Training a Large-Scale Word Detection Network Via Multiple GPUs

Stephen Miller
554 Salvatierra Walk
Stanford, CA 94305
sdmiller@stanford.edu

Abstract

Neural Networks have shown great promise in their ability to extract patterns from data and use them to perform exceedingly well on discriminative tasks, often in domains for which they were not explicitly designed. Yet training time often remains a bottleneck to exploring new architectures and scaling up existing ones. In this work, I speed up forward and backward propogation for a network which attempts to classify variable-sized words in audio. To do so, I exploit the parallelism of the GPU while maintaining simple, flexible code, to allow us to experiment with new architectures. I show that the GPU drastically improves training speed, and give examples of the boosted performance the quick training speed allowed us to reach.

1. Introduction

Deep Belief Networks have proven to be incredible tools for a variety of learning tasks, capable of detecting and exploiting patterns in both in unsupervised, and discriminatively trained settings. These have shown impressive results in a wide variety benchmarks, including Vision and Natural Language Processing [2] [7]. Unlike standard state-of-the-art approaches to these tasks, which often succeed by virtue of the expert knowledge engineered into them by their creators, the DBN paradigm has the attractive property of being extremely general, requiring little to know domain-specific knowledge on the part of the designer.

Yet, as anyone who has trained such a network can attest, generality does not equate to a one-size-fits-all algorithm. As with most any learning task, a number of decisions may drastically affect the success of the network. These may be architectural (how many hidden units to use, choice of receptive field, pooling methods, number of layers) or optimization-specific (which method to use, what batch size to consider, etc). While intuition and well-reasoned decisions are valuable in making such choices, there is an undeniable empirical component: in order to select the best architecture, one must evaluate its performance, note trends, and make changes accordingly. In order to evaluate performance frequently, one must be able to train in a reasonable amount of time. Furthermore, real world problems are rarely as simple as toy examples: any tool which hopes to see real-world application must be able to handle the massive variance inherent to the domain. In other words, the optimal network—regardless of architectural decisions—must be able to handle massive amounts of training data, and increase its own size to retain the ability to represent that data. This, like the selection process, makes training time a crucial bottleneck. A network which takes two weeks to train does not allow for fine evaluation, nor will it handle a 10x increase in data or layer size gracefully.

There is, however, much processing power at the programmer's disposal. In particular, GPU Processors allow for massively parallel computations, speeding up linear algebra and graphics calculations by orders of magnitude. Moreover, while researcher time is valuable, machine time is often cheap: a given task may be parallelized across multiple machines. Well-parallelized, the speed of a computation may then scale almost linearly with the number of machines.

In this work, I attempt to exploit this parallelism to train a convolutional neural network for word detection in audio data; parallelizing training across multiple machines with multiple GPUs, to make scaling-up feasible and architectural decisions realistic to test. I then show a few example results, of the type of improvements this was able to give our system.

2. Prior Work

Much work has been done in the realm of Convolutional Neural Networks. Far too much, unfortunately, for the scope of this paper. For a few example works on convolutional networks, readers are invited to refer to the work of [5] and, more recently, [6].

Speeding up the training of networks via GPUs has become quite popular in recent years. Bergstra et al [1] introduce a Python library for full training of neural networks in a very modular way. R. Collobert presents a similar library

for Lua.[1] Recent work by Farabet implemented fast neural network training on FPGAs [4]. Finally, A. Krizhevsky has produced much code for training neural networks via GPUs.[2] However, as our detection task is unique given its variable-sized labelled training data, these do not, as simple black box methods, apply to our task.

3. Preliminaries

In this paper, the terms "Network", "Neural Network", and "Deep Belief Network" are used loosely and synonymously to refer to any learning system consisting of interconnected stacks of units, called "neurons", with associated magnitudes, called "activations", whose values follow a simple update rules determined solely by the activations of their connected neighbors. For the sake of brevity, a full discussion of these is impossible, but I invite readers to visit [3] for an excellent introduction to the subject.

4. Problem Statement

An integral part of any speech recognition system is word detection: this is yet another example of a field in which many experts, particularly linguists, have designed methods based on strong domain-specific expertise. Thus, in this particular work, we explore the power of Deep Belief Networks on such a task . Namely, given an input audio feed and indices indicating a candidate word, we wish to learn to classify that word as one of the top 100 from a candidate dictionary.

This problem poses a number of unique challenges to fast optimization. In particular:

- The candidate words we are asked to classify are of variable length. Thus, standard techniques which exploit the fixed structure of the input cannot be used as is.
- Our input data is extremely sparse: that is, while we have many frames of audio, only a (comparatively) small number of them are words.
- Our input is multidimensional, but we convolve it in a single dimension: time. Thus, we can neither use standard one-dimensional convolution techniques (which assume a dimensionality of 1) nor conventiently use two-dimensional ones (which are either needlessly slow, or make "square window, square image" assumptions which we cannot satisfy.
- Due to the difficult nature of speech, we must be able to process very large amounts of input data.

