

---

# Exploiting Redundancies in Convolutional Networks

---

Emily Denton                      DENTON@CS.NYU.EDU  
Wojciech Zaremba                ZAREMBA@CS.NYU.EDU  
Joan Bruna                        BRUNA@CS.NYU.EDU  
Rob Fergus                        FERGUS@CS.NYU.EDU

## Abstract

### 1. Introduction

Training large neural networks (NN) takes weeks, or even months. This can hinder research, and there have been extensive effort devoted to speed up training procedure. However, resource-wise from perspective of companies executing neural networks on internet-scale data (e.g. annotating images), this is not the main cost. Major cost is in the final stage, where network is evaluated on the target data, which is present in quantities of billions. We focus here on speeding up evaluation of *trained* NN, which directly maps to the cost of executing NN on internet-scale data. Our techniques speed up execution by factor of 2–4 while keeping prediction accuracy within 1% from the original model.

We focus in this work on convolutional neural networks used for computer vision tasks. Most of computation time during evaluation is spend on convolutional layers i.e.  $\sim 90\% - 95\%$ , while it takes only the small fraction of time  $\sim 5\% - 10\%$  to evaluate rest of layers (pooling, local contrast normalization, fully connected). It is worth to note, that most of learnable parameters are kept in fully connected layers  $\sim 90\% - 95\%$ , and convolutional layers constitutes of very small fraction of parameters  $\sim 5\% - 10\%$ .

We achieve forward pass speed up by constructing approximations to the convolutional layer kernel. Convolutional kernel is a 4-dimensional tensor, with two spacial dimensions, and two feature maps-to-feature maps dimensions. Kernel of trained network has a lot of redundancies in parameters, which we exploit to

speed up forward pass, while mildly training off prediction accuracy (approximated kernels give prediction within  $\sim 1\%$  of the original prediction).

### 2. Related Work

There have been extensive research devoted speeding up forward pass of neural network. There are few different pathways, how the speed up can be achieved.

Vanhoucke et. al. (Vanhoucke et al., 2011) examined how to exploit properties of CPUs to speed up execution. They present many solutions specific to Intel and AMD CPUs, however some of their propositions are general enough to be used for any type of processor. They describe how to align memory, and use SIMD operations (vectorized operations on CPU) to get more efficient matrix multiplication. Moreover, they propose to use linearly quantize for weights, and input. Linear quantization replaces 32-bit values of weights with 8-bits (range  $[-128, 127]$ ). This approximation is similar to our approach, which replaces kernel  $W$  with few other operations, which give result approximately equal to convolution with  $W$ . Moreover, linear quantization can be used in conjunction with methods presented in this paper.

Most expensive operation in neural networks for vision is convolution. Complexity of this operation grows proportionally to the square size of local receptive field (in the direct, common implementation). However, Mathieu et. al. (Mathieu et al., 2013) shown that convolution can be efficiently computed in Fourier domain, where it becomes element-wise multiplication (there is no cost associated with size of receptive field). It is worth to note, that to operate in Fourier domain, one has to perform Fourier transform, and inverse Fourier transform to bring data back to original domain, which are operations not without cost. They report speed up in forward pass up to  $10x$  (depending on the kernel size, number of features etc.). Moreover, this method can be used jointly with most of techniques presented in this paper.

Our low-rank approximations get inspired by work of Denil et. al (Denil et al., 2013) on redundancies in neural networks. They shown that based on small fraction of weights (e.g.  $\sim 5\%$ ), one can accurately predict rest of weights. This indicates that neural networks are heavily over-parametrized. All the methods presented here focus on exploiting linear structure of this over-parametrization.

### 3. Tensor Low Rank Approximations

In this section, we give theoretical background on low rank approximations. First, we discuss simplest setting, which is for matrices (two dimensional tensors). Further, we move to approximation of 4-dimensional tensors with 2 convolutional (spacial) dimensions.

