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 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* neural network, 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 ($90\% - 95\%$, while it takes only small fraction of time ($5\% - 10\%$) 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 ($90\% - 95\%$) , and convolutional layers store very small fraction of parameters ($5\% - 10\%$).

We achieve evaluation speed up by constructing approximations to the convolutional kernels. Convolutional kernel is a 4-dimensional tensor, with two spacial dimensions which are convolutional. It turns out, that kernel of trained network has a lot of redundancies in parameters, which we exploit to speed up forward pass.

Proceedings of the 31st International Conference on Machine Learning, Beijing, China, 2014. JMLR: W&CP volume 32. Copyright 2014 by the author(s).

2. Related Work

Marc'Aurelio on weight redundancy.

Classical optimizations for convolutions.

Quantization.

Sparse arrays of signatures for online character recognition.

Understanding of ConvNets.

3. Tensor Low Rank Approximation (Joan writing, Wojciech Interfaces)

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

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

3.1. 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 runtime 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

$$\|W - \sum_{k \leq K} \alpha_k \otimes \beta_k \otimes \gamma_k \otimes \delta_k \|_F \ , \tag{2}$$

where α_k, β_k, γ_k and δ_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 (4) can be obtained using a greedy algorithm, which computes for a given tensor X its best rank-1 approximation:

$$\min_{\alpha, \beta, \gamma, \delta} \|X - \alpha \otimes \beta \otimes \gamma \otimes \delta\|_F \ . \tag{3}$$

Table 1. Complexity measurement of convolutional layer versus its low-rank approximation

Method	Full	K-rank approximation
Ops per pixel	$X\,Y\,C\,F\,\Delta^{-2}$	$K\cdot(C+X\Delta^{-1}+Y\Delta^{-2}+F\Delta^{-2})$

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.

3.2. Analysis of Complexity

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

Let us assume a fixed stride of Δ in each spatial dimension.

Table 3.2 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.

3.3. 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 $G_1, ..G_k, .., G_N$ of the first coordinate space C and a partition $H_1, ...H_l, ...H_M$ 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:

$$\widetilde{W}=\sum_{k\leq K}\alpha_{i(k)}\otimes\beta_{j(k)}\otimes\gamma_{h(k)}\otimes\delta_{m(k)}\ , \qquad (4)$$

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

4. Convolution in Transformed Domains

4.1. FFT (referring to Micheal results

Suppose we have to compute F spatial convolutions, with kernels $W_1,\ldots,W_F$ of maximum size M. The FFT is computed on chunks of data of size $\tilde{M}$. The resulting complexity is

$$O(F\,\tilde{M}\,\log(\tilde{M}))+O(\frac{N}{\tilde{M}-M}(\tilde{M}\,\log(\tilde{M})+F\tilde{M}\Delta^{-2}\log(\tilde{M}\Delta^{-2}))\ .$$

The value of $\tilde{M}$ is optimized as a function of N, the size of the input, and M, the size of the kernel. Figure 2 shows for each pair (N,M) the optimum value of $\tilde{M}$, and compares the resulting cost with the spatial convolution, which has a complexity of $O(F\,M\,N\,\Delta^{-2})$. For standard choices of $\Delta=2^2$ and $F=32$, and using a matlab implementation, figure 2 shows that the optimum strategy depends upon the value of parameters. For small kernels, spatial implementation is more efficient. is always to implement the convolution in the Fourier domain, but for kernels of size ≥ 8, the resulting using windowed transforms instead of the fully delocalized transform.

Figure 1. optimum value of $\tilde{M}$ as a function of N and M using default parameters $\Delta=2^2$ and $F=96$. The value 0 corresponds to the spatial implementation.

