Parallel Training of Deep Stacking Networks

Li Deng[1], *Brian Hutchinson*[2], *and Dong Yu*[1]

[1]Microsoft Research, Redmond, WA, USA
[2]University of Washington, Seattle, WA, USA

{deng,dongyu}@microsoft.com; brianhutchinson@ee.washington.edu

Abstract

The Deep Stacking Network (DSN) is a special type of deep architecture developed to enable and benefit from parallel learning of its model parameters on large CPU clusters. As a prospective key component of future speech recognizers, the architectural design of the DSN and its parallel training endow the DSN with scalability over a vast amount of training data. In this paper, we present our first parallel implementation of the DSN training algorithm. Particularly, we show the tradeoff between the time/memory saving via training parallelism and the associated cost arising from inter-CPU communication. Further, in phone classification experiments, we demonstrate a significantly lowered error rate using parallel full-batch training distributed over a CPU cluster, compared with sequential mini-batch training implemented in a single CPU machine under otherwise identical experimental conditions and as exploited prior to the work reported in this paper.

Index Terms: parallel and distributed computing, deep stacking networks, full-batch training, phone classification

1. Introduction

Since the birth of deep learning around 2006 [10][2][14], deep models with various types have recently been developed and successfully evaluated for a number of speech processing applications [3][4][7][11][15]. Among these models, the Deep Stacking Network (DSN), presented recently in [5][6], is particularly attractive due to the potential of using parallel computing for learning its weights. Other popular deep models, e.g., deep belief nets and deep neural nets [10][4], have not benefitted from parallel training capability and scalability, making it difficult for large scale applications.

The earlier papers introduced the DSN, outlined the algorithms for learning its weight parameters emphasizing its parallel learning capability, and presented experimental evidence for its effectiveness in speech and image classification tasks [5][6][17]. In this paper, we build upon and extend the earlier work by describing our new parallel implementation of the DSN learning algorithm. In particular, we show how the gradient is computed in CPU clusters and what the tradeoff looks like between the gain derived from the parallelism and the cost due to inter-CPU communication.

Unlike most other deep models, the implementation of learning algorithms for the DSN does not require GPU units even for large datasets. The importance of our parallel implementation lies in the scalability to vast amounts of training data. This capability is a key for the potential use of the DSN as a component of future speech recognizers based on deep models.

Parallel implementation for DSN learning makes it possible to use a full-batch, gradient descent training procedure. The classification results presented in earlier papers [5][6] were limited to mini-batch training due to the memory constraint within a single CPU. The new classification results shown in this paper further confirm the effectiveness of the DSN and its learning algorithm. Specifically, we demonstrate that the full-batch training, enabled by parallel learning distributed over CPU clusters, gives substantially fewer classification errors than the corresponding mini-batch training under otherwise identical experimental conditions.

2. Deep Stacking Networks

In this and the next sections, we provide an overview of the DSN architecture and its learning which are relevant to the implementation of parallel training as the main theme of this paper. The philosophy of DSN's architecture design rests in the concept of stacking, as proposed originally in [16], where simple blocks of functions or classifiers are composed first and then they are "stacked" on top of each other so as to learn more complex functions or classifiers. Following this philosophy, [5] presented the basic form of the DSN architecture that consists of many stacking blocks, each of which takes a simplified form of shallow multilayer perceptron using convex optimization for learning the perceptron weights.

Fig. 1 gives an example of a three-block DSN, each consisting of three layers and each illustrated with a separate color. Two common and equivalent ways of showing the DSN architecture are depicted here, where "stacking" is accomplished by concatenating the immediately previous block's output prediction vector with the original input vector to form the new "input" vector in the new block. (Other types of stacking are possible, e.g. [3], and will not be covered in this paper.) The hidden layers in all blocks have the same sigmoid nonlinearity. Prediction and input layers in all blocks are linear. The DSN weight parameters $\mathbf{W}$ and $\mathbf{U}$ in each block are learned efficiently from training data, which we will describe in the next and the following sections from the algorithmic and implementational perspectives, respectively.

3. Training in Deep Stacking Networks

The linearity of the output layer in each block of the DSN is a key design decision. Such linearity enables highly efficient, parallelizable, and closed-form estimation for the output network weight matrices $\mathbf{U}$ given the hidden units' activities, denoted by $\mathbf{H}$. Due to the closed-form constraints between the input weight matrix $\mathbf{W}$ and the output weight matrix $\mathbf{U}$, the former can also be elegantly estimated in an efficient, parallelizable, batch-mode manner.

