ON PARALLELIZABILITY OF STOCHASTIC GRADIENT DESCENT FOR SPEECH DNNS

Frank Seide[1], *Hao Fu*[1,2], *Jasha Droppo*[3], *Gang Li*[1], *and Dong Yu*[3]

[1] Microsoft Research Asia, 5 Danling Street, Haidian District, Beijing 100080, P.R.C.
[2] Institute of Microelectronics, Tsinghua University, 10084 Beijing, P.R.C
[3] Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
{fseide,jdroppo,ganl,dongyu}@microsoft.com, fuhao9202@hotmail.com

ABSTRACT

This paper compares the theoretical efficiency of model-parallel and data-parallel distributed stochastic gradient descent training of DNNs. For a typical Switchboard DNN with 46M parameters, the results are not pretty: With modern GPUs and interconnects, model parallelism is optimal with only 3 GPUs in a single server, while data parallelism with a minibatch size of 1024 does not even scale to 2 GPUs.

We further show that data-parallel training efficiency can be improved by increasing the minibatch size (through a combination of AdaGrad and automatic adjustments of learning rate and minibatch size) and data compression. We arrive at an estimated possible end-to-end speed-up of 5 times or more.

We do not address issues of robustness to process failure or other issues that might occur during training, nor of speed of convergence differences between ASGD and SGD parameter update patterns.

1. INTRODUCTION AND RELATED WORK

At present, the best context-dependent deep-neural-network HMMs, or CD-DNN-HMMs [1, 2], are typically trained primarily with error back-propagation (BP), a form of stochastic gradient descent, or SGD. For production-size models and corpora, this is a time-consuming task that can take many days or weeks, even on modern GPGPU hardware.

While DNNs for vision are commonly convolutional with local connectivity [3, 4], typical DNNs for acoustic modelling are fully connected between layers. This makes parallelizing back-propagation over multiple compute nodes inefficient. E.g., Google's DistBelief system successfully utilizes 16,000 cores for the ImageNet task through asynchronous SGD [3], while for a speech model with 42M parameters, a 1,600-core DistBelief [5] is only marginally faster than a single recent GPU. [4] achieved a 28-fold speed-up with 64 GPUs for their best, 1.9B-parameter vision network, while [6] reports a 3.2-times speed-up from ASGD using 4 GPUs for speech.

Unlike, e.g., [7, 8] which factored the network into a hierarchy, or low-rank approximations [9], or ADMM which cleverly tweaks the objective function for better parallelizability [10], we focus in this paper on parallelizing plain no-frills SGD, through *model parallelism* and *data parallelism*.

In earlier work, we attempted to parallelize SGD over layers—each node (GPU) processed one or more consecutive layers, where data flowed up and down through the layers, and gradients only became available at a delay of several minibatches. The lower the layer, the larger the delay [11]. That work showed that such delayed update can in fact still work, and it motivated an alternative understanding of minibatches that we present in this paper. Layer parallelism achieved a 3.3-times speed-up on 4 GPUs, but it does not scale beyond the number of layers, and with a vastly larger output layer, load balancing is a prohibitive issue.

This paper analyses the upper bound of parallelizability of SGD using either *model parallelism* or *data parallelism*. We introduce a formalism to estimate the optimal number of compute nodes to maximize the efficiency of computation, using measured hardware parameters such as computation speed and data-exchange bandwidth. A direct consequence of this analysis is that to maximize the efficiency of data parallelism, one must maximize the minibatch size (while retaining convergence speed). We introduce an approach to do that, and evaluate its effectiveness on the 309h Switchboard corpus.

The paper is organized as follows. We will first introduce the formalism in sections 2 and 3. We will then motivate and introduce our approach to effectively increase the minibatch size in section 4. Section 5 shows the results for the estimated optimal number of nodes, as well as runtime measurements for model parallelism and word-error rates for growing the minibatch size for data parallelism.

