The Tensor Deep Stacking Network Toolkit

David Palzer and Brian Hutchinson
Computer Science Department
Western Washington University
{David.Palzer, Brian.Hutchinson}@wwu.edu

***Abstract*—In this paper we introduce the Tensor Deep Stacking Network (T-DSN) Toolkit, an implementation of the T-DSN deep learning architecture. The toolkit consists of a Python library and a set of accompanying helper scripts that allow you to train and evaluate T-DSN models. The toolkit is designed to be portable, modular, efficient and parallelized. Our goal for the toolkit is to promote research on this and related deep learning architectures. The T-DSN Toolkit is open source and free for non-commercial use. In this paper, we summarize the core functionality of the toolkit and discuss its design and implementation. We also present a new set of experiments on standard machine learning datasets, demonstrating the model's effectiveness.**

I. Introduction

Deep learning continues to gain prominence in machine learning, driven in large part by the impressive results it has achieved, particularly in application areas like speech recognition [1], [2] and image classification [3], [4]. In recent years, a substantial amount of work has gone into improving existing deep architectures (e.g. with linear rectified units [5], dropout [6], discriminative pre-training [7], Hessian-free second order optimization [8], etc.) and developing new ones (e.g. Deep Boltzmann Machines [9], sum-product-networks [10], etc.).

Of particular relevance to this work, in [11], Deng and Yu introduced the novel "Deep Convex Network," also known as the Deep Stacking Network (DSN). Unlike standard deep architectures [12], which center around the idea of learning a deep hidden representation of the data, propagating the hidden representations from layer to layer, the DSN is a stacked architecture that propagates the (increasingly accurate) output predictions from layer to layer. In addition to its strong empirical performance [13], [14], [15], the DSN has several desirable properties: it does not require any pre-training nor does it need end-to-end fine-tuning, and it can be naturally parallelized using CPU or GPU clusters [16].

The Tensor Deep Stacking Network [11], [17] (T-DSN) is an extension of DSN architecture. Rather than rely upon a single hidden layer in each block, it uses two parallel hidden representations in order to incorporate second order information from the non-linearly transformed input data into the model. This has the effect of moving more of the parameters within a block from the lower layer, the optimization of which is non-convex, to the upper layer, which has a convex, closed-form solution, which in turn has the effect of making each layer more powerful (for a fixed number of free parameters). It was shown that the T-DSN improves performance over the DSN in [17], [11], and a strategy for parallelizing the computation was presented in [16], [11].

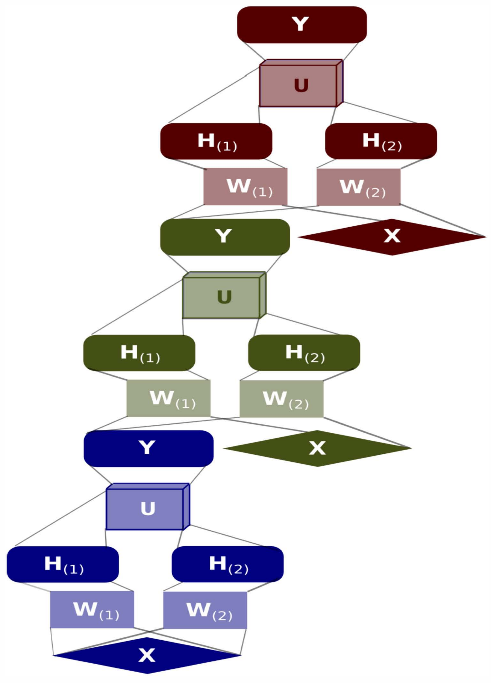

Fig. 1: An illustration of the stacked nature of the T-DSN.

In this paper we introduce the Tensor Deep Stacking Network Toolkit,[1] an open source Python implementation of the T-DSN that is freely available for non-commercial use. Our goal for developing and releasing this toolkit is to facilitate research on this and related deep architectures. We have designed it to be portable, easy-to-use and efficient. In this paper we also report several new results on standard datasets that demonstrate the T-DSN's effectiveness.