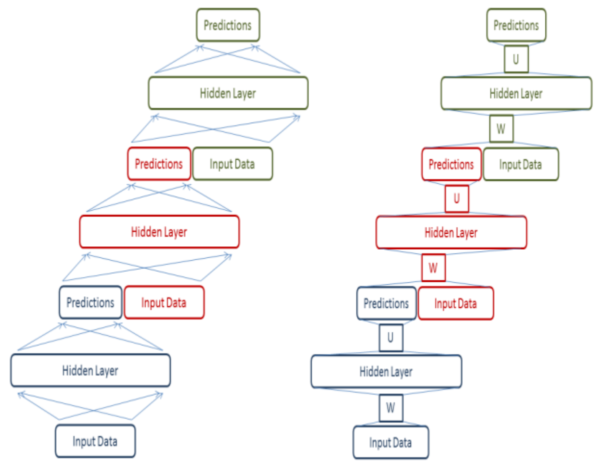

Fig. 1: *Two common and equivalent ways of illustrating a typical DSN architecture used in this and recent studies. Hidden layers in all blocks are sigmoid nonlinear. Prediction and input layers are linear. Three blocks are shown here, each with a distinct color.*

Denote the training vectors by $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_i, \cdots, \mathbf{x}_N]$, in which each vector is denoted by $\mathbf{x}_i = \left[x_{1i}, \cdots, x_{ji}, \cdots, x_{Di}\right]^{\mathrm{T}}$ where D is the dimension of the input vector, which is a function of the block, and N is the total number of training samples. Denote by L the number of hidden units and by C the dimension of the output vector. Then, the output of a DSN block is $\mathbf{y}_i = \mathbf{U}^{\mathrm{T}}\mathbf{h}_i$, where $\mathbf{h}_i = \sigma(\mathbf{W}^{\mathrm{T}}\mathbf{x}_i)$ is the hidden-layer output vector, $\mathbf{U}$ is an $L \times C$ weight matrix at the upper layer of a block. $\mathbf{W}$ is a $D \times L$ weight matrix at the lower layer of a block, and $\sigma(\cdot)$ is a sigmoid function. Bias terms are implicitly represented in the above formulation if $\mathbf{x}_i$ and $\mathbf{h}_i$ are augmented with ones, which we adopt in this work.

Given target vectors in the full training set with a total of N samples, $\mathbf{T} = [\mathbf{t}_1, \cdots, \mathbf{t}_i, \cdots, \mathbf{t}_N]$, where each vector is $\mathbf{t}_i = \left[t_{1i}, \cdots, t_{ji}, \cdots, t_{Ci}\right]^{\mathrm{T}}$, the parameters $\mathbf{U}$ and $\mathbf{W}$ are learned so as to minimize the average of the total square error below:

$$\mathrm{Error} = \|\mathbf{Y} - \mathbf{T}\|_{\mathrm{F}}^2 = \mathrm{Tr}[(\mathbf{Y} - \mathbf{T})(\mathbf{Y} - \mathbf{T})^{\mathrm{T}}], \tag{1}$$

where $\mathbf{Y} = [\mathbf{y}_1, \cdots, \mathbf{y}_i, \cdots, \mathbf{y}_N]$ and subscript F denotes Frobenius norm. Note that once the lower layer weights $\mathbf{W}$ are fixed (e.g., by random numbers or by a fixed restricted Boltzmann machine's weights [5][6]), the hidden layer values $\mathbf{H} = [\mathbf{h}_1, \cdots, \mathbf{h}_i, \cdots, \mathbf{h}_N]$ are also determined uniquely. Consequently, the upper layer weights $\mathbf{U}$ can be determined by setting the gradient

$$\frac{\partial \mathrm{Error}}{\partial \mathbf{U}} = \frac{\partial \mathrm{Tr}\left[\left(\mathbf{U}^{\mathrm{T}}\mathbf{H} - \mathbf{T}\right)\left(\mathbf{U}^{\mathrm{T}}\mathbf{H} - \mathbf{T}\right)^{\mathrm{T}}\right]}{\partial \mathbf{U}} = 2\mathbf{H}\left(\mathbf{U}^{\mathrm{T}}\mathbf{H} - \mathbf{T}\right)^{\mathrm{T}} \tag{2}$$

to zero, leading to the closed-form solution

$$\mathbf{U} = (\mathbf{H}\mathbf{H}^{\mathrm{T}})^{-1}\mathbf{H}\mathbf{T}^{\mathrm{T}}. \tag{3}$$

