Tensor Deep Stacking Networks

Brian Hutchinson, Student Member, IEEE, Li Deng, Fellow, IEEE, and Dong Yu, Senior Member, IEEE

Abstract—A novel deep architecture, the tensor deep stacking network (T-DSN), is presented. The T-DSN consists of multiple, stacked blocks, where each block contains a bilinear mapping from two hidden layers to the output layer, using a weight tensor to incorporate higher order statistics of the hidden binary \((0,1)\) features. A learning algorithm for the T-DSN’s weight matrices and tensors is developed and described in which the main parameter estimation burden is shifted to a convex subproblem with a closed-form solution. Using an efficient and scalable parallel implementation for CPU clusters, we train sets of T-DSNs in three popular tasks in increasing order of the data size: handwritten digit recognition using MNIST (60k), isolated state/phone classification and continuous phone recognition using TIMIT (1.1 m), and isolated phone classification using WSJ0 (5.2 m). Experimental results in all three tasks demonstrate the effectiveness of the T-DSN and the associated learning methods in a consistent manner. In particular, a sufficient depth of the T-DSN, a symmetry in the two hidden layers structure in each T-DSN block, our model parameter learning algorithm, and a softmax layer on top of T-DSN are shown to have all contributed to the low error rates observed in the experiments for all three tasks.

Index Terms—Deep learning, stacking networks, tensor, bilinear models, handwriting image classification, phone classification and recognition, MNIST, TIMIT, WSJ

1 INTRODUCTION

RECENTLY, a deep classification architecture built upon blocks of simplified neural network modules, each with a single nonlinear hidden layer and linear input and output layers was proposed, developed, and evaluated [1], [2]. It was called the deep convex network since learning the upper layer weights of each block could be formulated as solving a convex optimization problem with a closed-form solution, after having initialized the lower layer weights of each block with a fixed restricted Boltzmann machine. The network was later renamed the deep stacking network (DSN) [3], emphasizing that the mechanism in this network for establishing the deep architecture shares the same philosophy as “stacked generalization” [4]. This name also recognizes that the lower layer weights are, in practice, learned for greater effectiveness in classification tasks, so the overall weight learning problem in the DSN is no longer convex. In Section 2.1, we provide a short review of the previous DSN as the background for the current work.

The new deep architecture presented in this paper, which we call the tensor deep stacking network (T-DSN), improves and extends the earlier DSN architecture in two significant ways. First, information about higher order, covariance statistics in the data, which was not represented in DSN, is now embedded into the T-DSN via a bilinear mapping from two hidden representations to predictions using a third-order tensor. Second, while the T-DSN retains the same linear-nonlinear interleaving structure as DSN in building up the deep architecture, it shifts the major learning problem from the lower layer, nonconvex optimization component to the upper layer, convex subproblem with a closed-form solution.

Modeling covariance structure directly on raw speech or image data, rather than on the more compact binary \((0,1)\) hidden feature layers as achieved in T-DSN, was previously proposed in an architecture called mcRBM [5], [6], [7]. One key distinction is the different domains in which the higher order structure is represented: one in the visible data, as in the mcRBM, and another in the hidden units, as in our T-DSN. In addition, mcRBM can only be used in one single bottom layer in deep architectures and cannot be easily extended to deeper layers. This is due to the model and learning complexity that are incurred by the factorization required to reduce the cubic growth in the size of the weight parameters. Factorization incurs very high computational cost which, together with the high cost of hybrid Monte Carlo in learning, makes it impossible to scale up to very large datasets. These difficulties are removed in the proposed T-DSN presented in this paper. Specifically, the same interleaving nature of linear and nonlinear layers inherited from DSN makes it straightforward to stack up deeper layers, and the closed-form solution for the upper layer weights enables efficient, parallel training. Because of the relatively small sizes in the hidden layers, no factorization is needed for the T-DSN’s tensor weights. The mcRBM and T-DSN differ in other ways; in particular, the mcRBM is a generative model optimizing a maximum likelihood objective, while the T-DSN is a discriminative model optimizing a least-squares objective. The preliminary work that introduced the T-DSN and its key advantages was described previously in [8]. This paper significantly expands the work and contains comprehensive experimental results plus details of the learning algorithm and its implementation.

One major motivation for developing the recent DSN is the lack of scalability and parallelization in the learning
algorithms for the deep neural network (DNN) [9], [10], [11] which have achieved high success in large vocabulary speech recognition (e.g., [10]). In [1], [2], and [3], it was shown that all computational steps of the learning algorithm for DSN are batch-mode based, and are thus amenable to parallel implementation on a cluster of CPU (and/or GPU) nodes. The same computational advantage is retained for the T-DSN architecture introduced in this paper. We are able to parallelize all computations necessary for training and evaluation and thus scale our experiments to larger training sets using a cluster. The ability to continue to benefit from increasingly large batch sizes leads the T-DSN training to use parallelism in a very different way from that of [12], which in contrast distributes asynchronous computation of mini-batch gradients for training a deep sparse autoencoder. Unlike the DNN and other deep architectures that demand GPUs in learning, all results presented in this paper are obtained using exclusively CPU-based clusters.

The organization of this paper is as follows: In Section 2, we provide a brief review of the DSN and discuss how the T-DSN generalizes the DSN. Section 2 also presents an overview of the T-DSN architecture and its bilinear structure that uses tensor weights to make predictions from two hidden representations of the input data. In Section 3, we describe a solution, including its derivation, to the T-DSN learning problem from the algorithmic perspective. A parallel implementation of the learning algorithm for practical applications is detailed in Section 4, which for comparison also includes a computational analysis of sequential learning time complexity. Section 5 presents three sets of evaluation experiments, from a smaller scale (MNIST with 60k training samples for image classification) to a larger scale (TIMIT with 1.12m training samples for both isolated phone classification and continuous phone recognition) and to a still larger scale (WSJ with 5.23 m training samples for phone classification). We show the experimental results that consistently demonstrate the effectiveness of the T-DSN architecture and related learning methods.

2 TENSOR DEEP STACKING NETWORK: AN OVERVIEW

In this section, we first briefly review the DSN as it relates to the T-DSN, and then describe the general architecture of the T-DSN and its key properties.

2.1 Deep Stacking Networks: A Review

The DSN is a scalable deep architecture amenable to parallel weight learning [1]. It is trained in a supervised, block-wise fashion, without the need for back propagation over all blocks, as is common in other popular deep architectures [13]. The DSN blocks, each consisting of a simple, easy-to-learn module, are stacked to form the overall deep network.

Each DSN block, as developed in [1] and [2] and which also forms the basis of the T-DSN, is a simplified multilayer perceptron with a single hidden layer. It consists of an upper layer weight matrix $U$ that connects the logistic sigmoidal nonlinear hidden layer $h$ to the linear output layer $y$, and a lower layer weight matrix $W$ that links the input and hidden layers. Let the target vectors $t$ be arranged to form the columns of $T$, let the input data vectors $x$ be arranged to form the columns of $X$, let $H = \sigma(W^TX)$ denote the matrix of hidden units, and assume the lower layer weights $W$ are known. The function $\sigma$ performs the element-wise logistic sigmoid operation $\sigma(x) = 1/(1 + \exp(-x))$. Then, learning the upper layer weight matrix $U$ can be formulated as a convex optimization problem

$$\min_U f = ||U^TH - T||_F^2,$$

which has a closed-form solution

$$U^T = TH^t, \quad \text{where} \quad H^t = H^T(HH^T)^{-1}.$$  

At the bottom block, $X$ contains only the raw input data, but for higher blocks of the DSN (as well as the T-DSN), the input data are concatenated with one or more output representations (typically $y$) from the previous blocks. The lower layer weight matrix $W$ in a DSN block can be optimized using an accelerated gradient descent [14] algorithm to minimize the squared error objective in (1). Embedding the solution of (2) into the objective and deriving the gradient, we obtain

$$\nabla_W f = X[H^t \circ (1 - H^t) \circ \Theta],$$

where $1$ is the matrix of all ones, $\circ$ denotes element-wise multiplication, and

$$\Theta = 2H^t(HH^T)(HH^T) - T^T(HH^T).$$

To train a DSN block one iteratively updates $W$ using the gradient in (3), which by design takes into consideration the optimal $U$; after $W$ has been estimated, the closed-form $U$ is computed once, explicitly.

