Large Scale Distributed Deep Networks

Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen,
Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato,
Andrew Senior, Paul Tucker, Ke Yang, Andrew Y. Ng
{jeff, gcorrado}@google.com
Google Inc., Mountain View, CA

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 30x larger than previously reported in the literature, and achieves state-of-the-art performance on ImageNet, a visual object recognition task with 16 million images and 21k categories. We show that these same techniques dramatically accelerate the training of a more modestly- sized deep network for a commercial speech recognition service. Although we focus on and report performance of these methods as applied to training large neural networks, the underlying algorithms are applicable to any gradient-based machine learning algorithm.

1 Introduction

Deep learning and unsupervised feature learning have shown great promise in many practical applications. State-of-the-art performance has been reported in several domains, ranging from speech recognition [1, 2], visual object recognition [3, 4], to text processing [5, 6].

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 modestly sized deep networks practical. A known limitation of the GPU approach is that the training speed-up is small when the model does not fit in GPU memory (typically less than 6 gigabytes). To use a GPU effectively, researchers often reduce the size of the data or parameters so that CPU-to-GPU transfers are not a significant bottleneck. While data and parameter reduction work well for small problems (e.g. acoustic modeling for speech recognition), they are less attractive for problems with a large number of examples and dimensions (e.g., high-resolution images).

In this paper, we describe an alternative approach: using large-scale clusters of machines to distribute training and inference in deep networks. We have developed a software framework called *DistBelief* that enables model parallelism within a machine (via multithreading) and across machines (via

DistBelief 能够在单机（通过多线程）或多机（通过数据传输）上考虑并行、同步和通信地支持模型并行。该框架也能支持数据并行。

DistBelief 主要由两个算法构成。算法一是 Downpour SGD，一个异步 SGD 过程，能够自适应学习率，支持大规模的模型副本；算法二是 Sandblaster L–BFGS，L–BFGS 的分布式实现，同时使用数据和模型并行。

message passing), with the details of parallelism, synchronization and communication managed by the framework. In addition to supporting model parallelism, the DistBelief framework also supports data parallelism, where multiple replicas of a model are used to optimize a single objective. Within this framework, we have designed and implemented two novel methods for large-scale distributed training: (i) *Downpour SGD*, an asynchronous stochastic gradient descent procedure which leverages adaptive learning rates and supports a large number of model replicas, and (ii) *Sandblaster L-BFGS*, a distributed implementation of L-BFGS that uses both data and model parallelism.[1] Both Downpour SGD and Sandblaster L-BFGS enjoy significant speed gains compared to more conventional implementations of SGD and L-BFGS.

揭示了异步 SGD（之前很少用在非凸问题）在分布式训练中表现良好，特别是和 Adagrad 自适应学习率方法结合时

Our experiments reveal several surprising results about large-scale nonconvex optimization. Firstly, asynchronous SGD, rarely applied to nonconvex problems, works very well for training deep networks, particularly when combined with Adagrad [10] adaptive learning rates. Secondly, we show that given sufficient resources, L-BFGS is competitive with or faster than many variants of SGD.

With regard to specific applications in deep learning, we report two main findings: that our distributed optimization approach can both greatly accelerate the training of modestly sized models, and that it can also train models that are larger than could be contemplated otherwise. To illustrate the first point, we show that we can use a cluster of machines to train a modestly sized speech model to the same classification accuracy in less than 1/10th the time required on a GPU. To illustrate the second point, we trained a large neural network of more than 1 billion parameters and used this network to drastically improve on state-of-the-art performance on the ImageNet dataset, one of the largest datasets in computer vision.

