IMPROVING DEEP NEURAL NETWORK ACOUSTIC MODELS USING GENERALIZED MAXOUT NETWORKS

Xiaohui Zhang, Jan Trmal, Daniel Povey, Sanjeev Khudanpur

Center for Language and Speech Processing & Human Language Technology Center of Excellence
The Johns Hopkins University, Baltimore, MD 21218, USA
{xiaohui,khudanpur@jhu.edu}, {dpovey,jtrmal}@gmail.com

ABSTRACT

Recently, maxout networks have brought significant improvements to various speech recognition and computer vision tasks. In this paper we introduce two new types of generalized maxout units, which we call p-norm and soft-maxout. We investigate their performance in Large Vocabulary Continuous Speech Recognition (LVCSR) tasks in various languages with 10 hours and 60 hours of data, and find that the p-norm generalization of maxout consistently performs well. Because, in our training setup, we sometimes see instability during training when training unbounded-output nonlinearities such as these, we also present a method to control that instability. This is the "normalization layer", which is a nonlinearity that scales down all dimensions of its input in order to stop the average squared output from exceeding one. The performance of our proposed nonlinearities are compared with maxout, rectified linear units (ReLU), tanh units, and also with a discriminatively trained SGMM/HMM system, and our p-norm units with p equal to 2 are found to perform best.

Index Terms— Maxout Networks, Acoustic Modeling, Deep Learning, Speech Recognition

1. INTRODUCTION

Following the recent success of pre-trained deep neural networks based on sigmoidal units [1, 2, 3] and the popularity of "deep learning", a number of different nonlinearities (activation functions) have been proposed for neural network training. A nonlinearity that has recently become popular is the Rectified Linear Unit (ReLU) [4], which is a simple activation function $y = \max(0, x)$. Significant performance gain is reported in [4] and [5], where ReLU networks, without pre-training, outperform the standard sigmoidal networks. More recently, the maxout nonlinearity [6], which can be regarded as a generalization of ReLU, was proposed. This is a function $y = \max_i x_i$ that takes the maximum over groups of inputs which are arranged in groups of, say, 3. Maxout networks, combined with "dropout" [7], have given state-of-the-art performance in various computer vision tasks [6], and have also achieved improvements in speech recognition tasks [8, 9].

In this this paper, we present two "dimension-reducing" nonlinearities that are inspired by maxout. One is a "soft-maxout", formed by replacing the $\max$ function with the function $y = \log(\sum_i exp(x_i))$. The other one is p-norm, which is the nonlinearity $y = ||\mathbf{x}||_p = (\sum_i |x_i|^p)^{1/p}$, where the vector $\mathbf{x}$ represents a small group of inputs (say, 5). Note: if all the x_i were known to be positive, the original maxout would be equivalent to the p-norm with $p = \infty$. It should also be noted that the p-norm pooling strategy has been used for learning image features [10, 11, 12], and a recent work [13] has proposed a learned-norm pooling strategy for deep feedforward and recurrent neural networks.

In Section 2 we describe our baseline DNN training recipe. In Section 3, we describe our proposed nonlinearities. In Section 4 we describe our experimental setup; and in Sections 5 and 6 we give experiments and conclusions.

2. OUR DNN RECIPE

In this section we explain key features of our baseline DNN training recipe. This recipe is part of the Kaldi ASR toolkit [14]. In order to avoid confusion we should explain that Kaldi currently contains two parallel implementations for DNN training. Both of these recipes are deep neural networks where the last (output) layer is a softmax layer whose output dimension equals the number of context-dependent states in the system (typically several thousand). The neural net is trained to predict the posterior probability of each context-dependent state. During decoding the output probabilities are divided by the prior probability of each state to form a "pseudo-likelihood" that is used in place of the state emission probabilities in the HMM.

2.1. First Kaldi DNN implementation

The first implementation [1] is as described in [15]. This implementation supports Restricted Boltzmann Machines (RBM) pre-training [1, 2, 3], stochastic gradient descent training using NVidia graphics processing units (GPUs), and discriminative training such as boosted MMI [16] and state-level minimum Bayes risk (sMBR) [17, 18].

2.2. Second Kaldi DNN implementation

The work done in this paper was done using the second implementation of DNNs in Kaldi [2]. This recipe was originally written to support parallel training on multiple CPUs, although it has now been extended to support parallel GPU-based training. All our work here was performed on CPUs, in parallel. It does not support discriminative training.