[1] In current proceeding of NIPS 2011. See http://torch5.sourceforge.net/credits.html

[2] http://www.cs.toronto.edu/ kriz/

[3] http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial

We wish to explore networks with the following components:

- The input layer: $X_1 \in R^{13\times F}$, where $X_1^{(:,t)}$ are the Mel-Frequency Cepstral Coefficients (MFCCs, [3]) at the frame t. Here, F denotes the number of frames in our training example.
- The output layer: $D \in R^{100\times N}$, where $D^{(i,j)}$ is the output of a logistic regression unit corresponding to the ith word in our dictionary, on the jth labeled word in the input.
- Convolutional hidden layers, $X_l \in R^{H_l\times F_l}$ formed by temporally convolving a lower layer with a filter $W_l \in R^{d\times\omega_l}$ and applying an activation function h_l elementwise to the output, where H_l are the number of hidden units, F_l the resulting signal size, d the dimensionality of the lower layer, ω_l a fixed window size, and h_l either sigmoid or soft sign[4].
- Fixed pooling layers, P_f^l, which combine equally-sized neighborhoods of their input layer into one output, via a function $f_p(x_1, x_2, \dots) = x_o$. Here we consider f_p to be either the mean of the inputs, or an L-p norm of the inputs. The former case is mean pooling: when p is large, the latter approximates max pooling.
- A local pooling layer, P_L, which locates words and splits them into l_p equally sized subsets, combining them again via some function f_p.

Finally, given an architecture, there must be an optimization scheme: a mechanism for minimizing the given cost function. Multiple methods were considered, including Stochastic Gradient Descent and Conjugate Gradient. However, we have found the more complex line search approach of LBFGS [8] to have the best performance. As our data size is large (and presumed infinite for the intents of this work), we run the LBFGS algorithm only on small batches of data, deemed minibatch-LBFGS.

5. Our Approach

A key goal in optimizing our network is to allow us to vary training and architectural decisions and choose the best among them. Therefore, we do not at this time wish to fit our approach too tightly to a particular one of these decisions. Thus, we presently choose to:

- Restrict our parallelism to individual gradient computations. This allows us to use any gradient-based optimization method with no change to the underlying code, while retaining consistent speed-ups.

[4] $h(x) = \frac{x}{1+|x|}$

- Optimize individual pieces of this network, while retaining modularity.

When processing with a GPU, it is extremely costly to move data on and off memory. Therefore, I first slice the data into small enough pieces, such that an entire gradient computation can be done for that piece while keeping all necessary variables in GPU memory. The slicing, and subsequent operations in the gradient computation, are given below. Finally, I explain how this can be parallelized across multiple machines and GPUs.

All linear algebra computations that follow were computed using Jacket, a fast GPU-based linear algebra package for MATLAB [5].

5.1. Data preprocessing

In our optimization, the smallest unit of labelled input data—a minibatch—corresponds to an hour, or 36000 frames, of audio. This, unfortunately, is far too large for the GPU to perform any relevant computations. As a preprocessing step when an hour of data is loaded into our machine, then, we consider the particular architecture of our network and estimate the maximum length of time which can be processed entirely in memory. The data and labels are then sliced according to this time, and the gradient computation is done separately for each. The sum of all resulting outputs yields the full gradient.

5.2. Hidden Layers

Our hidden layers are convolutional. We would expect, then, that an explicit convolution (such as MATLAB's conv2 function) would be well suited to the task. However, time results show this to be far slower than another, more naive approach: computing each window of the input data separately, and performing a left-matrix multiply by the weight matrix, stacked in vector form. Each of these computations is extremely well optimized in Jacket.

Forward propogation (ignoring nonlinear activation) then becomes:

$$X_{l+1}^{(i,j)} = \sum_{k,l} W_l^{(m,n)} w_j(X_l^{(m,n)})$$

Where w_j selects the jth sliding window of X. In more compact, MATLAB notation:

$$X_{l+1} = W_l \textsc{im2col}\left(X_l\right)$$

And backpropogation a symmetric step, with minor bookkeeping of indices rendering it difficult to make explicit here.