2 Previous work

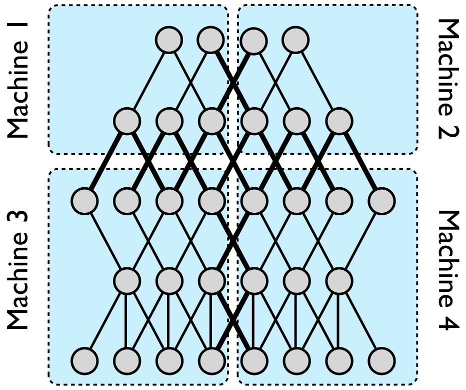
In recent years commercial and academic machine learning data sets have grown at an unprecedented pace. In response, a great many authors have explored scaling up machine learning algorithms through parallelization and distribution [11, 12, 13, 14, 15, 16, 17]. Much of this research has focused on linear, convex models, where distributed gradient computation is the natural first step. Within this area, some groups have relaxed synchronization requirements, exploring delayed gradient updates for convex problems [12, 17]. In parallel, other groups working on problems with sparse gradients (problems where only a tiny fraction of the coordinates of the gradient vector are non-zero for any given training example) have explored lock-less asynchronous stochastic gradient descent on shared-memory architectures (i.e. single machines) [5, 18]. We are interested in an approach that captures the best of both worlds, allowing the use of a cluster of machines asynchronously computing gradients, but without requiring that the problem be either convex or sparse.

In the context of deep learning, most work has focused on training relatively small models on a single machine (e.g., Theano [19]). Suggestions for scaling up deep learning include the use of a farm of GPUs to train a collection of many small models and subsequently averaging their predictions [20], or modifying standard deep networks to make them inherently more parallelizable [21]. Our focus is scaling deep learning techniques in the direction of training very large models, those with a few billion parameters, but without introducing restrictions on the form of the model. In special cases where one layer dominates computation, some authors have considered distributing computation in that one layer and replicating computation in the remaining layers [5]. But in the general case where *many* layers of the model are computationally intensive, full model parallelism in a spirit similar to [22] is required. To be successful, however, we believe that model parallelism must be combined with clever distributed optimization techniques that leverage data parallelism.

MapReduce设计用于并行数据处理，不适合深度网络训练中固有的迭代计算；GraphLab设计用于一般（非结构化）图计算，不会利用深度网络中典型的结构化图中的计算效率

We considered a number of existing large-scale computational tools for application to our problem, MapReduce [23] and GraphLab [24] being notable examples. We concluded that MapReduce, designed for parallel data processing, was ill-suited for the iterative computations inherent in deep network training; whereas GraphLab, designed for general (unstructured) graph computations, would not exploit computing efficiencies available in the structured graphs typically found in deep networks.

[1] We implemented L-BFGS within the Sandblaster framework, but the general approach is also suitable for a variety of other batch optimization methods.



只有包含跨越 machines 的边的 nodes 才需要在 machines 间传输状态

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 the parallelized across all available CPU cores.

3 Model parallelism

注意，模型并行包括不同 layers 分布在不同 machines 上，以及相同 layer 中的不同子 tensors 分布在不同 machines 上。

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.[2] 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.

短板效应可能存在，许多 machines 可能需要等待那个最慢的完成特定的计算阶段

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

[2] In the case of a neural network 'upward' and 'downward' might equally well be called 'feedforward' and 'backprop', while for a Hidden Markov Model, they might be more familiar as 'forward' and 'backward'.

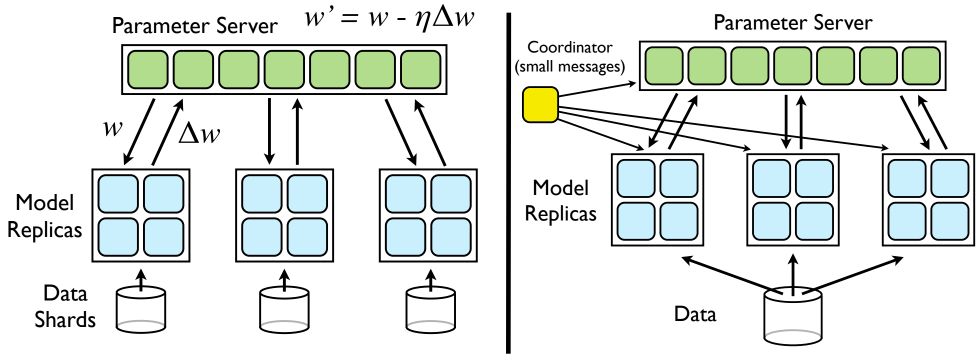


Figure 2: Left: Downpour SGD. Model replicas asynchronously fetch parameters w and push gradients Δw to the parameter server. Right: Sandblaster L-BFGS. A single 'coordinator' sends small messages to replicas and the parameter server to orchestrate batch optimization.

instance of the model, but to distribute training *across* multiple model instances. In this section we describe this second level of parallelism, where we employ a set of DistBelief model instances, or replicas, to simultaneously solve a single optimization problem.

