, with respect to the number of training examples, the number of model parameters, or both, can drastically improve ultimate classification accuracy [3, 4, 7]. These results have led to a surge of interest in scaling up the training and inference algorithms used for these models [8] and in improving applicable optimization procedures [7, 9]. The use of GPUs [1, 2, 3, 8] is a significant advance in recent years that makes the training of very large deep networks practical. A known limitation of the GPU approach is the long time required for training large neural networks.

Recent work in unsupervised feature learning and deep learning has shown that being able to train large models can dramatically improve performance. In this paper, we consider the problem of training a deep network with billions of parameters using tens of thousands of CPU cores. We have developed a software framework called DistBelief that can utilize computing clusters with thousands of machines to train large models. Within this framework, we have developed two algorithms for large-scale distributed training: (i) Downpour SGD, an asynchronous stochastic gradient descent procedure supporting a large number of model replicas, and (ii) Sandblaster, a framework that supports a variety of distributed batch optimization procedures, including a distributed implementation of L-BFGS. Downpour SGD and Sandblaster L-BFGS both increase the scale and speed of deep network training. We have successfully used our system to train a deep network 10x larger than previously reported in the literature, and achieves state-of-the-art accuracy.

1 Introduction

Deep learning and unsupervised feature extraction methods have achieved great success in many practical applications. State-of-the-art performance in speech recognition [1, 2], visual object recognition [3], and natural language processing [4, 5, 6] have been achieved by deep learning.

It has also been observed that increasing the scale of deep learning, with respect to the number of training examples, the number of model parameters, or both, can drastically improve ultimate classification accuracy [3, 4, 7]. These results have led to a surge of interest in scaling up the training and inference algorithms used for these models [8] and in improving applicable optimization procedures [7, 9]. The use of GPUs [1, 2, 3, 8] is a significant advance in recent years that makes the training of very large deep networks practical. A known limitation of the GPU approach is the long time required for training large neural networks.
Building High-Level Features Using Large Scale Unsupervised Learning

Abstract

We consider the problem of building high-level, class-specific feature detectors from only unlabeled data. For example, is it possible to learn a face detector using only unlabeled images from unlabeled data? To answer this, we train a 9-layered locally connected sparse autoencoder with pooling and local contrast normalization on a large dataset of images (the model has 1 billion connections, the dataset has 10 million 200x200 pixel images downloaded from the internet). We train this network using model parallelism and asynchronous SGD on a cluster with 1,000 machines (10,000 cores) for three days. Contrary to what appears to be a widely-held intuition, our experimental results reveal that it is possible to train a face detector without having to label images as containing a face or not. Importantly, it answers an intriguing question as to whether it is possible to learn a face detector using only unlabeled data. A positive answer to this question would be a significant challenge for problems where labeled data are rare. Although approaches that make use of non-proper unlabeled data are often preferred, they have not been shown to work well for building high-level features.

This work investigates the feasibility of building high-level features from only unlabeled data. A positive answer to this question would have important implications for future AI research.
Combine Model and Data Parallelism

Machine 1
Machine 2
Machine 3
Machine 4

This appears in earlier work on graph systems ...

Model Parallelism

Data Parallelism

Parameter Server

$w' = w - \eta \Delta w$

Model Replicas

Data Shards

Downpour SGD
Combine Model and Data Parallelism

Figure 1: An example of model parallelism in DistBelief. A five layer deep neural network with local connectivity is shown here, partitioned across four machines (blue rectangles). Only those nodes with edges that cross partition boundaries (thick lines) will need to have their state transmitted between machines. Even in cases where a node has multiple edges crossing a partition boundary, its state is only sent to the machine on the other side of that boundary once. Within each partition, computation for individual nodes will parallelized across all available CPU cores.

3 Model parallelism

To facilitate the training of very large deep networks, we have developed a software framework, DistBelief, that supports distributed computation in neural networks and layered graphical models. The user defines the computation that takes place at each node in each layer of the model, and the messages that should be passed during the upward and downward phases of computation.

For large models, the user may partition the model across several machines (Figure 1), so that responsibility for the computation for different nodes is assigned to different machines. The framework automatically parallelizes computation in each machine using all available cores, and manages communication, synchronization and data transfer between machines during both training and inference.