2. TRAINING CONTEXT-DEPENDENT DEEP-NEURAL-NETWORK HMMS

A deep neural network (DNN) is a conventional multi-layer perceptron (MLP [12]) with many layers, where training is commonly initialized by a pretraining algorithm [13, 14, 15]. A CD-DNN-HMM models the posterior probability $P(s|o)$ of a tied triphone state, or senone s [16, 1], given an observation vector o. For details, please see, for example, [15].

The best known DNNs to this date are trained using the common error back-propagation (BP) technique [17]. Even when other techniques are used, such as the Hessian-free method [18, 19, 9], BP still often constitutes a significant portion of their training time.

BP is a form of stochastic gradient descent, which we want to write in a slightly unusual way as as an iteration over *individual samples* (indexed by sample index t) rather than, as common, over minibatches:

$$\lambda^{(t+1)} = \lambda^{(t)} + \epsilon^{(t)} \cdot \frac{\partial \mathcal{F}_\lambda(o(t))}{\partial \lambda}\bigg|_{\lambda=\lambda^{(\tau)}} \quad (1)$$

Here, $\lambda^{(t)}$ denotes the model at "current" sample index t,

while $\lambda^{(\tau)}$ is meant to denote a slightly "outdated" model at index $\tau \leq t$. It is this model that the partial gradient of the objective function $\mathcal{F}_\lambda$ for the current sample vector $o(t)$ is evaluated on. $\epsilon^{(t)}$ is the learning rate at this training stage.

With this, minibatching can be described by defining $\tau = t - (t \bmod N)$ with minibatch size N. I.e., τ is rounded down to multiples of N. As long as t falls within the same minibatch, the formula simply sums up individual frames' gradients computed on the same model $\lambda(\tau)$. Note that we define a minibatch as a *sum*; and not, as common, as an average. We account for this by using an N times smaller value for ϵ.

This notation is useful because more generally, $\tau < t$ means that gradients are computed using a model that is $t - \tau$ samples "outdated." Any optimal variant of data parallelism necessarily implies some form of such *delayed update*—new samples are processed while results from previous samples are still being transferred concurrently. A popular variant of this is asynchronous SGD, or ASGD [5], where τ varies non-deterministically across model parameters.

Hence, Eq. (1) allows us to understand more complex forms of delayed updates as something *qualitatively similar to minibatching*, and thus we can *expect similar convergence behavior* as long as the update delay stays in a similar range.

3. OPTIMALITY

We define a parallel system that is training optimally as one where computation and data exchange happen *concurrently with perfect overlap*; that is, simultaneously saturating the communcation channel and the processing resources. If the communication channel is not saturated, then the system could be improved by parallelizing more. If the processing resources are not saturated, it could be improved by parallelizing (communicating) less or by processing more data.

If K denotes the number of compute nodes,[1] the optimal number of nodes $\hat{K}$ is reached when communication and processing time are balanced:

$$T_{\text{calc}}(\hat{K}) \quad = \quad T_{\text{comm}}(\hat{K}). \tag{2}$$

Here T_{calc} and T_{comm} are the time *per minibatch* for concurrent per-node calculation and inter-node communication, respectively.[2] Calculation is dominated by 3 large equal-size matrix products—forward propagation, error back-propagation, and gradient computation—so without any parallelization ($K = 1$), it takes

$$T_{\text{calc}}(1) \quad = \quad \frac{M}{f} \cdot \left(3\,N \cdot \eta_{\text{mbs}}(N) + C \cdot \eta_{\text{fix}}\right) \tag{3}$$

to compute an N-frame minibatch, where M is the number of model parameters and f the computation speed (FLOPS).

[1] A node can be defined as a GPU, a CPU core, or a multi-core server.

[2] Note that our aim is to maintain convergence rate w.r.t. samples presented to the learning algorithm—that the parallelized version will present the same number of samples (or do the same number of data passes) until the target model is achieved. It is conceivable that a modification exists that might slow down convergence speed while allowing for a parallelization speed-up that more than makes up for it. We do not consider this case here.