We present a comparison of two large-scale distributed optimization procedures: Downpour SGD, an online method, and Sandblaster L-BFGS, a batch method. Both methods leverage the concept of a centralized sharded parameter server, which model replicas use to share their parameters. Both methods take advantage of the distributed computation DistBelief allows within each individual replica. But most importantly, both methods are designed to tolerate variance in the processing speed of different model replicas, and even the wholesale failure of model replicas which may be taken offline or restarted at random.

In a sense, these two optimization algorithms implement an intelligent version of data parallelism. Both approaches allow us to simultaneously process distinct training examples in each of the many model replicas, and periodically combine their results to optimize our objective function.

4.1 Downpour SGD

Stochastic gradient descent (SGD) is perhaps the most commonly used optimization procedure for training deep neural networks [25, 26, 3]. Unfortunately, the traditional formulation of SGD is inherently sequential, making it impractical to apply to very large data sets where the time required to move through the data in an entirely serial fashion is prohibitive.

SGD的传统公式本质上是顺序的，因此不适用于非常大的数据集，因为这些数据集中，以完全串行的方式传输数据所需的时间是不允许的

To apply SGD to large data sets, we introduce *Downpour SGD*, a variant of asynchronous stochastic gradient descent that uses multiple replicas of a single DistBelief model. The basic approach is as follows: We divide the training data into a number of subsets and run a copy of the model on each of these subsets. The models communicate updates through a centralized parameter server, which keeps the current state of all parameters for the model, sharded across many machines (e.g., if we have 10 parameter server shards, each shard is responsible for storing and applying updates to 1/10th of the model parameters) (Figure 2). This approach is asynchronous in two distinct aspects: the model replicas run independently of each other, and the parameter server shards also run independently of one another.

Downpour SGD 就是异步 parameter server (PS) 架构的模型并行版本，不同的 workers（都保存一份独立的 replica，用独立的数据集独立地进行 fp 和 bp；被划分到不同的 machines 上，因此每个 machine 只需要跟一部分 ps 通信，注意 machine 发送和接收时需要进行同步，以 replica 为单位，尽量避免（但没有保证）算力差别导致的异步累积，即 ps shards 之间的参数更新代数不同，这实际上会带来更多的随机性）之间和不同的 ps （体现在更新的快的 shards 先把这部分更新后的参数返回给 worker 中特定的 machine）之间都是异步的

In the simplest implementation, before processing **each** mini-batch, a model replica asks the parameter server service for an updated copy of its model parameters. Because DistBelief models are themselves partitioned across multiple machines, each machine needs to communicate with just the subset of parameter server shards that hold the model parameters relevant to its partition. After receiving an updated copy of its parameters, the DistBelief model replica processes a mini-batch of data to compute a parameter gradient, and sends the gradient to the parameter server, which then applies the gradient to the current value of the model parameters.

It is possible to reduce the communication overhead of Downpour SGD by limiting each model replica to request updated parameters only every n_{fetch} steps and send updated gradient values only every n_{push} steps (where n_{fetch} might not be equal to n_{push}). In fact, the process of fetching parameters, pushing gradients, and processing training data can be carried out in three only weakly synchronized threads (see the Appendix for pseudocode). In the experiments reported below we fixed $n_{fetch} = n_{push} = 1$ for simplicity and ease of comparison to traditional SGD.