The remainder of the paper is organized as follows: Section II summarizes the T-DSN model and training algorithms, Section III describes the key features of the T-DSN Toolkit, in Section IV we report a new set of experimental results on standard machine learning datasets, and in Section V we conclude and address priorities for future extensions to the toolkit.

II. The Tensor Deep Stacking Network

As shown in Figure 1[2], a T-DSN consists of several stacked blocks, each of which is a modified neural network with a linear activation on the output. More specifically, the

[1] http://fw.cs.wwu.edu/~hutchib2/software/tdsn/
[2] A variant this figure previously appeared in [11].

input representation $x \in \mathbb{R}^D$ is mapped through two weight matrices, W_1 and W_2 and through a sigmoid non-linearity to two hidden representations, $h_1 \in \mathbb{R}^{L_1}$ and $h_2 \in \mathbb{R}^{L_2}$. These two hidden representations are mapped multilinearly through an upper layer third-order weight tensor $\mathcal{U} \in \mathbb{R}^{L_1 \times L_2 \times C}$. That is, the output prediction vector $y \in \mathbb{R}^C$ is defined by:

$$y_k = \sum_{i=1}^{L_1} \sum_{j=1}^{L_2} \mathcal{U}_{ijk} h_{1_i} h_{2_j} \quad (1)$$

Although this formulation is powerful, matrices are in practice more convenient to work with than tensors, so the following equivalent formulation can be made. Let $U \in \mathbb{R}^{L_1 L_2 \times C}$ denote the unfolding of tensor $\mathcal{U}$ [18], and let $\tilde{h}$ denote the Kronecker product of the hidden representations: $\tilde{h} = h_1 \otimes h_2$. Then y can be equivalently written as:

$$y = U^T \tilde{h}. \quad (2)$$

One defining characteristic of the DSN and the T-DSN is that the upper layer weights, U, are chosen to be the solution to the least squares problem:

$$\min_U \|U^T \tilde{H} - T\|_F^2, \quad (3)$$

Where the N columns of $\tilde{H}$ are the hidden representations $\tilde{h}$ of the N data points in the training set, and likewise the columns of $T \in \mathbb{R}^{C \times N}$ are the desired (target) output vectors. For classification tasks, each column of T is typically a one-hot (indicator) representation of the true output class. The input datapoints are themselves assembled as columns of a matrix $X \in \mathbb{R}^{D \times N}$. Given the closed form solution for U, the gradient of the objective in Eqn. 3 can be found for W_1 and W_2. Given the gradients, any first-order optimization technique can be used; by default, the T-DSN Toolkit uses scipy's l-BFGS optimizer [19].

One important practical consideration for the computation of the gradient and objective function is breaking large matrices into smaller "chunks." This serves two purposes: first, it reduces the amount of data that must be stored in memory simultaneously, and second, it makes it easy to parallelize the problem over local cores or even over nodes in a compute cluster. The T-DSN Toolkit is currently locally parallelized. Table I[3] describes how the overall computation and each of the intermediate variables can be computed in terms of the individual batches. Two types of superscript are required to indicate a given matrix in a set of matrices:

1) The angle bracketed superscript k (e.g. $\mathbf{\Psi}^{\langle k \rangle}$) denotes the k'th chunk of a matrix, split along the second dimension (of size N) into P equal sized chunks; for example,

$$\mathbf{\Psi} = [\mathbf{\Psi}^{\langle 1 \rangle}\ \mathbf{\Psi}^{\langle 2 \rangle}\ \cdots\ \mathbf{\Psi}^{\langle P \rangle}]$$

2) The square bracketed superscript k (e.g. $\mathbf{B}^{[k]}$) denotes one of a set of matrices, all with the same dimension, each computed using data from a single chunk, that will ultimately be summed together into a single matrix.

