Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

We present techniques for speeding up the test-time evaluation of large convolutional networks, designed for object recognition tasks. These models deliver impressive accuracy but each image evaluation requires millions of floating point operations, making their deployment on smartphones and Internet-scale clusters problematic. The computation is dominated by the convolution operations in the lower layers of the model. We exploit the linear structure present within the convolutional filters to derive approximations that significantly reduce the required computation. Using large state-of-the-art models, we demonstrate speedups by a factor of $2\times$, while keeping the accuracy within 1% of the original model.

1 Introduction

Large neural networks have recently demonstrated impressive performance on a range of speech and vision tasks. However the size of these models can make their deployment at test time problematic. For example, mobile computing platforms are limited in their CPU speed, memory and battery life. At the other end of the spectrum, Internet-scale deployment of these models requires thousands of servers to process the 100's of millions of images per day. The electrical and cooling costs of these servers required is significant.

Training large neural networks can take weeks, or even months. This hinders research and consequently there have been extensive efforts devoted to speeding up training procedure. However, there are relatively few efforts aimed at improving the *test-time* performance of the models.

In this paper we focus on speeding up the evaluation of *trained* networks, with minimal compromise to performance. We consider convolutional neural networks used for computer vision tasks, since they are large and widely used in commercial applications. Within these models, most of the time ($\sim 90\%$) is spent in the convolution operations in the lower layers of the model. The remaining operations: pooling, contrast normalization and the upper fully-connected layers collectively take up the remaning 10%.

We present several novel methods for speeding up the convolution operations. They are based on various low-dimensional approximations of convolution operators (which are 4-dimensional tensors), and exploit sparsity to deliver speedups in performance. Viability of methods is architecture dependent, and speedup by factor of 2 is quite achievable without fine-tuning. However, a much larger speedup should be attainable if the models were trained for a few more epochs after imposing the filter approximation.

Neural networks are stacked linear transforms alternated with simple point-wise non-linearities. From mathematical perspective, we have a good understanding of linear operators, however properties of composed operators are much more difficult to understand. Our studies of filter compressibility reveal some of underlying low-dimensional structure. Effectively, it decreases number of parameters, and potentially might lead to better model generalization if used during training. We observe some indications that it is indeed feasible. In particular, the first layer filters look “cleaner” after approximation and can, on occasion, yield a slightly lower test error than the original versions, provided the approximations are mild.

2 Related Work

Vanhoucke *et al.* [1] explored the properties of CPUs to speed up execution. They present many solutions specific to Intel and AMD CPUs, however some of their techniques 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 boost the efficiency of matrix multiplication. Additionally, they propose the linear quantization of the network weights and input. This involves representing weights as 8-bit integers (range $[-127, 128]$), rather than 32-bit floats. This approximation is similar in spirit to our approach, but differs in that it is applied to each weight element independently. By contrast, our approximation approach models the structure within each filter. Potentially, the two approaches could be used in conjunction.

The most expensive operations in convolutional networks are the convolutions in the first few layers. The complexity of this operation is linear in the area of the receptive field of the filters, which is relatively large for these layers. However, Mathieu *et al.* [2] have shown that convolution can be efficiently computed in Fourier domain, where it becomes element-wise multiplication (and there is no cost associated with size of receptive field). They report a forward-pass speed up of around $2\times$ for convolution layers in state-of-the-art models. Importantly, the FFT method can be used jointly with most of techniques presented in this paper.

The use of low-rank approximations in our approach is inspired by work of Denil *et al.* [3] who demonstrate the redundancies in neural network parameters. They show that the weights within a layer can be accurately predicted from a small (e.g. $\sim 5\%$) subset of them. This indicates that neural networks are heavily over-parametrized. All the methods presented here focus on exploiting the linear structure of this over-parametrization.

3 Tensor Approximation Techniques

In typical object recognition architectures, the weights of convolutional layers at the end of training exhibit strong redundancy and regularity across all dimensions. In this section we describe techniques then can be used to exploit this structure to compress the 4 dimensional weight tensors into a representation that permits efficient computation.