The GPU matrix product (SGEMM) is less efficient for smaller matrices due to caching. Eq. (3) accounts for this by a "reality factor" $\eta_{\text{mbs}}(N)$ which denotes the slow-down due to N being smaller than its optimal value. E.g., for our model size, $\eta_{\text{mbs}}(256) \approx 1.4$ on a K20X GPU. The subsequent formulas below will contain more such "reality factors."

Lastly, C counts "fixed cost" steps (AdaGrad accumulation+weighting and momentum+model update: $C = 4$), and η_{fix} is their overhead (e.g. memory latency on GPUs).

We will now derive formulas for T_{calc} and T_{comm} for two forms of parallelization: *model* and *data* parallelism.

3.1. Model Parallelism

Model parallelism means that *model matrices* are distributed over nodes. Each of the K nodes holds and computes a $1/K$-th sub-section of each weight matrix. In forward propagation, each node takes full-dimension input vectors and computes partial vectors of $1/K$-th that dimension. This takes $1/K$-th of the time. Each node must then send that vector to all other $K - 1$ nodes. All this happens in half-batches such that one half can be computed while the previous half's data is concurrently being transferred (*double buffering*). Back propagation has a very similar pattern. For either, we get:

$$T_{\text{calc}}(K) \quad = \quad \frac{M}{f} \cdot N \cdot \frac{1}{K} \cdot \left[\eta_{\text{mbs}}(\tfrac{N}{2}) \cdot \eta_{\text{strp}}(K)\right]$$

$$T_{\text{comm}}(K) \quad = \quad \frac{A_{\text{fr}} \cdot w_A}{b} \cdot N \cdot \frac{K-1}{K} \cdot \left[\lambda_{\text{blk}}(\tfrac{N}{K})\, \lambda_{\text{conn}}(K)\right]$$

with A_{fr} denoting the total number of activation parameters of byte size w_A to be passed per minibatch. η_{strp} accounts for a further SGEMM inefficiency due to smaller matrix height, and λ_{blk} for sub-peak data speed due to using suboptimally small blocks. λ_{conn} accounts for peer-to-peer overhead and connectivity limits, like the number of PCIe lanes. Altogether, the optimal $\hat{K}$ for forward and back propagation is:

$$(\hat{K} - 1) \cdot \left[\frac{\lambda_{\text{conn}}(\hat{K}) \cdot \lambda_{\text{blk}}(\frac{N}{\hat{K}})}{\eta_{\text{strp}}(\hat{K}) \cdot \lambda_{\text{blk}}(N)}\right] \quad = \quad \frac{M}{A_{\text{fr}} \cdot w_A} \cdot \frac{b}{f} \cdot \left[\frac{\eta_{\text{mbs}}(\frac{N}{2})}{\lambda_{\text{blk}}(N)}\right]$$

3.2. Data Parallelism

Data parallelism partitions each *minibatch* across the K nodes, which compute K sub-gradients, which then must be aggregated ("all-reduced") across nodes. The all-reduce operation is of $\mathcal{O}(1)$ by applying a similar pattern as for model parallelism (K steps of K concurrent exchanges of M/K values) twice (aggregation, redistribution). Again assuming double-buffering of half-minibatches for concurrency, we get:

$$T_{\text{calc}}(K) \quad = \quad 2\,\frac{M}{f}\left(3\,\frac{N}{2} \cdot \frac{1}{K} \cdot \eta_{\text{mbs}}(\tfrac{N}{2K}) + (C+1)\,\eta_{\text{fix}}\right)$$

$$T_{\text{comm}}(K) \quad = \quad 2\,\frac{M \cdot w_M}{b} \cdot \frac{2K}{K} \cdot \left[\lambda_{\text{blk}}(\tfrac{M w_M}{K})\, \lambda_{\text{conn}}(K)\right]$$

where w_M is the width in bytes of a model parameter (4 for single-precision floating point). The optimal $\hat{K}$ is:

$$\hat{K} \quad = \quad \frac{N \cdot \frac{3}{2}\frac{1}{f} \cdot \eta_{\text{mbs}}(\frac{N}{2\hat{K}})}{w_M \cdot \frac{1}{b} \cdot 2\left[\lambda_{\text{blk}}(\frac{M w_M}{K})\, \lambda_{\text{conn}}(\hat{K})\right] - \frac{1}{f}\,(C+1)\,\eta_{\text{fix}}}$$

This reveals two main avenues to increase parallelizablity: by growing N—*maximizing minibatch size*— and by decreasing w_M—*data compression*, where the former is limited by training stability, and the latter is bounded by the fixed cost as well as accuracy. This paper only explores the former.