[3]A variant of this table appeared previously in [11]

Once a block is trained, and we have our prediction matrix (Y), we can prepare the next block. This involves concatenating our prediction matrix with the original feature matrix X. This concatenated data plays the role of the input data X for the training of the next block. This can be interpreted as augmenting the original features (X) with an increasingly accurate estimate of the predictions (Y).

III. Toolkit Design and Implementation

A. Design Goals

The T-DSN Toolkit is written in the Python scripting language. This language was chosen to help us achieve the following design goals:

- **Portability.** One of our aims in designing the toolkit was to have it be portable so that it can be used in a wide range of environments with minimal dependence on third party packages.
- **Modularity.** The T-DSN is heavily modularized into a hierarchical set of functions so that researchers can easily manipulate the operation of the network to modify and extend T-DSN functionality.
- **Efficiency.** At the core of our computation is the numpy module. This module uses the BLAS library[4], and our operations depend upon standard, highly optimized numerical computation software.
- **Parallelizability.** Parallelizing the T-DSN is discussed in detail in [11]. The T-DSN toolkit is locally parallelized on a CPU, though it could be extended to parallelize locally on a GPU or over nodes in a compute cluster.

B. Major Classes of Functions

1) Loading and Saving: Four functions are used for file I/O. `matload` and `matsave` are used to load and save dense matrices, while `smatload` and `smatsave` load and save sparse matrices. All of the binary matrix file formats used are described in Sec. III-E. Our loading and saving functions have been optimized for speed.

2) Computational: A set of functions compute intermediate variables during the evaluation of the gradient or objective value. Each of these is named `compute_X`, where `X` is the name of the intermediate variable. See Table I for the mapping between intermediate variable and function.

3) Parallelization: There are two functions that are responsible for setting up and spawning our parallel processes.

- `parallel_compute_and_sum` spawns processes for each chunk, and then sums the resulting matrices; used to compute each variable with a superscript $[k]$ and the summed version (e.g. $\mathbf{B}$ from each $\mathbf{B}^{[k]}$).
- `parallel_compute_and_list` used to spawn processes that compute each variable in Table I with a superscript $\langle k \rangle$ (e.g. computing each $\mathbf{H}_1^{\langle k \rangle}$).