Section 3.1 reviews techniques for low-rank tensor approximations, Section 3.2 shows how to use clustering algorithms to discover, and later exploit, patterns between input and output features.

Notation: Convolution weights can be described as a 4-dimensional tensor: $W \in \mathbb{R}^{C\times X\times Y\times F}$. C is the number of number of input channels, X and Y are the spatial dimensions of the kernel, and F is the target number of feature maps. It is common for the first convolutional layer to have a stride associated with the kernel. We denote the stride by Δ. Let $I \in \mathbb{R}^{C\times N\times M}$ denote an input signal where C is the number of input maps, and N and M are the spatial dimensions of the maps. The target value, $T = I * W$, of a generic convolutional layer, with $\Delta = 1$, for a particular output feature, f, and spatial location, (x, y), is defined as

$$T(f,x,y) = \sum_{c=1}^{C}\sum_{x'=1}^{X}\sum_{y'=1}^{Y} I(c, x+x', y+y')W(c, x', y', f)$$

Moreover, we define $W_C \in \mathbb{R}^{C\times(XYF)}$, and $W_F \in \mathbb{R}^{(CXY)\times F}$, and $W_S \in \mathbb{R}^{C\times(XY)\times F}$ to be the folded, with respect to different dimensions, versions of W.

3.1 Low-rank Approximations

A particularly simple way to exploit the regularity present in trained convolutional network weights is to linearly compress the tensors, which amounts to finding low-rank approximations. In this section we describe two different techniques for expressing matrices and tensors as a product of matrices or vectors of smaller size.

3.1.1 Singular Value Decomposition for Matrices:

Let $I \in \mathbb{R}^{n \times m}$ denote the input to a fully connected layer of a neural network and let $W \in \mathbb{R}^{m \times k}$ denote the weight matrix for the layer. Matrix multiplication, the main operation for fully connected layers, costs $O(nmk)$. However, W is likely to have a low-rank structure and thus have several eigenvalues close to zero. These dimensions can be interpreted as noise, and thus can be eliminated without harming the accuracy of the network. We now show how to exploit this low-rank structure to compute IW much faster than $O(nmk)$.

Every matrix $W \in \mathbb{R}^{m \times k}$ can be expressed using singular value decomposition:

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

S is a diagonal matrix with singular value on the diagonal, and U, V are orthogonal matrices. If the singular values of W decay rapidly, W can be well approximated by keeping only the t largest singular values from S. We can write the approximation as

$$\tilde{W} = \tilde{U}\tilde{S}\tilde{V}^\top, \text{ where } \tilde{U} \in \mathbb{R}^{m \times t}, \tilde{S} \in \mathbb{R}^{t \times t}, \tilde{V} \in \mathbb{R}^{t \times k} \tag{1}$$

The approximation error $\|I\tilde{W} - IW\|_F$ satisfies

$$\|I\tilde{W} - IW\|_F \leq S(t+1, t+1)\|I\|_F \ , \tag{2}$$

and thus is controlled by the decay along the diagonal of S. Now the computation $I\tilde{W}$ can be done in $O(nmt + nt^2 + ntk)$, which, for sufficiently small t is significantly smaller than $O(nmk)$.

3.1.2 Singular Value Decomposition for Tensors:

Any tensor can be converted to a matrix by folding all but two dimensions together. For example, let $W_C \in \mathbb{R}^{C \times (XYF)}$ be a folded version of W where the spatial and output dimensions have been folded together. Singular value decomposition of W_C can decrease number of input colors on which the convolution has to operate in exchange for an additional matrix multiplication operation. More formally,

$$W_C = USV^T \approx \tilde{U}\tilde{S}\tilde{V}^T$$

$\tilde{U}\tilde{S}$ is a matrix that transforms input colors to intermediate output. Then $\tilde{V}$ is considered as convolution operator on intermediate output space. Similarly, we can apply SVD to $W_F \in \mathbb{R}^{(CXY) \times F}$ or $\tilde{V}$.