When L2 regularization is applied in learning $\mathbf{U}$, it can be easily shown that the learning formula becomes

$$\mathbf{U} = (\rho\mathbf{I} + \mathbf{H}\mathbf{H}^{\mathrm{T}})^{-1}\mathbf{H}\mathbf{T}^{\mathrm{T}}. \tag{4}$$

The weight matrix $\mathbf{W}$ in each module of the DSN can be further learned using batch-mode gradient descent [17]. The computation of the error gradient makes use of Eq. (4) and proceeds by

$$\begin{aligned}
\frac{\partial \mathrm{Error}}{\partial \mathbf{W}} &= \frac{\partial\, \mathrm{Tr}\left[\left(\mathbf{U}^{\mathrm{T}}\mathbf{H}-\mathbf{T}\right)\left(\mathbf{U}^{\mathrm{T}}\mathbf{H}-\mathbf{T}\right)^{\mathrm{T}}\right]}{\partial \mathbf{W}} \\
&= \frac{\partial\, \mathrm{Tr}[([(\rho\mathbf{I} + \mathbf{H}\mathbf{H}^{\mathrm{T}})^{-1}\mathbf{H}\mathbf{T}^{\mathrm{T}}]^{\mathrm{T}}\mathbf{H} - \mathbf{T})([(\rho\mathbf{I} + \mathbf{H}\mathbf{H}^{\mathrm{T}})^{-1}\mathbf{H}\mathbf{T}^{\mathrm{T}}]^{\mathrm{T}}\mathbf{H} - \mathbf{T})}{\partial \mathbf{W}} \\
&\xrightarrow{\rho \to 0} \frac{\partial\, \mathrm{Tr}[\mathbf{T}\mathbf{T}^{\mathrm{T}} - \mathbf{T}\mathbf{H}^{\mathrm{T}}(\mathbf{H}\mathbf{H}^{\mathrm{T}})^{-1}\mathbf{H}\mathbf{T}^{\mathrm{T}}]}{\partial \mathbf{W}} \\
&= -\frac{\partial\, \mathrm{Tr}[(\mathbf{H}\mathbf{H}^{\mathrm{T}})^{-1}\mathbf{H}\mathbf{T}^{\mathrm{T}}\mathbf{T}\mathbf{H}^{\mathrm{T}}]}{\partial \mathbf{W}} \\
&= -\frac{\partial\, \mathrm{Tr}[(\boldsymbol{\sigma}(\mathbf{W}^{\mathrm{T}}\mathbf{X})[\boldsymbol{\sigma}(\mathbf{W}^{\mathrm{T}}\mathbf{X})]^{\mathrm{T}})^{-1}\boldsymbol{\sigma}(\mathbf{W}^{\mathrm{T}}\mathbf{X})\mathbf{T}^{\mathrm{T}}\mathbf{T}[\boldsymbol{\sigma}(\mathbf{W}^{\mathrm{T}}\mathbf{X})]^{\mathrm{T}}]}{\partial \mathbf{W}} \\
&= \mathbf{2X}\left[\mathbf{H}^{\mathrm{T}} \circ (\mathbf{1} - \mathbf{H})^{\mathrm{T}} \circ [\mathbf{H}^{\dagger}(\mathbf{H}\mathbf{T}^{\mathrm{T}})(\mathbf{T}\mathbf{H}^{\dagger}) - \mathbf{T}^{\mathrm{T}}(\mathbf{T}\mathbf{H}^{\dagger})]\right]
\end{aligned} \tag{5}$$

where $\mathbf{H}^{\dagger} = \mathbf{H}^{\mathrm{T}}(\mathbf{H}\mathbf{H}^{\mathrm{T}})^{-1}$ is pseudo-inverse of $\mathbf{H}$ and symbol $\circ$ denotes component-wise multiplication. Note the approximation used in deriving the gradient above, where we assume that the regularization parameter ρ is sufficiently small. Otherwise, the gradient would not have the simple, closed form of (5) due to lack of cancellation of a number of terms, and parallel computation of the gradient would become more complicated. The way to initialize $\mathbf{W}$ in gradient descent as required for stacking each block of DSN is not trivial; four methods were discussed in [6].