[4]http://www.netlib.org/blas

Variable	Dimensions	Definition	Toolkit Function
$\mathbf{H}_{(i)}^{\langle k\rangle}$	$L_i \times N_k$	$\sigma(\mathbf{W}_{(i)}^T \mathbf{X}^{\langle k\rangle})$	`compute_h`
$\tilde{\mathbf{H}}_{(i)}^{\langle k\rangle}$	$L \times N_k$	$\mathbf{H}_{(1)}^{\langle k\rangle} \odot \mathbf{H}_{(2)}^{\langle k\rangle}$	`compute_hh`
$\mathbf{B}^{[k]}$	$L \times L$	$\tilde{\mathbf{H}}^{\langle k\rangle} \tilde{\mathbf{H}}^{\langle k\rangle T}$	`compute_b_aux`
$\mathbf{F}^{[k]}$	$L \times C$	$\tilde{\mathbf{H}}^{\langle k\rangle} \mathbf{T}^{\langle k\rangle T}$	`compute_f_aux`
$\mathbf{B}$	$L \times L$	$\sum_{k=1}^{P} \mathbf{B}^{[k]}$	`compute_b`
$\mathbf{F}$	$L \times C$	$\sum_{k=1}^{P} \mathbf{F}^{[k]}$	`compute_f`
$\tilde{\mathbf{H}}^{\dagger\langle k\rangle}$	$N_k \times L$	$\tilde{\mathbf{H}}^{\langle k\rangle T} \mathbf{B}^{-1}$	`compute_hht`
$\mathbf{U}$	$L \times C$	$\mathbf{B}^{-1}\mathbf{F}$	`compute_u`
$\mathbf{D}^{[k]}$	$C \times L$	$\mathbf{T}^{\langle k\rangle} \tilde{\mathbf{H}}^{\dagger\langle k\rangle}$	`compute_d_aux`
$\mathbf{D}$	$C \times L$	$\sum_{k=1}^{P} \mathbf{D}^{[k]}$	`compute_d`
$s^{[k]}$	1×1	$\Vert \mathbf{U}^T \tilde{\mathbf{H}}^{\langle k\rangle} - \mathbf{T}^{\langle k\rangle} \Vert_F^2$	`compute_s_aux`
s	1×1	$\sum_{k=1}^{P} s^{[k]}$	`compute_s`
$\tilde{\mathbf{\Theta}}^{T\langle k\rangle}$	$L \times N_k$	$2\tilde{\mathbf{H}}^{\dagger\langle k\rangle}\mathbf{F}\mathbf{D} - \mathbf{T}^{\langle k\rangle T}\mathbf{D}$	`compute_theta_t`
$\mathbf{\Psi}_{(1)}^{\langle k\rangle}$	$L_1 \times N_k$	$\Psi_{(1)in} = \langle \mathbf{E}_{(i,n)}^{L_1 \times N} \odot \mathbf{H}_{(2)}^{\langle k\rangle}, \tilde{\mathbf{\Theta}}^{T\langle k\rangle} \rangle$	`compute_psi1`
$\mathbf{\Psi}_{(2)}^{\langle k\rangle}$	$L_2 \times N_k$	$\Psi_{(2)jn} = \langle \mathbf{H}_{(1)}^{\langle k\rangle} \odot \mathbf{E}_{(j,n)}^{L_2 \times N}, \tilde{\mathbf{\Theta}}^{T\langle k\rangle} \rangle$	`compute_psi2`
$\mathbf{G}_{(i)}^{[k]}$	$D \times L_i$	$\mathbf{X}^{\langle k\rangle}(\mathbf{H}_{(i)}^{\langle k\rangle} \circ (\mathbf{1} - \mathbf{H}_{(i)}^{\langle k\rangle}) \circ \mathbf{\Psi}_{(i)}^{\langle k\rangle})$	`compute_gi_aux`
$\mathbf{G}^{[k]}$	$D \times (L_1 + L_2)$	$[\mathbf{G}_{(1)}^{[k]}\ \mathbf{G}_{(2)}^{[k]}]$	`compute_gi`
$\mathbf{G}$	$D \times (L_1 + L_2)$	$\sum_{k=1}^{P} \mathbf{G}^{[k]}$	`compute_g`

TABLE I: Details of the steps to compute the objective function and gradient, including the mathematical definition of intermediate variables and the corresponding toolkit functions to perform the computation.

We opt to spawn multiple processes, rather than multiple threads, due to a threading limitation in Python's Global Interpreter Lock. Each process must be passed the relevant intermediate input variables for its chunk (k) of the data.

C. Core Functionality

The functionality supported by the toolkit can either be imported as a module, or run using our standalone script from the command-line. In the latter case, the script is responsible for parsing the command-line arguments, setting default values, reading in the initial files, starting the appropriate T-DSN operation, and reporting any results, if requested.

1) Training: To train a T-DSN, you must provide a dense input feature file X and a sparse target file T. You may also specify the chunk size, number of blocks to stack, among other options. After the training of a block has converged, the toolkit will write the prediction matrix, Y, to disk, along with the weight matrices W_1, W_2 and U.

2) Testing: In test-mode, given a trained model, the T-DSN toolkit allows one to generate predictions for a new test set. These new predictions can be evaluated if the corresponding labels are provided in a new target matrix.

D. Dependencies

While we try to minimize the number of third party modules, we do make use of several standard Python modules for the sake of efficiency and reliability. Specifically, this toolkit uses the following Python modules: numpy, scipy [20], sys, struct, gc, time, argparse, and multiprocessing. All of the modules are distributed by default with python, except for numpy and scipy.