Unlike for model parallelism, double-buffering with half-minibatches *changes* the model being trained, because the update delay τ (cf. Eq. (1)) is changed from $t - (t \bmod N)$ to $t - \frac{N}{2} - (t \bmod \frac{N}{2})$. The delay remains in a similar range, though, so its convergence behavior should remain similar.

3.2.1. Asynchronous SGD

ASGD [5] is a form of data parallelism, where nodes ("model replica") communicate through parameter servers, and parameters are read and written while being accessed unsynchronized for computation. Thus, individual model parameters may be outdated by one iteration, in a non-determistic fashion. This does not cause detriment due to the linearized nature of SGD. Rather, it is another form of delayed update.

The above estimate of $\hat{K}$ applies to ASGD. ASGD allows for more flexible rounding where $\hat{K}$ is not an integer. This can reduce the effective update delay by the order of N/K, but not more: If it did, one should add another node.

The lesson here is that ASGD does not improve parallelizability in a fundamental way. If deterministic double-buffered data parallelism does not scale well, ASGD won't either.

4. GROWING THE MINIBATCH SIZE

Section 3.2 shows that one approach to maximizing parallelizability of data parallelism (including ASGD) without affecting convergence speed, is to drive up minibatch size N.

We find that at any point during the training, there is an upper limit to N (let's call it the *feasible* limit), above which convergence slows notably and eventually fails [11]. Below the feasible limit, N does not affect convergence speed (assuming we do not divide the gradient sum by N, per Eq. (1)). Luckily, we also observe that the feasible limit is larger for more mature models, and that a k times increase of the learning rate can allow for an up to k times larger feasible limit.

This informs our approach to drive up the minibatch size, which consists of four steps as follows.

First, we implement "AdaGrad" [20], which we found to lead to more mature models earlier, thus allowing for larger N. AdaGrad normalizes the components of the gradient $g(t) = \partial \mathcal{F}_\lambda(o(t))/\partial\lambda$ by their standard deviation over time:

$$\begin{aligned} g'_{i,j}(t) &= g_{i,j}(t) \cdot (\sigma_{i,j}(t)/\overline{\sigma_{i,j}(t)})^{-1} \\ \sigma^2_{i,j}(t) &= h(t) * g^2_{i,j}(t) \end{aligned}$$

where $h(t)$ is a unit-gain first-order low-pass filter (we use a time constant 2h of data). We gracefully ignore the mean in the variance calculation. Our version differs from the original AdaGrad [20] which can be described as $h(t) = 1$ and implies an undesired $1/t$ learning-rate schedule ("AdaDec" [21] addresses this similarly). We also differ in that we normalize the standard deviations $\sigma_{i,j}$ by their per-matrix averages $\overline{\sigma_{i,j}}$. The aim is to keep gradients in a similar numeric range, hoping to keep previously tuned learning rates valid.

Secondly, we use an automatic method to shrink the learning rate. Smaller learning rates allow for larger N. Every 120h of data, we compute frame accuracy on a cross-validation set. If it grew only slowly, we halve the learning rate; and if it dropped, we revert the model—fairly standard.

Lastly, we dynamically determine the feasible limit of N: After every 24h-block of data, we try a range of minibatch sizes on about 2% of the next block and pick the largest feasible one based on the objective function's value.

5. EXPERIMENTAL RESULTS

We will provide the optimal $\hat{K}$ estimates for model and data parallelism for training a realistic state-of-the-art speech-to-text transcription system. For model parallelism, we will also give actual runtime measurements; while for data parallelism, we will report on an experiment on driving up minibatch size that aims to inform the design of a future actual optimized implementation of data parallelism.

