Deep learning with COTS HPC systems

Adam Coates ACOATES@CS.STANFORD.EDU
Brody Huval BRODYH@STANFORD.EDU
Tao Wang TWANGCAT@STANFORD.EDU
David J. Wu DWU4@CS.STANFORD.EDU
Andrew Y. Ng ANG@CS.STANFORD.EDU
Stanford University Computer Science Dept., 353 Serra Mall, Stanford, CA 94305 USA

Bryan Catanzaro BCATANZARO@NVIDIA.COM
NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050

Abstract

Scaling up deep learning algorithms has been shown to lead to increased performance in benchmark tasks and to enable discovery of complex high-level features. Recent efforts to train extremely large networks (with over 1 billion parameters) have relied on cloud-like computing infrastructure and thousands of CPU cores. In this paper, we present technical details and results from our own system based on Commodity Off-The-Shelf High Performance Computing (COTS HPC) technology: a cluster of GPU servers with Infiniband interconnects and MPI. Our system is able to train 1 billion parameter networks on just 3 machines in a couple of days, and we show that it can scale to networks with over 11 billion parameters using just 16 machines. As this infrastructure is much more easily marshaled by others, the approach enables much wider-spread research with extremely large neural networks.

1. Introduction

A significant amount of effort has been put into developing deep learning systems that can scale to very large models and large training sets. With each leap in scale new results proliferate: large models in the literature are now top performers in supervised visual recognition tasks (Krizhevsky et al., 2012; Ciresan et al., 2012; Le et al., 2012), and can even learn to detect objects when trained from unlabeled images alone (Coates et al., 2012; Le et al., 2012). The very largest of these systems has been constructed by Le et al. (Le et al., 2012) and Dean et al. (Dean et al., 2012), which is able to train neural networks with over 1 billion trainable parameters. While such extremely large networks are potentially valuable objects of AI research, the expense to train them is overwhelming: the distributed computing infrastructure (known as "DistBelief") used for the experiments in (Le et al., 2012) manages to train a neural network using 16000 CPU cores (in 1000 machines) in just a few days, yet this level of resource is likely beyond those available to most deep learning researchers. Less clear still is how to continue scaling significantly beyond this size of network. In this paper we present an alternative approach to training such networks that leverages inexpensive computing power in the form of GPUs and introduces the use of high-speed communications infrastructure to tightly coordinate distributed gradient computations. Our system trains neural networks at scales comparable to DistBelief with just 3 machines. We demonstrate the ability to train a network with more than 11 billion parameters—6.5 times larger than the model in (Dean et al., 2012)—in only a few days with 2% as many machines.

Buoyed by many empirical successes (Uetz & Behnke, 2009; Raina et al., 2009; Ciresan et al., 2012; Krizhevsky, 2010; Coates et al., 2011) much deep learning research has focused on the goal of building larger models with more parameters. Though some techniques (such as locally connected networks (LeCun et al., 1989; Raina et al., 2009; Krizhevsky, 2010), and improved optimizers (Martens, 2010; Le et al., 2011)) have enabled scaling by algorithmic advantage, another main approach has been to achieve scale

Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28.

through greater computing power. Two axes are available along which researchers have tried to expand: (i) using multiple machines in a large cluster to increase the available computing power, ("scaling out"), or (ii) leveraging graphics processing units (GPUs), which can perform more arithmetic than typical CPUs ("scaling up"). Each of these approaches comes with its own set of engineering complications, yet significant progress has been made along each axis (Raina et al., 2009; Krizhevsky et al., 2012; Ciresan et al., 2012; Uetz & Behnke, 2009; Dean et al., 2012). A clear advantage might be obtained if we can combine improvements in both of these directions (i.e., if we can make use of many GPUs distributed over a large cluster). Unfortunately, obvious attempts to build large-scale systems based on this idea run across several major hurdles.

First, attempting to build large clusters of GPUs is difficult due to communications bottlenecks. Consider, for instance, using widely-implemented map-reduce infrastructure (Dean & Ghemawat, 2004; Chu et al., 2007) to employ our GPUs in a "data parallel" mode, where each GPU keeps a complete copy of the neural network parameters but computes a gradient using a different subset of the training data. The network parameters must fit on a single GPU—limiting us to, say, 250 million floating-point parameters (1 GB of storage). Our GPU code is capable of computing a gradient for these parameters in just milliseconds per training image, yet copying parameters or gradients to other machines will take at least 8 seconds over commodity Ethernet—several orders of magnitude slower. Parallelizing the gradient computations with "model parallelism", where each GPU is responsible for only a piece of the whole neural network, reduces bandwidth requirements considerably but also requires frequent synchronization (usually once for each forward- or backward-propagation step). This approach works well for GPUs in a single server (which share a high-speed bus) (Krizhevsky, 2010; Krizhevsky et al., 2012) but is still too inefficient to be used with Ethernet networks. For these reasons, we turn to the use of high-end networking infrastructure to remove the communications bottleneck between servers and enable us to exploit both fast GPU computation and to "scale out" to many servers. Our cluster incorporates Infiniband interconnects, which are dramatically faster (in terms of both bandwidth and latency) than typical Ethernet networks.

