One weird trick for parallelizing convolutional neural networks

Alex Krizhevsky
Google Inc.
akrizhevsky@google.com

April 29, 2014

Abstract

I present a new way to parallelize the training of convolutional neural networks across multiple GPUs. The method scales significantly better than all alternatives when applied to modern convolutional neural networks.

1 Introduction

This is meant to be a short note introducing a new way to parallelize the training of convolutional neural networks with stochastic gradient descent (SGD). I present two variants of the algorithm. The first variant perfectly simulates the synchronous execution of SGD on one core, while the second introduces an approximation such that it no longer perfectly simulates SGD, but nonetheless works better in practice.

2 Existing approaches

Convolutional neural networks are big models trained on big datasets. So there are two obvious ways to parallelize their training:

- across the model dimension, where different workers train different parts of the model, and
- across the data dimension, where different workers train on different data examples.

These are called model parallelism and data parallelism, respectively.

In model parallelism, whenever the model part (subset of neuron activities) trained by one worker requires output from a model part trained by another worker, the two workers must synchronize. In contrast, in data parallelism the workers must synchronize model parameters (or parameter gradients) to ensure that they are training a consistent model.

In general, we should exploit all dimensions of parallelism. Neither scheme is better than the other a priori. But the relative degrees to which we exploit each scheme should be informed by model architecture. In particular, model parallelism is efficient when the amount of computation per neuron activity is high (because the neuron activity is the unit being communicated), while data parallelism is efficient when the amount of computation per weight is high (because the weight is the unit being communicated).

Another factor affecting all of this is batch size. We can make data parallelism arbitrarily efficient if we are willing to increase the batch size (because the weight synchronization step is performed once per batch). But very big batch sizes adversely affect the rate at which SGD converges as well as the quality of the final solution. So here I target batch sizes in the hundreds or possibly thousands of examples.

3 Some observations

Modern convolutional neural nets consist of two types of layers with rather different properties:

- Convolutional layers cumulatively contain about 90-95% of the computation, about 5% of the parameters, and have large representations.
- Fully-connected layers contain about 5-10% of the computation, about 95% of the parameters, and have small representations.

Knowing this, it is natural to ask whether we should parallelize these two in different ways. In particular, data parallelism appears attractive for convolutional layers, while model parallelism appears attractive for fully-connected layers.

This is precisely what I'm proposing. In the remainder of this note I will explain the scheme in more detail and also mention several nice properties.

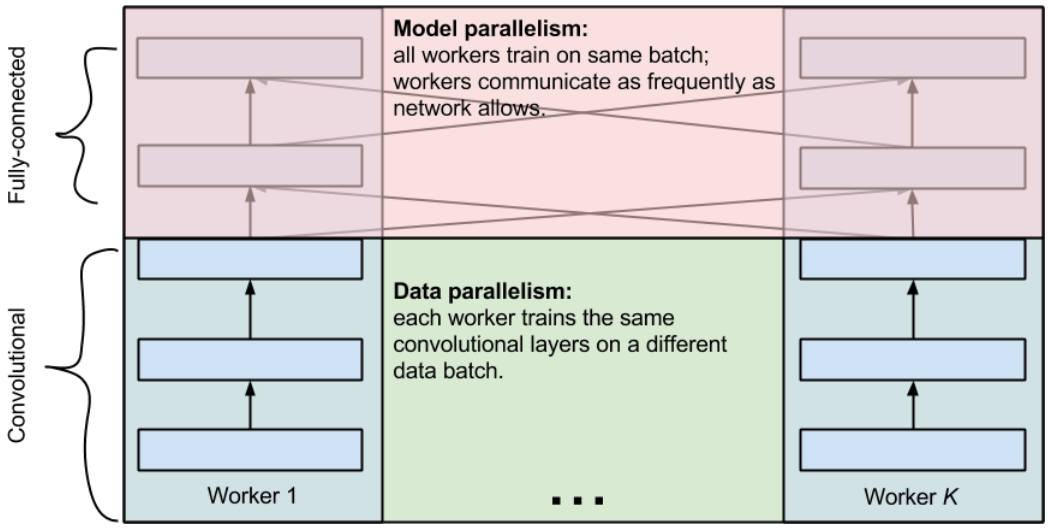

Figure 1: K workers training a convolutional neural net with three convolutional layers and two fully-connected layers.