The performance benefits of distributing a deep network across multiple machines depends on the connectivity structure and computational needs of the model. Models with a large number of parameters or high computational demands typically benefit from access to more CPUs and memory, up to the point where communication costs dominate. We have successfully run large models with up to 144 partitions in the DistBelief framework with significant speedups, while more modestly sized models show decent speedups for up to 8 or 16 partitions. (See Section 5, under the heading Model Parallelism Benchmarks, for experimental results.) Obviously, models with local connectivity structures tend to be more amenable to extensive distribution than fully-connected structures, given their lower communication requirements. The typical cause of less-than-ideal speedups is variance in processing times across the different machines, leading to many machines waiting for the single slowest machine to finish a given phase of computation. Nonetheless, for our largest models, we can efficiently use 32 machines where each machine achieves an average CPU utilization of 16 cores, for a total of 512 CPU cores training a single large neural network. When combined with the distributed optimization algorithms described in the next section, which utilize multiple replicas of the entire neural network, it is possible to use tens of thousands of CPU cores for training a single model, leading to significant reductions in overall training times.

4 Distributed optimization algorithms

Parallelizing computation within the DistBelief framework allows us to instantiate and run neural networks considerably larger than have been previously reported. But in order to train such large models in a reasonable amount of time, we need to parallelize computation not only within a single...
Sandblaster L-BFGS

- **L-BFGS**
  - Commonly used for convex opt. problems
  - Requires repeated scans of all data
  - Robust, minimal tuning

- Naturally fits map-reduce pattern

- **Innovations:**
  - Accumulate gradients and store outputs in a sharded key value store (parameter server)
  - Tiny tasks + backup tasks to mitigate stragglers
Combine Model and Data Parallelism

3 Model parallelism

To facilitate the training of very large deep networks, we have developed a software framework, DistBelief, that supports distributed computation in neural networks and layered graphical models. The user defines the computation that takes place at each node in each layer of the model, and the messages that should be passed during the upward and downward phases of computation.

Large models, the user may partition the model across several machines (Figure 1), so that responsibility for the computation for different nodes is assigned to different machines. The framework automatically parallelizes computation in each machine using all available cores, and manages communication, synchronization and data transfer between machines during both training and inference.

The performance benefits of distributing a deep network across multiple machines depends on the connectivity structure and computational needs of the model. Models with a large number of parameters or high computational demands typically benefit from access to more CPUs and memory, up to the point where communication costs dominate. We have successfully run large models with up to 144 partitions in the DistBelief framework with significant speedups, while more modestly sized models show decent speedups for up to 8 or 16 partitions. (See Section 5, under the heading Model Parallelism Benchmarks, for experimental results.) Obviously, models with local connectivity structures tend to be more amenable to extensive distribution than fully-connected structures, given their lower communication requirements. The typical cause of less-than-ideal speedups is variance in processing times across the different machines, leading to many machines waiting for the single slowest machine to finish a given phase of computation. Nonetheless, for our largest models, we can efficiently use 32 machines where each machine achieves an average CPU utilization of 16 cores, for a total of 512 CPU cores training a single large neural network. When combined with the distributed optimization algorithms described in the next section, which utilize multiple replicas of the entire neural network, it is possible to use tens of thousands of CPU cores for training a single model, leading to significant reductions in overall training times.

4 Distributed optimization algorithms

Parallelizing computation within the DistBelief framework allows us to instantiate and run neural networks considerably larger than have been previously reported. But in order to train such large models in a reasonable amount of time, we need to parallelize computation not only within a single

Asynchronous

Parameter Server

$w' = w - \eta \Delta w$

$w$

$\Delta w$

Model Replicas

Data Shards

Downpour SGD

Sandblaster L-BFGS

Synchronous

Parameter Server

Coordinator (small messages)

Model Replicas

Data
Downpour SGD

Claimed Innovations

- Parameter Server
- Combine model and data parallelism in an async. execution.
- Adagrad stabilization
- Warmstarting

\[ w' = w - \eta \Delta w \]
Parameter Servers

Essentially a **sharded** key-value store
- support for put, get, **add**

Idea appears in earlier papers:

"An Architecture for Parallel Topic Models", Smola and Narayanamruthy. (VLDB’10)