#### 3.1. Matrix Low Rank Approximation

Let's consider input  $X \in \mathbb{R}^{n \times m}$ , and matrix of weights  $W \in \mathbb{R}^{m \times k}$ . Matrix multiplication, which is the main operation for fully connected layers costs  $O(nmk)$ . However, potentially  $W$  might have a low-rank (many eigenvalues close to zero), and operation  $XW$  can be computed much faster.

Every matrix can be expressed using singular value decomposition:

$$W = USV^T, \text{ where } U \in \mathbb{R}^{n \times m}, S \in \mathbb{R}^{m \times k}, V \in \mathbb{R}^{k \times k}$$

$S$  has eigenvalues on the diagonal, and apart from it has zeros.  $W$  can be approximated by using  $t$  most significant eigenvalues from  $S$ . We can write approximation as

$$\hat{W} = \hat{U}\hat{S}\hat{V}^T, \text{ where } \hat{U} \in \mathbb{R}^{n \times t}, \hat{S} \in \mathbb{R}^{t \times t}, \hat{V} \in \mathbb{R}^{t \times k}$$

Cost of multiplication  $X$  with  $\hat{W}$  is  $O(nmt + nt^2 + ntk)$ , which can be significantly smaller than  $O(nmk)$  (for sufficiently small  $t$ ). It is worth to note, that often major cost is in memory management, and aforementioned complexities might be only relevant for large matrices.

### 4. Tensor Low Rank Approximations

This section describes a low-rank approximation of a generic convolutional layer.

Let  $W$  be a 4-dimensional tensor of dimensions  $(C, X, Y, F)$ , and let  $I(c, x, y)$  denote an input signal, where  $c = 1 \dots C$  and  $(x, y) \in \{1, \dots, N\} \times \{1, \dots, M\}$ . A generic convolutional layer is defined as

$$I * W(f, x, y) = \sum_{c=1}^C \sum_{x'=-X/2}^{X/2} \sum_{y'=-Y/2}^{Y/2} I(c, x - x', y - y') W(c, x', y', f)$$

#### 4.1. Linear Compression of Convolutional Filter bank

In typical object recognition architectures, the convolutional tensors resulting from the training exhibit

strong redundancy and regularity across all its dimensions. This redundancy affects performance since it exposes the architecture to more overfitting, and run-time speed. A particularly simple way to exploit such regularity is to linearly compress the tensors, which amounts to finding low-rank approximations.

Given a 4-tensor  $W$  of dimensions  $(C, X, Y, F)$ , we search for decompositions that minimize

$$\|W - \sum_{k \leq K} \alpha_k \otimes \beta_k \otimes \gamma_k \otimes \delta_k\|_F, \quad (1)$$

where  $\alpha_k, \beta_k, \gamma_k$  and  $\delta_k$  are rank 1 vectors of dimensions  $C, X, Y$  and  $F$  respectively, and  $\|X\|_F$  denotes the Frobenius norm. Generalization of the SVD.

The rank  $K$  approximation (3) can be obtained using a greedy algorithm, which computes for a given tensor  $X$  its best rank-1 approximation:

$$\min_{\alpha, \beta, \gamma, \delta} \|X - \alpha \otimes \beta \otimes \gamma \otimes \delta\|_F. \quad (2)$$

This problem is solved by iteratively minimizing one of the monoids while keeping the rest fixed. Each of the step consists in solving a least squares problem. (todo expand).

Figures ? and ?? show low-rank approximations of the first two convolutional layers of the Imagenet architecture.

#### 4.2. Analysis of Complexity

A good low-rank approximation allows a computational speed-up.

Let us assume a fixed stride of  $\Delta$  in each spatial dimension.

Table ?? shows the number of multiplications required to perform the convolution. In order to optimize the complexity, it is not always a good idea to decompose the full tensor  $W$ . Indeed, depending on its dimensions, the approximation cost might be superior than the original. We might consider instead low-rank approximations of  $W$  which partition the coordinate space in the most efficient manner.