4 The proposed algorithm

I propose that to parallelize the training of convolutional nets, we rely heavily on data parallelism in the convolutional layers and on model parallelism in the fully-connected layers. This is illustrated in Figure 1 for K workers.

In reference to the figure, the forward pass works like this:

1. Each of the K workers is given a different data batch of (let's say) 128 examples.
2. Each of the K workers computes all of the convolutional layer activities on its batch.
3. To compute the fully-connected layer activities, the workers switch to model parallelism. There are several ways to accomplish this:
   (a) Each worker sends its last-stage convolutional layer activities to each other worker. The workers then assemble a big batch of activities for $128K$ examples and compute the fully-connected activities on this batch as usual.
   (b) One of the workers sends its last-stage convolutional layer activities to all other workers. The workers then compute the fully-connected activities on this batch of 128 examples and then begin to backpropagate the gradients (more on this below) for these 128 examples. **In parallel with this computation**, the next worker sends its last-stage convolutional layer activities to all other workers. Then the workers compute the fully-connected activities on this second batch of 128 examples, and so on.
   (c) All of the workers send $128/K$ of their last-stage convolutional layer activities to all other workers. The workers then proceed as in (b).

It is worth thinking about the consequences of these three schemes.

In scheme (a), all useful work has to pause while the big batch of $128K$ images is assembled at each worker. Big batches also consume lots of memory, and this may be undesirable if our workers run on devices with limited memory (e.g. GPUs). On the other hand, GPUs are typically able to operate on big batches more efficiently.

In scheme (b), the workers essentially take turns broadcasting their last-stage convolutional layer activities. The main consequence of this is that much (i.e. $\frac{K-1}{K}$) of the communication can be hidden – it can be done in parallel with the computation of the fully-connected layers. This seems fantastic, because this is by far the most significant communication in the network.

Scheme (c) is very similar to scheme (b). Its one advantage is that the communication-to-computation ratio is constant in K. In schemes (a) and (b), it is proportional to K. This is

because schemes (a) and (b) are always bottlenecked by the outbound bandwidth of the worker that has to send data at a given "step", while scheme (c) is able to utilize many workers for this task. This is a major advantage for large K.

The backward pass is quite similar:

1. The workers compute the gradients in the fully-connected layers in the usual way.

2. The next step depends on which of the three schemes was chosen in the forward pass:

   (a) **In scheme (a)**, each worker has computed last-stage convolutional layer activity gradients for the entire batch of $128K$ examples. So each worker must send the gradient for each example to the worker which generated that example in the forward pass. Then the backward pass continues through the convolutional layers in the usual way.

   (b) **In scheme (b)**, each worker has computed the last-stage convolutional layer activity gradients for one batch of 128 examples. Each worker then sends these gradients to the worker which is responsible for this batch of 128 examples. **In parallel with this**, the workers compute the fully-connected forward pass on the next batch of 128 examples. After K such forward-and-backward iterations through the fully-connected layers, the workers propagate the gradients all the way through the convolutional layers.

   (c) **Scheme (c)** is very similar to scheme (b). Each worker has computed the last-stage convolutional layer activity gradients for 128 examples. This 128-example batch was assembled from $128/K$ examples contributed by each worker, so to distribute the gradients correctly we must reverse this operation. The rest proceeds as in scheme (b).

   I note again that, as in the forward pass, scheme (c) is the most efficient of the three, for the same reasons.

The forward and backward propagations for scheme (b) are illustrated in Figure 2 for the case of $K = 2$ workers.

4.1 Weight synchronization

Once the backward pass is complete, the workers can update the weights. In the convolutional layers, the workers must also synchronize the weights (or weight gradients) with one another. The simplest way that I can think of doing this is the following:

1. Each worker is designated $1/K$th of the gradient matrix to synchronize.

2. Each worker accumulates the corresponding $1/K$th of the gradient from every other worker.

3. Each worker broadcasts this accumulated $1/K$th of the gradient to every other worker.

It's pretty hard to implement this step badly because there are so few convolutional weights.