The second major problem with building larger systems is a software challenge: managing computation and communication amongst many GPUs significantly complicates algorithm design. For instance, we must create well-optimized code for the GPUs themselves, sometimes requiring algorithm-specific assumptions to maximize performance. Similarly, while low-level message passing software deals with some of the communications difficulties, we have found the message-passing metaphor cumbersome for building deep learning algorithms. In this paper, we will highlight several useful engineering solutions we have come across that greatly simplify development for systems like ours. Conceivably, these solutions could be boiled down to a software library, packaged and optimized for use by other researchers in the future.

In the remainder of this paper we will detail the implementation of our large-scale model-parallel training system for deep neural networks as developed for COTS HPC computing infrastructure. After describing our base hardware and software setup, we will detail several pieces of our software implementation in Section 4. We will then verify the scalability of our approach experimentally and present several results obtained from our implementation in Section 5. In particular, we will demonstrate the ability to replicate some of the experiments from (Le et al., 2012) (the largest training system in the literature) with just 3 machines, and also give results from an 11 billion parameter network trained with our cluster in just a few days.

2. Cluster Setup

Our cluster is comprised of 16 servers, each with 2 quad-core processors. Each server contains 4 NVIDIA GTX680 GPUs and an FDR Infiniband adapter. The GPUs each have 4GB of memory and are capable of performing about 1 TFLOPS (single-precision) with well-optimized code. The Infiniband adapter connects to the other servers through an Infiniband switch, and has a maximum throughput of 56 Gbps along with very low end-to-end latency (usually microseconds for small messages).

This particular server configuration was chosen to balance the number of GPUs with CPUs, which we have found to be important for large-scale deep learning. In previous work, multi-GPU systems have demonstrated their ability to rapidly train very large neural networks (Ciresan et al., 2011; Krizhevsky et al., 2012) (usually convolutional neural networks). Such systems rely on high-speed access to other GPUs across the host PCI bus to avoid communications bottlenecks—and it makes sense to put many GPUs into a single server in this case. But this approach scales only to 4 GPUs or perhaps 8 GPUs before the host machine becomes overburdened by I/O, power, cooling, and CPU compute demands. As a result, we have limited our-

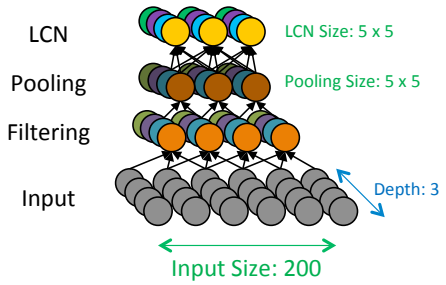


Figure 1. Basic structure of our network. The full network is constructed from 3 stacks of the filtering, pooling and local contrast normalize (LCN) layers as in (Le et al., 2012).

selves to 4 GPUs per server and relied on Infiniband to make communication feasible amongst GPUs in separate servers.

All of our software is written in C++ and built atop the MVAPICH2 (Wang et al., 2011) MPI implementation. MPI provides a standard message passing interface that allows multiple processes in a cluster to exchange blocks of data. MVAPICH2 handles all of the low-level communications over Infiniband in response to MPI API calls and includes integrated support for GPUs. Pointers to data in GPU memory can be provided as arguments to MPI calls to initiate transfers from one GPU to another, even when the destination GPU is in another server.

This off-the-shelf software infrastructure gives us a foundation on top of which to build our deep learning system. When our system starts up, every server spawns one process for each of its GPUs. Each process claims one GPU and is assigned an ID number ("rank") by the MPI implementation. Since each GPU has its own process, all communication amongst GPUs occurs through MPI. Though message-passing is a very low-level operation (and is not especially natural for building deep learning systems), we will show later how most of the communication can be abstracted easily making it much simpler to build deep learning algorithms on top of MPI.

3. Algorithm and Network Architecture

In this paper we will focus on the implementation of the sparse autoencoder described in (Le et al., 2012), though other variants could be implemented as well (Ranzato et al., 2007; Glorot et al., 2011). Closely following (Le et al., 2012), our network is constructed from stacks of neurons with each stack composed of three layers: a linear filtering layer, a pooling layer, and a local contrast normalization layer (Figure 1). This stack is replicated 3 times to form a 9 layer network.