#### 4.3. Optimizing Cost with Subspace Clustering

We can decompose the 4-tensor  $W$  in a collection  $W_{k,l}$  of 4-tensors, by considering a partition  $G_1, \dots, G_N$  of the first coordinate space  $C$  and a partition  $H_1, \dots, H_l, \dots, H_M$  of the last coordinate space  $F$ . If we assume a uniform partition with  $N$  groups of  $C/N$  coordinates and  $M$  groups of  $F/M$  coordinates respectively, and that each tensor  $W_{k,l}$  is approximated with

$K$  rank-1 tensors, the resulting complexity is

$$K \cdot N \cdot M \cdot \left( \frac{C}{N} + X\Delta^{-1} + Y\Delta^{-2} + \frac{F}{M}\Delta^{-2} \right)$$

How to optimize the groupings on each of the variables? We perform a subspace clustering.

Sharing between blocks:

$$\widetilde{W} = \sum_{k \leq K} \alpha_{i(k)} \otimes \beta_{j(k)} \otimes \gamma_{h(k)} \otimes \delta_{m(k)} , \quad (3)$$

If now each of the separable filters is taken out of a collection smaller than  $K$ , we can gain in computation. This can be for instance implemented with a K-Means on the tensor decompositions.

Examples: Monochromatic filtering

Spatially Separable

Memory access constraints

## 5. Numerical Experiments

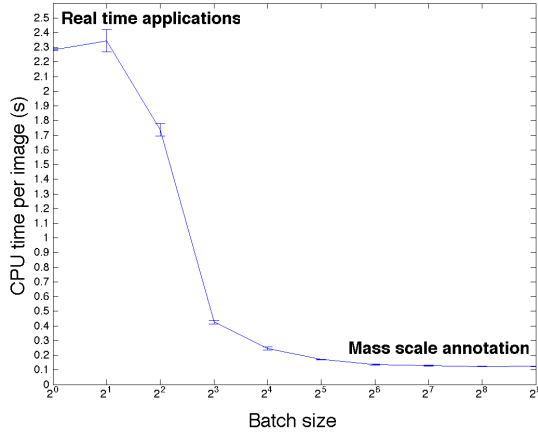


Figure 1. CPU computational time per image for various batch sizes.

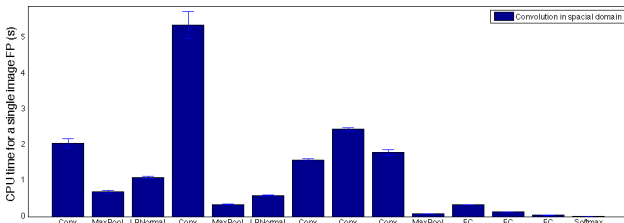


Figure 2. Per layer breakdown of execution time for mini batch of size 128. Such size of mini batch gives optimal per image CPU time.

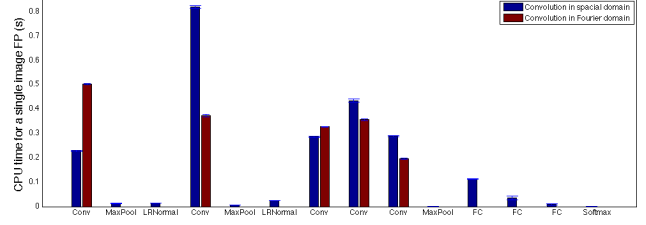


Figure 3. Per layer breakdown of execution for mini batch of size 1. Use for real time applications.

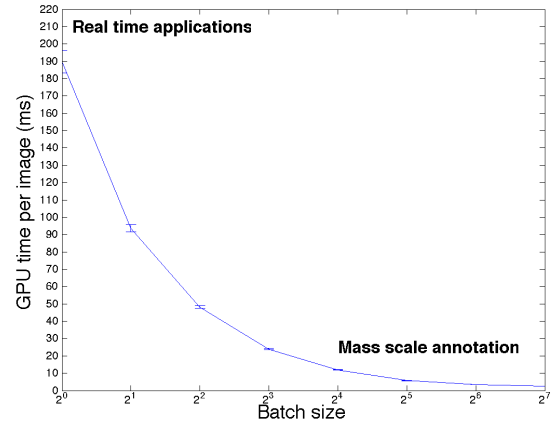


Figure 4. GPU computational time per image for various batch sizes.