4.2 Variable batch size

So what we have here in schemes (b) and (c) is a slight modification to the standard forward-backward propagation which is, nonetheless, completely equivalent to running synchronous SGD with a batch size of $128K$. Notice also that schemes (b) and (c) perform K forward and backward passes through the fully-connected layers, each time with a different batch of 128 examples. This means that we can, if we wish, update the fully-connected weights after each of these partial backward passes, at virtually no extra computational cost. We can think of this as using a batch size of 128 in the fully-connected layers and $128K$ in the convolutional layers. With this kind of variable batch size, the algorithm ceases to be a pure parallelization of SGD, since it no longer computes a gradient update for any consistent model in the convolutional layers. But it turns out that this doesn't matter much in practice. As we take the effective batch size, $128K$, into the thousands, using a smaller batch size in the fully-connected layers leads to faster convergence to better minima.

5 Experiments

The first question that I investigate is the accuracy cost of larger batch sizes. This is a somewhat complicated question because the answer is dataset-dependent. Small, relatively homogeneous datasets benefit from smaller batch sizes more so than large, heterogeneous, noisy datasets. Here, I report experiments on the widely-used ImageNet 2012 contest dataset (ILSVRC 2012) [Deng et al., 2009]. At 1.2 million images in 1000 categories, it falls somewhere in between the two extremes. It isn't tiny, but it isn't "internet-scale" either. With current GPUs (and