The first two layers implement selective features ("simple cells") and invariant features (Hyvärinen & Hoyer, 2000; Hyvärinen et al., 2001) ("complex cells" (Hubel & Wiesel, 1959)). These elements are common to many other architectures (Garrigues & Olshausen, 2010; LeCun et al., 2004; Riesenhuber & Poggio, 1999), though we note that like (Le et al., 2012) we use *untied* filter banks—every neuron has its own parameters, in contrast to convolutional networks (LeCun et al., 1989; Krizhevsky et al., 2012) where spatially translated neurons use the same filter. The contrast normalization layer has been found empirically to be useful in many systems (Jarrett et al., 2009) and appears to aid training of higher layers in the network. Each of the layers makes use of "local receptive fields" (LeCun et al., 1989; Raina et al., 2009; Krizhevsky, 2010): each neuron (linear filter, pooling unit, or local contrast unit) uses only a small window of inputs from the layer below to compute its output, which will be a necessary feature for our distributed implementation.

We train this network in a greedy, layer-wise fashion (Hinton et al., 2006; Bengio et al., 2006). The pooling and normalization layers have fixed parameters like those in (Le et al., 2012), and thus we need only train the filter layers. To do so, we optimize the following unsupervised learning objective over the linear filter parameters, W, and a scalar scaling parameter α:

$$\underset{W,\alpha}{\text{minimize}} \sum_i ||W^\top(\alpha W x^{(i)}) - x^{(i)}||_2^2 + \lambda \sum_j \sqrt{V_j (\alpha W x^{(i)})^2} \quad (1)$$

$$\text{subject to } ||W^{(k)}||_2 = 1, \forall k.$$

Here $x^{(i)}$ is the i'th training example (a training image, or features computed by lower layers of the network), V_j is a vector of weights for the j'th pooling unit[1] and λ is a sparsity penalty (set to 0.1 in all of our experiments). $W^{(k)}$ is the filter for the k'th neuron, which is constrained to have unit norm. Note also that in this formulation the reconstruction penalty (first line of (1)) uses $W^\top$ as the "decoding" weights rather than using a separate matrix of weights. This saves a substantial amount of memory, which will be important for training our largest networks.

The optimization problem (1) is solved using a standard mini-batch stochastic gradient descent procedure with momentum (Rumelhart et al., 1986; Hinton, 2010). The gradient for the entire objective can be computed using gradient back-propagation. To en-

[1]The weights for all of our pooling units are fixed to 1 with a 5x5 square receptive field.

force the normalization constraint in our gradient descent procedure we define $W^{(k)} = \tilde{W}^{(k)}/||\tilde{W}^{(k)}||_2$ in the objective above and then optimize over $\tilde{W}$.

4. Implementation

We now describe in more technical detail several key aspects of our implementation of the training algorithm above for our HPC cluster. To begin, we note that we must solve two basic problems to arrive at an efficient (and hopefully not too complex) implementation: (i) we require highly optimized GPU code ("kernels") for all major computational operations, and (ii) we must develop a scheme for distributing the computations over many GPUs and managing the communication between them (which, since we are using MPI, will involve passing messages between GPUs). Fortunately, these problems can be dealt with separately, so we will visit each in turn.

As a preliminary, we note that the first layer of our network takes in a mini-batch of images which can be represented as a 4D array of size M-by-w-by-w-by-c, where M is the mini-batch size, w is the image width and height, and c is the number of input channels. In our experiments we will use a large unlabeled dataset of 200x200 color images so each image may be thought of as a 3D grid of 200-by-200-by-3 values, and each mini-batch is just an array of M such 3D grids. The output of the network layer can similarly be represented as a 4D grid of M-by-r-by-r-by-d responses, where r and d are determined by the size and number of filters. This layout is shown in Figure 2, which we will explain in more detail below. We will think of our GPU and MPI code as operating on these 4D arrays.

4.1. CUDA kernels

Our cluster uses NVIDIA GTX680 GPUs, and our GPU code is written with NVIDIA's CUDA language (NVI). We will not detail the particulars of the code, but instead describe a few basic observations that have enabled us to write highly optimized kernels for the most important computations.

Deep learning systems, including the sparse autoencoder in Section 3, rely on just a few major operations. Point-wise operations (e.g., addition or scalar nonlinearities) are very easy to implement efficiently. The more difficult operations to implement are those that involve local connectivity. In particular, the weight matrix W in Eq. 1 is very sparse: the filter $W^{(k)}$ for each neuron has non-zero entries only for indices j corresponding to x_j in a local spatial region. This sparsity means that we cannot use optimized linear algebra code designed for dense matrices, and generic sparse linear algebra code tends to be much slower.