DistBelief was probably the first paper to call a sharded key-value store a Parameter Server.
Downpour SGD

Claimed Innovations

- Parameter Server
- Combine model and data parallelism in an *async.* execution.
- Adagrad stabilization
- Warmstarting

\[ w' = w - \eta \Delta w \]
Key Results: Training and Test Error

Weird 20K Error Metric

Accuracy on Training Set

Accuracy on Test Set

Optimization method comparisons:

To evaluate the proposed distributed optimization procedures, we ran the speech model described above in a variety of configurations. We consider two baseline optimization procedures: training a DistBelief model (on 8 partitions) using conventional (single replica) SGD, and training the identical model on a GPU using CUDA [27]. The three distributed optimization methods we compare to these baseline methods are: Downpour SGD with a fixed learning rate, Downpour SGD with Adagrad learning rates, and Sandblaster L-BFGS.

Figure 4 shows classification performance as a function of training time for each of these methods on both the training and test sets. Our goal is to obtain the maximum test set accuracy in the minimum amount of training time, regardless of resource requirements. Conventional single replica SGD (black curves) is the slowest to train. Downpour SGD with 20 model replicas (blue curves) shows a significant improvement. Downpour SGD with 20 replicas plus Adagrad (orange curve) is modestly faster. Sandblaster L-BFGS using 2000 model replicas (green curves) is considerably faster yet again. The fastest, however, is Downpour SGD plus Adagrad with 200 model replicas (red curves). Given access to sufficient CPU resources, both Sandblaster L-BFGS and Downpour SGD with Adagrad can train models substantially faster than a high performance GPU.

Though we did not confine the above experiments to a fixed resource budget, it is interesting to consider how the various methods trade off resource consumption for performance. We analyze this by arbitrarily choosing a fixed test set accuracy (16%), and measuring the time each method took to reach that accuracy as a function of machines and utilized CPU cores, Figure 5. One of the four points on each traces corresponds to a training configuration shown in Figure 4, the other three points are alternative configurations.

In this plot, points closer to the origin are preferable in that they take less time while using fewer resources. In this regard Downpour SGD using Adagrad appears to be the best trade-off: For any fixed budget of machines or cores, Downpour SGD with Adagrad takes less time to reach the accuracy target than either Downpour SGD with a fixed learning rate or Sandblaster L-BFGS. For any allotted training time to reach the accuracy target, Downpour SGD with Adagrad used few resources than Sandblaster L-BFGS, and in many cases Downpour SGD with a fixed learning rate could not even reach the target within the deadline. The Sandblaster L-BFGS system does show promise in terms...
Why are they in the NY Times

- Trained a 1.7 billion parameter model (30x larger than state-of-the-art) (was it necessary?)

- Using 16,000 cores (efficiently?)

- Achieves 15.8 accuracy on ImageNet 20K (70% improvement over state of the art).
  - Non-standard benchmark

- Qualitatively interesting results

Figure 6. Visualization of the cat face neuron (left) and human body neuron (right).
Long-term Impact

- The parameter server appears in many later machine learning systems

- Downpour (Asynchronous) SGD has been largely replaced by synchronous systems for supervised training
  - Asynchrony is still popular in RL research
  - Why?

- Model parallelism is still used for large language models
  - Predated this work

- The neural network architectures studied here have been largely replaced by convolutional networks
More recent large-scale training

- Generated a lot of press
- Surpassed by Fast.ai: “Now anyone can train ImageNet in 18 minutes for $40.” blog post
- Popularized linear learning rate scaling