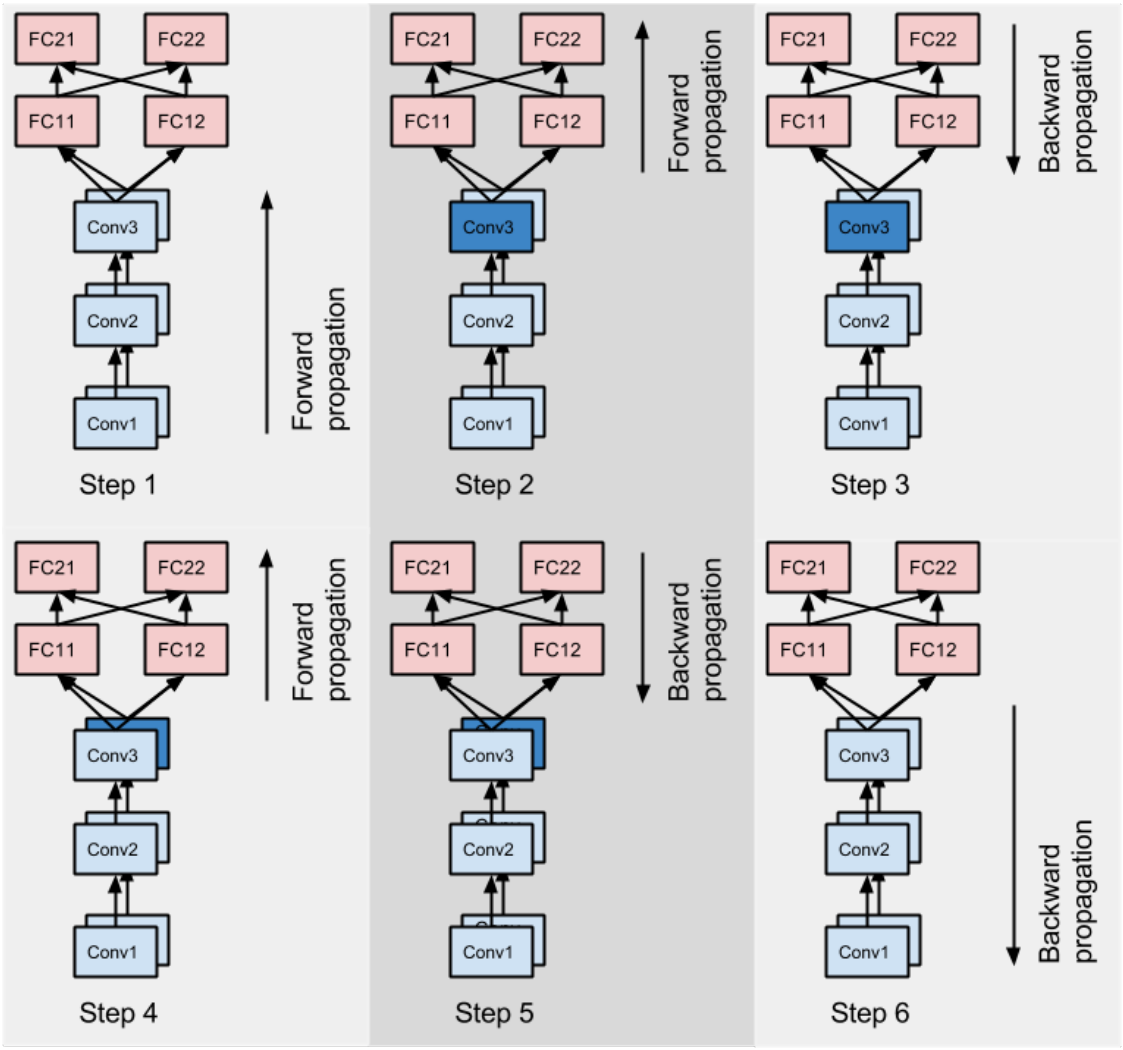

Figure 2: Illustration of the forward and backward propagations for scheme (b) of Section 4, for $K = 2$ workers. Two-way data parallelism in the three convolutional layers is represented with layers stacked on top of one another, while two-way model parallelism in the two fully-connected layers is represented with layers laid out next to one another. The standard two passes are replaced here with six passes. Shading in the final convolutional layer indicates the batch that is processed by the fully-connected layer. Note that, as mentioned in Section 4.2, we are free either to update the fully-connected weights during each of the backward passes, or to accumulate a gradient and then update the entire net's weights after the final backward pass.

CPUs) we can afford to iterate through it many times when training a model.

The model that I consider is a minor variation on the winning model from the ILSVRC 2012 contest [Krizhevsky et al., 2012]. The main difference is that it consists of one "tower" instead of two. This model has 0.2% more parameters and 2.4% fewer connections than the two-tower model. It has the same number of layers as the two-tower model, and the (x, y) map dimensions in each layer are equivalent to the (x, y) map dimensions in the two-tower model. The minor difference in parameters and connections arises from a necessary adjustment in the number of kernels in the convolutional layers, due to the unrestricted layer-to-layer connectivity in the single-tower model.[1] Another difference is that instead of a softmax final layer with multinomial logistic regression cost, this model's final layer has 1000 independent logistic units, trained to minimize cross-entropy. This cost function performs equivalently to multinomial logistic regression but it is easier to parallelize, because it does not require a normalization across classes.[2] I trained all models for exactly 90 epochs, and multiplied the learning rate by $250^{-1/3}$ at 25%, 50%, and 75% training progress.

The weight update rule that I used was

$$\begin{aligned} \Delta w &:= \mu \Delta w + \epsilon \left(\langle \frac{\partial E}{\partial w} \rangle_i - \omega w \right) \\ w &:= w + \Delta w \end{aligned}$$

where μ is the coefficient of momentum, ω is the coefficient of weight decay, ϵ is the learning rate, and $\langle \frac{\partial E}{\partial w} \rangle_i$ denotes the expectation of the weight gradient for a batch i.

When experimenting with different batch sizes, one must decide how to adjust the hyperparameters μ, ω, and ϵ. It seems plausible that the smoothing effects of momentum may be less necessary with bigger batch sizes, but in my experiments I used $\mu = 0.9$ for all batch sizes. Theory suggests that when multiplying the batch size by k, one should multiply the learning rate ϵ by $\sqrt{k}$ to keep the variance in the gradient expectation constant. How should we adjust the weight decay ω? Given old batch size N and new batch size $N' = k \cdot N$, we'd like to keep the total weight decay penalty constant. Note that with batch size N, we apply the weight decay penalty k times more frequently than we do with batch size N'. So we'd like k applications of the weight decay penalty under batch size N to have the same effect as one application of the weight decay penalty under batch size N'. Assuming $\mu = 0$ for now, k applications of the weight decay penalty under batch size N, learning rate ϵ, and weight decay coefficient ω give

$$\begin{aligned} w_k &= w_{k-1} - \epsilon\omega w_{k-1} \\ &= w_{k-1}(1 - \epsilon\omega) \\ &= w_0(1 - \epsilon\omega)^k. \end{aligned}$$

While one application of weight decay under batch size N', learning rate ϵ' and weight decay coefficient ω' gives

$$\begin{aligned} w'_1 &= w_0 - \epsilon'\omega' w_0 \\ &= w_0(1 - \epsilon'\omega') \end{aligned}$$

so we want to pick ω' such that

$$(1 - \epsilon\omega)^k = 1 - \epsilon'\omega'$$

which gives

$$\begin{aligned} \omega' &= \frac{1}{\epsilon'} \cdot \left(1 - (1 - \epsilon\omega)^{\,k}\right) \\ &= \frac{1}{\sqrt{k}\epsilon} \cdot \left(1 - (1 - \epsilon\omega)^{\,k}\right). \end{aligned}$$