It is to be emphasized that a key element of the standard DSN is that each block outputs an estimate of the final label class (expressed as the vector $y$) and this estimate is concatenated with the original input vector to form the expanded “input” vector for the next block of the DSN. Because the original input is retained for each higher block, it is guaranteed to perform better on the training set than the previous block. In contrast to other deep architectures (e.g., the deep belief network [9]), the DSN does not aim to discover transformed feature representations. Due to this restrictive nature of building hierarchical structures as well as to the simplicity of each block, the core of the DSN is considerably simplified and optimizing network weights is naturally parallelizable. It is noteworthy that for purely discriminative tasks experiments have shown that DSN, despite its simplicity, performs better than the deep belief network [3].

It is also to be clarified that the estimate of the label class, vector $y$, is continuous valued, computed as a linear combination of the hidden layer in each block. The classification decision is made at the top block where the index of the maximal value in vector $y$ determines the label. At the lower blocks of the DSN, the output vector $y$ is not used for decision but is used to concatenate with the original input vector to feed to its immediately upper block. All these attributes have been inherited by the T-DSN to be presented next.
Fig. 1. An example T-DSN architecture with three stacking blocks, where each block consists of three layers and superscript is used to indicate the block number. Inputs (X) and outputs (Y(i−1)) are concatenated to link two adjacent blocks. The hidden layer in each block has two parallel branches (H(i)1 and H(i)2).

2.2 T-DSN: An Architectural Overview

The DSN just reviewed is a special case of the T-DSN we describe now. In Fig. 1, we illustrate the modular architecture of a T-DSN, where three complete blocks are stacked one on another. The stacking operation of the T-DSN is exactly the same as that for the DSN described in [1]. Unlike the DSN, however, each block of the T-DSN has two sets of lower layer weight matrices \( W(1) \) and \( W(2) \). They connect the input layer X with two parallel branches of sigmoidal hidden layers \( H(1) \) and \( H(2) \), shown in Fig. 1. Each T-DSN block also contains a three-way, upper layer weight tensor \( U \) that connects the two branches of the hidden layer with the output layer.

Note that if the T-DSN is used for regression or for classification, then the basic architecture shown in Fig. 1 is sufficient. However, if the T-DSN is to be interfaced with a hidden Markov model (HMM) for structured prediction, such as continuous phonetic or word recognition, it is desirable to convert the final output in Fig. 1 into posterior probabilities via an additional softmax layer (the softmax operation applied to a vector exponentiates each entry and then normalizes the vector’s entries to sum to one, yielding a distribution). One set of the experiments reported in Section 5.2 is obtained with an additional softmax layer added to the top of Fig. 1.

2.3 Bilinear Prediction from Parallel Hidden Layers

Whereas each block of the DSN produces a single hidden representation of the data and linearly maps from the hidden representation to predictions, each block of the T-DSN uses two hidden representations and combines them bilinearly to produce the predictions. A map \( F(u, v) \) is bilinear if it is linear in \( u \) for every fixed \( v \) and linear in \( v \) for every fixed \( u \) [15]. We will see that the T-DSN generalizes the single block structure, replacing a linear map from hidden representation to output with a bilinear mapping while retaining its desirable modeling and estimation properties. That is, we have a generalization from the mapping of \( \mathbb{R}^d \to \mathbb{R}^C \) in the DSN to the mapping of \( \mathbb{R}^{d_1} \times \mathbb{R}^{d_2} \to \mathbb{R}^C \) in the T-DSN.

As illustrated in Fig. 1, the first step in the T-DSN operation is to map an input data vector \( x \in \mathbb{R}^D \) to two parallel branches of hidden representations, \( h(1) \in \mathbb{R}^{d_1} \) and \( h(2) \in \mathbb{R}^{d_2} \). Conceptually, these represent two different views of the data (see Appendix C for an analysis of these views). Each hidden representation is obtained nonlinearly from the input data according to \( h(i) = \sigma(W(i)h(x)) \), where \( W(1) \in \mathbb{R}^{D \times d_1} \) and \( W(2) \in \mathbb{R}^{D \times d_2} \) are two weight matrices to be estimated. The interactions between these two hidden-layer branches and the prediction vector \( y \) are modeled by a third-order weight tensor, \( U \in \mathbb{R}^{d_1 \times d_2 \times C} \). Specifically, in the second step, the two hidden representations are bilinearly mapped to the prediction vector via \( U \). In tensor notation, the operation is

\[
U(h(1), h(2)) = (U \times_1 h(1)) \times_2 h(2) = y,
\]

where \( \times_i \) denotes multiplying along the \( i \)th dimension (mode) of the tensor [16]. In more common notation:

\[
y_k = \sum_{i=1}^{L_1} \sum_{j=1}^{L_2} u_{ijk} h(1)_i h(2)_j = h(1)^T U_k h(2),
\]

where \( U_k \in \mathbb{R}^{L_1 \times L_2} \) denotes the (matrix) slice of \( U \) obtained by fixing the third index to \( k \) and allowing the first two indices to vary.

It is instructive to link the T-DSN’s behavior to that of the DSN, which we can accomplish by manipulating the bilinear notation above. First, define \( \tilde{h} = h(1) \otimes h(2) \in \mathbb{R}^{L_1 \times L_2} \), where \( \otimes \) denotes the Kronecker product: The \( ((i-1)L_2 + j) \)th element of \( h(1) \otimes h(2) \) is equal to \( h(1)_i h(2)_j \), for \( i \in \{1, 2, \ldots, L_1\}, j \in \{1, 2, \ldots, L_2\} \). By definition, \( h \) contains all pairs of products between elements in \( h(1) \) and elements in \( h(2) \). One can then vectorize \( U_k \) to create \( \tilde{u}_k = \text{vec}(U_k) \in \mathbb{R}^{L_1 \times L_2} \) using the ordering that matches \( h \), that is, if the \( \ell \)th element of \( h \) is \( h(1)_i h(2)_j \), then the \( \ell \)th element of \( \tilde{u}_k \) is \( U_{ijk} \). Then:

\[
y_k = \sum_{i=1}^{L_1} \sum_{j=1}^{L_2} U_{ijk} h(1)_i h(2)_j = \tilde{u}_k^T \tilde{h}.
\]

Arranging all \( \tilde{u}_k, k = 1, 2, \ldots, C \), into a matrix \( \tilde{U} = [\tilde{u}_1 \tilde{u}_2 \ldots \tilde{u}_C] \), the overall prediction then becomes

\[
y = \tilde{U}^T \tilde{h}.
\]

Thus, the bilinear mapping from the two hidden-layer branches can be viewed as a linear mapping from a single, implicit hidden representation, \( \tilde{h} \). The linear mapping uses matrix \( \tilde{U} \), which contains all of the elements of the tensor \( U \).
This leads to the same prediction equation as in the DSN, but with an implicit hidden representation \( \tilde{H} \) that contains pairwise multiplicative interactions between \( h_1 \) and \( h_2 \), incorporating second-order statistics of the input data in a parsimonious manner. In Fig. 2, we present an equivalent architecture to the bottom block of Fig. 1, illustrating how the two hidden layers are expanded into an implicit hidden layer. The relationship between the matrices of explicit, lower dimensional hidden units, \( H_1 = \sigma(W_1^T X) \) and \( H_2 = \sigma(W_2^T X) \), and the matrix of implicit hidden units, \( \tilde{H} \), is

\[
\tilde{H} = H_{(1)} \odot H_{(2)}.
\]

The \( \odot \) operation is the Khatri-Rao product [16], which performs a column-wise Kronecker product.