E. File Formats

The T-DSN toolkit uses two binary file formats for its files. A dense format is used for feature, weight and prediction matrices. The dense format for an $N \times M$ matrix consists of:

1) N as a 8-byte signed integer.
2) M as a 8-byte signed integer.
3) The $N \times M$ values of the matrix, stored in column major order, each as 8-byte double precision values.

A sparse matrix format is used for the targets. This format for an $N \times M$ matrix with K non-zero entries consists of:

1) N as a 8-byte signed integer.
2) M as a 8-byte signed integer.
3) K as a 8-byte signed integer.
4) All non-zero entries are then stored in any order as triples (i, j, x):
    a) i as a 8-byte double precision value (row index[5] for non-zero value).
    b) j as a 8-byte double precision value (column index for non-zero value).
    c) x as a 8-byte double precision value (value).

F. Hyper-Parameters

The T-DSN has a few tunable hyper-parameters; most notably, the number of hidden units in each hidden layer (L_1 and L_2) and the total number of blocks in the network. Ultimately, these hyper-parameters should be tuned empirically for your problem, but to give intuition for these we report results on several datasets over a range of values in the following section.

G. Parallelizing

Because memory, and not speed, was the original motivation for parallelizing the T-DSN, the toolkit currently only parallelizes over CPU cores. By default, the toolkit detects the number of cores and amount of memory available on the system, and then breaks the problem into sufficiently small chunks such that each core can be processing a chunk in memory at the same time. Each time a parallelizable calculation is run, each core is assigned a set of chunks to process. Inter-process

[5]Row and column indexing starts at 1 for Matlab compatibility.

Data Set	$\mathbf{N_{train}}$	$\mathbf{N_{test}}$	D	C	$\mathbf{L_i}$	Depth
Iris	136	7	4	3	4	2
Car	1,296	432	21	4	75	3
Abalone	3,113	1,044	10	3	5	7

TABLE II: Dataset statistics, including the number of training samples (N_{train}), the number of test samples (N_{test}), the input feature dimension (D) and the number of classes (C), and T-DSN hyperparameters selected by grid-search, including hidden layer size (L_i) and model depth in blocks.

communication is accomplished via disk. For simplicity, the toolkit currently supports only local parallelization on a CPU, although we plan to parallelize over CPU or GPU clusters in future releases.

IV. Experiments

In this section, we evaluate the T-DSN Toolkit on several standard machine learning datasets and report the results.

A. Datasets

Our three datasets are freely available from the UCI Machine Learning Repository.[6] Each dataset is split into disjoint train and test sets. Table II provides a summary of the dataset characteristics, including the number of samples in train and test, the dimension of the input and the number of classes.

1) Iris: Roland Fisher's famous Iris data set consists of a set of iris flowers. There are only four features (sepal length, sepal width, petal length, petal width), and each flower has one of three class labels (iris setosa, iris virginica, and iris versicolor).

2) Car: The Car Evaluation data set is a collection of vehicles, whose attributes capture various properties of a car, and whose labels are whether the car is unacceptable, acceptable, good, or very good.

3) Abalone: The Abalone data set is a collection of physical measurements of abalone that can be used to predict the age of a specimen. Each specimen has eight features; sex, length, diameter, height, whole weight, shucked weight, viscera weight, and shell weight. The three class labels are age ranges: 1-8 years old, 9-10 years old, and 11+ years old.

B. Tuning

There are two hyperparameters of the T-DSN to tune: 1) $L_i (= L_1 = L_2)$, the number of hidden nodes in each of the two parallel layers, and 2) the depth of the model, in blocks. We use five-fold cross-validation on the training set to select these hyperparameters. The specific hyperparameter values found in experiments our are presented in Table II.

C. Results

We report model test accuracy in Table III.[7] For comparison, we cite several previously reported results.

[6]https://archive.ics.uci.edu/ml/datasets.html

[7]Due to the small sample size of the Iris test set, we report an averaged accuracy, over 50 systems trained on the fixed training set. (There is variation in test set accuracy because training is non-deterministic due to random weight initialization and non-convexity.)