So, for example, if we trained a net with batch size $N = 128$ and $\epsilon = 0.01, \omega = 0.0005$, the theory suggests that for batch size $N' = 1024$ we should use $\epsilon' = \sqrt{8} \cdot 0.01$ and $\omega' \approx 0.0014141888$. Note that, as $\epsilon \to 0$, $\omega' = \frac{1}{\sqrt{k}\epsilon} \cdot \left(1 - (1 - \epsilon\omega)^{\,k}\right) \to \sqrt{k} \cdot \omega$, an easy approximation which works for the typical ϵs used in neural nets. In our case, the approximation yields $\omega' \approx \sqrt{8} \cdot \omega \approx 0.0014142136$. The acceleration obtained due to momentum $\mu = 0.9$ is no greater than that obtained by multiplying ϵ by 10, so the $\sqrt{8} \cdot \omega$ approximation remains very accurate.

Theory aside, for the batch sizes considered in this note, the heuristic that I found to work the best was to multiply the learning rate by k when multiplying the batch size by k. I can't explain this discrepancy between theory and practice[3]. Since I multiplied the learning rate ϵ by k instead of $\sqrt{k}$, and the total weight decay coefficient is $\epsilon'\omega'$, I used $\omega' = \omega = 0.0005$ for all experiments.

As in [Krizhevsky et al., 2012], I trained on random 224×224 patches extracted from 256×256 images, as well as their horizontal reflections. I computed the validation error from the center 224×224 patch.

[1] In detail, the single-column model has 64, 192, 384, 384, 256 filters in the five convolutional layers, respectively.

[2] This is not an important point with only 1000 classes. But with tens of thousands of classes, the cost of normalization becomes noticeable.

[3] This heuristic does eventually break down for batch sizes larger than the ones considered in this note.

The machine on which I performed the experiments has eight NVIDIA K20 GPUs and two Intel 12-core CPUs. Each CPU provides two PCI-Express 2.0 lanes for four GPUs. GPUs which have the same CPU "parent" can communicate amongst themselves simultaneously at the full PCI-Express 2.0 rate (about 6GB/sec) through a PCI-Express switch. Communication outside this set must happen through the host memory and incurs a latency penalty, as well as a throughput penalty of 50% if all GPUs wish to communicate simultaneously.

5.1 Results

Table 1 summarizes the error rates and training times of this model using scheme (b) of Section 4. The main take-away is that there is an accuracy cost associated with bigger batch sizes, but it can be greatly reduced by using the variable batch size trick described in Section 4.2. The parallelization scheme scales pretty well for the model considered here, but the scaling is not quite linear. Here are some reasons for this:

- The network has three dense matrix multiplications near the output. Parallel dense matrix multiplication is quite inefficient for the matrix sizes used in this network. With 6GB/s PCI-Express links and 2 TFLOP GPUs, more time is spent communicating than computing the matrix products for 4096×4096 matrices.[4] We can expect better scaling if we increase the sizes of the matrices, or replace the dense connectivity of the last two hidden layers with some kind of restricted connectivity.

- One-to-all broadcast/reduction of scheme (b) is starting to show its cost. Scheme (c), or some hybrid between scheme (b) and scheme (c), should be better.

- Our 8-GPU machine does not permit simultaneous full-speed communication between all 8 GPUs, but it does permit simultaneous full-speed communication between certain subsets of 4 GPUs. This particularly hurts scaling from 4 to 8 GPUs.

6 Comparisons to other work on parallel convolutional neural network training

The results of Table 1 compare favorably to published alternatives. In [Yadan et al., 2013], the authors parallelize the training of the convolutional neural net from [Krizhevsky et al., 2012] using model parallelism and data parallelism, but they use the same form of parallelism in every layer. They achieved a speedup of 2.2x on 4 GPUs, relative to a 1-GPU implementation that takes 226.8 hours to train for 90 epochs on an NVIDIA GeForce Titan. In [Paine et al., 2013], the authors implement asynchronous SGD [Niu et al., 2011, Dean et al., 2012] on a GPU cluster with fast interconnects and use it to train the convolutional neural net of [Krizhevsky et al., 2012] using model parallelism and data parallelism. They achieved a speedup of 3.2x on 8 GPUs, relative to a 1-GPU implementation that takes 256.8 hours to train on an NVIDIA K20X. Furthermore, this 3.2x speedup came at a rather significant accuracy cost: their 8-GPU model achieved a final validation error rate of 45%.