### 3 Learning T-DSN Weights—Algorithm

Because the architectures shown in Figs. 1 and 2 are equivalent, learning the second layer weights given the implicit hidden representation is the same least-squares problem encountered by the DSN. Specifically, the Tikho-nov regularized optimization problem (with training target matrix \( T \)),

\[
\min_{\tilde{U}} \|T - \tilde{U}^T \tilde{H}\|_F^2 + \lambda \|U\|_F^2,
\]

has the closed-form solution of

\[
\tilde{U}^T = T \tilde{H}^{-1} = T \tilde{H}^T (\tilde{H} \tilde{H}^T + \lambda I)^{-1}.
\]

Substituting the constraint of (11) into the overall objective of (10), we can make the full learning more effective by coupling the lower weight matrices with the upper weight matrix. In other words, as with the DSN, the upper layer weights of a T-DSN block are deterministically computed given the lower layer weights and thus they do not need to be learned separately and independently. This contrasts with the standard neural network learning algorithms, where both sets of weights are updated incrementally with no direct constraints on each other. Such constraints are available for T-DSN due to the use of linear output layers in each block, and are not available for standard neural networks with nonlinear output layers.

We now address the more difficult learning problem for the two lower weight matrices \( W_{(1)} \) and \( W_{(2)} \) since \( \tilde{H} \) is a deterministic function of the lower layer weights. In this paper, we adopt the strategy of optimizing \( W_{(1)} \) and \( W_{(2)} \) using methods requiring only first-order oracle information, i.e., those requiring only objective function evaluations and the gradients of the objective function with respect to \( W_{(1)} \) and \( W_{(2)} \). Note that this includes approximate second-order optimization methods such as L-BFGS. The derivation of these gradients has commonality to the DSN case, with an extra step necessitated by the Khatri-Rao product. To simplify the notation throughout the paper we assume \( \lambda = 0 \) in (10) (a detailed derivation of the gradient for arbitrary \( \lambda \) is provided in Appendix A).

With notation analogous to (4), let \( \tilde{\Theta} \) denote

\[
\tilde{\Theta} = \nabla_{\tilde{H}^T f} = 2 \tilde{H}^T (\tilde{H}^T \tilde{H})(\tilde{H}^T) - T^T (\tilde{H} \tilde{H}^T).
\]

By the chain rule, we obtain

\[
\nabla_{H_{(1)} f}_{(1)} = \langle \tilde{\Theta}^T, E_{(1)n} \rangle \odot H_{(2)}
\]

\[
= \sum_{k=1}^{L_2} H_{(2)kn} \tilde{\Theta}((i-1)L_1+k,n),
\]

\[
\nabla_{H_{(2)} f}_{(2)} = \langle \tilde{\Theta}^T, H_{(1)} \rangle \odot E_{(2)n}^{l_2 \times n}
\]

\[
= \sum_{k=1}^{L_1} H_{(1)kn} \tilde{\Theta}((k-1)L_2+j,n),
\]

where \( E_{m \times n} \) denotes an \( m \times n \) matrix with entry \( (i,j) \) equal to one and all other entries zero. Let \( \Psi_{(1)} \in \mathbb{R}^{L_1 \times N} \) and \( \Psi_{(2)} \in \mathbb{R}^{L_2 \times N} \) denote the matrices \( \nabla_{H_{(1)} f} \) and \( \nabla_{H_{(2)} f} \), respectively. Then, following the derivation of the DSN, we obtain

\[
\nabla_{W_{(1)} f} = X(H_{(1)}^T \odot (1 - H_{(1)}^T) \odot \Psi_{(1)})
\]

and

\[
\nabla_{W_{(2)} f} = X(H_{(2)}^T \odot (1 - H_{(2)}^T) \odot \Psi_{(2)}).
\]

The \( \Psi_{(1)} \) and \( \Psi_{(2)} \) matrices above have the effect of bridging the high-dimensional representation used in \( \tilde{\Theta} \) with the low-dimensional representation in \( H_{(1)} \) and \( H_{(2)} \), and are needed due to the Khatri-Rao product.

Using the above gradients one can train a T-DSN block using a number of algorithms; in our experiments, we use the L-BFGS and gradient descent implementations in the Poblano optimization toolbox [17]. Our experience suggests that training a T-DSN block requires around 10-15 iterations of L-BFGS, with up to seven gradient evaluations per iteration for line search. In our experiments, the weight matrices \( W_{(1)} \) and \( W_{(2)} \) are initialized with random values in a range that is tuned using the validation set.
From (12) and (17), it is clear that the bulk of the gradient computation is in matrix operations, including matrix multiplications and element-wise matrix products. To bypass memory limitations and to speed up training, we parallelize these matrix operations to run on a CPU cluster. The ability to parallelize training in this manner is key for the scalability of T-DSN training.

### 3.1 Connections between T-DSN and DSN

We show here that the DSN is, in fact, a special case of the T-DSN. To distinguish this special case from the general case, we will use $\hat{H}$ and $\hat{U}$ to denote the T-DSN’s $H$ and $U$, respectively. As before, let $h_{(1)}$ and $h_{(2)}$ denote the two hidden representations in a T-DSN block and let $h$ denote the (only) hidden representation in a DSN block. Let $W_{(1)}$, $W_{(2)}$, and $W$ denote the corresponding first-layer weight matrices. A DSN is then a special case of a T-DSN when $L_1 = 1$, $L_2 = 1$, $W_{(1)} = 0$, and $W_{(2)} = W \in \mathbb{R}^{L \times N}$. Then $h_{(1)} = \sigma(0^T x) = 1/2$ and $h_{(2)} = h$, and it follows that

$$\hat{H} = h_{(1)} \odot h_{(2)} = \frac{1}{2} H.$$  

(18)

In this case, the pseudo-inverse is

$$\hat{H}^\dagger = \hat{H}^T (\hat{H} \hat{H}^T)^{-1} = \frac{1}{2} H^T \left( \frac{1}{4} H \hat{H}^T \right)^{-1} = 2H^\dagger,$$

(19)

giving the least-squares solution $\hat{U}^T = \hat{H}^T \hat{H}^\dagger = 2TH^\dagger$. Hence, the T-DSN and DSN predictions are identical:

$$\hat{U}^T \hat{H} = (2U^T) \left( \frac{1}{2} H \right) = U^T H = Y.$$  

(20)

In Appendix B, we show that under these same conditions, gradients $\nabla_{W_{(1)}} f$ and $\nabla_{W} f$ are identical as well. While the DSN is a special, extremely asymmetric, case of the T-DSN, we have found that the closer the two hidden-layer branches’ dimensions are, the better the classification performance (see Section 5.2 for empirical evidence). In the nondegenerated cases ($L_1 \approx L_2 \gg 1$), the T-DSN dramatically shifts the balance of parameters from the lower layer weights to the upper layer weights. Whereas the DSN has $D \times L$ lower layer (explicit) parameters and $L \times C$ upper layer (implicit) parameters, the T-DSN has $D \times (L_1 + L_2)$ lower layer parameters and $L_1 \times L_2 \times C$ upper layer parameters. We conjecture that the observed advantages of using equal numbers of hidden units is because the symmetric case maximizes the ratio of implicit feature dimension $L_1 L_2$ over explicit feature dimension $L_1 + L_2$, and thus makes the best use of the closed-form upper layer parameters. The key advantage of the nondegenerated T-DSN over the degenerated one (i.e., DSN) is the new ability to capture higher order feature interactions via the cross product.

### 4 Learning T-DSN Weights—Parallel Implementation

Stochastic minibatch training is commonly employed in deep learning training. Empirically, researchers often observe diminishing returns in classification accuracy performance for gradient methods as the minibatch size increases. In contrast, due to the embedded least-squares problem, T-DSN’s accuracy in classification tasks continues to improve as the minibatch size increases. For this reason, it is desirable to use the largest possible amount of training data at each iteration. In this section, we analyze the time and space complexities of our T-DSN training algorithm, and introduce a parallel training method that allows us to scale to large training sets.

