Neural network pruning with simultaneous matrix tri-factorization and non-negative matrix factorization

Teja Roštan

Abstract

In this paper we present the Matrix Factorization-based Brain Pruning (MFBP) method for pruning neural networks. MFBP finds and prunes redundant connections by mapping connection weight matrices into a low-dimensional space. Simultaneous matrix tri-factorization is used to prune fully connected layers, and non-negative matrix factorization is used to prune the convolutional layers. We evaluated the MFBP method on two data sets. There, the reduced neural networks maintained their generalization performance, while requiring less computation.

1 Introduction

Deep neural networks (DNN) are a popular approach to machine learning that is being used to solve widely different and complex problems, such as image, sound and text recognition. However, they are computationally intensive to train and yield black-box models that are difficult to interpret. Success of neural networks largely depends on their architecture. While the size of the input layer and the output layer is given in advance, the number and size of hidden layers depends on the complexity of the problem [2].

Generally, a network with large number of hidden nodes is able to learn fast and avoids local minima, but an oversized network may overfit the training data and lose its generalization ability. Smaller networks are preferred because they usually achieve good generalization performance and are easier to interpret. Networks that are too small may be sensitive to initial conditions, choice of learning parameters and usually do not generalize well. The most popular approach to obtain the optimal architecture of a DNN is to start with a slightly larger DNN and then apply pruning. Pruning algorithms remove the redundant connections while maintaining and sometimes improving the network generalization performance [2].

Modern DNN can be extremely large, evolving many hidden layers with millions of parameters and requiring large computation and storage cost. This prohibits their usage on resource limited mobile or embedded devices, which are most suited for applications involving DNNs [8].

In this work we present a novel approach to pruning, named matrix factorization-based brain pruning (MFBP). The method is based on low-dimensional matrix factorization and is able to consider the entire structure of a neural network when pruning individual parts of the network. The method uses simultaneous matrix tri-factorization to prune the fully connected layers, and non-negative matrix factorization (NMF) to prune the convolutional layers.

Source code for the MFBP method and code for the supporting experiments is available at `https://github.com/teja-rostan/mfbp`.

2 Related work

Because there is significant redundancy in the parametrization of networks, many researchers proposed solutions to prune DNNs where initial extensive pruning is followed by backpropagation to fine-tune the pruned network and restore its generalization performance.

In convolutional neural network (CNN), about 90% of the model size is taken up by the fully connected layers and more than 90% of the running time is taken by the convolutional layers [24]. Compressing the most storage demanding fully connected layers is possible by pruning with low-rank matrix factorization (MF) methods [16, 10, 5, 20]. Optimal Brain Damage (OBD) [16] and Optimal Brain Surgeon (OBS) [10] are state-of-the-art approaches where OBS is one of the best methods for pruning. Likewise OBD, it should hold the Hessian matrix, thus requiring additional memory. Because the last fully connected layer is most parametrized, by pruning only last layer we can gain 30-50% model size reduction [20]. Method proposed by *Bondarenko* [5] can significantly simplify networks but it is not applicable on larger data sets and bigger models. Alternative approach [23] uses a singular value decomposition (SVD) as low-rank MF to reconstruct the weight matrices of the fully connected layers, reducing model size for 80% and because of fine-tuning, with negligible accuracy loss. Opposite to our approach, they did not focus on convolutional layers to reduce testing time.

Low-rank MF can also be used to reduce time complexity [25]. In [13] CNN kernel maps were approximated with a low rank basis of kernels that are separable in the spatial domain and drastically speedup computations. Alternatively, each convolutional and fully connected layer can be compressed by finding an appropriate low-rank approximation with considering several elementary tensor decompositions based on SVDs [7]. Both methods were successful at reducing model size by compressing to 90% with negligible loss, meanwhile compressing convolutional layers up to 50%.