Method	Accuracy
T-DSN	0.986
HIDER [21]	0.967
CORE [22]	0.966
Naive-Bayes [22]	0.955
MCADT [23]	0.953
C4.5 [22]	0.937

(a) Iris

Method	Accuracy
T-DSN	0.970
TAN [24]	0.941
BAN [24]	0.940
Naive-Bayes [24]	0.866
GBN [24]	0.861

(b) Car

Method	Accuracy
HFRBCS [25]	0.702
Chi-5 [25]	0.667
T-DSN	0.663
Ishibuchi05 [25]	0.661
Chi-3 [25]	0.630
C4.5 [25]	0.156

(c) Abalone

TABLE III: Experimental results for the Iris (a), Car (b) and Abalone (c) datasets. Results within one standard deviation of T-DSN are colored in blue.

To understand the effect of our hyperparameters, we visualize the average accuracy over 10 random train-test splits in Figure 2. For brevity and clarity of the display, we show only the values for L_i and depths that are adjacent (in our grid search space) to the optimal hyperparameters selected during tuning. The figure also visualizes a ± 1 standard deviation interval, also estimated from our 10 random train-test splits, to give a sense of the variability of the results.

D. Analysis

1) Iris: Although our test set accuracy shown in Table III is higher than the baseline methods we compare against, the small test set size means that all of the baselines fall within one standard deviation of our result. Due to the simplicity of this task, we need only a few hidden nodes in each parallel layer. Our tuning selected $L_1 = L_2 = 4$, although Figure 2 shows that $L_1 = L_2 = 3$ actually slightly outperforms on average. Although it is not shown in Figure 2, using only one or two hidden nodes per parallel layer leads to heavy degradations in performance. Moderately deep (3-6) block T-DSNs seem to work best for this dataset.

2) Car: On the car evaluation dataset, the T-DSN outperforms the baseline methods, with an almost 50% relative reduction in error over the closest baseline. With a relatively larger amount of data, tuning our model finds that hidden layer sizes of $L_i = 75$ is preferable, although Figure 2 shows that there is in fact a wide range of L_i that perform comparably. The figure also suggests that our selected block depth of 3 may not be optimal, and that performance may be further improved with slightly deeper models.

3) Abalone: The T-DSN is outperformed by one of the methods reported in [24], while it performs comparably to two others and outperforms the last two. Despite the larger

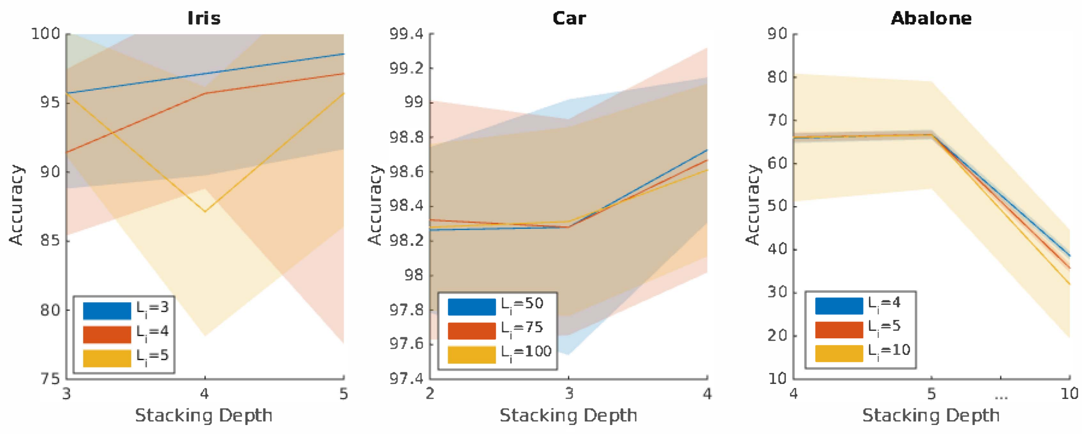

Fig. 2: Accuracy confidence intervals for each dataset in the neighborhood of the optimal tuned hyperparameters.

training set size, tuning select the small $L_i = 5$, presumably due to the small input and output dimensions. Tuning favored a deeper model, of depth seven, although Figure 2 shows that performance degrades as the model gets too deep.