### 5.1. Testing time

on GPU: Michael can help.

on CPU

#### 5.1.1. MONOCHROMATIC

#### 5.1.2. LINEAR COMBINATION OF FILTERS

#### 5.1.3. SEPARABLE FILTERS

### 5.2. Denoising

## 6. Implications

### 6.1. Denoising Aspect

we can improve training by simple linear denoting.

### 6.2. Low-Rank training

Low-rank to avoid over-fitting.

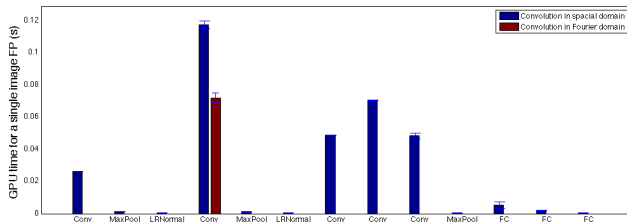


Figure 5. Per layer breakdown of execution time for mini batch of size 128. Such size of mini batch gives optimal per image GPU time.

## 7. Discussion

### References

- Denil, Misha, Shakibi, Babak, Dinh, Laurent, Ranzato, Marc'Aurelio, and de Freitas, Nando. Predicting parameters in deep learning. *arXiv preprint arXiv:1306.0543*, 2013.
- Le, Q. V., Ngiam, J., Chen, Z., Chia, D., Koh, P. W., and Ng, A. Y. Tiled convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2010.
- Le, Quoc V, Ranzato, Marc'Aurelio, Monga, Rajat, Devin, Matthieu, Chen, Kai, Corrado, Greg S, Dean, Jeff, and Ng, Andrew Y. Building high-level features using large scale unsupervised learning. *arXiv preprint arXiv:1112.6209*, 2011.
- Lowe, David G. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pp. 1150–1157. Ieee, 1999.
- Mathieu, Michael, Henaff, Mikael, and LeCun, Yann. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.
- Vanhoucke, Vincent, Senior, Andrew, and Mao, Mark Z. Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011.
- Zeiler, Matthew D, Taylor, Graham W, and Fergus, Rob. Adaptive deconvolutional networks for mid and high level feature learning. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pp. 2018–2025. IEEE, 2011.

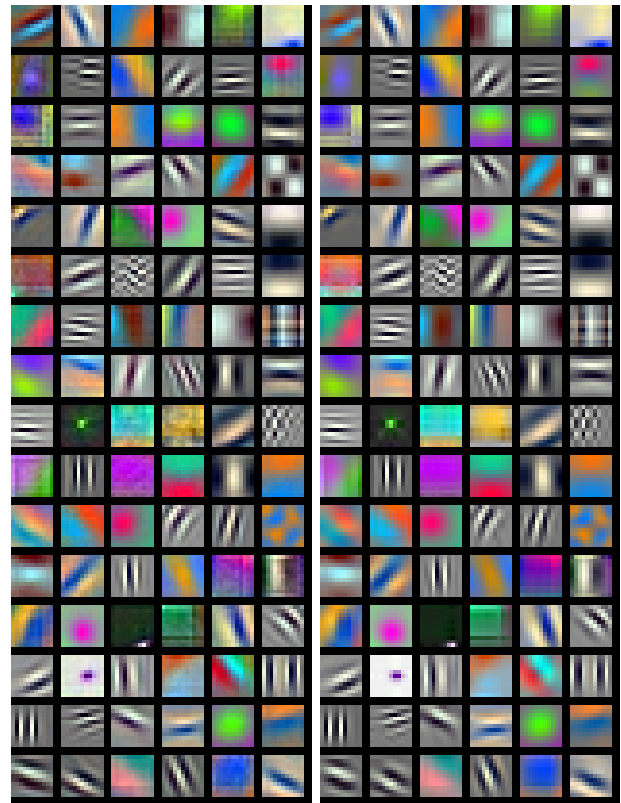


Figure 6. (Left) Original filters, (Right) approximated filters. (this pictures are too big, and should contain white separation).