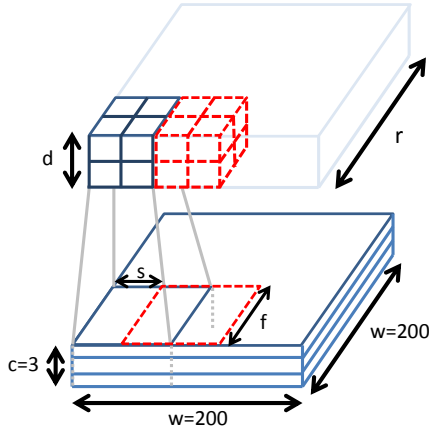


Figure 2. Schematic showing the notation and local receptive field connectivity of our network. Our input is an M-by-w-by-w-by-c image. (The fourth dimension M, the image index within a mini-batch, is not shown here.) Each neuron has a filter of size f that connects to all c input channels. All neurons sharing a receptive field are organized into 3D blocks (see Section 4.1; a 2-by-2-by-2 block arrangement is shown here). Each block of neurons sees a different receptive field shifted by step size s. The output representation is also a 4D array, but of size M-by-r-by-r-by-d. The setup for our higher-layer filters, pooling units, and contrast normalization units is analogous.

Unfortunately, writing optimized code for this operation turns out to be difficult. Recent GPUs rely heavily on instruction-level parallelism (ILP) in addition to thread parallelism and have fairly sophisticated cache hierarchies and instruction pipelines. Optimizing code for such architectures is thus increasingly difficult to perform without expert knowledge of each GPU architecture. Indeed, our own code to compute $y = WX$ (where X is a matrix representing a mini-batch of images [2]) achieved disappointing performance: 300 GFLOPS on GPUs capable of more than 1 TFLOPS peak. As well, experience from convolutional neural network implementations (Krizhevsky, 2010), like storing the filter coefficients in cache memory, has turned out not to be applicable: for our largest networks, a single filter can be larger than the entire shared memory cache of the GPU.

The main insight that we have used to implement much better kernels is to make a small change to our neural network structure so that computation of $Y = WX$ may be very efficiently implemented as a large number of smaller dense matrix multiplies. In particular, if we have a set of neurons F that share a single receptive field (i.e., for every $k \in F$ the filters

[2] For a mini-batch of size M, X would have M columns.

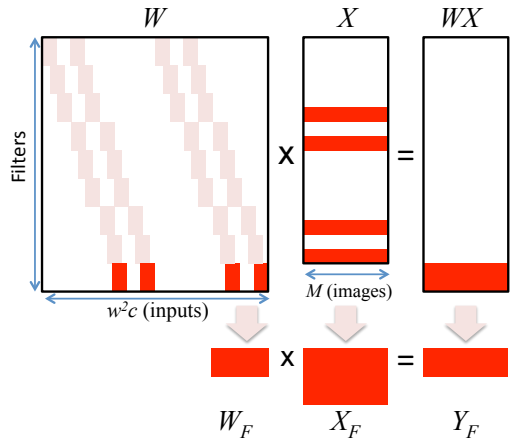


Figure 3. Our locally-connected filter operation is, intuitively, equivalent to a block-sparse matrix multiply. For large blocks, the operation can run nearly as efficiently as a dense matrix-matrix multiply.

$W^{(k)}$ have the same sparsity pattern), then we can compute the responses using a dense matrix-matrix multiplication: $Y_F = W_F X_F$. Here, W_F is the matrix of filters for neurons in F obtained by extracting the non-zero columns of W and the corresponding rows of X (denoted X_F). This situation is depicted in Figure 3. Provided the number of neurons in each set F (number of rows of W_F) and the number of images in a mini-batch (columns of X) are large enough, each block of filter responses Y_F may be computed almost identically to standard matrix-matrix multiplications—-we need only alter the fetching of columns in W and rows in X to follow the correct pattern.

For our implementation we referenced the highly-optimized MAGMA BLAS matrix-matrix multiply kernels (Tomov et al., 2011), which make use of advanced techniques including pre-fetching, exploitation of ILP, and careful register usage. By following the basic skeleton but mapping row and column fetches to the appropriate locations in our filter array W and inputs X we are able to execute operations like $Y = WX$ at speeds competitive with a full dense multiply.

To compute the linear responses $Y = WX$ efficiently with our optimized kernel, we must have a set F of many neurons sharing identical receptive fields. To ensure that we have such structure, we use block local connectivity as shown in Figure 2. In this setup, we group neurons into 3D blocks where all of the neurons in each block share an identical receptive field. We aim to make the blocks large enough to ensure efficient execution of our GPU code.[3] We can make the blocks in Figure 2 larger by expanding their width or depth, but in order to keep the total number of neurons constant we must also use a larger step size s (e.g., $s = 4$).