[5] http://www.accelereyes.com/

5.3. Pooling Layers

As words may take on variable lengths, the local pooling step—which separates a word into a fixed number of groups— does not lend itself to simple matrix computations. Yet the cost of manually computing each pooled output, be it in MATLAB or C++, is extraordinarily high. Instead, I noted that any pooling step, fixed or variable, could be written as:

$$\begin{aligned} X_{l+1}^{(i,j)} &= \sqrt[p]{\sum_k X_l^{(i,k)^p} S^{(k,i)}} \\ X_{l+1} &= \sqrt[p]{X_l S} \end{aligned}$$

Where S is a sparse matrix, where $S^{(i,j)}! = 0$ indicates that the jth frame of the input lies in pooling region i. If the nonzero elements are 1, this corresponds to L-p pooling. If p = 1 and S is normalized such that its columns sum to 1, this becomes mean pooling. Thus, the problem reduces to efficiently computing pooling matrix S. So doing, we may exploit MATLAB/Jacket's efficient matrix multiplication routines as usual, losing little memory as long as S is sparse.

For L-p, the forward and backward steps become:

$$\begin{aligned} X_{l+1}^{(i,j)} &= \sqrt[p]{\sum_k X_l^{(i,k)^p} S^{(k,i)}} \\ X_{l+1} &= \sqrt[p]{X_l S} \\ \frac{\delta C}{\delta X_l^{(i,j)}} &= X_l^{(i,j)^{p-1}} \sum_k \frac{\delta C}{\delta X_{l+1}^{(i,k)}} X_{l+1}^{(i,k)^{1-p}} S^{(i,k)} \\ \nabla_{X_l} C &= X_l^{p-1} \circ \left(\left(\nabla_{X_{l+1}} C \circ X_{l+1}^{1-p} \right) S^T \right) \end{aligned}$$

In the case of average pooling, this simplifies to:

$$\begin{aligned} X_{l+1}^{(i,j)} &= \sum_k X_l^{(i,k)} S^{(k,i)} \\ X_{l+1} &= X_l S \\ \frac{\delta C}{\delta X_l^{(i,j)}} &= \sum_k \frac{\delta C}{\delta X_{l+1}^{(i,k)}} S^{(i,k)} \\ \nabla_{X_l} C &= \nabla_{X_{l+1}} S^T \end{aligned}$$

With fixed pooling regions, computing S becomes quite trivial, and may be done efficiently in MATLAB. Variable length regions, however, have no clean vectorized solution. Thus, I implemented this step in C++, using MEX to read labels and data sizes from MATLAB, and output S in sparse format.

5.4. Multiple Computers, Multiple GPUs

The above details how computation can be done on a single machine with a single GPU. Both multiple machines,

and multiple GPUs, are currently handled quite simply. Every GPU on every machine is considered a slave. When training, the data and labels are then distributed evenly among all slaves, and the gradient computations of each are combined.

6. Results

The above changes, as well as efficient memory management and a reordering of operations wherever possible, were implemented on our system. Fig. 1 shows a comparison of the gradient computation steps on a GPU vs multiple CPUs, as naturally exploited by MATLAB.

Furthermore, it is important to keep in mind that the goal of this work was not solely to improve speed for its own sake, but also to do so in order to train a working Speech system. Prior to its implementation, only relatively small networks (500 hidden units, a convolutional window of 7) were able to be trained, as the procedure spanned days. With the speedup, we have been able to examine a number of different architectural and optimization decisions, including number of hidden units, number of layers, and number of minibatch-LBFGS iterations. Some examples can be seen in Fig. 2.

7. Conclusion

Data-driven approaches to learning are a clear path for the future of Artificial Intelligence. As this continues to grow, the need for fast, scalable systems must increase to handle that influx of data. In this work, I present a small step towards that goal, showing how a GPU was utilized to speed up the training of a Word Detector network. This allowed us to go from a training time of days to one of hours, and thus, learn more about the proper architecture for the system while it is in its infantile stages, and improve efficiency as the amount of data increases.

References

[1] 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 *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010.

[2] A. Coates and A. Ng. The importance of encoding versus training with sparse coding and vector quantization. In *Proc. of the 28th Int. Conf. on Machine Learning*, 2011.

[3] S. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 28(4):357–366, 1980.

[4] C. Farabet, Y. LeCun, K. Kavukcuoglu, E. Culurciello, B. Martini, P. Akselrod, and S. Talay. Large-scale fpga-based convolutional networks, 2011.

[5] Y. LeCun and Y. Bengio. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361, 1995.

[6] H. Lee, R. Grosse, R. Ranganath, and A. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 609–616. ACM, 2009.

[7] R. Socher, C. Manning, and A. Ng. Learning continuous phrase representations and syntactic parsing with recursive neural networks. In *Proceedings of the NIPS-2010 Deep Learning and Unsupervised Feature Learning Workshop*, 2010.

