Neural network pruning with simultaneous matrix tri-factorization

Teja Roštan

Abstract

In this paper we present an approach for pruning neural networks, which significantly reduces the model size and running time while maintaining its generalization performance. We apply a simultaneous matrix tri-factorization to map weight matrices to a low-dimensional space, therefore reducing them and partially eliminating noise. Factorized models are thus more robust and have a better generalization ability.

1 Introduction

Deep neural networks are a popular tool that is being used to solve widely different problems. They are widely used for complex or abstract problems such as image, sound and text recognition. However, they are computationally intensive to train and are known for black box problem as they will not tell you why they reached a certain conclusion. Success of neural networks largely depends on their architecture. While the size of the input layer and the output layer is known, the number of hidden layers and the number of nodes in each hidden layer 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 when a network is oversized, the network may overfit the training data and lose its generalization ability while still having unnecessary calculations as they are using more nodes than necessary. Better generalization performance can be achieved only by small networks. They are easier to interpret but their training may require a lot of effort. Also too small networks are very sensitive to initial conditions and learning parameters and do not generalize well. The most popular approach to obtain the most optimal architecture of neural network is pruning. Pruning is defined as a network trimming within the assumed initial architecture, which is larger than necessary. Pruning algorithms are used to remove the redundant connections while maintaining the networks performance. So one can use the larger networks for training and its generalization can be improved by the process of pruning [2].

More recent researches have tackled upon an issue of deep learning which is that they involve many layers with millions of parameters, making the size of the network model to be extremely large to store and with high computation cost. This prohibits the usage on resource limited hardware especially mobile devices or other embedded devices even though deep neural networks are increasingly used in applications suited for mobile devices [12].

In this work we present a novel approach using low-dimensional matrix factorization. With pruning we wanted to acknowledge all relations that exists in network. We used simultaneous matrix tri-factorization, also known as data fusion for dense layers and non-negative matrix factorization for convolutional layers. Pruning with these approaches was named matrix factorization-based brain pruning (MFBP).

2 Related work

Because there is significant redundancy in the parametrization of networks, many researchers found solutions to prune neural networks with possible accuracy loss in order to reduce the model size extensively. But were able to fine-tune the compressed layers with added learning iterations to recover the performance and improve the accuracy back. In convolutional neural network, about 90% of the model size is taken up by the dense connected layers and more than 90% of the running time is taken by the convolutional layers [32].

Compressing the most storage demanding dense connected layers is possible by neural network pruning with low-rank matrix factorization methods [5, 27, 26], where network pruning has been used both to reduce model size and to reduce over-fitting [13]. State-of-the-art approaches [20, 14] opened a rich field of studies using matrix factorization to prune the networks.

Besides neural network pruning with matrix factorization many alternatives have been used in numerous ways to optimize neural network architecture. One of the latest studies [12] used vector quantization methods for which they said have a clear gain over existing matrix factorization methods. Alternative approach [31] is application of singular value decomposition (SVD) on the weight matrices to decompose and reconstruct the model based on the sparseness of the original matrices. A simple solution to reduce the model size and preserve the generalization ability is to train models that have a constant number of simpler neurons which was presented in article [9]. Another examined strong method uses the significance of neurons by evaluating the information on weight variation and consequently prune the insignificant nodes [13]. There the first phase learns which connections are important and removes the unimportant ones using multiple iterations. Hashing is also an effective strategy for dimensionality reduction while preserving generalization performance [30, 28, 7].

Running time complexity is depended from the computation which is dominated by convolution operations in the lower layers of the model. In contrast to model size compression, fewer approaches focused on reducing the time complexity. One of the earlier approaches of reducing the time complexity is FFT algorithm [21] which by computing the Fourier transforms of the matrices in each set efficiently performs convolutions as pairwise products. However, the FFT based approach uses a significant amount of temporary memory, since the filters must be padded to be the same size as the inputs [8]. One approach is to lower the convolutions into a matrix multiplication by reshaping the filter tensor to provide performance as close as possible to matrix multiplication, while using no auxiliary memory [8]. This avoids the usage of nested loops and speeds up the computation [6]. However redundant data and kernels storage has its own cost of extra memory usage as said in article [1] where they also proposed to reduce this complexity with structured pruning and fixed point optimization.