The CD-DNN-HMM in this system is trained using the 309-hour Switchboard-I training set [22]. The model has 7-hidden layer of dimension 2k, and an output dimension of 9304. This results in $M = 46$M model parameters and $A_{\text{fr}} = 12$k activation values to be propagated per frame in both forward and back-propagation.

Our main hardware is an AMAX ServMax equipped with 8 NVidia Tesla K20X GPU cards. For these, we measure $f = 976$ Gop/s for our model when using large minibatches (16k). $N = 1024$ is only marginally slower, but the slow-down from 1024 to 512 is a noticeable $\eta_{\text{mbs}}(512)/\eta_{\text{mbs}}(1024) = 1.2$. The fixed cost is $M/f \cdot C\,\eta_{\text{fix}} = 18.2$ ms.

Our standard training uses a minibatch size of $N = 1024$ (except for the first 24h, where we set $N = 256$ to ensure convergence). Single-channel peer-to-peer bandwidth when exchanging activations in chunks of 1024 frames was measured to $b/\lambda_{\text{blk}}(1024) = 4.83$ GB/s (PCIe 2.0 maximum: 6 GB/s). With our communication structure, the GPU hardware allows for fully concurrent data transfers, but is limited by the PCIe bus' 40 lanes in our 8-GPU server. We find the aggregate bandwidth to fluctuate strongly and max out at about 14 GB/s for 4 GPUs. Reliably, for 3 GPUs, we observe a slow-down of $\lambda_{\text{conn}} = 2$.

5.1. Model Parallelism

Based on the above, we can estimate the optimal number of nodes. It is not pretty: $\hat{K} = 3.3$. Model parallelism for a 46M-parameter DNN cannot benefit from more than 3 GPUs. Accounting for the slow-down from smaller matrices (striping, double-buffering), the estimated speed-up is 2.5.

Reality is not too far off: Table 1 shows processing speed in frames per second (fps) of an optimized, double-buffered implementation, for both the entire end-to-end BP (including data loading, objective-function tracking, etc.) and broken out by the three main steps, forward propagation, error back-propagation, and model-parameter update (which requires no data exchange). We see that with three K20X GPUs, we can

Table 1. *Processing speed in frames (=samples) per second for model parallelism on two kinds of GPUs. Using more than 3 GPUs did not lead to a further speed-up.*

configuration	speed [fps] (speed-up) 1 GPU	2 GPUs	3 GPUs
K20X, end-to-end	**6861**	10722	**12311 (1.8×)**
forward prop only	21248	31351	39565 (1.9×)
error back-prop only	25966	33211	36720 (1.4×)
param update only	18027	32787	43233 (2.4×)
C2075, end-to-end	2236	3906	-

achieve a speed-up from 6861 to 12311 fps, a factor of 1.8 (60% efficiency). Using 4 GPUs is slower; the PCIe data-transfer concurrency does not hold up. Using slower C2075 GPUs in a desktop computer, the speed-up for two is 1.8.

5.2. Data Parallelism

Using the standard minibatch size of $N = 1024$, the situation is even bleaker for data parallelism: The optimal number of nodes is $\hat{K} \approx 1.5$. For $N = 4096$, $\hat{K}$ increases only to 2 (limited by PCIe's $\lambda_{\text{conn}} \approx 2$), and to 6 for $N = 16$k (assuming $\lambda_{\text{conn}} \approx 3$), with expected speed-ups of 1.1 and 5.1, respectively. Assuming moderate data compression to 8-bit values would raise $\hat{K}$ to 5 and 12 with estimated speed-ups of 4 and 10.3, respectively. A critical factor is λ_{conn}, which, in our experience so far, can behave unexpectedly. We do not have an optimized implementation of data parallelism at this point in time, so we cannot provide actual time measurements.

We can, however, evaluate whether our method of Section 4 even makes such minibatch sizes feasible, to inform a future implementation. We use a non-parallelized full-minibatch update as a proxy, which has an update delay and thus convergence behavior that should be similar to an actual double-buffered half-minibatch update.