[8] C. Zhu, R. Byrd, P. Lu, and J. Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software (TOMS)*, 23(4):550–560, 1997.

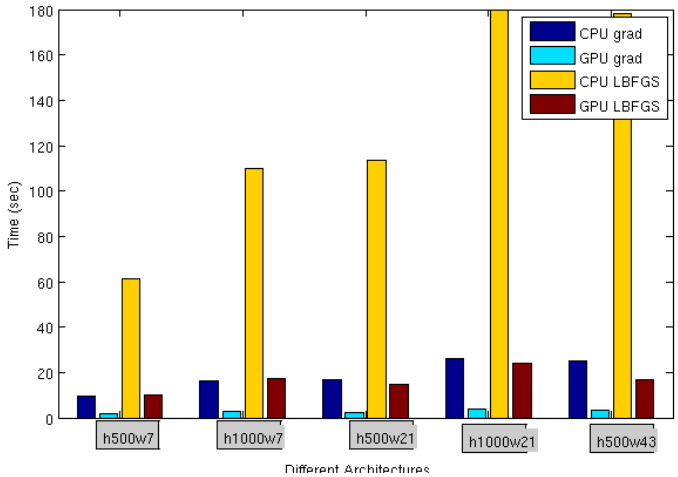

Figure 1. Timing comparisons, running on 1 hour of audio. h500w7 indicates a single hidden layer size of 500, with a convolutional window of 7. In each group, the left two bars show the time it takes to do one gradient computation. The right shows the time to run minibatch-LBFGS using 5 line search iterations. Note that in all of these cases, Jacket's Sparse Library is no longer available for testing here, so the pooling matrices are forced to be represented densely. When S is made sparse, speed increased by another factor of roughly 3.

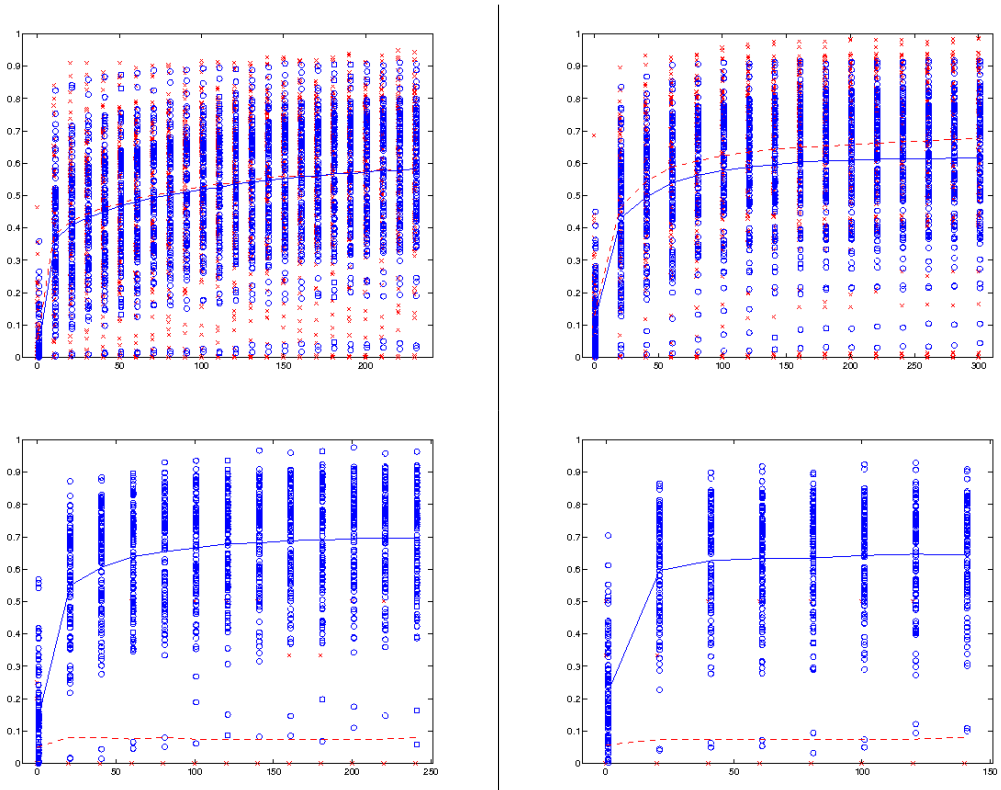

Figure 2. Improved AP on larger networks, trained on 500 hours of audio. TL: The initial performance on 500 hiden units. BL: Our current best, with 1000 hidden units and a window size of 21. TR: 1000 hidden units, 20 LBFGS iterations. With the ability to run many experiments quickly, we found that varying the optimization helped boost performance, with BR: 1000 hidden units, 40 LBFGS iterations.