4. Parallel Computing for Weight Training

In this section, we describe our parallel implementation of the DSN weight learning algorithm. Specifically, we describe how the gradient of Eq. (5), denoted by $\mathbf{G}$ in Fig. 2, involving all training data is computed over a CPU cluster.

In Fig. 2 we show data flow and the computation procedure in evaluating the gradient in Eq. (5). Each of the matrices in green is computed in parallel and its result is stored to disk. The parallelization is accomplished by parallelizing the matrix multiplies, having split each matrix in the product along the inner (shared) dimension (of size *N*). The jobs in orange color are accumulators: they simply sum the individual green matrices. Because the summing can begin before all of the individual matrices have been computed, the accumlators introduce minimal delay into the parallel computing pipeline. The job in red color requires synchronizing over the parallel batches distributed over the CPU cluster; i.e. it must wait for all of its dependencies. The quantities in blue color are recomputed as needed instead of being distributed, as this reduces network and disk load and optimizes overall speed.

While a main motivation for parallel implementation of the DSN learning is to scale beyond the memory limits of a single machine, we also examine the effect of parallelization on overall computation speed. There is cost associated with parallel computation arising from the inter-process communication. Because our implementation uses network disk to store and load cached variables, this cost is non-trivial. In Fig. 3, we show the measured wall-clock run-time, over three repeated single instances at different times of a day, as a function of the total number of distributed processors. The task is to compute the gradient and to evaluate the training objective on the training data consisting of a total of 1.12 million training samples. As can be observed, the lowest total computation time is achieved when distributed over between four to ten machines. This gives an average speed-up of approximately three times over the single processor case. When more than ten parallel processors are used, we observe a rise in the total computation time, as a result of the additional disk access costs and of the inter-process communication.

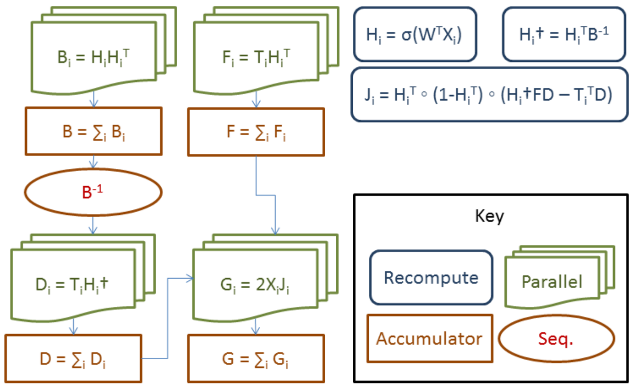

Fig. 2: *Parallel computation for the gradient,* **G**, *of Eq. (5) in implementing the DSN learning algorithm. Specifically, four quantities in green are computed in parallel over a CPU cluster.*

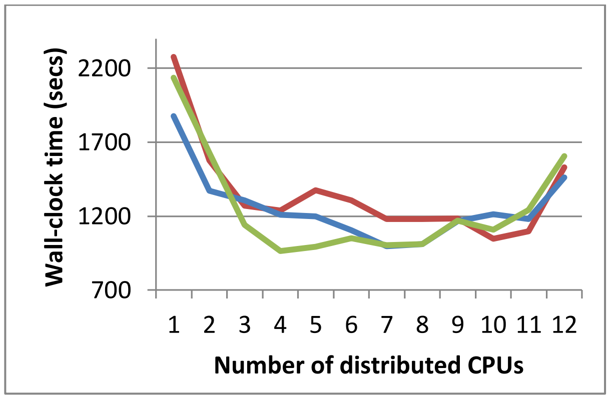

Fig. 3: *Run time of learning DSN as a function of the total number of distributed processors. Data from three independent runs at different times of a day are shown.*

5. State and Phone Classification Experiments

Prior to the work reported in this paper, gradient-descent learning of the DSN's weights was carried out in a mini-batch mode in a single processor [5][6]. This was mainly due to the memory limitation of the processor used in the experiments. Parallel implementation of the DSN's weight learning enables distributing the training data as well as computation over multiple processors and thus supports the training in a full-batch mode, whose results are reported here.