Downpour SGD is more robust to machines failures than standard (synchronous) SGD. For synchronous SGD, if one machine fails, the entire training process is delayed; whereas for asynchronous SGD, if one machine in a model replica fails, the other model replicas continue processing their training data and updating the model parameters via the parameter servers. On the other hand, the multiple forms of asynchronous processing in Downpour SGD introduce a great deal of additional stochasticity in the optimization procedure. Most obviously, a model replica is almost certainly computing its gradients based on a set of parameters that are slightly out of date, in that some other model replica will likely have updated the parameters on the parameter server in the meantime. But there are several other sources of stochasticity beyond this: Because the parameter server shards act independently, there is no guarantee that at any given moment the parameters on each shard of the parameter server have undergone the same number of updates, or that the updates were applied in the same order. Moreover, because the model replicas are permitted to fetch parameters and push gradients in separate threads, there may be additional subtle inconsistencies in the timestamps of parameters. There is little theoretical grounding for the safety of these operations for nonconvex problems, but in practice we found relaxing consistency requirements to be remarkably effective.

不同 ps shards 之间的更新代数不同能够带来更多的随机性

放宽一致性在非凸问题中并无理论依据，但在实际中非常有效

One technique that we have found to greatly increase the robustness of Downpour SGD is the use of the Adagrad [10] adaptive learning rate procedure. Rather than using a single fixed learning rate on the parameter sever (η in Figure 2), Adagrad uses a separate adaptive learning rate for each parameter. Let $\eta_{i,K}$ be the learning rate of the i-th parameter at iteration K and $\Delta w_{i,K}$ its gradient, then we set: $\eta_{i,K} = \gamma/\sqrt{\sum_{j=1}^{K} \Delta w_{i,j}{}^2}$. Because these learning rates are computed only from the summed squared gradients of each parameter, Adagrad is easily implemented locally within each parameter server shard. The value of γ, the constant scaling factor for all learning rates, is generally larger (perhaps by an order of magnitude) than the best fixed learning rate used without Adagrad. The use of Adagrad extends the maximum number of model replicas that can productively work simultaneously, and combined with a practice of "warmstarting" model training with only a single model replica before unleashing the other replicas, it has virtually eliminated stability concerns in training deep networks using Downpour SGD (see results in Section 5).

Adagrad 自适应学习率策略直接在 ps shard 中，用梯度来算学习率，易实现，缩放常量远大于固定的学习率。该策略能够扩展 model replicas 能实际使用的个数，且与“少数几个 replicas warmstarting，再逐步加入其他 replicas”策略结合使用时，能够很好地解决 Downpour SGD 训练时的稳定性问题

Sandblaster 框架下，优化算法（如 L-BFGS）在协调进程中，该进程并不直接访问模型参数，而是向 ps shards 发送一系列子操作（点积，放缩，考虑系数的加，乘等），计算结果保存在 shard 本地。这样做可以避免需要把所有的参数和梯度都汇聚到一个中心 server 上。
为了缓解短板效应，提出了一个负载均衡策略：协调器给每个 replica 分配一个很小比例的工作（相较于 1/N batch），并给那些完成比较快的 replicas 分配新的更多工作。对于 batch 最后的工作，协调器会让多个 replicas 同时运算，并用完成最快的那个。上述策略意味着，无需像 Downpour SGD 那样将 dataset 划分为多个独立的 subset，而是以一个个 batch 的形式，在协调器的指挥下，供多个 replicas 进行处理。

4.2 Sandblaster L-BFGS

Batch methods have been shown to work well in training small deep networks [7]. To apply these methods to large models and large datasets, we introduce the *Sandblaster* batch optimization framework and discuss an implementation of L-BFGS using this framework.

A key idea in *Sandblaster* is distributed parameter storage and manipulation. The core of the optimization algorithm (e.g L-BFGS) resides in a coordinator process (Figure 2), which does not have direct access to the model parameters. Instead, the coordinator issues commands drawn from a small set of operations (e.g., dot product, scaling, coefficient-wise addition, multiplication) that can be performed by each parameter server shard independently, with the results being stored locally on the same shard. Additional information, e.g the history cache for L-BFGS, is also stored on the parameter server shard on which it was computed. This allows running large models (billions of parameters) without incurring the overhead of sending all the parameters and gradients to a single central server. (See the Appendix for pseudocode.)