The four most-used computations in our code are: Wx, $W^\top x$, $\delta\ x^\top$[4] and (surprisingly) the normalization of the columns of W. This last operation is simple but memory bandwidth-limited. The first three, on the other hand, all use the approach described above. Their computational throughput is shown in Table 1. On some models of consumer GPUs (e.g., an overclocked GTX580) the fastest kernel can exceed 1 TFLOPS.

GPU	sgemm	Wx	$W^\top x$	$\delta\ x^\top$
GTX 680	1080	885	808	702
GTX 580 OC	1221	1015	887	798

Table 1. Average computation throughput (GFLOPS) for the most heavily-used compute kernels. Compare to sgemm applied to comparably-sized dense matrices. (For larger matrices, sgemm peak may be higher.)

4.2. Communication with MPI

Given implementations of the basic mathematical operations on the GPU, it is possible to build a single-GPU system to train neural networks of the form in Section 3. To expand to multiple GPUs we need to first divide up the computational work amongst the GPUs in the cluster and then organize their communication so that the end result matches what would be produced on a single GPU. In this paper, we will parallelize across GPUs using a strictly *model parallel* scheme: each GPU is responsible for a separate part of the computation, but all of the GPUs work together on the same mini-batch of input samples.

We think of the GPUs in our cluster as being arranged in a multi-dimensional grid. For simplicity we will use a 2D grid of 2-by-2 GPUs as an example. Recall that our input mini-batch as well as all of the neuron responses can be thought of as 4D arrays of values. A natural way to break up these arrays over our GPUs is to partition the spatial dimensions (width and height) evenly over our 2D grid. For example, at the input layer where we have a M-by-200-by-200-by-3 pixel array, each GPU claims a M-by-100-by-100-by-3 chunk of the input image. The colored regions in Figure 4 show how we might split an input image and neuron responses for the next layer over a grid of 4 GPUs.

[3]For current Fermi- and Kepler-class Nvidia GPUs, we aim to have blocks of 96 neurons and minibatches of $M =$ 96 images.

[4]δ is the gradient computed during back-propagation—$\delta\ x^\top$ is the gradient with respect to the filters W, which requires us to deal with the local receptive field structure.

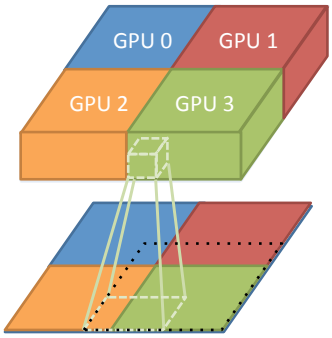


Figure 4. The image arrays are partitioned over the GPUs along the spatial dimensions. (Best viewed in color.) In this example, GPU 3 is responsible for computing responses of neurons in the green block. To facilitate this, the distributed array ensures that all of the inputs within the dotted black rectangle (the "input window") in the input layer are copied to GPU 3 before the responses are calculated.

This distribution of the arrays also implies one possible distribution for computation: each GPU is responsible for computing the responses of any neurons that have been assigned to it by the above partitioning scheme. For example, in Figure 4, GPU 3 must compute all of the neuron responses in the large green block of the top layer. Given this partition of neurons over GPUs we can also partition the filter weights W so that filter $W^{(k)}$ is stored on the same GPU as the k'th neuron. In principle, computation of the filter responses is carried out using the same GPU code as for a single-GPU implementation, but with one caveat: it is often the case that the inputs needed to compute the output of the k'th neuron will reside on several GPUs and thus we need to arrange for these "missing" inputs to be fetched from their homes before computation can be performed. In Figure 4, one block of neurons on GPU 3 might need to access inputs from both GPU 3 and GPU 2.

Implementing these fetches is messy and often difficult to think about during development. We have found that a simple distributed array abstraction hides virtually all communication from the rest of the computational code. For each such array, GPU i specifies an output window $\texttt{output}_i$ in the global array that it plans to fill with results. Similarly, it specifies a second window $\texttt{input}_i$ that it will need to read in order to continue the computation. At runtime GPU i will send a message to GPU j containing all of the values in the window $\texttt{output}_i \cap \texttt{input}_j$ and receive a message from GPU j containing values from $\texttt{input}_i \cap \texttt{output}_j$. The received values are written to a local array (along with the outputs from GPU i), yielding a globally consistent view of data in window $\texttt{input}_i$. Computation can then proceed as usual. In Figure 4, the input window $\texttt{input}_3$ for GPU 3 in our example is shown with a black dotted line.