5.1 TIMIT experiments

Detailed experimental setup and procedure for the TIMIT frame-level state and segment-level phone classification tasks have been described in [5][6] and will not be repeated here. Here we focus on our new experimental results. Table 1 presents the frame-level state classification and segment-level phone classification error rates for a wide sweep of the mini-batch sizes (i.e. the number of training samples) used in each weight update of the gradient descent algorithm for learning weight matrix **W** according to Eq. (5). The results with full-batch fine-tuning are shown at the bottom row, and those with no fine-tuning shown at the top row. In these experiments, the DSN uses 3000 hidden units in each block and has a total of five blocks. Note that after each **W** update, the weight matrix **U** is estimated always with the full-batch data according to Eq. (4) after using the new **W** to compute **H**. Also, a sufficient number of iterations in gradient descent are carried out to reach convergence as judged by the TIMIT development set behavior. L-BFGS is used to update **W** [8] so that we do not need to tune the learning rate. The use of manually tuned learning rates and a FISTA procedure [1] for training is found to give similar results (not shown here).

The frame-level state classification results in the middle column of Table 1 are obtained using the straightforward DSN, where the total number of state classes is 183. When post-processing with dynamic programming [4][13] is applied to the three states for each phone and the 183 classes are merged into 39, we obtain the segment-level phone classification results as shown in the final column of Table 1.

Table 1: *State (frame-level) and phone (segment-level) classification error rates for the TIMIT core test set as a function of the mini-batch size in the gradient-based learning algorithm for training each DSN module.*

MiniBatch Size	**State Err Rate**	**Phone Err Rate**
No fine tuning	50.50%	30.10%
1,000	49.30%	29.66%
2,000	48.55%	28.50%
5,000	47.75%	27.88%
10,000	46.25%	26.20%
20,000	45.30%	25.25%
50,000	44.99%	24.90%
100,000	44.60%	24.77%
200,000	44.29%	24.10%
250,000	43.98%	23.60%
1,124,589 (full)	42.70%	**22.20%**

It is clear from Table 1 that the error rates decrease strictly as the mini-batch size increases. Full-batch training gives a significantly lower error rate than all sizes of mini-batch training shown. Note the full-batch training is made possible via the parallel and distributed training over a CPU cluster as described in Section 4. Using a single CPU machine (with 48G memory) to implement the DSN's fine tuning, we were limited by the maximal mini-batch size of 250k training samples (each sample with the dimensionality of 429), beyond which all memory in the machine became exhausted.

5.2 WSJ0 experiments

To verify the effectiveness of the full-batch training, we also use the 5k-WSJ0 database [12] to run frame-level phone classification experiments. 5k-WSJ0 has 5000 words in the vocabulary. The training material from the SI84 set in the database contains 7077 utterances (15.3 hours of speech data) from 84 speakers. They are separated into a 6877-utterance training set and a 200-utterance cross validation set. The test set consists of the Nov92 evaluation data with 330 utterances from eight speakers [9]. For the short-time spectral representation of the speech signal we use the same MFCCs and their deltas as in the TIMIT experiments; 11 frames are grouped as a single feature vector with 429 elements and are input to the DSN classifier. With the ten millisecond frame rate, the training set has a total of 5,232,244 frames as training samples, substantially larger than TIMIT. Further, unlike the TIMIT database where phone boundaries in training, development, and test sets are provided, no phone boundaries are given in WSJ0. In this work, we generate the phone labels and their boundaries in the data from the forced alignments using a tied-state cross-word tri-phone Gaussian-mixture-HMM speech recognizer. These phone labels, with a total of 40 in size, together with their boundaries provide one-to-one mapping between each speech frame and its phone label as the target for training and evaluating the DSN.

In Table 2, we show again the effectiveness of the full-batch training enabled by parallel implementation in training the DSN, where the hidden layer contains 3000 units in each block and a total of five blocks are constructed for the DSN. The task is frame-level phone classification, without any post-processing as required for segment-level phone classification carried out for TIMIT. Use of no segment constraint creates more errors than when constraints are imposed via post-processing as done for TIMIT. But the number of classes on WSJ0 is relatively small (40 of them), phones and words in WSJ0 are reasonably clearly enunciated, and there are about five times more training samples in WSJ0 than in TIMIT. These account for similar phone classification error rates between TIMIT and WSJ0.