Thanks to Shangsi Wang for useful discussions on the p-norm operation. The authors were supported by DARPA BOLT contract No HR0011-12-C-0015, and IARPA BABEL contract No W911NF-12-C-0015. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, IARPA, DoD/ARL or the U.S. Government.

[1] Location in code: `src/{nnet,nnetbin}`, in scripts: `local/run_dnn.sh`

[2] Location in code: src/{nnet-cpu,nnet-cpubin}/, soon to be moved to: src/{nnet2,nnet2bin}/. Location in scripts: local/run_nnet_cpu.sh, soon to be moved to: local/run_nnet2.sh

2.2.1. Greedy layer-wise supervised training

This recipe does not support Restricted Boltzmann Machine pre-training. Instead we do something similar to the greedy layer-wise supervised training [19] or the "layer-wise backpropagation" of [3]. We initialize the network randomly with one hidden layer, train it for a short time (typically less than an epoch, meaning less than one fullpass through the data), then remove the layer of weights that go to the softmax layer, add a new hidden layer and two sets of randomly initialized weights, and train again. This is repeated until we have the desired number of layers.

2.2.2. Use of multiple CPUs

Our parallelization of the neural network training has two levels: we parallelize on a single machine, and also across machines (by "machine" we mean a single physical server with shared memory and multiple CPUs). In a typical setup we run on 16 machines, each running 16 threads, so we have 256 CPUs running in total. The training time for 80 hours of data is about 48 hours.

The parallelization method on a single machine is to have multiple threads simultaneously updating the parameters while simply ignoring any synchronization issues. This is similar to the Hogwild! approach [20]. Another approach that we considered was to use a fairly large minibatch size and to use a multi-threaded implementation of BLAS. However, we tried this and found the speedup was much less than linear in the number of CPUs used.

We will now explain the parallelization method we use *across* machines. It does not matter, for this method, whether the individual machines are using a single CPU or multiple CPUs. Our method is to have multiple machines train independently using SGD, on different random subsets of the data. After processing a specified amount of data (typically 300000 samples per machine, which can take 10-20 minutes when using CPUs), each machine writes its model to disk and we average the model parameters. The averaged model parameters become the starting point for the next iteration of training. Because we average model parameters, rather than gradients as in the normal approaches [21], we only need to average the models quite infrequently. We have found that the lack of convexity of the neural network objective function is simply not an issue. However, for this method to make fast progress we must use a higher learning rate than we would use if we were training on a single machine, and this can sometimes lead to parameter divergence or saturation of the sigmoidal units. Our parallel model training method tends to give a small degradation in WER when compared with training on a single machine, but we do it anyway because it is much faster.

2.3. Methods to stabilize training

We use a number of methods to help keep the training stable and to stop the neurons from becoming "over-saturated" when using sigmoidal units. We can only summarize these here.

2.3.1. Preconditioning the SGD

Preconditioned SGD means SGD where, instead of using a fixed learning rate, we use some arbitrary matrix-valued learning rate where the matrix is symmetric positive definite. This matrix can either be fixed or can vary in some random way, although to be able to prove convergence it will generally be necessary to have upper and lower bounds on its eigenvalues that decrease in a suitable way during training. Also, this matrix should not depend on the current sample or it may bias the direction taken by the learning algorithm. There are reasons from information geometry to think that this matrix should be a multiple of the inverse of a multiple of the Fisher information matrix [22]. However, this is a very high-dimensional matrix. Previous work has used a diagonal approximation [23]; our approach is a full-matrix approximation, but factored in a special way. Essentially, for the weights of each layer, we have a matrix-valued factor corresponding to the input dimension and one corresponding to the output dimension, which scale the input values and the output derivatives respectively. For each sample we train, and for each of these factors, we derive the matrix from the inverse of the Fisher information matrix computed over the other members of the minibatch, smoothed using a multiple of the identity matrix. This can be implemented much more efficiently than one might think. We multiply the resulting matrices by a constant for each (minibatch and layer) that is computed to ensure that the total parameter-change is roughly the same as what it would have been if we were not using this method.

We find that this method improves convergence as well as improving stability when learning rates are high.

2.3.2. Maximum change per minibatch

We enforce a maximum change in the parameters per minibatch. For each layer, we scale the parameter-change to ensure that the sum over the examples in the minibatch, of the 2-norm change in parameters due to that example, is no more than a constant (e.g. 20). We formulate it in this slightly un-intuitive way in order to avoid having to store a temporary matrix.