Beside DNN pruning with MF many alternatives have been used in numerous ways to reduce model size. One of the latest studies [8] used strong vector quantization methods. A simple solution uses a constant number of simpler neurons with sparsity-inducing regularizers, which was presented in article [6]. Another method uses the significance of neurons by evaluating the information on weight variation and consequently prune the insignificant nodes on fully connected and on convolutional layers [9]. There, the first phase learns which connections are important and removes the unimportant ones using multiple iterations. They were also more successful at compressing fully connected layers comparing to convolutional layers. Because iterative pruning dramatically increased pruning performance [9], we also introduced iterative pruning in convolutional layers.

In article [1] they proposed to reduce time complexity with structured pruning and fixed point optimization. This work is the most similar to ours, as they prune by changing redundant weight values to zero. They introduced structured sparsity at various scales for CNN. With channel pruning all weights to a feature map may be pruned. With kernel pruning full kernels can be dropped. Intra-kernel pruning prunes specific weights within a kernel [1]. In our work, we focus in intra-kernel pruning.

Dropout [21] and Dropconnect [22] randomly zeroes neuron outputs and weights only during training for generalization purposes and the network architecture does not change at evaluation time. Our work drops parameters permanently, therefore, yields network with fewer parameters at test time.

3 Approximation of weights in neural network

Matrix factorization (MF) is a technique to approximate the data in low-dimensional space and to find latent features, which are a linear representation of data.

With ordinary artificial neural network, when there is only one hidden layer, the two weight matrices share the same dimension. In such cases, the two matrices can be concatenated and MF can be applied. Deep neural networks 1 have a multi-layer architecture where only neighbour weight matrices share the same dimension. We can directly apply co-dependency between two neighbouring weight matrices, but we can not apply dependency between, for example, between the first and third weight matrix. Our goal is to consider the entire structure of a DNN, including relations between layers that are not connected directly.

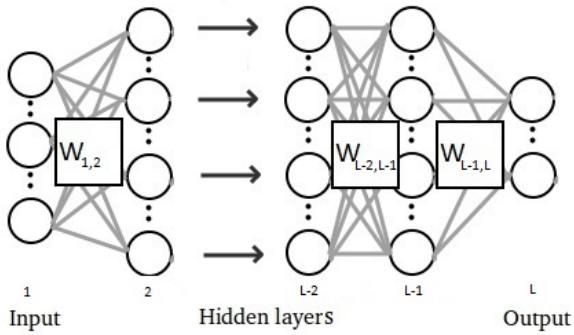

Figure 1: Deep neural network with fully connected layers. Relation matrix $W_{i,j}$ stores the weights of connections between neurons at layer i and j.

3.1 Approximation of weights in fully connected layers

We use simultaneous matrix tri-factorization for pruning in fully connected layers. The theorem of simultaneous matrix tri-factorization states that all available relation matrices W_{ij} can be simultaneously factorized into $G_i \in \Re^{m \times k}$, $G_j \in \Re^{n \times h}$ and $S_{ij} \in \Re^{k \times h}$. The factorization can be regularized through constrain matrices θ_i and θ_j, such that $W_{ij} \approx G_i S_{ij} G_j^T$ [27] 2.

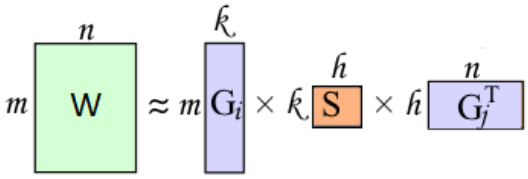

Figure 2: Graphical visualization of simultaneous matrix tri-factorization.

Figure 1 shows a neural network with hidden layers and their relation weight matrices W_{ij} between layers. The weight matrices can be placed into a grid, i.e., into a matrix of relations W, as shown in equation 1. A block in the i-th row and j-th column (W_{ij}) of matrix W represents the relationship between object type ξ_i and ξ_j. In case of a DNN, these represent neurons at layers i and j, respectively. The block matrix W is tri-factorized into block matrix factors G and S. A factorization rank k_i is assigned to ξ_i during inference of the factorized system. Factors S_{ij} define the relations between layers ξ_i and ξ_j, while factors G_i are specific to layers ξ_i and are used in the reconstruction of every relation with this layer. In this way, each weight matrix W_{ij} obtains its own factorization ${G_i S_{ij} G_j}^T$ with factor G_i (G_j) that is shared across relations that involve layers ξ_i (ξ_j). The objective function used for tri-factorization, as proposed by [27], ensures good approximation of the weight matrices.