In practice, the input window used by Layer N overlaps substantially with the output window used by Layer $N-1$, and thus most of the needed data is already available. This abstraction is most useful though when this arrangement does not quite work out—for instance when a neuron has a large receptive field and thus requires values from many GPUs (not just neighboring GPUs in the grid), or when uneven partitions result in strange configurations of the input and output areas. These "edge cases" are hidden by the array implementation and the GPU code sees only a local memory buffer that is properly populated with data.

Though we will not describe it in further detail here, we note that a small tweak is needed to allow *overlapping* output windows—i.e., regions of the global distributed array that are written by multiple GPUs. Simply: the overlapping response values are summed up. We use this primarily to implement back-propagation.

5. Experiments

5.1. Scaling efficiency

We have evaluated the efficiency of our system as we scale out to many GPUs. To measure the efficiency of scaling, we performed short optimization runs with varying numbers of GPUs and varying sizes of neural networks. For each run we recorded the average time taken to compute an update to *all* of the layers, as would be done during full joint training of the network. This computation requires us to do a feed-forward pass through all layers and compute the objective function for each stack. We must then perform a complete backward pass from the top of the network to the bottom and compute gradient updates for all of the parameters. This exercises all of the components of our system and is representative of the most demanding computations performed by deep learning systems, including those that use fine-tuning from supervised objectives.

We report the time taken to perform a mini-batch update for several network sizes in Figure 5. Figure 6 shows the factor speedup obtained relative to a single GPU, normalized by the number of parameters in each network. As can be seen in Figure 6, using many GPUs does not yield significant increases in computational throughput for small networks, but our system excels when working with much larger networks.

In absolute terms, our system is fast enough to train the largest networks we have used (with over 11 billion

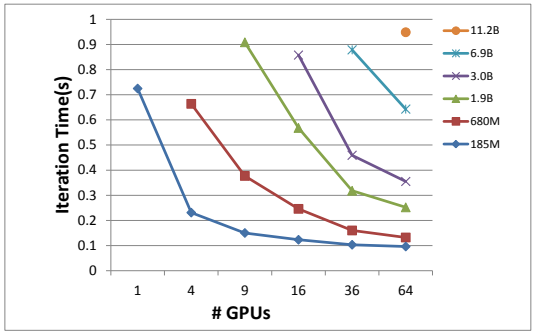


Figure 5. Time taken to perform a mini-batch update for all weights in large neural networks of sizes ranging from 180 million parameters up to 11.2 billion parameters.

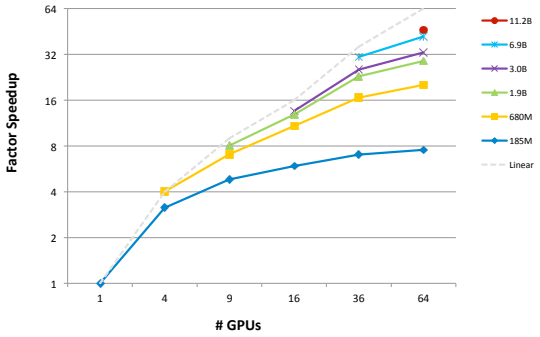


Figure 6. Factor speedup obtained for varying sizes of network and number of GPUs, normalized for the size of the network.

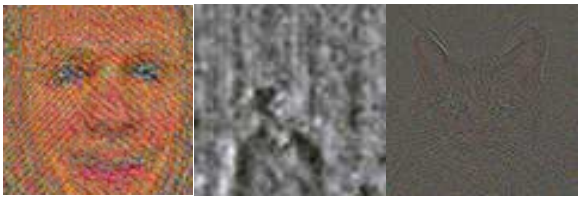
parameters) in just a few days. A single mini-batch update (mini-batch size of 96 images) for the 3rd stack of such a network takes less than 0.6 seconds and a full epoch through our 10 million image training set takes about 17 hours. In total, allowing for 2 epochs to train the 2nd and 3rd stacks in our networks (the 1st stack trains very quickly), our 11 billion parameter models can be trained in roughly 3 days.

5.2. High-level, object-selective features

It was recently shown that very large neural networks trained from only unlabeled data can learn to identify objects, such as human faces, in images (Le et al., 2012). We have replicated such a result using our system but at drastically reduced expense. We will briefly describe these experiments.

We construct an unlabeled training set by harvesting 10 million YouTube video thumbnails. We rescale each thumbnail so that its height is 200 pixels, and crop out the central 200-by-200 pixel region. These 200-by-200 color images are contrast normalized and whitened offline then distributed to local disk on our cluster for training.