In typical parallelized implementations of L-BFGS, data is distributed to many machines and each machine is responsible for computing the gradient on a specific subset of data examples. The gradients are sent back to a central server (or aggregated via a tree [16]). Many such methods wait for the slowest machine, and therefore do not scale well to large shared clusters. To account for this problem, we employ the following load balancing scheme: The coordinator assigns each of the N model replicas a small portion of work, much smaller than 1/Nth of the total size of a batch, and assigns replicas new portions whenever they are free. With this approach, faster model replicas do more work than slower replicas. To further manage slow model replicas at the end of a batch, the coordinator schedules multiple copies of the outstanding portions and uses the result from whichever model replica finishes first. This scheme is similar to the use of "backup tasks" in the MapReduce framework [23]. Prefetching of data, along with supporting data affinity by assigning sequential portions of data to the same worker makes data access a non-issue. In contrast with Downpour SGD, which requires relatively high frequency, high bandwidth parameter synchronization with the parameter server, Sandblaster workers only fetch parameters at the beginning of each batch (when they have been updated by the coordinator), and only send the gradients every few completed portions (to protect against replica failures and restarts).

5 Experiments

We evaluated our optimization algorithms by applying them to training models for two different deep learning problems: object recognition in still images and acoustic processing for speech recognition.

The speech recognition task was to classify the central region (or frame) in a short snippet of audio as one of several thousand acoustic states. We used a deep network with five layers: four hidden layer with sigmoidal activations and 2560 nodes each, and a softmax output layer with 8192 nodes. The input representation was 11 consecutive overlapping 25 ms frames of speech, each represented by 40 log-energy values. The network was fully-connected layer-to-layer, for a total of approximately 42 million model parameters. We trained on a data set of 1.1 billion weakly labeled examples, and evaluated on a hold out test set. See [27] for similar deep network configurations and training procedures.

For visual object recognition we trained a larger neural network with locally-connected receptive fields on the ImageNet data set of 16 million images, each of which we scaled to 100x100 pixels [28]. The network had three stages, each composed of filtering, pooling and local contrast normalization, where each node in the filtering layer was connected to a 10x10 patch in the layer below. Our infrastructure allows many nodes to connect to the same input patch, and we ran experiments varying the number of identically connected nodes from 8 to 36. The output layer consisted of 21 thousand one-vs-all logistic classifier nodes, one for each of the ImageNet object categories. See [29] for similar deep network configurations and training procedures.

Model parallelism benchmarks: To explore the scaling behavior of DistBelief model parallelism (Section 3), we measured the mean time to process a single mini-batch for simple SGD training as a function of the number of partitions (machines) used in a single model instance. In Figure 3 we quantify the impact of parallelizing across N machines by reporting the average training speed-up: the ratio of the time taken using only a single machine to the time taken using N. Speedups for inference steps in these models are similar and are not shown here.

The moderately sized speech model runs fastest on 8 machines, computing 2.2× faster than using a single machine. (Models were configured to use no more than 20 cores per machine.) Partitioning

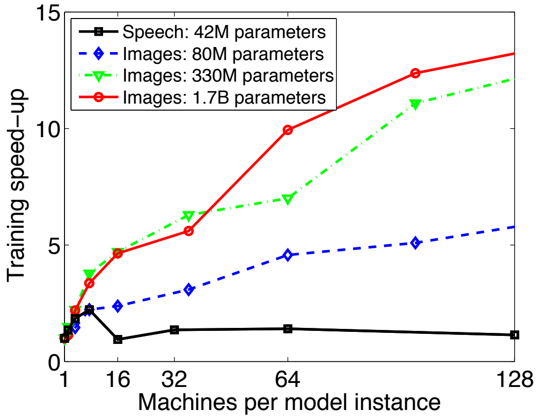


Figure 3: Training speed-up for four different deep networks as a function of machines allocated to a single DistBelief model instance. Models with more parameters benefit more from the use of additional machines than do models with fewer parameters.