In articles [17, 24] they use an intuition that CNN filter maps can be approximated using a low rank basis of filters that are separable in the spatial domain where in [17] substantial speedups can be achieved by also exploiting the cross-channel redundancy to perform low-rank decomposition in the channel dimension. Alternatively in article [10] they compressed each convolutional layer by finding an appropriate low-rank approximation with considering several elementary tensor decompositions based on singular value decompositions, as well as filter clustering methods to take advantage of similarities between learned features.

Method, presented in article [33], takes the nonlinear units (ReLU) into account as they minimize the reconstruction error of the nonlinear responses, subject to a low-rank constraint which helped to reduce the complexity of filters, therefore reduced computation.

3 Approximation of weights in neural network

Matrix factorization is a technique to search linear representation with factorizing. Approximation of matrix with matrix factorization is used to approximate the data in low-dimensional space in order to find latent features.

With ordinary artificial neural network, we have only one hidden layer and therefore two weight matrices with sharing dimension. Because of this property, we are able to concatenate the matrices through sharing dimension and apply a matrix factorization. With deep neural networks 1 we have a multi-layer architecture where only neighbour weight matrices share the same dimension. We can apply co-dependency between neighbour weight matrices but we can not apply dependency between, for example, first and third weight matrix. Our goal was to consider all relations that exist between weight matrices in deep neural network.

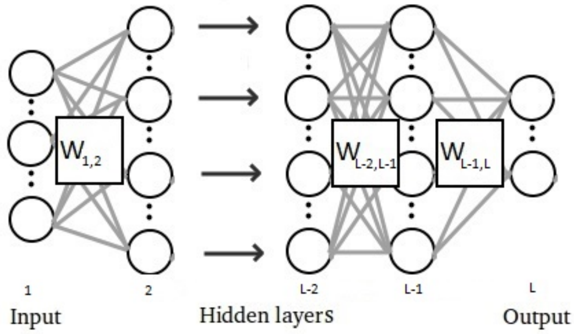

Figure 1: Deep neural network with dense connected layers. Relation matrix $W_{i,j}$ stores the weights of connections between neurons at layer i and j.

3.1 Approximation of weights in dense layers

Simultaneous matrix tri-factorization applies our criteria for pruning in dense connected layers. The theorem of simultaneous matrix tri-factorization 3.1.

Theorem 3.1 *Simultaneous tri-factorization of multiple matrices simultaneously factorize all available relation matrices W_{ij} into $G_i \in \Re^{m\times k}$, $G_j \in \Re^{n\times h}$ and $S_{ij} \in \Re^{k\times h}$ and regularize their approximations through constrained matrices θ_i and θ_j, such that $W_{ij} \approx G_i S_{ij} G_j^T$ [35] 2.*

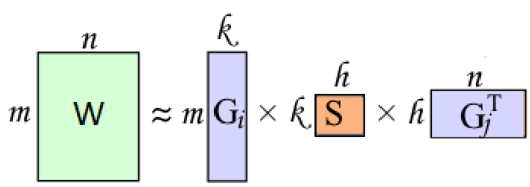

Figure 2: Graphical visualization of simultaneous matrix tri-factorization.

In a figure 1 is shown a neural network with hidden layers and their relation weight matrices W_{ij} between them. The weight matrices are collected from neural network and configured in 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 neural network, these represent neurons at layers i and j, respectively. Configuration is set on diagonal because the neighbour weight matrices share the dimension from shared hidden layer. 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 which involve layers ξ_i (ξ_j). The objective function minimized by penalized matrix tri-factorization ensures good approximation of the input data and adherence to must-link and cannot-link constraints [35].