#### 4.1 Sequential Training Computational Complexity

To learn the T-DSN weights, we need to evaluate both the objective function and its gradients with respect to $W_{(1)}$ and $W_{(2)}$ at each iteration. Let $L = L_1 L_2$ denote the number of implicit hidden units. Then, computing the gradients defined by (17) involves a sequence of cached intermediate steps with the time complexities listed in Table 1, and has an overall space complexity of $O((L + D + C)N)$ (due to the need to store $X$, $\hat{H}$, and $\hat{H}^\dagger$ in memory). Evaluating the objective function in (10) has a time complexity of $O(N L (C + D))$ with the same space complexity. In practice, for large enough $N$ this space complexity will exceed the main memory of a single machine, creating the ability to parallelize over many machines crucial for handling very large datasets.

### 4.2 Parallelizing Matrix Operations

There are well-known techniques for parallel matrix multiplication of the form $A = B C^T \in \mathbb{R}^{r \times r}$; in general, they break $B \in \mathbb{R}^{q \times N}$ and $C \in \mathbb{R}^{N \times N}$ into submatrices that can be combined to produce $A$. Because our multiplies will involve instances where common dimension of $B$ and $C^T$ (i.e., the number of training samples) is significantly larger than both $q$ and $r$, we use the following basic matrix multiplication parallelization strategy:

$$A = \sum_{k=1}^{P} B^{(k)} C^{(k)T},$$  

(21)

where $B^{(k)}$ denotes the $k$th subblock of matrix $B$ that has been divided into $P$ subblocks along the second dimension:

$$B = [B^{(1)} \ B^{(2)} \ldots \ B^{(P)}].$$  

(22)

And likewise, we have for matrix $C^T$:

$$C^T = \begin{bmatrix} C^{(1)T} \\ \vdots \\ C^{(P)T} \end{bmatrix}.$$  

(23)

Decomposing the large matrix operations into many small submatrix operations is the key to parallelizing the gradient and objective function computation.
4.3 Parallel Pipeline

Given current values for the T-DSN parameters $W_1(1)$ and $W_1(2)$, we use the above parallelization strategy to compute the objective function value and gradients. Our parallelization is over training data points: We split any matrix $M$ with a second dimension $N$ into $P$ submatrices $M_{hk}$, each of which has $N_k$ columns.

The computation pipeline is broken into a large number of jobs, as illustrated by the directed acyclic graph in Fig. 3. The arrows denote dependence: A job can run once all of the jobs feeding into it have completed. The results of each job in Fig. 3 are cached and used in subsequent processing. There are three qualitatively different kinds of jobs, denoted by the different shapes in the figure. The green three-dimensional boxes each denote a set of $P$ jobs, where the individual jobs process a fraction of the total dataset (the $k$th batch). The orange rectangle jobs are accumulators, and only need to sum over the $P$ individual files on which they depend. Because they begin accumulating the sum of their ancestors as each one finishes, they introduce minimal delay into the pipeline. The jobs marked with red octagons cannot be run until all ancestor jobs have been finished since they require synchronizing over the $P$ batches.

Table 2 lists the full set of intermediate variables that must be computed to construct the gradient and evaluate the function; it is a superset of the variables listed in Fig. 3. Note that no variable that is dependent on the dataset size (i.e., has a dimension equal to $N_k$) is cached. These variables are recomputed according to their definitions each time as the cost of caching these variables to the disk was found to exceed the cost of recomputing them.

At the conclusion of the parallel pipeline, the variable $s$ denotes the objective function $f$ evaluated at $f(W_1(1), W_1(2))$, and the variable $G$ denotes the concatenation of the gradients, $\nabla W_1 f$.

4.3.1 Parallel Timings

Our original and primary motivation for the parallel implementation was to allow us to scale beyond the memory limit of a single machine. We also measure the effect of parallelization on speed. As usual, there is cost associated with parallelization, namely, the interprocess communication time. Because our implementation uses network disk to store and load cached variables, this cost is nontrivial. Fig. 4 measures empirical wall-clock runtimes over repeated single instances of the parallel pipeline (i.e., each computing the gradient and evaluating the objective) on the TIMIT data (1.12m training samples). Specifically, the mean and the upper and lower 95 percent confidence intervals are plotted. To produce the timing results in this figure, we use hidden representations of dimension $L_1 = L_2 = 40$, and repeat a single instance of the parallel pipeline eight times over the number of machines, $P$, across the number of cores.

### Table 2

<table>
<thead>
<tr>
<th>Variable</th>
<th>Dimension</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H^{(k)}_1$</td>
<td>$L_1 \times N_k$</td>
<td>$\sigma(W_1(1) X^{(k)})$</td>
</tr>
<tr>
<td>$\bar{H}^{(k)}_1$</td>
<td>$L \times N_k$</td>
<td>$H^{(k)}_1 \odot H^{(k)}_2$</td>
</tr>
<tr>
<td>$B^{(k)}$</td>
<td>$L \times L$</td>
<td>$H^{(k)}_1 \bar{H}^{(k)}_1^T$</td>
</tr>
<tr>
<td>$F^{(k)}$</td>
<td>$L \times C$</td>
<td>$\sum_{i=1}^P B^{(k)}_i$</td>
</tr>
<tr>
<td>$B$</td>
<td>$L \times L$</td>
<td>$F^{(k)}$</td>
</tr>
<tr>
<td>$C$</td>
<td>$L \times C$</td>
<td>$H^{(k)}_1 \bar{H}^{(k)}_1^T B_1^{-1}$</td>
</tr>
<tr>
<td>$D^{(k)}$</td>
<td>$C \times L$</td>
<td>$B_1^{-1} F$</td>
</tr>
<tr>
<td>$D$</td>
<td>$C \times L$</td>
<td>$T^{(k)} \bar{H}^{(k)}_1$</td>
</tr>
<tr>
<td>$s^{(k)}$</td>
<td>$1 \times 1$</td>
<td>$\sum_{i=1}^P s^{(k)}_i$</td>
</tr>
<tr>
<td>$s$</td>
<td>$1 \times 1$</td>
<td>$\sum_{k=1}^K s^{(k)}_i$</td>
</tr>
<tr>
<td>$\bar{G}^{T(k)}$</td>
<td>$L \times N_k$</td>
<td>$2 \bar{H}^{(k)}_1 F D - T^{(k)} T D$</td>
</tr>
<tr>
<td>$\Psi^{(1)}$</td>
<td>$L_1 \times N_k$</td>
<td>$\Psi^{(1)}<em>{in} = (E^{(1)}</em>{L_1} \otimes H^{(k)}_2) \theta^{(k)}$</td>
</tr>
<tr>
<td>$\Psi^{(2)}$</td>
<td>$L_2 \times N_k$</td>
<td>$\Psi^{(2)}_{in} = (H^{(k)}<em>1 \otimes E^{(2)}</em>{L_2} \otimes \theta^{(k)}$</td>
</tr>
<tr>
<td>$G^{(k)}$</td>
<td>$D \times L_i$</td>
<td>$X^{(k)}(H^{(k)}_1 \circ (1 - H^{(k)}<em>2) \circ \Psi^{(k)}</em>{in}$</td>
</tr>
<tr>
<td>$G$</td>
<td>$D \times (L_1 + L_2)$</td>
<td>$G^{(k)} \circ C^{(k)} + \sum_{k=1}^K G^{(k)}$</td>
</tr>
</tbody>
</table>

Variables with a dimension equal to $N_k$ are recomputed each time; others are cached to disk.
which the parallel training is distributed. On each machine processing is parallelized over eight cores. A fixed, additional overhead associated with initializing the data is also included in the presented times; in practice, these overhead costs would become negligible over the course of training a multiple-block T-DSN. The minimum value for \( P \) is four since lower values caused the compute nodes’ memory to be exceeded. For this dataset, the lowest average times are achieved in the range between \( P = 10 \) and \( P = 25 \) (80-200 cores). After this, there is a gradual rise in the total computation times as improvements in computation time are outpaced by the additional disk access and communication costs. In practice, because the speedup is relatively insensitive to the degree of parallelization, we simply set \( P \) to be sufficiently large that training does not exceed the compute nodes’ memory limits. Note that the stacking nature of the T-DSN means that one cannot parallelize over blocks, only within blocks. Because, in practice, the number of blocks is limited and the bulk of the computation is spent within blocks, this does not prove to be a major obstacle to training.