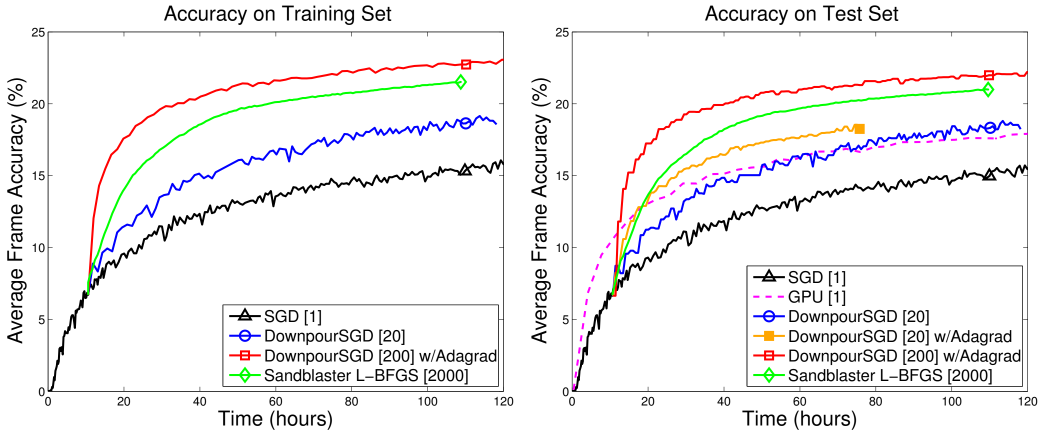


Figure 4: Left: Training accuracy (on a portion of the training set) for different optimization methods. Right: Classification accuracy on the hold out test set as a function of training time. Downpour and Sandblaster experiments initialized using the same ∼10 hour warmstart of simple SGD.

the model on more than 8 machines actually slows training, as network overhead starts to dominate in the fully-connected network structure and there is less work for each machine to perform with more partitions.

In contrast, the much larger, locally-connected image models can benefit from using many more machines per model replica. The largest model, with 1.7 billion parameters benefits the most, giving a speedup of more than 12× using 81 machines. For these large models using more machines continues to increase speed, but with diminishing returns.

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 resourses, 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

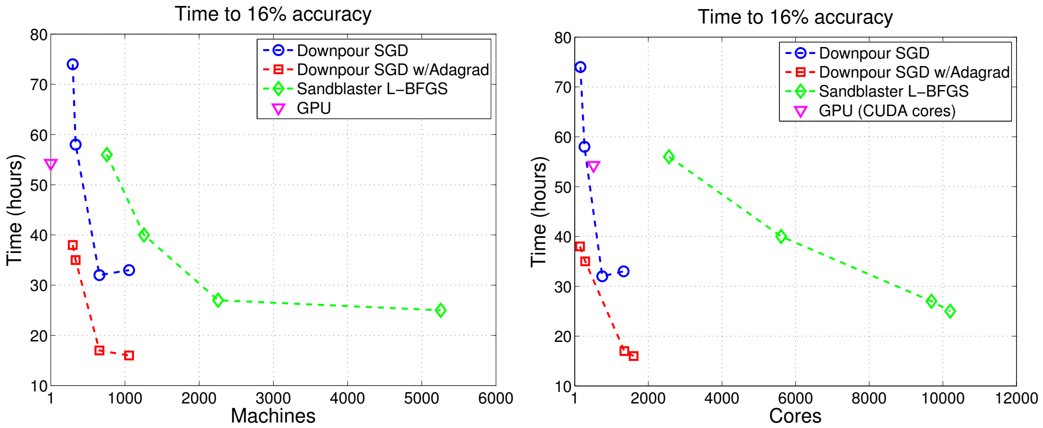


Figure 5: Time to reach a fixed accuracy (16%) for different optimization strategies as a function of number of the machines (left) and cores (right).

of its scaling with additional cores, suggesting that it may ultimately produce the fastest training times if used with an extremely large resource budget (e.g., 30k cores).

Application to ImageNet: The previous experiments demonstrate that our techniques can accelerate the training of neural networks with tens of millions of parameters. However, the more significant advantage of our cluster-based approach to distributed optimization is its ability to scale to models that are much larger than can be comfortably fit on single machine, let alone a single GPU. As a first step toward exploring the capabilities of very large neural networks, we used Downpour SGD to train the 1.7 billion parameter image model described above on the ImageNet object classification task. As detailed in [29], this network achieved a cross-validated classification accuracy of over 15%, a relative improvement over 60% from the best performance we are aware of on the 21k category ImageNet classification task.