3.1.3 Outer Product Decomposition for Tensors:

The linear approximation of matrices can be easily extended to higher order tensors. Let $v \otimes l$ denote the outer product. For a 3-tensor, $W_S \in \mathbb{R}^{C \times (XY) \times F}$, we can construct a rank 1 approximation by finding a decomposition that minimizes

$$\|W_S - \alpha \otimes \beta \otimes \gamma\|_F \ , \tag{3}$$

where $\alpha \in \mathbb{R}^C$, $\beta \in \mathbb{R}^{XY}$, $\gamma \in \mathbb{R}^F$ and $\|X\|_F$ denotes the Frobenius norm. Problem (3) is solved efficiently by performing alternate least squares on α, β and γ respectively.

This easily extends to a rank K approximation using a greedy algorithm: Given a tensor M, we compute (α, β, γ) using (3), and we update $W_S \leftarrow W_S - \alpha \otimes \beta \otimes \gamma$. Repeating this operation K times results in

$$\tilde{W_S} = \sum_{k=1}^{K} \alpha_k \otimes \beta_k \otimes \gamma_k \ . \tag{4}$$

where $\alpha \in \mathbb{R}^C$, $\beta \in \mathbb{R}^{XY}$, $\gamma \in \mathbb{R}^F$

The approximations (3) and (4) are extended to q-tensors by adding more terms in the separable approximations. As opposed to the SVD for matrices, the rank-1 tensors are not orthogonal in general.

3.2 Clustering

Another form of structure within the 4-D weight tensors that can be exploited is the similarity between different filters. We can capture this by first splitting the filters into groups and then approximating the weights within a group using a low-rank factorization method. We found it to be most efficient to independently cluster over both input and output feature channels. We start by clustering the rows of $W_C \in \mathbb{R}^{C\times(XYF)}$, which results in clusters $C_1, \ldots, C_a$. Then we cluster the columns of $W_F \in \mathbb{R}^{(CXY)\times F}$. This procedure returns clusters $F_1, \ldots, F_b$. These two operations break the original weight tensor W into ab sub-tensors $\{W_{C_i,F_j}\}_{i=1,\ldots,a,j=1,\ldots,b}$ as shown in Figure 1(b)). Each sub-tensor contains similar elements making this approach more efficient than attempting to fit a low rank approximation to all filters.

In order to efficiently exploit the parallelism inherent in CPU and GPU architectures it is useful to constrain clusters to be of equal sizes. We therefore perform the biclustering operations (or clustering for monochromatic filters 4.1) using a modified version of the k-means algorithms which balances the cluster count at each iteration It is implemented with the Floyd algorithm, by modifying the Euclidean distance with a subspace projection distance.

4 Application to Convolutional Networks

In this section we describe how to use the techniques described in Section 3 to approximate covolutional weight tensors in ways that allow for a more efficient computation of the convolution. The approximations are more efficient in two senses: both the number of floating point operations required to compute the convolution output and the number of parameters that need to be stored are dramatically reduced.

The first convolutional layer in the standard architecture receives three color channels, typically in RGB or YUV space, as input whereas later hidden layers typically receive a much larger number of feature maps that have resulted from computations performed in previous layers. As a result, the first layer weights often have a markedly different structure than the weights in later convolutional layers. We have found that different approximation techniques are well suited to the different layers. The first approach, which we call *monochromatic approximation*, can be applied to the weights in the first convolutional layer. For the remaining layers, where the number of input and output maps are large, we use the biclustering approximation, outlined in Section **??**.

We will compare the number of floating-point operations for different approximation methods. Table 1 shows a breakdown of the number of operations required for the standard convolution and for our approximations.