5 Evaluation Experiments

In this section, we detail the experiments and present the results aimed at evaluating the effectiveness of the T-DSN architecture described in the preceding sections. Three well-known image and speech databases for benchmarking are used: MNIST, TIMIT, and WSJ.

5.1 MNIST Handwriting Image Recognition

In the first set of experiments, we evaluate the T-DSN architecture and the learning algorithms on the MNIST database of binary images of handwritten digits [18]. The digits have been size-normalized while preserving their aspect ratio. Each original image is centered by computing and translating the center of mass of the pixels, yielding a \( 28 \times 28 \) image. The task is to classify each \( 28 \times 28 \) image into one of the 10 digits. The MNIST training set is composed of 60,000 examples from approximately 250 writers. The test set is composed of 10,000 patterns. The sets of writers of the training set and test set are disjoint. In the experiments, a small fraction of the training data is held out as a validation set to tune hyperparameters in the T-DSN. The properties of the validation and test sets in MNIST are found to be very similar to each other.

The architecture of the T-DSN used in the MNIST experiment was shown in Fig. 1, except with four stacking blocks instead of three. The input layer at the bottom block consists of 784 units, one for each black-white pixel in the \( 28 \times 28 \) image. Each of the two hidden layers consists of \( L_1 \) and \( L_2 \) sigmoidal units, denoted by \( L_1 \times L_2 \) as its hidden-layer configuration. All blocks have the same hidden-layer configuration in all experiments reported in this section. The prediction layer of all blocks has 10 linear units, corresponding to 10 digit output classes. The input layer at the nonbottom blocks is a concatenation of the raw input data and the prediction layer’s output from the previous block, thus having the dimensionality of 794.

Fig. 5 shows the training objective (mean square error) between the prediction layer’s output and the zero-one target averaged over the full training set as a function of each T-DSN block and also as a function of epochs in the batch-mode gradient-decent training for each T-DSN block. The corresponding test-set classification error rate is shown in Fig. 6. The hidden-layer configuration of the T-DSN used here is \( L_1 \times L_2 = 90 \times 90 \). The two sets of input matrix weights to the two hidden layers in each T-DSN block are initialized with small uniform random numbers for parallel gradient-decent training.

The results in Figs. 5 and 6 show that when a new block is added to the T-DSN, both the training objective function and the test error rate are reduced even if the training in the previous block already reached near convergence. Further, we observe in Fig. 5 that as more blocks are added, the training objective continues to drop to close to zero and, although not shown in the figure, the error rate for the training set drops to zero as well. There is no obvious rise in errors observed for the test set (see Fig. 6).

In Table 3, we show the relationship between the test error rate and the hidden-layer configuration. The error rate is obtained at the convergence of the training with the optimal number of blocks and several other hyperparameters (learning rates, size of the initial random weights, and so on) determined on the validation set. We observe that with the same number of implicit hidden units, the symmetric configuration is significantly better than non-symmetric configurations. Also, the hidden layers should...
be sufficiently large to produce low error rate, which is possibly limited by the amount of training data and can be determined on the validation set.

As an extreme case, when one of the two hidden layers in each block reduces to a single unit, the corresponding T-DSN behaves like a DSN with a significantly higher error rate—e.g., $30 \times 30$ versus $900 \times 1$—as shown in Table 3.

The MNIST website provides the results of 68 classifiers. A very large and deep convolutional neural network gives the state-of-the-art error rate of 0.39 percent [19]. The use of distortions to augment the training data is important to achieve this lowest error rate. Without the use of the distortions, which is impractical in real applications, the error rate was increased to 0.53 percent. Without the use of convolutional structure and distortions or any other types of special preprocessing, the lowest error rate, 0.83 percent, was reported in [3] by a carefully tuned and optimized DSN. The error rate of 1.21 percent reported in this paper is comparable to that achieved using the deep belief network as reported in [9]. It was obtained without careful tuning, without passing of the learned weights from one block to another, and without pretraining of weights using restricted Boltzmann machines, which were all exploited in [3].

### 5.2 TIMIT Phone Classification and Recognition

In the second set of experiments, the TIMIT database is used for evaluating the T-DSN. The training set consists of 462 speakers. The total number of frames in the training data is 1,124,589. The validation set provided by the database contains 50 speakers, with a total of 122,488 frames. Results are reported using the standard 24-speaker core test set consisting of 192 sentences with 7,333 phone tokens and 57,920 frames.

The speech data is analyzed using standard Mel frequency cepstral coefficients (MFCCs). All experiments used a context window of 11 frames. This gives a total of $39 \cdot 11 = 429$ elements in each feature vector as the raw input to T-DSN. This window size was shown to be optimal for the TIMIT phone recognition task in different kinds of deep networks published earlier (e.g., [20]) and has not been customized for the T-DSN in this study. For the prediction at each layer of the T-DSN, we use 183 target class labels (i.e., three states for each of the 61 phones), which we call “phone states,” with a zero-one (also called one-hot) encoding scheme. Phone boundaries are labeled in the corpus; we obtain the phone state labels by a state-frame alignment scheme. Phone boundaries are labeled in the corpus; we obtain the phone state labels by a state-frame alignment scheme.

The main T-DSN architecture illustrated in Fig. 1, where the number of stacking blocks is between 8 to 13 as determined on the validation set. In some experiments, additional one or more hidden layer(s) and a softmax layer are added to the top of the T-DSN for computing frame-level state posterior probabilities. This latter step is needed for the phone recognition task when a further dynamic programming step is used to reach the phone recognition decision. Only symmetric hidden-layer configurations are used, and we tune configurations between $L_1 \times L_2 = 70 \times 70$ and $100 \times 100$.

In Tables 4a and 4b, we compare the frame-level State classification error rates (F-SER) with (b) and without (a) using a trained softmax layer on top of the T-DSN. Obtaining the results of F-SER requires no additional postprocessing, making the TIMIT experiment as simple as MNIST. Comparing (a) and (b), we observe noticeable error reduction after the softmax layer is added. In each case, we also compare T-DSN and its corresponding DSN. Similarly to the MNIST experiments, the T-DSN gives lower errors, especially in the case of softmax output (b). With the softmax output, we can also evaluate using the cross entropy (CE) measure for the test set. Cross entropy is the average of negative log (base-\(e\)) posterior probabilities over all frames in the test set, computed from the softmax layer. The lower the cross entropy, the better the performance. In Table 4b, we further compare T-DSN and its corresponding DSN. Using the softmax output, we can also evaluate using the cross entropy (CE) measure for the test set. Cross entropy is the average of negative log (base-$e$) posterior probabilities over all frames in the test set, computed from the softmax layer. The lower the cross entropy, the better the performance.

We now present a new set of TIMIT results, which are more meaningful to speech researchers, after postprocessing of the T-DSN outputs in Table 5. The first measure is framework phone error rate, computed by 1) collapsing three sequential units (states) associated with each phone into one single phone class using majority voting over all frames within the phone boundaries in each test sentence as provided by the TIMIT database and 2) collapsing a total of 183 output units into 39 phone-like units [22]. Using this new measure after collapsing, we observe a lower error rate using T-DSN than DSN and two versions of DNN. When the outputs of T-DSN are fed further to a five-hidden-layer DNN, denoted as “T-DSN + DNN” in Table 5, the framework phone error rate is dropped further.