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

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}. If we would like to reduce the number of parameters in W by a fraction of p [26], we require the equation 2 to hold.

$$m_1k_1 + k_1h_2 + h_2n_2 + ... + m_{L-1}k_{L-1} + k_{L-1}h_L + h_Ln_L < p(m_1n_2 + ... + m_{L-1}n_L) \quad (2)$$

3.2 Approximation of weights in convolutional layers

With convolutional layers we did not use the same approach. The weights (with convolutional networks more regularly called as kernels) in different layers are usually independent as they share feature maps with their dimensions dependent from each input image. Because of this property we have not found a solution based on matrix factorization that considers all connections that exists in all layers concurrently.

In turn we used a non-negative matrix factorization (NMF) method for approximation. NMF is a recent method for finding such a representation. 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), in contrast to many other linear representations [16]. 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 a NMF method from Nimfa library [34] which with preprocessing handles negative values in input matrix.

4 Factorization-based brain pruning (MFBP)

With approximations we determined which weights are better to prune. We pruned weights which hold followed criteria and were forced to a zero value to be considered as pruned:

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

The pruning procedure is defined in Algorithm 1. The code of pruning modern neural network with simultaneous matrix tri-factorization is available online [25].

Data: weight matrices W of learned neural network
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 dense 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
for *every approximated weight matrix* A **do**
    Wp_i = W_i * (absolute(W_i) - absolute(A_i) $<$ threshold);
end

Algorithm 1: Pruning neural network with matrix factorization.

5 Experimental setup

We evaluated matrix factorization-based brain pruning on MNIST [19] and Cifar-10 [18] datasets.

We used a modern neural network, presented in [22]. There are two main contributions to a modern neural network. One is changing of activation function. Instead of sigmoid function it uses a rectifier (Rectified linear unit (ReLU) $f(x) = max(0, x)$, where x is the input to a neuron. With rectifier only the input above zero activates. This activation function has been argued to be more biologically plausible [11]. It induces the sparsity in the hidden neurons and does not face gradient vanishing problem. Deep neural networks can be trained efficiently using rectifier even without pre-training. The other contribution is regularizing the model with dropout [29]. It addresses the main problem in deep learning that is overfitting. The purpose of dropout is to add some noise by dropping out a random number of some neuron activations in a given layer. With every iteration a different random set of neurons are chosen to drop, therefore it prevents co-adaptation of neurons. There was also a change of update rule. Instead of a standard stochastic gradient descent (SGD) backpropagation method we used RMSprop (A mini-batch version of rpop). The idea behind SGD is to approximate the real update step by taking the average of the all given mini batches. RMSprop keeps a running average of its recent gradient magnitudes and divides the next gradient by this average so that loosely gradient values are normalized [15].

To evaluate our experiments, we implemented algorithm on Python with the help of Theano [3, 4]. Theano is a Python library that is suitable for building an optimized neural network. We chose it as it gives a comprehensive control over neural network formation which is suitable for our problem. Another reason we used Theano is because the implementation of modern neural net described above is available at [22]. Data fusion algorithm which performs simultaneous matrix tri-factorization is available in a python library Scikit-fusion [35]. Algorithm NMF [34] was used for pruning in convolutional layers.

6 Results

To measure our results with area under ROC curve (AUC), we used a machine learning library Scikit-learn [23].

6.1 Results on dense layers

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

From results we can see that the network which had less amount of pruning were able to recover the accuracy fast (after few iterations). With higher amount of pruning, the non-pruned weights needed more iterations to recover to the accuracy 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 higher accuracy than before pruning.

6.2 Results on convolutional layers

We trained a network with three convolutional layers and one dense connected layer. We introduced also a CIFAR-10 dataset in this approach. We used two previously mentioned approaches of kernel pruning with NMF. We also used multiple iterations of pruning, where pruning gets stronger with every iteration. Iterations between pruning were reserved for recovering of weights that survived. A recovering period was not hard-coded but lasted until there were five iterations of recovering where the neurons did not improve for a small amount.