V. Conclusions and Future Work

In this paper we introduce the Tensor Deep Stacking Network Toolkit, an open-source implementation of the Tensor Deep Stacking Network [11] that is freely available for non-commercial use. The toolkit is implemented in Python and is designed to be easy to use, portable and efficient, with minimal dependencies. Our hope is that the availability of this toolkit will accelerate research on this and related architectures. We also present several new results and analyses on standard machine learning datasets, demonstrating the effectiveness of the model. Cumulatively over the three reported datasets, our model outperforms six of our baselines, is beaten by one, and performs comparably to seven. There are many ways the toolkit could be extended, and development is on-going. Priorities for future extensions include parallelization over a compute cluster and support for GPU parallelization.

References

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

[2] G. H. Li Deng and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: An overview," in *Proc. ICASSP*, 2013.

[3] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *Proc. Joint Conference on Artificial Intelligence*, 2011, pp. 1237–1242.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Neural Information Processing Systems*, 2012, pp. 1097–1105.

[5] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proc. ICML*, 2010.

[6] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *The Computing Research Repository*, 2012.

[7] X. C. Frank Seide, Gang Li and D. Yu, "Feature engineering in context-dependent deep neural networks for conversational speech transcription," in *Proc. ASRU*, 2011.

[8] J. Martens, "Deep learning via hessian-free optimization," in *Proc. ICML*, 2010.

[9] R. R. Salakhutdinov and G. E. Hinton, "Deep boltzmann machines," in *Proc. AISTATS*, 2009.

[10] H. Poon and P. Domingos, "Sum-product networks: A new deep architecture," in *Proc. UAI*, 2011.

[11] B. Hutchinson, L. Deng, and D. Yu, "Tensor deep stacking network," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, pp. 1944–1957, 2013.

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

[13] L. Deng and D. Yu, "Deep convex network: A scalable architecture for speech pattern classification," in *Proc. Interspeech*, 2011.

[14] D. Y. Li Deng and J. Platt, "Scalable stacking and learning for building deep architectures," in *Proc. ICASSP*, 2012.

[15] D. H.-T. Gokhan Tur, Li Deng and X. He, "Towards deeper understanding deep convex networks for semantic utterance classification," in *Proc. ICASSP*, 2012.

[16] L. Deng, B. Hutchinson, and D. Yu, "Parallel training of deep stacking networks," in *Interspeech*, 2012.

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

[18] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, September 2009.

[19] Y. Nesterov, *Introductory Lectures on Convex Optimization: A Basic Course*. Kluwer Academic Publishers, 2004.

[20] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/

[21] J. S. Aguilar-Ruiz, R. Giráldez, and J. C. Riquelme, "Natural encoding for evolutionary supervised learning," *IEEE Transactions on Evolutionary Computation*, 2007.

[22] K. C. Tan, Q. Yu, and J. H. Ang, "A coevolutionary algorithm for rules discovery in data mining," *International Journal of Systems Science*, 2006.

[23] G. Holmes, B. Pfahringer, R. Kirkby, E. Frank, and M. Hall, "Multiclass alternating decision trees," in *Proc. ECML*, 2002, pp. 161–172.

[24] J. Cheng and R. Greiner, "Comparing bayesian network classifiers," in *Proc. UAI*, 1999, p. 101.

[25] A. Fernández, M. J. del Jesus, and F. Herrera, "Hierarchical fuzzy rule based classification systems with genetic rule selection for imbalanced data sets," *International Journal of Approximate Reasoning*, 2009.