$$W = \begin{bmatrix} W_{1,2} & & \\ & \ddots & \\ & & W_{L-2,L-1} \\ & & & W_{L-1,L} \end{bmatrix} \approx \begin{bmatrix} G_1 S_{1,2} G_2^T & & \\ & \ddots & \\ & & G_{L-2} S_{L-2,L-1} G_{L-1}^T \\ & & & G_{L-1} S_{L-1,L} G_L^T \end{bmatrix} \quad (1)$$

We can reduce the number of neurons (parameters) in network as long as the number of parameters in G_i and G_j is less than the number of parameters in W_{ij}. To reduce the number of parameters in W by a fraction of p [20], a selection of ranks k_i must be made so that the equation 2 holds.

$$m_1 k_1 + k_1 h_2 + h_2 n_2 + ... + m_{L-1} k_{L-1} + k_{L-1} h_L + h_L n_L < p(m_1 n_2 + ... + m_{L-1} n_L) \quad (2)$$

3.2 Approximation of weights in convolutional layers

For convolutional layers we used a different approach. The kernel weights in different layers are usually independent. They share feature maps with their dimensions dependent from each input image. Because of this property, we have not found a solution based on MF that considers all connections that exist in all layers concurrently. We focused on every convolutional layer separately.

We used a non-negative matrix factorization (NMF) method for approximation. NMF is a recent method for finding such a representation [12]. Given a non-negative data matrix W (kernel), NMF finds an approximate factorization $W \approx UV$ into non-negative factors U and V. The non-negativity constraints make the representation purely additive (allowing no subtractions). We used two approaches:

- In every layer we approximated every kernel separately with NMF.
- In every layer we reshaped kernels to column vectors and concatenated them into a matrix, which we used for approximation with NMF (used in pseudocode 1).

Because the kernels in network are not constrained to non-negative values, we used the NMF method from Nimfa library [26], which with preprocessing handles negative values in input matrix.

4 Factorization-based brain pruning (MFBP)

Candidates for pruning are those weights that when approximated move towards zero for a specified threshold, as given by the following formula:

$(abs(originalWeight) - abs(approximatedWeight)) >= threshold$

The threshold is set to user-specified percentile of all observed differences. Pruned weights are set to zero. The pruning procedure is defined in Algorithm 1. The code of MFBP is available on-line [19].

Data: weight matrices W of learned neural network, degree of pruning p
Result: pruned weight matrices Wp
for *every convolutional layer* **do**
  reshape kernels to column vectors;
  concatenate kernels into a matrix W_i;
  A_i := approximation of W_i with NMF;
end
for *every weight matrix* W_i *in fully connected layers* **do**
  make relations;
  add to relations graph R;
end
apply simultaneous matrix tri-factorization on relations graph R;
for *every weight matrix* W *in relations graph* R **do**
  A_i := approximations of W_i;
end
$diff\_matrix$:= absolute(W) - absolute(A);
$threshold$ = percentile p of $diff\_matrix$;
for *every approximated weight matrix* A **do**
  $Wp_i = W_i * (diff\_matrix_i \leq threshold)$;
end

Algorithm 1: Pruning neural network with matrix factorization.

5 Experimental setup

We evaluated MFBP on MNIST [15] and Cifar-10 [14] data sets.

We pruned the neural network available at github [17]. The network inclused the ReLU (Rectified linear unit) activation function. It regularizes the network model with Dropout [21]. Instead of a standard stochastic gradient descent (SGD) backpropagation method, the network uses a RMSprop [11].

To evaluate our experiments, we implemented MFBP in Python, using the Theano library [3, 4], which offers great control over neural network formation. For simultaneous matrix tri-factorization, we used the data fusion algorithm, which is available in a python library Scikit-fusion [27]. For NMF we used the algorithm available in the Nimfa library [26].

6 Results

We evaluated the predictive performance of the pruned networks with cross-validation. We used the Scikit-learn library [18] to measure the area under ROC curve (AUC).