2.4. Miscellaneous methods

In the following, note that an "iteration" of training corresponds to however long it takes for each of the (say) 16 machines to process a pre-specified number of samples– typically 300000 or so. The number of epochs is fixed in advance, typically to around 20, and this together with the total amount of data and the number of machines we are using in parallel determines the number of iterations; the resulting number of iterations can vary between about 50 and 500 depending on the amount of training data.

2.4.1. Learning rates

The initial and final learning rates in our training setup must be specified by hand, and during training we decrease them exponentially, except for a few epochs at the end (typically 5) during which we keep them constant. In [24] an exponentially decreasing schedule was also found to work best, although it is not supported by theory. We should note that whatever the specified global learning rate is, we apply half that learning rate to the last and second-to-last layers of weights. We experimented with various automatic ways to set the learning rates but found it very hard to do so robustly.

2.4.2. Parameter Shrinking and "fixing"

These methods are only applicable when using sigmoid-type units. After each iteration we do either "shrinking" or "fixing", on an alternating schedule. Shrinking means scaling the parameters of each layer by a separate factor, with the factors being determined by non-linear optimization over a subset of the training data (we found this worked better than using validation data). We call it shrinking because we expect that the resulting scaling factors will generally be less than one (although this is not always the case). "Fixing" is an operation where we check whether neurons are "over-saturated", and scales down the parameters for that neuron if so. "Over-saturated" is defined as the derivative at the nonlinearity, averaged over training-data samples, being lower than a threshold.

2.4.3. *Mixing-up*

In a method inspired by Gaussian mixture models, about halfway through training we increase the dimension of the final output layer (the softmax layer) by letting each output class's probability be a sum over potentially multiple "mixture components". The mixture components are distributed using a power rule, proportional to the class priors. During mixing up, we copy and then perturb the parameters that we had before mixing up, and then modify the bias term so that the total posterior probability assigned to each class is roughly the same as before. The average number of new mixtures we assign per state is configuration-dependent, but generally about two or three. The WER improvement from this is quite small, and not always consistent.

2.4.4. *Model combination*

After the final iteration of training, we take the models from the last n iterations and combine them via a weighted-average operation into a single model. The weights are determined via nonlinear optimization, optimizing the cross-entropy on a randomly selected subset of the training data (again, we found this to perform better than using the validation data). Each layer is weighted separately, so the number of parameters being optimized is equal to n times the number of layers. This gives a consistent improvement in WER.

3. MAXOUT NETWORKS AND OUR GENERALIZATIONS

In a maxout network, the nonlinearity is dimension-reducing. Suppose we have K maxout units (e.g. $K = 500$) with group size G (e.g. $G = 5$), then in this example the maxout nonlinearity would reduce the dimension from 2500 to 500. For each group of 5 neurons, the output would be the maximum of all the inputs:

$$y = \max_{i=1}^{G} x_i \tag{1}$$

Our generalizations are soft-maxout:

$$y = \log \sum_{i=1}^{G} \exp(x_i) \tag{2}$$

and the p-norm:

$$y = ||\mathbf{x}||_p = \left(\sum_i |x_i|^p \right)^{1/p}. \tag{3}$$

The value of p is configurable; our experiments favor p=2. Another nonlinearity we experiment with here is the ReLU nonlinearity:

$$y = \max(x, 0). \tag{4}$$

All of these nonlinearities have unbounded output. This can lead to instability in training. What we generally observe is that after the objective function increases steadily for a number of epochs, it suddenly becomes very negative. When looking into the output of the parallel training runs on a particular iteration, we would observe that often the bad performance was limited to one or several of the runs. We tried a few methods to fix this, and the one we settled on was the following.

3.1. Normalization layers

A normalization layer is nonlinearity that goes from some dimension K to K. We apply it directly after the nonlinearities discussed above, without a layer of weights in between. Let the input be x_i, with $1 \leq i \leq K$. Compute $\sigma = \sqrt{1/K \sum_i x_i^2}$ which is the uncentered standard deviation of the x_i. The nonlinearity is:

$$y_i = \begin{cases} x_i, & \sigma \leq 1 \\ x_i/\sigma, & \sigma > 1 \end{cases} \tag{5}$$

That is, it scales down the whole set of activations if necessary to prevent the standard deviation from exceeding 1. Although this is intended to stabilize training, for consistency we apply it in both training and test conditions. We use normalization layers for all the unbounded-output nonlinearities (i.e. ReLU, and maxout and its variants).