Abstract

Deep learning thrives with large neural networks and large datasets. However, larger networks and larger datasets result in longer training times that impede research and development progress. Distributed synchronous SGD offers a potential solution to this problem by dividing SGD minibatches over a pool of parallel workers. Yet to make this scheme efficient, the per-worker workload must be large, which implies nontrivial growth in the SGD minibatch size. In this paper, we empirically show that on the ImageNet dataset large minibatches cause optimization difficulties, but when these are addressed the trained networks exhibit good generalization. Specifically, we show no loss of accuracy when training with large minibatch sizes up to 8k images. To achieve this result, we adopt a hyperparameter-free linear scaling rule for adjusting learning rates as a function of minibatch size and develop a new warmup scheme that overcomes optimization challenges early in training. With these simple techniques, our Caffe2-based system trains ResNet-50 with a minibatch size of 8192 on 256 GPUs in one hour while maintaining the top-1 error of small minibatch training. For all minibatch sizes we set the learning rate as a linear function of the minibatch size and apply a simple warmup phase for the first few epochs of training. All other hyperparameters are kept fixed. Using this simple approach, accuracy of our models is invariant to minibatch size (up to an 8k minibatch size). Our techniques enable a linear reduction in training time with $\approx 90\%$ efficiency as we scale to large minibatch sizes, allowing us to train an accurate 8k minibatch ResNet-50 model in 1 hour on 256 GPUs.
Contrasting to the first paper

- **Synchronous SGD**
  - Much of the recent work has focused on synchronous setting
  - Easier to reason about

- Focus exclusively on data parallelism: *batch-size scaling*

- Focuses on the *generalization gap problem*
How do you distribute SGD?

\[ \theta^{(0)} \leftarrow \text{initial vector (random, zeros ...)} \]

For \( t \) from 0 to convergence:

\[ \mathcal{B} \sim \text{Random subset of indices} \]

\[
\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta_t \left( \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_{\theta} L(y_i, f(x_i; \theta)) \bigg|_{\theta = \theta^{(t)}} \right)
\]

Slow? (~150ms)
Depending on size of \( \mathcal{B} \)
Batch Size Scaling

- Increase the batch size by adding machines

\[
\theta^{(t+1)} \leftarrow \theta^{(t)} - \hat{\eta} \left( \frac{1}{k} \sum_{j=1}^{k} \frac{1}{|B_j|} \sum_{i \in B_j} \nabla_{\theta} L(y_i, f(x_i; \theta)) \bigg|_{\theta=\theta^{(t)}} \right)
\]

- Each server processes a fixed batch size (e.g., n=32)

- As more servers are added (k) the effective overall batch size increases linearly

- Why do these additional servers help?
Bigger isn’t Always Better

Motivation for larger batch sizes
- More opportunities for parallelism → but is it useful?
- Recall (1/n variance reduction):

\[
\frac{1}{n} \sum_{i=1}^{n} \nabla_{\theta} \mathbf{L}(y_i, f(x_i; \theta)) \approx \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_{\theta} \mathbf{L}(y_i, f(x_i; \theta))
\]

Is a variance reduction helpful?
- Only if it let’s you take bigger steps (move faster)
- Does it affect the final prediction accuracy?
Generalization Gap Problem

Larger batch sizes harm generalization performance.
Rough “Intuition”

Key problem: Addressing the generalization gap for large batch sizes.
Solution: Linear Scaling Rule

- Scale the learning rate linearly with the batch size

\[
\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta k \left( \frac{1}{k} \sum_{j=1}^{k} \frac{1}{|B_j|} \sum_{i \in B_j} \nabla_{\theta} L(y_i, f(x_i; \theta)) \right)
\]

- Addresses generalization performance by taking larger steps (also improves training convergence)

- Sub-problem: Large learning rates can be destabilizing in the beginning. Why?
  - Gradual warmup solution: increase learning rate scaling from constant to linear in first few epochs
  - Doesn’t help for very large k…
Other Details

- **Independent Batch Norm**: Batch norm calculation applies only to local batch size (n).

- **All-Reduce**: Recursive halving and doubling algorithm
  - Used instead of popular ring reduction (fewer rounds)

- **Gloo**: A library for efficient collective communications

- **Big Basin GPU Servers**: Designed for deep learning workloads
  - Analysis of communication requirements ➔ latency bound

- **No discussion on straggler or fault-tolerance**
  - Why?!
Key Results

All curves closely match using the linear scaling rule.