Table 2 shows the progression of word-error rates (WERs) over training data passes for speaker-independent recognition of the NIST 2000 Hub5 evaluation set (SWB part). The rows of small numbers show learning-rate (LR) profiles, as the factor (range) by which the original LR $\epsilon^{(0)} = 1/320$ per frame was reduced in that data pass. The last row also includes auto-adjusted minibatch sizes N (as the inverse average within a pass, very roughly proportional to the estimated speed-up).

First, we see that AdaGrad leads to somewhat faster convergence, and reaches the optimum one data pass earlier. Secondly, our standard LR profile—cut LR after 3 data passes sharply by a factor of 40 [2]—does not lead to the best possible accuracy; automatic LR control reduces WER from 15.8 to 15.3%. Lastly, automatic adjustment of the minibatch size N using the proposed method allows for nearly the same convergence while driving the minibatch size up to a maximum of 181k. For the first four data passes, which nearly reach the final accuracy, N ranges from 3.6k to 14.3k, which would allow for an end-to-end speed-up of about 5 (based on the above 4k/16k-minibatch estimates). Parallelism for the later data passes ($N = 87$k and above) will be limited by the realizable bandwidth across servers, and also the fixed cost.

Table 2. *Word error rates, learning-rate profiles, and minibatch growth over data passes (various training setups).*

setup \ data passes:	1	2	3	4	5	6	7
baseline WER[%]	19.6	17.8	16.9	16.1	16.1	**15.9**	16.0
$\epsilon^{(t)} = \epsilon^{(0)}/\ldots$	1	1	1	40	40	40	40
+ AdaGrad	18.4	17.3	17.1	16.3	**15.8**	15.9	15.7
	1	1	1	40	40	40	40
+ automatic LR control	17.6	16.7	16.1	15.6	**15.3**	15.3	15.3
	1..2	2	2..4	4..16	16..64	128..512	..2048
+ automatic N adjustment	19.7	17.1	16.1	15.5	15.6	15.5	**15.4**
	1	1	1..2	2..4	4..8	16..64	..256
$\overline{N^{(t)}}$	*3.6k*	*4k*	*6.8k*	*14.3k*	*87k*	*181k*	*181k*

5.3. Can a CPU-based farm beat a GPU-based one?

Probably not. While slower computation speed would allow for proportionally greater parallelism at the same communication latency, parallelization gains will be limited by "reality factors" due to using smaller sub-batch sizes. The SGEMM performance of a 16-CPU core machine using Intel's Math Kernel Library was measured to be nearly 5 times worse than that of a single K20X GPU [23]—we'd need 5 times smaller stripes to make up for it. Moreover, at slower computation speed, the fixed cost may become a limiting factor.

Indeed, [5] reports that for speech DNNs of similar size as ours, the CPU-based ASGD system maxes out at a 5.8-times combined speed-up from model and data parallelism on 80 20-CPU core nodes (while the same system successfully harnesses 1,000 16-core nodes for ImageNet [3]).

6. CONCLUSION

We compared the theoretical efficiency of distributed stochastic-gradient descent training of DNNs, for model and data parallelism of a typical Switchboard DNN with 46M parameters. For this fully connected model, parallelization quickly runs into the bandwidth bottleneck: For model parallelism, there is no benefit to use more than a meager 3 GPUs; and data parallelism shows no benefit even from 2 GPUs as long as the standard minibatch size of 1024 is used.

However, combining AdaGrad, automatic adjustment of learning rate and minibatch size, and data compression may enable designs with end-to-end speed-ups of above 5. A suitable server farm would consist of few servers with many high-end GPUs and high, maximally concurrent peer-to-peer bandwidth. Our next step will be to actually realize such a system.

We also find that multi-core CPU farms are not likely to beat GPU farms; ASGD does not fundamentally change that.

Overall, we find that dramatic speed-ups from applying model and/or data parallelism to standard SGD are not to be expected for speech-scale DNNs. Any significant speed-up will have to come from a more fundamental change of the training algorithm that allows for greater parallelizability by its nature.