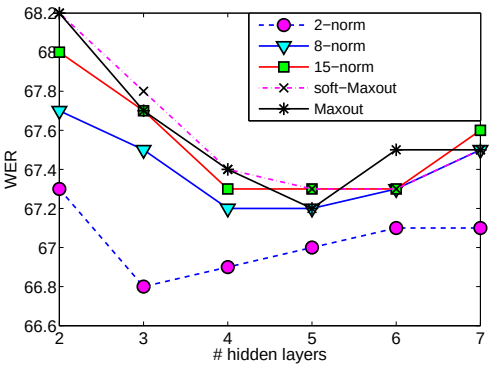

Fig. 1. Tuning #layers for maxout and variants: Bengali LimitedLP

3.2. Previous related work

We should note that soft versions of ReLU have already been investigated [25, 5]. Besides the *softplus* function used in [25], a "leaky" version of ReLU that allows for a small non-zero derivative when the unit is inactive, was tried in [5]. Neither modification resulted in a performance improvement.

4. EXPERIMENTAL SETUP

We test on a number of languages. In each language there are two evaluation conditions: *LimitedLP*, with 10 hours of training data, and *FullLP*, with 60 or 80 hours of training data. We measure based on Word Error Rate (WER), and Actual Term Weighted Value (ATWV), for which higher is better. Our baselines are our standard DNN system with tanh activations, and a SGMM [26] system trained with Boosted MMI (bMMI) [16]. Both are trained on top of features adapted with Constrained MLLR/fMLLR [27].

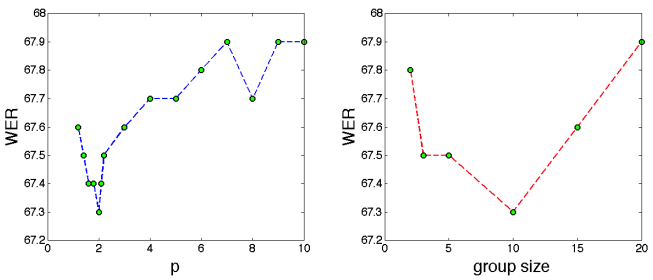

Fig. 2. Effect of p and the group size for p-norm units.

In this paper, we experiment with Babel Assamese[3], Bengali[4], Haitian Creole[5] and Zulu[6] databases, mainly on *limitedLP* (10 hours of training speech) setup. We used the Bengali *fullLP* (60 hours of training speech) setup for experiments with group size G for the Maxout networks.

The LimitedLP systems had around 2000 context-dependent states and the Bengali FullLP system had 4800. The base features used were PLP plus pitch and probability of voicing features based on SAcC [28], for all languages except Haitian Creole where we also added FFV features [29].

Note: for evaluation purposes we used the performer provided keywords releases for Assamese and Bengali. As these keyword sets weren't available at the time of writing this paper for Zulu and Haitian Creole, we generated our own development set of keywords for these two languages, using a criterion based on mutual information.

5. EXPERIMENTS

Figure 1 shows a tuning experiment where we varied the number of layers for a Bengali LimitedLP experiment. Here, we used a group size $G = 10$ and a number of groups $K = 290$ in each case (this was tuned to give about 3 million parameters in the 2-layer case). The optimal number of hidden layers seems to be lower for 2-norm (at 3 layers) than for the other nonlinearities (at around 5).

The left plot in Figure 2 shows further tuning experiments, with the same experimental setup as Figure 1, with two hidden layers. Here we vary the value of p in the p-norm, and find that $p = 2$ works best. The right plot in the figure shows the effect of varying the group size G, using the same data, two layers, p=2, and keeping the number of parameters fixed at 3 million. A group size G=10 seems to work best. Note that in [8], a group size of $G = 2$ worked best, but in that case the number of parameters was being decreased as the group size was increased.

We also tuned the initial and learning rates (experiments not shown). For all maxout variants, we used $0.016 \rightarrow 0.004$; for ReLU we used $0.004 \rightarrow 0.001$, and for tanh units we used $0.015 \rightarrow 0.002$ for LimitedLP and $0.01 \rightarrow 0.001$ for FullLP.

In Figures 3 and 4 we compare the various nonlinearity types across a number of languages, for the LimitedLP setting (10 hours of training data). All neural networks have 2 layers; unfortunately this is probably not the optimal setting. We used a group size G=10. We kept the number of parameters for all neural nets roughly fixed at around 3 million. We also compare with the baseline SGMM+bMMI system. A fairly consistent trend can be seen, with the 2-norm giving the best performance.