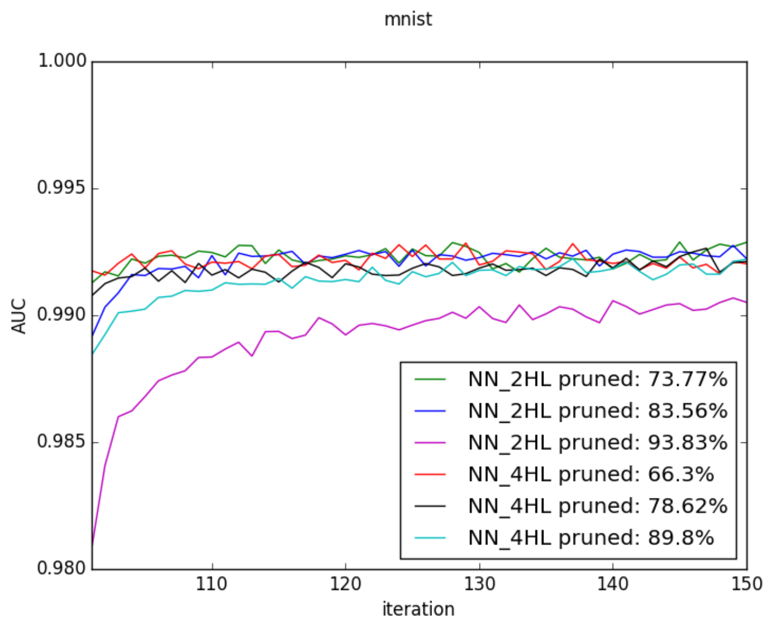

Figure 3: AUC results of six networks after pruning

	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).

The results in 4 show four AUC measures, with two approaches and two datasets. Pruning occurs at steps where there are coloured dots with a number that represents degree of pruning at that step. A figure 5 separately shows the degree of pruning when pruning occurs.

With results we came to similar conclusions. The network was able to recover lost accuracy to some amount. With stronger pruning, the recovery was less effective. When we pruned above 80 % there was a possibility that we lost all information that was learned. First, we need to consider that usually, the first convolution layer has fewer parameters than the following layers. Secondly it directly performs on the input layer and is therefore more sensitive to pruning. These reasons may affect the accuracy loss the most. In future, we might consider to protect the first layer from strong pruning.