The second measure (shown in the last column of Table 5) is continuous phonetic recognition error rate. This is computed by 1) collapsing a total of 183 T-DSN output

| TABLE 3 |
| MNIST: Test-Set Error Rate at Convergence as a Function of Hidden-Layer Configuration |
|---|---|
| Configuration | Test Error Rate |
| $100 \times 100$ | 1.21% |
| $90 \times 90$ | 1.22% |
| $100 \times 110$ | 1.30% |
| $180 \times 45$ | 1.38% |
| $80 \times 60$ | 1.50% |
| $90 \times 40$ | 1.65% |
| $50 \times 50$ | 1.70% |
| $40 \times 40$ | 1.95% |
| $1500 \times 1$ | 2.70% |
| $30 \times 30$ | 2.20% |
| $900 \times 1$ | 3.25% |

| TABLE 4 |
| TIMIT: Comparing T-DSN (and DSN) (a) Before and (b) After Adding Softmax Layers to Produce Posterior Probabilities, in Terms of Frame-Level State Error Rate (F-SER) and CE Value |
|---|---|---|
| (a) Linear Output | (b) Posterior Output |
| Model | F-SER | Model | F-SER | CE |
| TDSN | 42.6% | TDSN | 40.9% | 2.02 |
| DSN (mini-batch) | 44.3% | DSN | 41.8% | 2.16 |
| DSN (full-batch) | 42.7% | DNN | 45.0% | 2.26 |
| DNN (full-batch) | 42.7% | MMI-DNN | 43.0% | 2.20 |
In our experiments reported in the right column of Table 5, the output needs to be in the form of probabilities. To accomplish the use of an HMM to interface with the frame-level phone recognition, we have found little difference between mean square error and cross entropy as measures of frame-level error rates and cross entropy, both DSN and T-DSN outperform DNN and even DNN alone has not matched the low phonetic recognition error rate achieved by DNN. T-DSN produces a slightly lower error rate (22.8 percent versus 22.9 percent). Further, the two differences are due to 1) the use of specially designed convolutional structure, 2) the use of filterband instead of MFCC features, and 3) very expensive optimization using backpropagation. The first two of these differences can be incorporated into the T-DSN architecture (as future work).

For pure classification problems such as MNIST and TIMIT results, we also observe here that larger (and symmetric) hidden layers are better than smaller ones. Importantly, each entry in Table 7 shows a noticeably lower error rate than 7, which in turn gives a further lower error rate than a single frame. In Table 6, we show the performance of a single block of the T-DSN, measured by the frame-level phone classification error rate. In the results presented in Table 6, the two weight matrices $W_1$ and $W_2$ in Fig. 1 are randomized and not learned via gradient descent. Learning is applied only to tensor $U$ according to (11). Consistent with the MNIST and TIMIT results, we also observe here that larger and (symmetric) hidden layers are better than smaller ones in the T-DSN. Further, a window size of 11 gives a noticeably lower error rate than 7, which in turn gives a further lower error rate than a single frame.

In Table 7, frame-level phone classification error rates are shown after five blocks of T-DSN are built, where each block runs gradient-decent learning until convergence with details described in Sections 3 and 4. No softmax layer is added to top of the T-DSN. Again, at the learning convergence, the T-DSNs with larger hidden layer sizes and larger input window sizes are superior to those using smaller ones. Importantly, each entry in Table 7 shows a significantly lower error rate than the corresponding entry in Table 6, demonstrating the effectiveness of building deep units into 39 phone-like units, 2) normalizing the softmax outputs of T-DSN by state priors so that the posterior probabilities of states are converted to a quantity proportional to data likelihoods for each state, and 3) using a dynamic programming step across full sentences to determine phone recognition errors (substitution, deletion, and insertion errors). This last step is also called “phonetic decoding,” where a standard bigram phone-level “language” model is used with the language model weight and insertion penalty tuned using the validation data. The results in Table 5 using this measure again demonstrate superior performance of T-DSN, especially when the outputs of T-DSN are further processed by a DNN.

The original motivation of this work was to make the learning of deep networks scalable by replacing stochastic gradient descent algorithm for fine tuning with the parallelizable batch-mode learning. As both the DSN and T-DSN do “fine tuning” only within the block rather than through the entire deep network as carried out for DNN, we expected at best a matching performance to DNN. While DSN alone has not matched the low phonetic recognition error rate achieved by DNN, T-DSN produces a slightly lower error rate (22.8 percent versus 22.9 percent). Further, for the measures of frame-level error rates and cross entropy, both DSN and T-DSN outperform DNN and even MMI-DNN [20]. The state-of-the-art TIMIT phone recognition error rate is 20.1 percent, reported and analyzed very recently in [23], lower than 22.8 percent reported here. The differences are due to 1) the use of specially designed convolutional structure, 2) the use of filterband instead of MFCC features, and 3) very expensive optimization using backpropagation. The first two of these differences can be incorporated into the T-DSN architecture (as future work).

For pure classification problems such as MNIST and frame-level phone classification, we have found little difference between mean square error and cross entropy as the loss. However, for continuous phone recognition requiring the use of an HMM to interface with the frame-level classifier, the output needs to be in the form of probabilities. In our experiments reported in the right column of Table 5, we use cross entropy for learning, which is substantially more expensive in computation obtains good results.

### 5.3 WSJ Phone Classification

In the third set of experiments, we use another popular but larger speech database, called 5k-WSJ0, designed for speaker independent speech recognition tasks [25]. As suggested by the name, the 5k-WSJ0 database uses a 5,000 word vocabulary. The training material from the SI84 set (7,077 utterances, or 15.3 hours of speech from 84 speakers) in the database is separated into a 6,877-utterance training set and a 200-sentence validation set. Evaluation was carried out on the Nov92 evaluation data with 330 utterances from eight speakers. In this paper, we use the same MFCCs and their deltas as in the TIMIT experiments for the short-time spectral representation of the speech signal. With the 10 millisecond frame rate, this database gives over 5-million frames (i.e., samples) in the training data (5,232,244 to be exact), substantially larger than MNIST and TIMIT. Further, unlike the TIMIT database where the phone boundaries in training data are provided by human annotators, no phone boundaries are given in WSJ0. In this paper, we generate the phone labels and their boundaries in the training data from the forced alignments using a tied-state crossword triphone Gaussian-mixture-HMM speech recognizer. Test-set labels are produced in the same way. These phone labels, with a total of 40 of them, together with their boundaries provide one-to-one mapping between each speech frame with its phone label as the target for training the T-DSN.
T-DSN and of the learning algorithms with its parallel implementation described in Sections 3 and 4.

6 DISCUSSIONS AND CONCLUSION

A new architecture for deep learning is presented, the T-DSN, generalizing the earlier DSN architecture. The principal novelty is to split the original large hidden layer (in each block) into two smaller ones and, through their multiplicative outer product and the associated higher order covariance structure in binary (their multiplicative outer product and the associated information while effectively gating the other hidden-to-hidden representations to encode speaker or environmental factors by training one of the output mapping. Moreover, the T-DSN is equipped with speaker and/or environmental factors by training one of the output mapping. Moreover, the T-DSN is equipped with the well-known closed-form solution to this problem is

$$f = \|U^T \hat{H} - T\|^2_F + \lambda \|U\|^2_F.$$

The known closed-form solution to this problem is

$$U^T = T\hat{H}^\dagger, \quad \hat{H} = \tilde{H}^T A^{-1}, \quad A = (\tilde{H}\tilde{H}^T + \lambda I)$$

APPENDIX A

T-DSN GRADIENT DERIVATION

In this appendix, we derive the gradients used for training our lower level weight matrices, $W_1$ and $W_2$, under the most general conditions.

A.1 Finding the Optimal $U^T$

Given a fixed implicit hidden representation matrix $\hat{H}$, consider the Tikhonov regularized least-squares objective

$$f = \|U^T \hat{H} - T\|^2_F + \lambda \|U\|^2_F.$$

The known closed-form solution to this problem is

$$U^T = T\hat{H}^\dagger, \quad \hat{H} = \tilde{H}^T A^{-1}, \quad A = (\tilde{H}\tilde{H}^T + \lambda I)$$

A.2 Deriving $\nabla_{\hat{H}} f$

We can substitute the closed form solution for $U^T$ given in (25) into the objective function. Our ultimate goal is to express this as a function of $W_1$ and $W_2$, and compute the gradients with respect to these lower level weight matrices. As an intermediate step, we first compute the gradient of the objective with respect to $\hat{H}^\dagger$.