Note learning rate schedule drops.
Key Results

- Alternative learning rate rules.
- Setting, the training curves can be used as a reliable proxy (of the baseline at the start of training. However, in the cases of small minibatches, the gradual warmup strategy always starting with a linear learning rate from the beginning. After 8k epochs, the validation error begins to increase. Beyond 64k training error, the performance with smaller minibatch sizes is slightly worse than with larger minibatches.
- For small minibatch sizes, the results show evidence of the generality of the linear scaling rule used in R-CNN training. For each minibatch size, we pre-train 5 models and initialize Mask R-CNN with different minibatch sizes. The baseline (8k minibatch size) is used as a reference for these experiments.
- We have run experiments with 256 to 11264 minibatches, and use them to initialize Mask R-CNN. The results show improved performance compared to the baseline.
- We note that because of the availability of hardware, we were able to run experiments with 64k GPUs in a single server, while all other training runs distribute across tasks (from classification to detection/segmentation).
- We emphasize that we observed significant improvements in run-time performance and time per ImageNet epoch. The improvement is slightly larger for the 8k minibatch size.
- We also use gradual warmup for both minibatch sizes, and the results show that this technique helps improve performance with larger minibatches.
- Relative to a perfectly efficient extrapolation of the 8 GPU baseline, our implementation achieves a 1.28 million images / second of performance.

Machine Learning

- Epoch
- System

“Learning”

- Time per iteration (secs)
- Time per epoch (mins)
Key Results

- Train ResNet-50 to state-of-the-art on 256 GPUs in 1 hour
  - 90% scaling efficiency

- Fairly careful study of the linear scaling rule
  - Observed limits to linear scaling do not depend on dataset size
  - Cannot scale parallelism with dataset size
All-Reduce
All Reduce

Mechanism to sum and distribute data across machines.

- Used to sum and distribute the gradient

Machine A

- $a_1$
- $a_2$
- $a_3$
- $a_4$

Machine B

- $b_1$
- $b_2$
- $b_3$
- $b_4$

Machine D

- $d_1$
- $d_2$
- $d_3$
- $d_4$

Machine C

- $c_1$
- $c_2$
- $c_3$
- $c_4$
Single Master All-Reduce
Single Master All-Reduce

Machine A

Sends \((P-1) \times N\) Data
- \(P\) Machines
- \(N\) Parameters

Machine B

Machine C

Machine D
Single Master All-Reduce

Sends \((P-1) \times N\) Data
- \(P\) Machines
- \(N\) Parameters

\[ s_i = a_i + b_i + c_i + d_i \]
Single Master All-Reduce

**Sends** \((P-1) \times N\) Data

- **P** Machines
- **N** Parameters

\[ S_i = a_i + b_i + c_i + d_i \]
Single Master All-Reduce

Sends \((P-1) \times N\) Data

- **P** Machines
- **N** Parameters

\[ \sum \nabla_i = \sum a_i + \sum b_i + \sum c_i + \sum d_i \]
Single Master All-Reduce

Sends \((P-1) \times N^2\) Data

- \(P\) Machines
- \(N\) Parameters

Issues?

- High fan-in on Machine A
- \((P-1) \times N\) Bandwidth for Machine A
Parameter Server All Reduce
Send each entry to parameter server for that entry.
- Key 1 → A
- Key 2 → B
- Key 3 → C
- Key 4 → D
Each machine sends $\frac{N}{P}$ data to all other machines.

\[ P \times (P-1) \times \frac{N}{P} = (P-1) \times N \]

- **P** Machines
- **N** Parameters
Compute local sum on each machine

$$s_1 = a_i + b_i + c_i + d_i$$
Broadcast sum to each machine
Broadcast sum to each machine
Parameter Server All-Reduce

- Same amount of data transmitted as before

- Same high fan-in \((P-1)\)

- Reduced Inbound Bandwidth = \((P-1)N/P\)
  - Previously \((P-1)*N\)
Ring All Reduce

Send messages in a ring using to reduce fan-in.
Machine A

Machine B

Machine D

Machine C

Ring All Reduce

Note this depicts a partial sum and not a bigger message.
Machine A

Machine B

Ring All Reduce

Machine D

Machine C
Ring All Reduce
Ring All Reduce

Each machine sends \( \frac{N}{P} \) data to the next machine each of \( (p-1) \) rounds:

\[
(P-1) \times P \times \frac{N}{P} = (P-1) \times N
\]

- **Bandwidth** per round:
  - \( P \times \frac{N}{P} = N \) (doesn’t depend on \( P \))
- **Fan-in Per Round:**
  - 1 (doesn’t depend on \( P \))
Ring All Reduce