In Figure 5 we repeated the comparison with the FullLP Bengali data, this time using 4 layers and around 12 million parameters, where the 2-norm performed best in WER.

6. CONCLUSIONS

In this paper, we proposed two new generalized versions of Maxout nonlinearities, namely soft-Maxout and p-norm units. In experiments on a number of languages, we find that the p-norm units with p=2 perform consistently better than the activations which we used as baselines (various versions of Maxout, and tanh and ReLU). The p-norm units also seem to perform better with a smaller number of parameters and layers than the other nonlinearities.

[3]Language collection release `IARPA-babel102b-v0.4`.
[4]Language collection release `IARPA-babel103b-v0.3`.
[5]Language collection release `IARPA-babel201b-v0.2b`.
[6]Language collection release `IARPA-babel206b-v0.1d`.

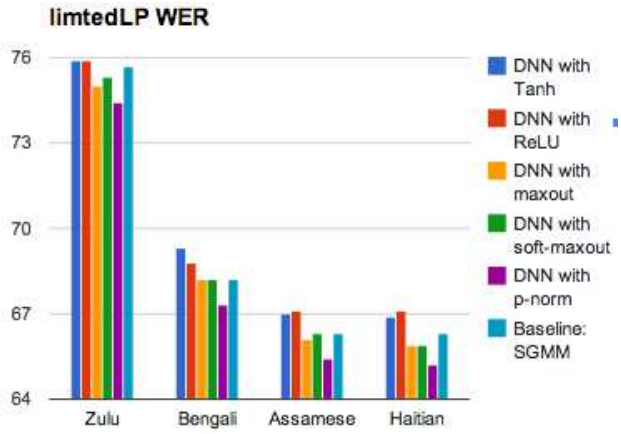

Fig. 3. Comparing nonlinearity types (%WER, LimitedLP)

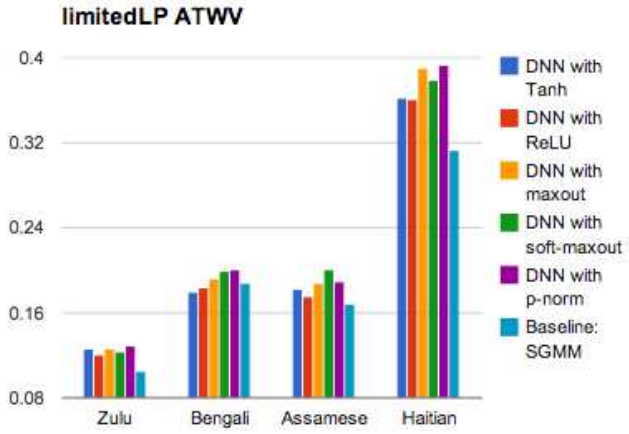

Fig. 4. Comparing nonlinearity types (ATWV, LimitedLP)

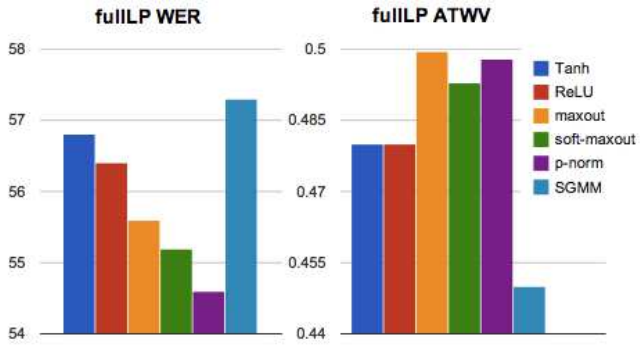

Fig. 5. Comparing nonlinearity types (Bengali FullLP)

7. REFERENCES

[1] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al., "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *Signal Processing Magazine, IEEE*, vol. 29, no. 6, pp. 82–97, 2012.

[2] George E Dahl, Dong Yu, Li Deng, and Alex Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 20, no. 1, pp. 30–42, 2012.

[3] Frank Seide, Gang Li, and Dong Yu, "Conversational speech transcription using context-dependent deep neural networks.," in *INTERSPEECH*, 2011, pp. 437–440.

[4] MD Zeiler, M Ranzato, R Monga, M Mao, K Yang, QV Le, P Nguyen, A Senior, V Vanhoucke, J Dean, et al., "On rectified linear units for speech processing," in *Proc. ICASSP*, 2013.