6 Conclusions

In this paper we introduced DistBelief, a framework for parallel distributed training of deep networks. Within this framework, we discovered several effective distributed optimization strategies. We found that Downpour SGD, a highly asynchronous variant of SGD works surprisingly well for training nonconvex deep learning models. Sandblaster L-BFGS, a distributed implementation of L-BFGS, can be competitive with SGD, and its more efficient use of network bandwidth enables it to scale to a larger number of concurrent cores for training a single model. That said, the combination of Downpour SGD with the Adagrad adaptive learning rate procedure emerges as the clearly dominant method when working with a computational budget of 2000 CPU cores or less.

Adagrad was not originally designed to be used with asynchronous SGD, and neither method is typically applied to nonconvex problems. It is surprising, therefore, that they work so well together, and on highly nonlinear deep networks. We conjecture that Adagrad automatically stabilizes volatile parameters in the face of the flurry of asynchronous updates, and naturally adjusts learning rates to the demands of different layers in the deep network.

Our experiments show that our new large-scale training methods can use a cluster of machines to train even modestly sized deep networks significantly faster than a GPU, and without the GPU's limitation on the maximum size of the model. To demonstrate the value of being able to train larger models, we have trained a model with over 1 billion parameters to achieve better than state-of-the-art performance on the ImageNet object recognition challenge.

Acknowledgments

The authors would like to thank Samy Bengio, Tom Dean, John Duchi, Yuval Netzer, Patrick Nguyen, Yoram Singer, Sebastian Thrun, and Vincent Vanhoucke for their indispensable advice, support, and comments.

References

[1] G. Dahl, D. Yu, L. Deng, and A. Acero. Context-dependent pre-trained deep neural networks for large vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 2012.

[2] G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 2012.

[3] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. *CoRR*, 2010.

[4] A. Coates, H. Lee, and A. Y. Ng. An analysis of single-layer networks in unsupervised feature learning. In *AISTATS 14*, 2011.

[5] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.

[6] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML*, 2008.

[7] Q.V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A.Y. Ng. On optimization methods for deep learning. In *ICML*, 2011.

[8] R. Raina, A. Madhavan, and A. Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *ICML*, 2009.

[9] J. Martens. Deep learning via hessian-free optimization. In *ICML*, 2010.

[10] J. C. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[11] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, A. Strehl, and V. Vishwanathan. Hash kernels. In *AISTATS*, 2009.

[12] J. Langford, A. Smola, and M. Zinkevich. Slow learners are fast. In *NIPS*, 2009.

[13] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. In *NIPS*, 2009.

[14] R. McDonald, K. Hall, and G. Mann. Distributed training strategies for the structured perceptron. In *NAACL*, 2010.

[15] M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized stochastic gradient descent. In *NIPS*, 2010.

[16] A. Agarwal, O. Chapelle, M. Dudik, and J. Langford. A reliable effective terascale linear learning system. In *AISTATS*, 2011.

[17] A. Agarwal and J. Duchi. Distributed delayed stochastic optimization. In *NIPS*, 2011.

[18] F. Niu, B. Retcht, C. Re, and S. J. Wright. Hogwild! A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[19] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *SciPy*, 2010.

[20] D. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. Technical report, IDSIA, 2012.

[21] L. Deng, D. Yu, and J. Platt. Scalable stacking and learning for building deep architectures. In *ICASSP*, 2012.

[22] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, U. Toronto, 2009.

[23] J. Dean and S. Ghemawat. Map-Reduce: simplified data processing on large clusters. *CACM*, 2008.

[24] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. In *VLDB*, 2012.

[25] L. Bottou. Stochastic gradient learning in neural networks. In *Proceedings of Neuro-Nîmes 91*, 1991.

[26] Y. LeCun, L. Bottou, G. Orr, and K. Muller. Efficient backprop. In *Neural Networks: Tricks of the trade*. Springer, 1998.

[27] V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[28] 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.

[29] Q.V. Le, M.A. Ranzato, R. Monga, M. Devin, K. Chen, G.S. Corrado, J. Dean, and A.Y. Ng. Building high-level features using large scale unsupervised learning. In *ICML*, 2012.