**Broadcast stage** repeats process sending messages forwarding sums (same communication costs).
Machine A

Machine B

Machine D

Machine C

Ring All Reduce

s_1  s_2

s_2  s_3

s_1  s_4

s_3  s_4
Ring All Reduce
Ring All Reduce
Ring All Reduce
Ring All-Reduce

- Simplified communication topology with low fan-in

- Overall communication
  - Same total communication: \(2(P-1)N\)
  - Bandwidth per round \((N)\) doesn’t depend on \(P\)
  - Fan-in is constant (doesn’t depend on \(P\))

- Issue: Number of communication rounds \((P-1)\)
Double Binary Tree All-Reduce

- Two overlaid binary reduction trees

- Double the fan-in $\rightarrow \log(p)$ rounds of communication
  - Currently used on Summit super-computer and latest NCCL

Review:

Dimensions of Parallelism
Data Parallelism

Parallelizing mini-batch gradient calculation with model replicated to all machines.

- **Synchronous Execution (Most Common)**
  - **Strengths:** deterministic, parallelism does not effect result
  - **Weaknesses:** need large batch sizes, frequent blocking comm., learning rate scaling, doesn't work with batch normalization

- **Asynchronous Execution (Popular in Research)**
  - **Strengths:** eliminate blocking and use background comm., batches don't need to span machines
  - **Weaknesses:** affects convergence (stability)

- **Issues:**
  - Model and activations must fit in each machine
Model Parallelism

Divide the model across machines and replicate the data.

- Supports large models and activations
- Requires communication within single evaluation

How to best divide a model?

- Split individual layers
  - which dimension?
    - Batch or Spatial → depends on operation

- Split across layers
  - Only one set of layers active a time → poor work balance
  - Soln: Pipelining Parallelism
Pipeline Parallelism

- Combine model and data parallelism to concurrently process multiple layers and batches.
- Originally described in GPipe*

*GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism
Operator Level Parallelism

- Exploiting the parallelism within linear algebra and convolution operations (a form of model parallelism)

- Multiple dimensions
  - Batch, spatial, time, ...

- Typically cast operators as linear alg. routines and leverage optimizes BLAS libraries
This weeks readings
Reading for the Week

- **Scaling Distributed Machine Learning with the Parameter Server** *(OSDI’14)*
  - Paper describing the parameter server system

- **PipeDream: Generalized Pipeline Parallelism for DNN Training** *(SOSP’19)*
  - Latest paper exploring pipeline parallel training

- **Adaptive Communication Strategies to Achieve the Best Error-Runtime Trade-off in Local-Update SGD** *(SysML’19)*
  - Dynamic averaging approach to distributed training
Scaling Distributed Machine Learning with the Parameter Server (OSDI’14)

- Describes the key-value store customized for machine learning
- Builds on earlier work in parameter servers

**Additional Context:** focused on topic modeling and sparse regression

**Key Ideas:** There are many ideas …

- Keys – Value pairs with **linear algebra** semantics (e.g., get by range)
- User defined **event handlers** on parameter servers and workers
- Several different **consistency models**
- **User defined filters** to determine when updates are communicated
PipeDream: Generalized Pipeline Parallelism for DNN Training (SOSP’19)

- Contemporaneously published with:
  - GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism (arXiv’18)

- **Key idea:** Leverage pipeline parallelism during training
  - **Automatically** constructs pipeline partition + schedule
  - Leverage *bounded staleness* + *versioned activations* to eliminate bubbles

---

**GPipe**

- All inputs use weights from last flush
- Pipeline flush: add gradients

---

**PipeDream**

- Startup State
- Steady State
- Time
- Forward Pass
- Backward Pass
- Idle
Bounded Staleness

- Developed as part of the parameter server work at CMU
- More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server (NIPS’13)

- Compromise between Hogwild and BSP
- Unclear implications for deep learning
- Non-convex loss
Adaptive Communication Strategies to Achieve the Best Error-Runtime Trade-off in Local-Update SGD (SysML’19)

- Studies Periodic Averaging SGD (PASGD)
- **Key Idea:** Change $\tau$ as algorithm converges
- More theoretical than previous reading
  - Theoretical results do not make convex assumptions!
Old Stuff