A.2.1 General Gradient

$$f = \|U^T \hat{H} - T\|^2_F + \lambda \|U\|^2_F$$

Denote the first term of (26) by $o(\hat{H})$ and the second term by $\beta(\hat{H})$. First, we derive $\nabla_{\hat{H}} \alpha(\hat{H})$. By the linearity of trace, we obtain

$$\nabla_{\hat{H}} \alpha(\hat{H}) = \nabla_{\hat{H}} \text{Tr}(THH^T\tilde{H}^\dagger U^T T\hat{H}^\dagger)$$

Before we can evaluate $\nabla_{\hat{H}} f$, let us introduce five lemmas.

Lemma A.1. Let $A$ denote $(\tilde{H}\tilde{H}^T + \lambda I)^{-1}$ (as before), and let $Z$ denote an arbitrary real $N \times N$ matrix. Then,

$$\nabla_{\hat{H}} \text{Tr}(A^{-1}\tilde{H}ZH\tilde{H}^T) = (I - \tilde{H}^\dagger \tilde{H})(Z + Z^T)\hat{H}^\dagger.$$
Lemma A.2.

Proof. Let $P$ denote the constant matrix that is the result of evaluating $A^{-1}$ with a fixed $\tilde{H}$. Let $Q$ denote the constant matrix that is the result of evaluating $\tilde{H}Z\tilde{H}^T$ with a fixed $\tilde{H}$. It follows that

$$\nabla_{\tilde{H}}^T \text{Tr}(A^{-1} \tilde{H}Z\tilde{H}^T) = \nabla_{\tilde{H}}^T \text{Tr}(P\tilde{H}Z\tilde{H}^T) + \nabla_{\tilde{H}}^T \text{Tr}(A^{-1}Q).$$

Using (107) from [29], and noting $A$'s symmetry, we can evaluate the first term in (30)

$$\nabla_{\tilde{H}}^T \text{Tr}(P\tilde{H}Z\tilde{H}^T) = \nabla_{\tilde{H}}^T \text{Tr}(Z\tilde{H}^T P\tilde{H}) = Z\tilde{H}^T A^{-1} + Z\tilde{H}^T A^{-1} = (Z + Z^T)\tilde{H}.$$  

Using (114) from [29] we can evaluate the second term in (30)

$$\nabla_{\tilde{H}}^T \text{Tr}(A^{-1}Q) = -\tilde{H}^T A^{-1} (\tilde{H}Z\tilde{H}^T + \tilde{H}Z\tilde{H}^T)A^{-1} = -\tilde{H}^T \tilde{H}(Z + Z^T)\tilde{H}.$$  

Substituting (31) and (32) into (30) completes the first part of the lemma. The second part is trivial: For symmetric $Z$ we have $Z + Z^T = 2Z$. □

Lemma A.3.

Proof. This follows from Lemma A.2 and the fact that $\text{Tr}(PQ) = \text{Tr}(QP)$.

$$\nabla_{\tilde{H}}^T \text{Tr}(T \tilde{H}^T \tilde{H}^T) = \nabla_{\tilde{H}}^T \text{Tr}(T \tilde{H}^T A^{-1} \tilde{H}^T) = \nabla_{\tilde{H}}^T \text{Tr}(\tilde{H}^T A^{-1} \tilde{H}^T) = 2(I - \tilde{H}^T \tilde{H})T^T \tilde{H}^T.$$  

Lemma A.4.

Proof. This follows from (27) and Lemmas A.2 and A.3. □

Lemma A.5.

Proof. Let $P$ denote a constant matrix that is the result of evaluating $U^T A^{-1}$ with a fixed $\tilde{H}$. Let $Q$ denote the constant matrix that is the result of evaluating $\tilde{H}T^T$ with a fixed $\tilde{H}$. It follows that

$$\nabla_{\tilde{H}}^T \text{Tr}(U^T) = \nabla_{\tilde{H}}^T \text{Tr}(P\tilde{H}T^T) + \nabla_{\tilde{H}}^T \text{Tr}(U^T A^{-1}Q).$$

The first term of (38) is

$$\nabla_{\tilde{H}}^T \text{Tr}(P\tilde{H}T^T) = \nabla_{\tilde{H}}^T \text{Tr}(\tilde{H}^T P) = 2T^T \tilde{H}^T A^{-1}.$$  

The second term of (38) is

$$- \nabla_{\tilde{H}}^T \text{Tr}(U^T A^{-1}Q) = -\tilde{H}^T (Q^T + UQ^T)A^{-1} = -\tilde{H}^T (\tilde{H}^T T^T \tilde{H}^T + \tilde{H}^T T^T \tilde{H}^T)A^{-1},$$  

where the first equality comes from [29, (114)]. Substituting (39) and (40) into (38) proves the lemma. □

Theorem A.6.

Proof. This follows from (26) with Lemmas A.4 and A.5. □
A.2.2 Simplified Gradient

When no regularization is used in the objective (i.e., \( \lambda = 0 \)), the gradient can be simplified. Importantly, in this case \( \tilde{H} = \hat{H} \) (the pseudo-inverse). Recall that \( \tilde{H} \tilde{H} \tilde{H} = \hat{H} \) and \( \tilde{H} \tilde{H} \tilde{H} = \hat{H} \). Under these conditions:

\[
\nabla_{\hat{H}}^T f = 2(1 - \hat{H}^\dagger \hat{H})^\dagger \hat{H}^T \tilde{T} \hat{H}^\dagger + 2(1 - \hat{H}^\dagger \hat{H})^T \tilde{T} \hat{H} \hat{H}^\dagger - 4(1 - \hat{H}^\dagger \hat{H})^T \tilde{T} \hat{H}^\dagger \hat{H}^\dagger \
\]

\[
= 2\hat{H}^\dagger \hat{H}^T \tilde{T} \hat{H}^\dagger - 2\hat{H}^T \tilde{T} \hat{H}^\dagger. \quad \tag{43}
\]

The notation \( \langle \cdot , \cdot \rangle \) denotes matrix inner product: It is an element-wise multiplication followed by a sum.

By another application of the chain rule (e.g., as stated in \([29, (126)]\)):

\[

\frac{\partial \hat{H}}{\partial H} = E_{(i,n)}^{L \times N} \odot H(2), \tag{46}
\]

where, as used earlier in the paper, \( E_m^{L \times N} \) denotes an \( L_1 \times N \) matrix that is zero everywhere except 1 in the \((i, n)\)th position. Let \( \Psi_{(i)} \) denote the matrix \( \nabla_{H_{(i)}} f(\hat{H}) \).

Similarly, we use \( \Psi_{(2)} \) to denote \( \nabla_{H_{(2)}} f(\hat{H}) \), where

\[
\nabla_{H_{(i)}} f(\hat{H}) \bigg|_{in} = \left\langle \Psi_{(i)}^T, \frac{\partial \hat{H}}{\partial H(1)} \bigg|_{in} \right\rangle. \quad \tag{47}
\]

Let \( Z_{(i)} \) denote \( W_{(i)}^T X \). By another application of the chain rule, we get

\[

\nabla_{Z_{(i)}} f(H(1)) \bigg|_{jk} = \left\langle \Psi_{(i)}^T, \frac{\partial H(1)}{\partial Z_{(i)}} \bigg|_{jk} \right\rangle. \quad \tag{48}
\]

Recall that \( H_{(i)} = \sigma(Z_{(i)}^T) \). The partial derivative is

\[

\frac{\partial H_{(i)}}{\partial Z_{(i)}} \bigg|_{jk} = E_{jk}^{L_1 \times N} \odot H_{(i)} \odot (1 - H_{(i)}), \tag{49}
\]

where \( 1 \in \mathbb{R}^{L_1} \) is the matrix of all ones. Since this fully decomposes over elements, we have

\[

\nabla_{Z_{(i)}} f(H(1)) = H_{(i)} \odot (1 - H_{(i)}) \odot \Psi_{(i)} \quad \tag{50}
\]

Finally, let \( \Omega_{(i)} \) denote \( \nabla_{Z_{(i)}} f(H(1)) \). Then,

\[

\nabla_{W_{(i)}} f(Z_{(i)}) \bigg|_{jk} = \left\langle \Omega_{(i)}^T, \frac{\partial Z_{(i)}}{\partial W_{(i)}} \bigg|_{jk} \right\rangle. \quad \tag{51}
\]

This gives us an expression for the gradients \( \nabla_{W_{(i)}} f \), which we can use to train a T-DSN block.