Table 2: *Frame-level phone classification error rate for the WSJ0 test set as a function of the mini-batch size in the gradient-based learning algorithm for training each DSN module.*

Mini-Batch Size	Phone Err Rate
10,000	29.95%
50,000	27.77%
100,000	26.10%
250,000	24.50%
5,232,244 (full batch)	**20.99%**

6. Discussions and Conclusion

In this paper, we report our first parallel implementation of the DSN learning algorithm. We explore the tradeoff between the multi-processor speed-up and inter-CPU communication cost by examining the run time required to complete a fixed DSN learning task as a function of the number of the distributed processors. The parallel nature of DSN learning presented in this paper is analogous to that in the batch-based EM learning algorithm prevailing in the current HMM speech recognition systems. This virtue is conspicuously missing in the recent deep neural network architectures [4].

In phone classification experiments using both TIMIT and WSJ0 databases, we demonstrate a significantly lowered error rate achieved by DSN with full-batch training, which would be impossible without parallel training, than with the corresponding mini-batch training carried out in earlier work [5][6]. This result forms a stark contrast to fine-tuning deep neural networks by back-propagation [4][10], where the error rate was found to saturate quickly (i.e., to stop decreasing) as the mini-batch size increases. This may account for why stochastic or mini-batch gradient descent has been popular for learning the deep neural networks; i.e., full-batch training would not lower the error rate but instead waste computing time using the same number of network weight updates.

The availability of parallel training and the effectiveness of batch-mode learning verified in this work have opened the door for a wide range of DSN applications to large-scale speech and related information processing in GPU-free computation environments. We are currently pursing applications of DSN in speech recognition, speech understanding, and information retrieval while refining and improving its architecture.

7. References

[1] A. Beck and M. Teboulle. "Gradient-based methods with application to signal recovery problems," In D. Palomar and Y. Eldar, editors, Convex Optimization in Signal Processing and Communications. Cambridge, University Press, 2010.

[2] Y. Bengio. "Learning deep architectures for AI," Foundations and Trends in Machine Learning, vol. 2, no. 1, pp. 1–127, 2009.

[3] B. Hutchinson, L. Deng, and D. Yu, "A deep architecture with bilinear modeling of hidden representations: Applications to phonetic recognition," Proc. ICASSP 2012.

[4] G.E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pretrained deep neural networks for large vocabulary speech recognition," IEEE Trans. Audio, Speech, and Lang. Proc. Jan. 2012.

[5] L. Deng and D. Yu. "Deep Convex Network: A scalable architecture for deep learning," In Proc. Interspeech, 2011.

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

[7] L. Deng, M. Seltzer, D. Yu, A. Acero, A. Mohamed, and G. Hinton. "Binary coding of speech spectrograms using a deep auto-encoder," In Proc. Interspeech, September 2010.

[8] D. Liu and J. Nocedal. "On the limited memory (BFGS) method for large scale optimization," J. Mathematical Programming, Vol. 45, No.3, 503-528, 1989.

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

[10] G. Hinton and R. Salakhutdinov. "Reducing the dimensionality of data with neural networks," Science, vol. 313, 504–507, 2006.

[11] A. Mohamed, G. E. Dahl, and G. E. Hinton, "Acoustic modeling using deep belief networks," IEEE Trans. on Audio, Speech, and Lang. Proc. Jan. 2012.

[12] D. B. Paul and J. M. Baker, "The design for the Wall Street Journal based CSR corpus," in Proc. Int. Conf. Spoken Lang. Processing, 1992.

[13] S. Renals, N. Morgan, H. Bourlard, M. Cohen, and H. Franco. "Connectionist Probability Estimators in HMM Speech Recognition," IEEE Trans. Speech and Audio Proc., January 1994.

[14] R. Salakhutdinov and G. Hinton. Deep Boltzmann machines. In Proc. AISTATS, 2009.

[15] G. Tur, L. Deng, D. Hakkani-Tür, and X. He. "Towards deep understanding: Deep convex networks for semantic utterance classification," In Proc. ICASSP, 2012.

[16] D. Wolpert. "Stacked generalization," In Neural Networks, vol. 5(2), pp 241-259, 1992.

[17] D. Yu, and L. Deng. "Accelerated parallelizable neural networks learning algorithms for speech recognition," In Proc. Interspeech, 2011.