7 Other work on parallel neural network training

In [Coates et al., 2013], the authors use a GPU cluster to train a locally-connected neural network on images. To parallelize training, they exploit the fact that their network is locally-connected but not convolutional. This allows them to distribute workers spatially across the image, and only neuron activations near the edges of the workers' areas of responsibility need to be communicated. This scheme could potentially work for convolutional nets as well, but the convolutional weights would need to be synchronized amongst the workers as well. This is probably not a significant handicap as there aren't many convolutional weights. The two other disadvantages of this approach are that it requires synchronization at every convolutional layer, and that with 8 or more workers, each worker is left with a rather small area of responsibility (particularly near the upper layers of the convolutional net), which has the potential to make computation inefficient. Nonetheless, this remains an attractive dimension of parallelization for convolutional neural nets, to be exploited alongside the other dimensions.

The work of [Coates et al., 2013] extends the work

[4] Per sample, each GPU must perform $4096 \times 512 \times 2$ FLOPs, which takes 2.09μs at 2 TFLOPs/sec, and each GPU must receive 4096 floats, which takes 2.73μs at 6GB/sec.

GPUs	Batch size	Cross-entropy	Top-1 error	Time	Speedup
1	$(128, 128)$	2.611	42.33%	98.05h	1x
2	$(256, 256)$	2.624	42.63%	50.24h	1.95x
2	$(256, 128)$	2.614	42.27%	50.90h	1.93x
4	$(512, 512)$	2.637	42.59%	26.20h	3.74x
4	$(512, 128)$	2.625	42.44%	26.78h	3.66x
8	$(1024, 1024)$	2.678	43.28%	15.68h	6.25x
8	$(1024, 128)$	2.651	42.86%	15.91h	6.16x

Table 1: Error rates on the validation set of ILSVRC 2012, with the model described in Section 5. *Batch size* (m, n) indicates an effective batch size of m in the convolutional layers and n in the fully-connected layers. All models use data parallelism in the convolutional layers and model parallelism in the fully-connected layers. *Time* indicates total training time in hours.

of [Dean et al., 2012], which introduced this particular form of model parallelism for training a locally-connected neural network. This work also introduced the version of the asynchronous SGD algorithm employed by [Paine et al., 2013]. Both of these works are in turn based on the work of [Niu et al., 2011] which introduced asynchronous SGD and demonstrated its efficacy for models with sparse gradients.

8 Conclusion

The scheme introduced in this note seems like a reasonable way to parallelize the training of convolutional neural networks. The fact that it works quite well on existing model architectures, which have not been adapted in any way to the multi-GPU setting, is promising. When we begin to consider architectures which are more suited to the multi-GPU setting, we can expect even better scaling. In particular, as we scale the algorithm past 8 GPUs, we should:

- Consider architectures with some sort of restricted connectivity in the upper layers, in place of the dense connectivity in current nets. We might also consider architectures in which a fully-connected layer on one GPU communicates only a small, linear projection of its activations to other GPUs.
- Switch from scheme (b) to scheme (c) of Section 4, or some hybrid between schemes (b) and (c).
- Reduce the effective batch size by using some form of restricted model parallelism in the convolutional layers, as in the two-column network of [Krizhevsky et al., 2012].

We can expect some loss of accuracy when training with bigger batch sizes. The magnitude of this loss is dataset-dependent, and it is generally smaller for larger, more varied datasets.

References

Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *Proceedings of The 30th International Conference on Machine Learning*, pages 1337–1345, 2013.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V Le, Mark Z Mao, Marc'Aurelio Ranzato, Andrew W Senior, Paul A Tucker, et al. Large scale distributed deep networks. In *NIPS*, pages 1232–1240, 2012.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, volume 1, page 4, 2012.

Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in Neural Information Processing Systems*, 24:693–701, 2011.

Thomas Paine, Hailin Jin, Jianchao Yang, Zhe Lin, and Thomas Huang. Gpu asynchronous stochastic gradient descent to speed up neural network training. *arXiv preprint arXiv:1312.6186*, 2013.

Omry Yadan, Keith Adams, Yaniv Taigman, and Marc'Aurelio Ranzato. Multi-gpu training of convnets. *arXiv preprint arXiv:1312.5853*, 2013.