7. ACKNOWLEDGEMENTS

We'd like to thank John Langford for helpful feedback.

8. REFERENCES

[1] D. Yu, L. Deng, and G. Dahl, "Roles of Pretraining and Fine-Tuning in Context-Dependent DNN-HMMs for Real-World Speech Recognition," NIPS Workshop on Deep Learning and Unsupervised Feature Learning, Dec. 2010.

[2] F. Seide, G. Li, and D. Yu, "Conversational Speech Transcription Using Context-Dependent Deep Neural Networks," Interspeech, 2011.

[3] 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," ICML, 2012.

[4] A. Coates, B. Huval, T. Wang, D.-J. Wu, and A.-Y. Ng, "Deep Learning with COTS HPC Systems," ICML, 2013.

[5] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M.-A. Ranzato, A. Senior, P. Tucker, K. Yang, A. Y. Ng, "Large Scale Distributed Deep Networks," NIPS, 2012.

[6] S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu, "Asynchronous Stochastic Gradient Descent for DNN Training," ICASSP, 2013.

[7] H. Franco *et al.*, "Context-Dependent Connectionist Probabilty Estimatation in a Hybrid Hidden Markov Model–Neural Net Speech Recognition System," Computer Speech and Language, vol. 8, pp. 211–222, 1994.

[8] P. Zhou, C. Liu, Q. Liu, L. Dai, and H. Jiang, "A Cluster-Based Multiple Deep Neural Networks Method for Large Vocabulary Continuous Speech Recognition," ICASSP, 2013.

[9] T.-N. Sainath, B. Kingsbury, H. Soltau, and B. Ramabhadran, "Optimization Techniques to Improve Training Speed of Deep Neural Networks for Large Speech Tasks," IEEE Trans. on Audio, Speech, and Language Processing, Vol. 21, No. 11, Nov. 2013.

[10] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers," in Foundations and Trends in Machine Learning, Vol. 3, No. 1 (2010) 1–122.

[11] X. Chen, A. Eversole, G. Li, D. Yu, and F. Seide, "Pipelined Back-Propagation for Context-Dependent Deep Neural Networks," Interspeech, 2012.

[12] F. Rosenblatt, "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms", Spartan Books, Wash. DC, 1961.

[13] G. Dahl, D. Yu, L. Deng, and A. Acero, "Context-Dependent Pre-Trained Deep Neural Networks for Large Vocabulary Speech Recognition," IEEE Trans. Speech and Audio Proc., Special Issue on Deep Learning for Speech and Language Processing, 2011.

[14] G. Hinton, S. Osindero, and Y. Teh, "A Fast Learning Algorithm for Deep Belief Nets", Neural Computation, vol. 18, pp. 1527–1554, 2006.

[15] F. Seide, G. Li, X. Chen, and D. Yu, "Feature Engineering in Context-Dependent Deep Neural Networks for Conversational Speech Transcription," Proc. ASRU, Waikoloa Village, 2011.

[16] B. Kingsbury, "Lattice-based optimization of sequence classification criteria for neural-network acoustic modeling," ICASSP, 2009.

[17] D. Rumelhart, G. Hinton, and R. Williams, "Learning Representations By Back-Propagating Errors," Nature, vol. 323, Oct. 1986.

[18] J. Martens, "Deep learning via Hessian-free optimization," ICML, 2010.

[19] B. Kingsbury, T. Sainath, and H. Soltau, "Scalable Minimum Bayes Risk Training of Deep Neural Network Acoustic Models Using Distributed Hessian-free Optimization," Interspeech, 2012.

[20] J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," http://www.cs.berkeley.edu/∼jduchi/projects/DuchiHaSi10.pdf, 2010.

[21] A. Senior, G. Heigold, M.-A. Ranzato, K. Yang, "An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition," ICASSP, 2013.

[22] J. Godfrey and E. Holliman, "Switchboard-1 Release 2," Linguistic Data Consortium, Philadelphia, 1997.

[23] W. Minjie *et al.*, "Minerva: A Scalable and Highly Efficient Training Platform for Deep Learning," under submission, 2013.