[5] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng, "Rectifier nonlinearities improve neural network acoustic models," in *ICML Workshop on Deep Learning for Audio, Speech, and Language Processing (WDLASL 2013)*, 2013.

[6] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio, "Maxout networks," *arXiv preprint arXiv:1302.4389*, 2013.

[7] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[8] Y. Miao, S. Rawat, and F. Metze, "Deep maxout networks for low resource speech recognition," in *Proc. ASRU*, 2013.

[9] M. Cai, Y. Shi, and J. Liu, "Deep maxout neural networks for speech recognition," in *Proc. ASRU*, 2013.

[10] Koray Kavukcuoglu, M Ranzato, Rob Fergus, and Yann LeCun, "Learning invariant features through topographic filter maps," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 1605–1612.

[11] Y-Lan Boureau, Jean Ponce, and Yann LeCun, "A theoretical analysis of feature pooling in visual recognition," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 111–118.

[12] Pierre Sermanet, Soumith Chintala, and Yann LeCun, "Convolutional neural networks applied to house numbers digit classification," in *Pattern Recognition (ICPR), 2012 21st International Conference on*. IEEE, 2012, pp. 3288–3291.

[13] Caglar Gulcehre, Kyunghyun Cho, Razvan Pascanu, and Yoshua Bengio, "Learned-norm pooling for deep neural networks," *arXiv preprint arXiv:1311.1780*, 2013.

[14] D. Povey, A. Ghoshal, et al., "The Kaldi Speech Recognition Toolkit," in *Proc. ASRU*, 2011.

[15] Karel Veselỳ, Arnab Ghoshal, Lukáš Burget, and Daniel Povey, "Sequence-discriminative training of deep neural networks," in *Interspeech*, 2013.

[16] D. Povey and D. Kanevsky and B. Kingsbury and B. Ramabhadran and G. Saon and K. Visweswariah, "Boosted MMI for Feature and Model Space Discriminative Training," in *ICASSP*, 2008.

[17] Gibson M. and Hain T., "Hypothesis Spaces For Minimum Bayes Risk Training In Large Vocabulary Speech Recognition," in *Interspeech*, 2006.

[18] Daniel Povey and Brian Kingsbury, "Evaluation of proposed modifications to MPE for large scale discriminative training," in *ICASSP*, 2007.

[19] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle, "Greedy layer-wise training of deep networks," *Advances in neural information processing systems (NIPS)*, vol. 19, pp. 153, 2007.

[20] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *arXiv preprint arXiv:1106.5730*, 2011.

[21] Jeffrey Dean and Greg S. Corrado and Rajat Monga and Kai Chen and Matthieu Devin and Quoc V. Le and Mark Z. Mao and Marc'Aurelio Ranzato and Andrew Senior and Paul Tucker and Ke Yang and Andrew Y. Ng, "Large Scale Distributed Deep Networks," in *Neural Information Processing Systems (NIPS)*, 2012.

[22] Noboru Murata and Shun-ichi Amari, "Statistical analysis of learning dynamics," *Signal Processing*, vol. 74, no. 1, pp. 3–28, 1999.

[23] Elad Hazan, Alexander Rakhlin, and Peter L Bartlett, "Adaptive online gradient descent," in *Advances in Neural Information Processing Systems (NIPS)*, 2007, pp. 65–72.

[24] Andrew Senior, Georg Heigold, Marc'Aurelio Ranzato, and Ke Yang, "An empirical study of learning rates in deep neural networks for speech recognition," in *Acoustics, Speech and Signal Processing (ICASSP)*, 2013.

[25] Xavier Glorot, Antoine Bordes, and Yoshua Bengio, "Deep sparse rectifier networks," in *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. JMLR W&CP Volume*, 2011, vol. 15, pp. 315–323.

[26] D. Povey, L. Burget, et al., "The Subspace Gaussian Mixture Model–A Structured Model for Speech Recognition," *Computer Speech & Language*, vol. 25, no. 2, pp. 404–439, April 2011.

[27] M. J. F. Gales and P. C. Woodland, "Mean and Variance Adaptation Within the MLLR Framework," *Computer Speech and Language*, vol. 10, pp. 249–264, 1996.

[28] Daniel PW Ellis and Byunk Suk Lee, "Noise robust pitch tracking by subband autocorrelation classification," in *13th Annual Conference of the International Speech Communication Association*, 2012.

[29] Kornel Laskowski, Mattias Heldner, and Jens Edlund, "The fundamental frequency variation spectrum," in *FONETIK*, 2008.