### Appendix B: Equivalence between the T-DSN and DSN Gradients

Under the conditions described in Section 3.1, namely, that \( W \) has dimension \( D \times 1 \) and is entry-wise zero, we show that the gradients \( \nabla_{W_{(i)}} f \) and \( \nabla_{W_{(i)}} f \) are equivalent. Let \( \tilde{\Theta} = \nabla_{X} f \) be defined analogously to (12). Using the simplified gradient, it follows that

\[

\tilde{\Theta} = 2(\hat{H}^\dagger (\hat{H}^T T) \hat{H}^\dagger) - T^T (\hat{H}^\dagger)\tag{53}
\]

Then, because each entry of \( \hat{H}(1) \) is \( 1/2 \),

\[

\tilde{\psi}_{(i)} = 2(\hat{H}_1(1) \odot E_{(i,n)}^{L_1 \times N}) \Rightarrow \tilde{\psi}_{(i)} = 1/2 \tilde{\Theta} = \Theta. \quad \tag{54}
\]

Which gives us the following gradient:

\[

\nabla_{W_{(i)}} f = \Theta (\hat{H}^T(1) \odot (1 - \hat{H}^T(1))) \tag{55}
\]

The gradients are identical under these conditions. For them to remain identical, \( W(1) \) must be clamped to zero.

### Appendix C: Complementarity of the Hidden Views

In the T-DSN, an input vector \( x \in \mathbb{R}^D \) is mapped simultaneously to two different hidden representations, \( h_1 \in \mathbb{R}^{L_1} \) and \( h_2 \in \mathbb{R}^{L_2} \), via

\[

h_1 = \sigma(W_1^T x), \quad h_2 = \sigma(W_2^T x). \quad \tag{56}
\]

In all cases considered in this paper, \( L_1 = L_2 \ll D \); so the two linear maps \( W_1^T \in \mathbb{R}^{L_1 \times D} \) have full row rank, but a \( D - L_1 \)-dimensional null space. The \( D \)-dimensional row spaces of \( W_i \), which we shall denote \( R_i \), can be used to characterize the linear map—it identifies which dimensions of the input space are preserved (and implicitly, which are destroyed). To assess the degree to which these two hidden representations capture different views of the data, we took a pair of matrices \( W_1 \) and \( W_2 \) trained on the TIMIT task (Section 5.2), where \( L_1 = L_2 = 90 \), and used two notions of
similarity of subspaces to infer whether the T-DSN does indeed push toward extracting different views of the data.

C.1 Measuring “Mass” Lost

C.1.1 Method

To measure the similarity between the two row spaces, we first consider the following score:

\[
\text{sim}(\mathbf{W}_1, \mathbf{W}_2) = \sum_{i} \sqrt{\sum_{j} (\mathbf{B}_{ij} \mathbf{B}_{ij})^2},
\]

where \( \mathbf{B}_{ij} \in \mathbb{R}^D \) denotes the \( i \)th basis vector in some basis of \( \mathcal{R}_1 \), and \( \mathbf{B}_{ij} \in \mathbb{R}^D \) likewise denotes the \( j \)th basis vector in some basis of \( \mathcal{R}_2 \). Assuming \( \mathbf{W}_i \) has singular value decomposition \( \mathbf{U}_i \mathbf{\Sigma}_i \mathbf{V}_i^T \), then in detail the process of computing \( \text{sim} \) is as follows:

1. Take the \( i \)th basis vector \( \mathbf{B}_{ij} \) and translate it into \( \mathcal{R}_2 \) via \( \mathbf{U}_j \), producing \( \mathbf{U}_j \mathbf{B}_{ij} \).
2. Take the \( \ell_2 \) norm of \( \mathbf{U}_j \mathbf{B}_{ij} \). If \( \| \mathbf{U}_j \mathbf{B}_{ij} \|_2 \) equals 1, then no mass has been lost during the change of basis; i.e., \( \mathbf{B}_{ij} \) is in the rowspace, \( \mathcal{R}_2 \). If it is equal to 0, then all mass has been lost; i.e., \( \mathbf{B}_{ij} \) is in the null space of \( \mathbf{W}_2 \).
3. \( \text{sim} \) is the sum over \( i \) of the scores computed in 2 above, aggregating the amount of “mass” that was preserved or lost in the change of basis, and thus characterizing the similarity between the two spaces.

The maximum value \( \text{sim} \) can attain is \( D \) if \( \mathcal{R}_1 = \mathcal{R}_2 \); the minimum value it can attain is 0 if the two spaces are orthogonal.

C.1.2 Results

For our pair of learned matrices, \( \mathbf{W}_1, \mathbf{W}_2 \in \mathbb{R}^{785 \times 90} \), we measured \( \text{sim} \) and obtained the result of 30.41 (out of 90 possible). We also randomly generated 10,000 pairs of orthonormal basis for 90D subspaces of \( \mathbb{R}^{785} \), and computed the corresponding \( \text{sim} \) values. On average, \( \text{sim} \) was 31.09, with a standard deviation of 0.21, putting \( \text{sim}(\mathbf{W}_1, \mathbf{W}_2) \) roughly 2.5 deviations below the mean. This suggests that the two spaces are being pushed to capture different information, but again, they are far from orthogonal.

C.2 Measuring Canonical Angles

We can also use canonical angles to assess subspace similarity.

C.2.1 Method

The cosine of the \( k \)th canonical angle can be defined recursively as the largest inner product between a basis vector of \( \mathcal{R}_1 \) and a basis vector of \( \mathcal{R}_2 \), excluding all basis vectors that have already been “used” in previous steps (e.g., that were part of a larger inner product). The cosine of canonical angles \( \theta_1, \theta_2, \ldots, \theta_D \) between two \( D \)-dimensional subspaces can be computed as the singular values of \( \mathbf{U}_1^T \mathbf{U}_2 \), where the columns of \( \mathbf{U}_i \) form a basis of \( \mathcal{R}_i \). One way to measure similarity is to sum the cosines of canonical angles; this notion of similarity too has a maximum value of \( D \) (if the spaces are equal) and a minimum value of 0 (if the spaces are orthogonal).

C.2.2 Results

For the same particular pair of estimated matrices, \( \mathbf{W}_1, \mathbf{W}_2 \in \mathbb{R}^{785 \times 90} \), we measured the sum of canonical angles and obtained the result of 26.20 (out of 90 possible). We also randomly generated 10,000 pairs of orthonormal basis for 90D subspaces of \( \mathbb{R}^{785} \), and computed the corresponding sum-of-canonical-angles values. On average, this was 26.71, with a standard deviation of 0.20, putting \( \text{sim}(\mathbf{W}_1, \mathbf{W}_2) \) roughly 2.5 deviations below the mean. This suggests that the two subspaces are being pushed to capture different information, but again, they are far from orthogonal.

C.3 Discussion

This analysis provides evidence in support of the notion that the T-DSN learns “different” views of the data. These different views capture a more diverse set of dimensions in the input that would occur by random chance, but are far from orthogonal.

ACKNOWLEDGMENTS

The authors would like to thank Karthik Mohan for his valuable feedback on the analysis of the T-DSN model.

REFERENCES

His research interests include speech and language processing, signal processing, machine learning, and pattern recognition. He has published more than 100 papers in these areas and is the inventor/co-inventor of close to 50 granted/pending patents. He is currently serving as an associate editor of the IEEE Transactions on Audio, Speech, and Language Processing (2011-) and as the lead guest editor of the IEEE Transactions on Audio, Speech, and Language Processing special issue on deep learning for speech and language processing (2010-2011). He is a senior member of the IEEE.

Brian Hutchinson received the BA degree in linguistics and the BS and MS degrees in computer science, all from Western Washington University, and the master’s degree in electrical engineering from the University of Washington, where he currently working toward the PhD degree.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.