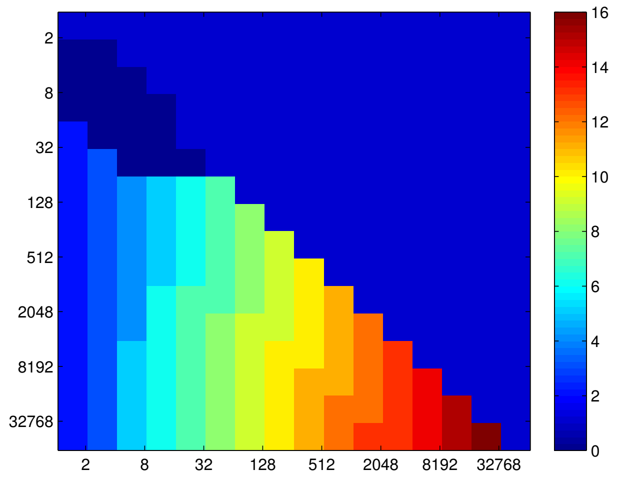
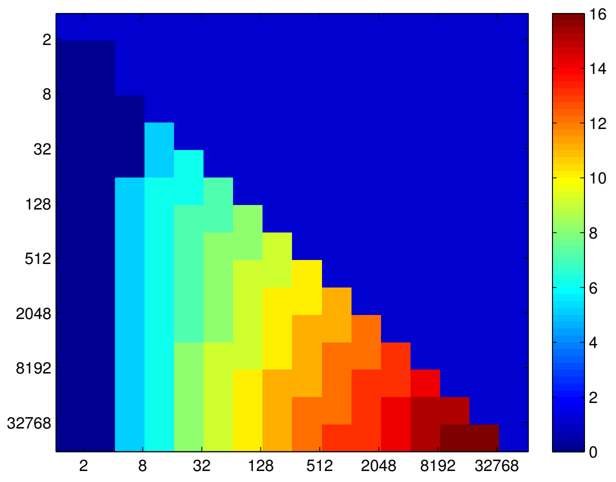


Figure 2. optimum value of $\tilde{M}$ as a function of N and M using default parameters $\Delta = 1$ and $F = 256$. The value 0 corresponds to the spatial implementation.



4.2. Multi-Resolution (Joan, reference to Wavelets and discret FFT.)

5. Numerical Experiments

5.1. Testing time

with FFT with FFT+Separable

on GPU: Michael can help.

on CPU

5.1.1. Monochromatic (Emily)

5.1.2. Linear combination of filters (Emily)

5.1.3. Separable filters (Emily)

5.2. Denoising (visual inspection, maybe meassure)

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.

7. Discussion

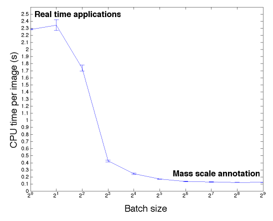


Figure 3. CPU computational time per image for various batch sizes.

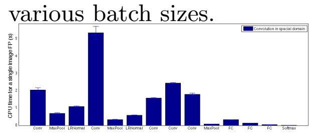


Figure 4. Per layer breakdown of execution time for mini batch of size 128. Such size of mini batch gives optimal per image CPU time.

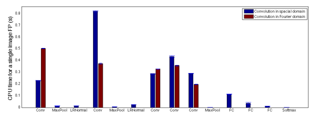


Figure 5. Per layer breakdown of execution for mini batch of size 1. Use for real time applications.

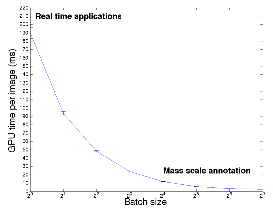


Figure 6. GPU computational time per image for various batch sizes.

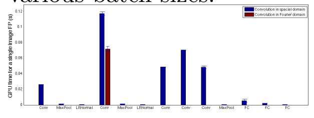


Figure 7. Per layer breakdown of execution time for mini batch of size 128. Such size of mini batch gives optimal per image GPU time.

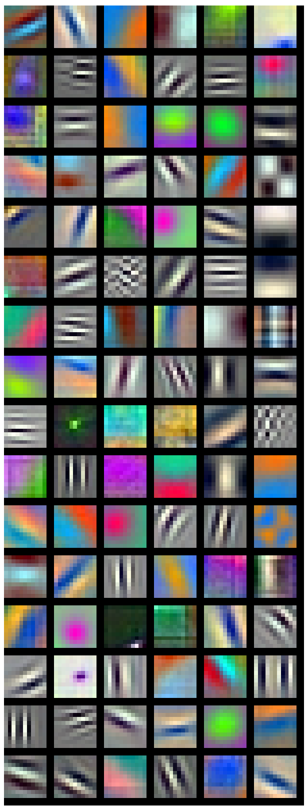


Figure 8. Original filters.

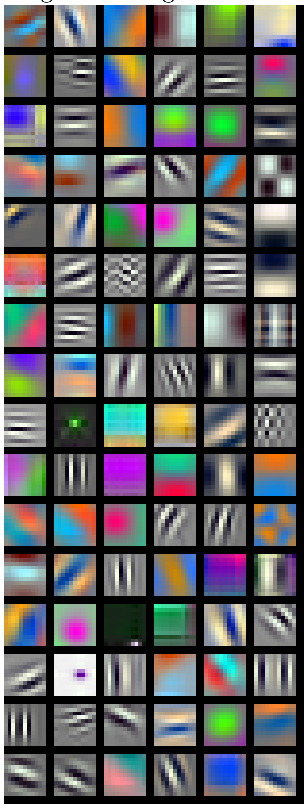


Figure 9. Approximated filters.