We trained a network with 3 stacks as in Section 3 where each filtering layer used 20-by-20-by-d receptive fields (where d is the depth of the input array), filter blocks of size 4-by-4-by-8 with a step size of s =4 pixels between receptive fields.[5] This setup is very close to the one in (Le et al., 2012). The entire network has about 1.8 billion filter parameters. Following (Le et al., 2012), we tested each neuron in the trained network by recording its response to images from a labeled diagnostic set containing 13152 labeled faces from the Labeled Faces in the Wild (Huang et al., 2007) dataset and 48000 distractor images from ImageNet. The neuron's selectivity for "faces" is quantified by computing the highest classification accuracy[6] it can achieve on this set. Like (Le et al., 2012) and a similar result in (Coates et al., 2012), we have found that some neurons in our network are selective for faces. These neurons are able to classify a diagnostic image as "face" or "not-face" with 88% accuracy (whereas random guessing would achieve only 64.7% on our benchmark). We have performed the same test for other objects with results summarized in Table 2. We have also visualized the optimal response for these object-selective neurons in Figure 7. This is done using the constrained optimization procedure as in (Le et al., 2012) with similar results.



(a) Face (b) Body (c) Cat

Figure 7. Optimal stimuli visualizations for highest-ranked object-selective neurons found in our 1.8 billion parameter network.

Finally, to demonstrate the scalability of our system, we also trained a much larger network from the same dataset. This network used 20-by-20-by-3 filters for the first layer but with filter blocks of size 4-by-4-by-18 (step size of s =4). The 2nd and 3rd stacks used the same filter block size, but larger filters: each filter is 32-by-32-by-18 elements. We continue to use the same 5-by-5 pooling and contrast normalization, though it is likely that improvements can be obtained by adjusting these parameters in the future.

Surprisingly, we have found that the most object-

[5] We used greedy, layer-wise training in our experiments. We found that jointly tuning the network yielded a lower objective function value but did not improve the results here.

[6] We report accuracies on a reweighted set to make comparisons with (Le et al., 2012) easier. In that work, they used 13026 faces and 37000 distractors.

Object	Random guess	Best in random net	Best in 1.8B param. net	Best in 11B param. net.
Human faces	64.7%	64.8%	88.2%	86.5%
Upper body	64.7%	64.8%	80.2%	74.5%
Cats	64.7%	64.8%	73.0%	69.4%

Table 2. Classification rates for neurons found to be selective for objects, similar to those found by Le et al. (Le et al., 2012). Numbers in the left columns are from a 1.8 billion parameter neural network; the right-most column gives results from our 11.2 billion parameter network.

selective neurons in this large network are less selective for our test object classes than in the smaller network, although they are still much better than random guessing. The classification performance of the best neurons in our 11 billion parameter network are listed in Table 2. We have found that the introduction of nonlinearities and substantial hyperparameter tuning improves these numbers slightly, but it is possible that new algorithms or analysis methods will be needed for the sizes of networks that we are now able to train.

6. Discussion and Conclusion

We have presented details of a very large scale deep learning system based on high performance computing infrastructure that is widely available. With our system we have shown that we can comfortably train networks with well over 11 billion parameters—more than 6.5 times as large as the one reported in (Dean et al., 2012) (the largest previous network), and using fewer than 2% as many machines.

Though it has taken significant effort to make the best use of HPC infrastructure, we have described in this paper several components useful for deep learning that are efficient and easily implemented. In particular, we found that distributed arrays allow us to hide communications during our forward and backward propagation steps and that highly optimized GPU kernels can be built with semantics and implementation akin to matrix-matrix multiplication code to handle locally-connected neuron computations—a major source of complexity and optimization issues in our other experiences with GPUs. These are components that, with wider adoption of HPC systems, might reasonably be packaged into optimized software libraries.

Though we are clearly able to train extremely large neural networks, we have not yet identified a combination of algorithms and architecture yielding much better results for our unsupervised tests. Our 11 billion parameter network relies heavily on an architecture that has not changed much from the ("small") 1 billion parameter network in (Le et al., 2012). Nevertheless, with such large networks now relatively straightforward to train, we hope that wider adoption of this type of training machinery in deep learning will help spur rapid progress in identifying how best to make use of these expansions in scale.

7. Acknowledgments

Thanks to the Stanford CSD-CF staff for assembling and supporting our cluster. Thanks to the authors of MAGMA BLAS and MVAPICH2 for technical support. We are grateful to Quoc Le for help reproducing the results of (Le et al., 2012) and for useful advice. Thanks also to Bill Daly, Cliff Woolley, Michael Bauer, and Geoffrey Fox for helpful conversations.

References

Bengio, Y, Lamblin, P, Popovici, D, and Larochelle, H. Greedy layer-wise training of deep networks. In *Neural Information Processing Systems*, 2006.

Chu, C, Kim, S. K, Lin, Y.-A, Yu, Y, Bradski, G, Ng, A. Y, and Olukotun, K. Map-reduce for machine learning on multicore. *Advances in Neural Information Processing Systems*, 19:281, 2007.