6.1 Results on fully connected layers

We trained six neural networks: three with two hidden layers (NN_2HL) and three with four hidden layers (NN_4HL). Every neural network had 100 iterations available to learn. After learning, the simultaneous matrix tri-factorization was performed to prune weights. After pruning, 50 iterations of fine-tuning was used to refine the non-pruned weight values, which have been biased by the pruned weight. Every type of network had different amount of pruning. The reported results are measured on test set shown in figure 3.

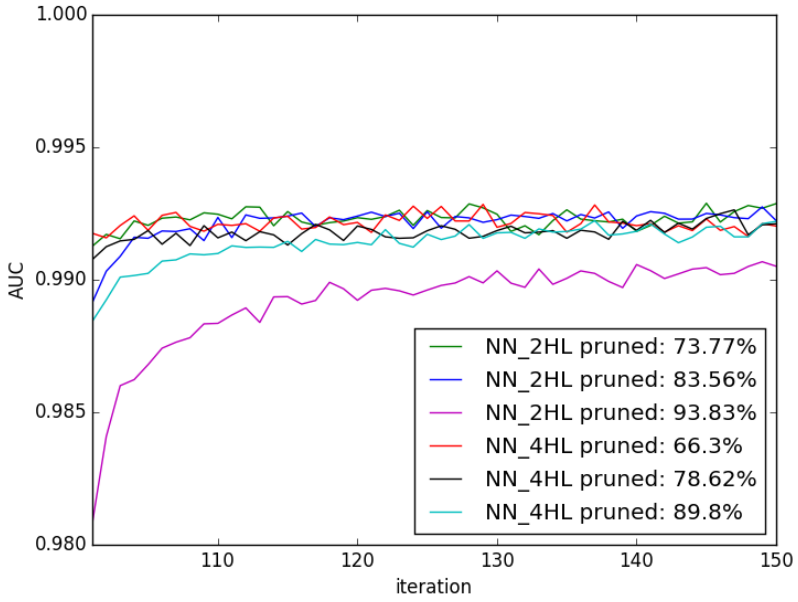

Figure 3: AUC results of six networks after pruning

We can observe that the networks with lower levels of pruning were able to recover the accuracy faster (after few iterations). In networks with higher amount of pruning, the non-pruned weights needed more iterations to recover to the accuracy from before pruning. Meanwhile, the network with two hidden layers, which was pruned the most (for 93,83 %) was not able to recover as the amount of pruning was too high. From table 1 we can see that in most cases, the pruning resulted in slightly higher accuracy than before pruning, by pruning neural network up to 80%.

6.2 Results on convolutional layers

We trained a network with three convolutional layers and one fully connected layer. Besides MNIST, we introduced the Cifar-10 data set. We used two previously mentioned approaches of kernel pruning with NMF. We also used multiple iterations of pruning, where the level of pruning increases with every pruning iteration. Iterations between pruning were reserved for recovering of weights that were kept. A recovering period lasted until five consecutive iterations where the network did not improve its performance.

	max AUC score BP	max AUC score AP	a first AUC AP >= max AUC BP
NN_2HL pruned: 73.77%	0.99272 at 72-iter	0.99289 at 145-iter	0.99275 at 112-iter
NN_2HL pruned: 83.56%	0.99291 at 88-iter	0.99275 at 149-iter	/
NN_2HL pruned: 93.83%	0.99293 at 83-iter	0.99068 at 149-iter	/
NN_4HL pruned: 66.3%	0.99236 at 97-iter	0.99284 at 129-iter	0.99241 at 104-iter
NN_4HL pruned: 78.62%	0.99236 at 78-iter	0.99284 at 147-iter	0.99249 at 146-iter
NN_4HL pruned: 89.8%	0.99201 at 99 iter	0.99223 at 137-iter	0.99208 at 128-iter

Table 1: AUC results from before pruning (BP) and after pruning (AP).