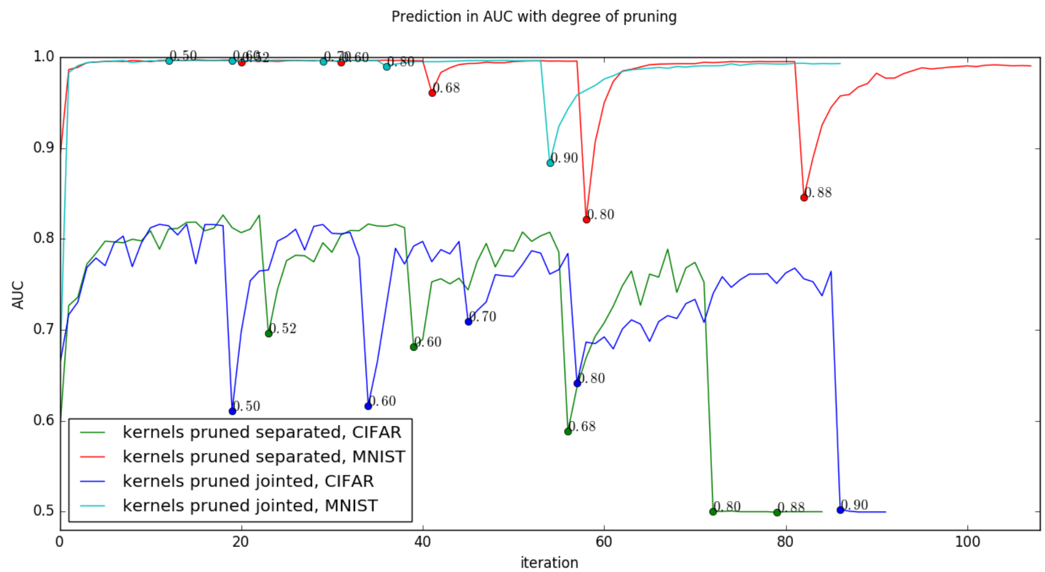

Figure 4: AUC results of two approaches on MNIST and CIFAR dataset.

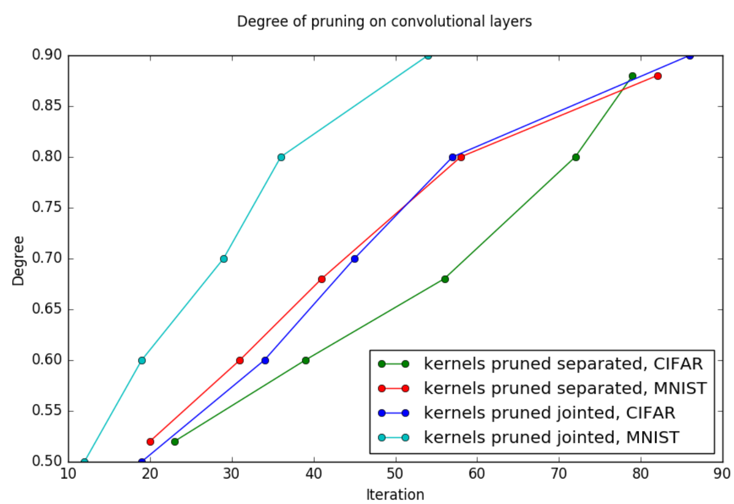

Figure 5: Degree of pruning.

7 Discussion and conclusion

In this paper, we have addressed size complexity by applying simultaneous matrix tri-factorization to compress network without loss of accuracy. We have addressed time complexity by applying NMF on kernels to speed up computations. Pruning values in kernels has the potential to bridge the gap between pruning and its computational advantages. Combined with convolution lowering, it can significantly reduce the computational cost [1].

We applied matrix factorization on weight matrices and used approximated weights to prune the weights which values moved closer to zero for the greatest amount. This allowed us to reduce the number of parameters of networks between 60-90 % without sacrificing the accuracy or sacrificing for the negligible amount.

The reduction of the parameters of neural network with higher amount of pruning required more iterations of fine-tuning to recover the non-pruned weights.

For future work, we will combine pruning on convolutional and dense connected layers on a bigger and more serious networks and datasets. We are planning on conducting experiments on evaluating its effectiveness for computational benefits.

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] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[7] Wenlin Chen, James T Wilson, Stephen Tyree, Kilian Q Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. *arXiv preprint arXiv:1504.04788*, 2015.

[8] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

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

[10] 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.

[11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey J. Gordon and David B. Dunson, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*, volume 15, pages 315–323. Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011.

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

[13] 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.

[14] 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.

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

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

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

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

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

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

[21] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

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

[23] 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.

[24] Roberto Rigamonti, Amos Sironi, Vincent Lepetit, and Pascal Fua. Learning separable filters. In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, pages 2754–2761. IEEE, 2013.

[25] Teja Rostan. osd, 2015. Avaliable: `http://www.github.com` [Reached 18.2.2016].

[26] 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.

[27] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[28] Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alex Smola, and SVN Vishwanathan. Hash kernels for structured data. *The Journal of Machine Learning Research*, 10:2615–2637, 2009.

[29] 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.

[30] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1113–1120. ACM, 2009.

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

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

[33] 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.

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

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