Ciresan, D. C, Meier, U, Masci, J, Gambardella, L. M, and Schmidhuber, J. Flexible, high performance convolutional neural networks for image classification. In *International Joint Conference on Artificial Intelligence*, pp. 1237–1242, 2011.

Ciresan, D. C, Meier, U, and Schmidhuber, J. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition*, pp. 3642–3649, 2012.

Coates, A, Lee, H, and Ng, A. Y. An analysis of single-layer networks in unsupervised feature learning. In *14th International Conference on AI and Statistics*, pp. 215–223, 2011.

Coates, A, Karpathy, A, and Ng, A. Y. On the emergence of object-selective features in unsupervised feature learning. In *Advances in Neural Information Processing Systems*, 2012.

Dean, J and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation*, 2004.

Dean, J, Corrado, G. S, Monga, R, Chen, K, Devin, M, Le, Q. V, Mao, M. Z, Ranzato, M, Senior, A, Tucker, P, Yang, K, and Ng, A. Y. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*, 2012.

Garrigues, P and Olshausen, B. Group sparse coding with a laplacian scale mixture prior. In *Advances in Neural Information Processing Systems 23*, pp. 676–684, 2010.

Glorot, X, Bordes, A, and Bengio, Y. Deep sparse rectifier neural networks. In *14th International Conference on Artificial Intelligence and Statistics*, pp. 315–323, 2011.

Hinton, G, Osindero, S, and Teh, Y. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

Hinton, G. A practical guide to training restricted boltzmann machines. Technical report, University of Toronto, 2010.

Huang, G. B, Ramesh, M, Berg, T, and Learned-Miller, E. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

Hubel, D and Wiesel, T. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148(3):574–591, 1959.

Hyvärinen, A and Hoyer, P. Emergence of phase-and shift-invariant features by decomposition of natural images into independent feature subspaces. *Neural Computation*, 12(7):1705–1720, 2000.

Hyvärinen, A, Hoyer, P, and Inki, M. Topographic independent component analysis. *Neural Computation*, 13(7):1527–1558, 2001.

Jarrett, K, Kavukcuoglu, K, Ranzato, M, and LeCun, Y. What is the best multi-stage architecture for object recognition? In *12th International Conference on Computer Vision*, pp. 2146–2153, 2009.

Krizhevsky, A. Convolutional Deep Belief Networks on CIFAR-10. Unpublished manuscript, 2010.

Krizhevsky, A, Sutskever, I, and Hinton, G. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pp. 1106–1114, 2012.

Le, Q. V, Ngiam, J, Coates, A, Lahiri, A, Prochnow, B, and Ng, A. Y. On optimization methods for deep learning. In *International Conference on Machine Learning*, 2011.

Le, Q, Ranzato, M, Monga, R, Devin, M, Chen, K, Corrado, G, Dean, J, and Ng., A. Building high-level features using large scale unsupervised learning. In *International Conference on Machine Learning*, 2012.

LeCun, Y, Boser, B, Denker, J. S, Henderson, D, Howard, R. E, Hubbard, W, and Jackel, L. D. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551, 1989.

LeCun, Y, Huang, F. J, and Bottou, L. Learning methods for generic object recognition with invariance to pose and lighting. In *Computer Vision and Pattern Recognition*, volume 2, pp. 97–104, 2004.

Martens, J. Deep learning via hessian-free optimization. In *27th International Conference on Machine Learning*, volume 951, pp. 2010, 2010.

nVidia CUDA Programming Guide. NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050. URL http://docs.nvidia.com/.

Raina, R, Madhavan, A, and Ng, A. Large-scale deep unsupervised learning using graphics processors. In *26th International Conference on Machine Learning*, 2009.

Ranzato, M, Poultney, C, Chopra, S, and LeCun, Y. Efficient learning of sparse representations with an energy-based model. In *Advances in Neural Information Processing Systems*, 2007.

Riesenhuber, M and Poggio, T. Hierarchical models of object recognition in cortex. *Nature neuroscience*, 2, 1999.

Rumelhart, D, Hinton, G, and Williams, R. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

Tomov, S, Nath, R, Du, P, and Dongarra, J. *MAGMA Users Guide*. UT Knoxville ICL, 2011. URL http://icl.cs.utk.edu/magma/.

Uetz, R and Behnke, S. Large-scale object recognition with CUDA-accelerated hierarchical neural networks. In *Intelligent Computing and Intelligent Systems*, 2009.

Wang, H, Potluri, S, Luo, M, Singh, A. K, Sur, S, and Panda, D. K. MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters. *Computer Science-Research and Development*, 26(3):257–266, 2011.