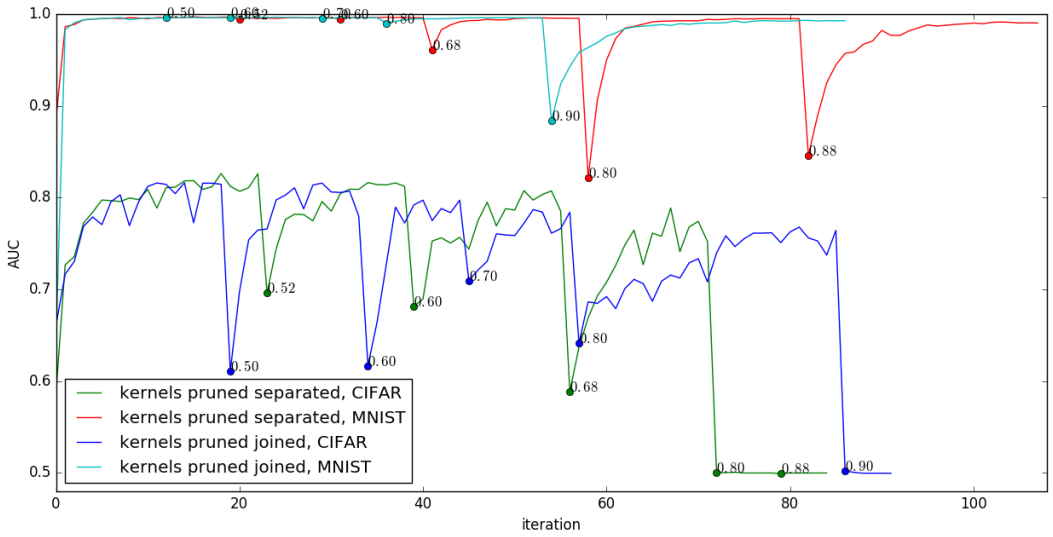

Figure 4: AUC results of two approaches on MNIST and CIFAR data set.

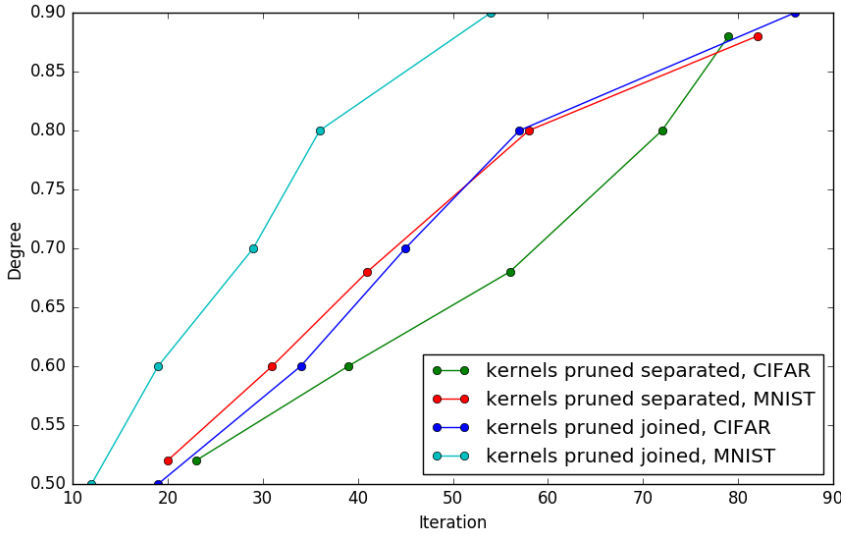

Figure 5: Degree of pruning.

Figure 4 show four AUC measures, for the two approaches and two data sets. Pruning occurs at steps indicated by coloured dots and the associated number represent the degree of pruning at that step. Figure 5 shows the degree of pruning when pruning occurs.

From Figure 4, we can see that the pruned networks were able to recover lost accuracy to some degree. With stronger pruning, the recovery was less effective. When pruning by more than 80% on the Cifar-10 data set, loss of all learned information occurred. The complete loss is related to the fact that the first convolutional layer has fewer parameters than the following layers. Because the first layer directly computes on the input data (layer), it is also more sensitive to pruning [1, 9]. In our future, we should limit the level of pruning of the first layer, and pruning should be stopped at 80%.

Overall, the proposed method *kernels pruned joined*, which prunes concatenated kernel matrices at every layer, performed better than *kernels pruned separated*, which prunes every kernel matrix separately. The *kernels pruned joined* method is thus able to consider the similarities between kernels when pruning them.

7 Discussion and conclusion

In this paper, we have addressed size complexity of DNN by applying simultaneous matrix trifactorization, which takes into account the entire structure of fully connected layers. We compressed network size considerably and with no loss in accuracy. We have addressed time complexity by applying NMF on kernels to speed up computations. We evaluated two methods for pruning kernels. One method prunes every kernel separately, while the other method concatenates kernels matrices from each layer into a matrix and prunes them simultaneously. The latter method performed better, as it can include similarities that exist between kernels in same layer. By pruning values in kernels we can exploit its computational advantages [1].

We did not compare our method with random pruning. Random pruning could be performed using Dropout [21] and permanently dropping values to reduce model size and running time. This comparison is left for future work. The question on how to simultaneously and efficiently prune all the elements of a modern neural network, including input, convolutional, pooling, fully connected, and loss layers remains open.

References

[1] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *arXiv preprint arXiv:1512.08571*, 2015.

[2] M Gethsiyal Augasta and T Kathirvalavakumar. Pruning algorithms of neural networksa comparative study. *Central European Journal of Computer Science*, 3(3):105–115, 2013.

[3] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[4] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[5] Andrey Bondarenko and Arkady Borisov. Artificial neural network generalization and simplification via pruning. *Information Technology and Management Science*, 17(1):132–137, 2014.

[6] Maxwell D Collins and Pushmeet Kohli. Memory bounded deep convolutional networks. *arXiv preprint arXiv:1412.1442*, 2014.

[7] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, pages 1269–1277, 2014.

[8] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir D. Bourdev. Compressing deep convolutional networks using vector quantization. *CoRR*, abs/1412.6115, 2014.

[9] Song Han, Jeff Pool, John Tran, and William J Dally. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626*, 2015.

[10] Babak Hassibi, David G Stork, and Gregory J Wolff. Optimal brain surgeon and general network pruning. In *Neural Networks, 1993., IEEE International Conference on*, pages 293–299. IEEE, 1993.

[11] Geoffrey Hinton. Lecture 6a overview of mini-batch gradient descent, 2014. Avaliable: `www.cs.toronto.edu/` [Reached 17.1.2016].

[12] Patrik O Hoyer. Non-negative matrix factorization with sparseness constraints. *Journal of machine learning research*, 5(Nov):1457–1469, 2004.

[13] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[14] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[15] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[16] Yann LeCun, John S Denker, Sara A Solla, Richard E Howard, and Lawrence D Jackel. Optimal brain damage. In *NIPs*, volume 89, 1989.

[17] Alec Radford (newmu). Theano-tutorials, 2015. Avaliable: `http://www.github.com` [Reached 27.1.2016].

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[19] Teja Rostan. mfbp, 2016. Avaliable: `http://www.github.com` [Reached 5.7.2016].

[20] Tara N Sainath, Brian Kingsbury, Vikas Sindhwani, Ebru Arisoy, and Bhuvana Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6655–6659. IEEE, 2013.

[21] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[22] Li Wan, Matthew Zeiler, Sixin Zhang, Yann L Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1058–1066, 2013.

[23] Jian Xue, Jinyu Li, and Yifan Gong. Restructuring of deep neural network acoustic models with singular value decomposition. In *INTERSPEECH*, pages 2365–2369, 2013.

[24] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014*, pages 818–833. Springer, 2014.

[25] Xiangyu Zhang, Jianhua Zou, Xiang Ming, Kaiming He, and Jian Sun. Efficient and accurate approximations of nonlinear convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1984–1992, 2015.

[26] Marinka Zitnik and Blaz Zupan. Nimfa: A python library for nonnegative matrix factorization. *Journal of Machine Learning Research*, 13:849–853, 2012.

[27] Marinka Zitnik and Blaz Zupan. Data fusion by matrix factorization. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 37(1):41–53, 2015.