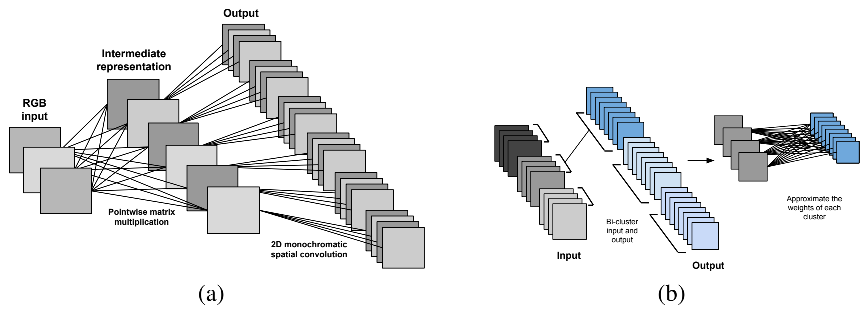
Output
Intermediate representation
RGB input
Pointwise matrix multiplication
2D monochromatic spatial convolution
(a)
Input
Bi-cluster input and output
Output
Approximate the weights of each cluster
(b)

Figure 1: A visualization of monochromatic and biclustering approximation structures. **(a)**: The monochromatic approximation, used for the first layer. Input color channels are projected by a set of intermediate color channels. After this transformation, output features need only to look at one intermediate color channel. This sparsity structure is what makes the speedups feasible. **(b)**: The biclustering approximation, used for higher convolution layers. Input and output features are clustered into equal sized groups. The weight tensor corresponding to each pair of input and output clusters is then approximated.

4.1 Monochromatic Approximation

Let $W \in \mathbb{R}^{C \times X \times Y \times F}$ denote the weights of the first convolutional layer of a trained network. The number of input channels, C, corresponds to a different color component (either RGB or YUV). We found that the color components of trained convolutional neural networks we considered (see section 5) tend to have low dimensional structure. In particular, the weights can be well approximated by projecting the color dimension down to a 1D subspace, i.e. a single color channel. The low-dimensional structure of the weights is apparent in 5.1.1. This figure shows the original first layer convolutional weights of a trained network and the weights after the color dimension has been projected into 1D lines.

The monochromatic approximation exploits this structure and is computed as follows. First, for every output feature, f, we consider consider the matrix $W_f \in \mathbb{R}^{C \times (X \cdot Y)}$, where the spatial dimensions of the filter corresponding to the output feature have been combined, and find the singular value decomposition,

$$W_f = U_f S_f V_f^\top$$

where $U_f \in \mathbb{R}^{C \times C}, S_f \in \mathbb{R}^{C \times XY}, V_f \in \mathbb{R}^{XY \times XY}$. We then take the rank 1 approximation to W_f

$$\tilde{W}_f = \tilde{U}_f \tilde{S}_f \tilde{V}_f^\top \tag{5}$$

where $\tilde{U}_f \in \mathbb{R}^{C \times 1}, \tilde{S}_f \in \mathbb{R}, \tilde{V}_f \in \mathbb{R}^{1 \times XY}$.

This approximation corresponds to shifting from C color channels to 1 color channel for each output feature. We can further exploit the regularity in the weights by sharing the color component basis between different output features. We do this by clustering the F left singular vectors, $\tilde{U}_f$, of each output feature f into C' clusters, where C' is much smaller than F. We constrain the clusters to be of equal size as discussed in section **??**. Then, for each of the $\frac{F}{C'}$ output features f that is assigned to cluster c_f, we can approximate W_f with

$$\tilde{W}_f = U_{c_f} \tilde{S}_f \tilde{V}_f^\top \tag{6}$$

where $U_{c_f} \in \mathbb{R}^{C \times 1}$ is the cluster center for cluster c_f and $\tilde{S}_f$ and $\tilde{V}_f$ as as before.

By decomposing the approximated weights into two tensors, this low-rank approximation allows for a more efficient computation of the convolutional layer output. Let $W_C \in \mathbb{R}^{C' \times C}$ denote the color transform matrix where the rows of W_C are the cluster centers $U_c^\top$. Let $W_{mono} \in \mathbb{R}^{X \times Y \times F}$ denote the monochromatic weight tensor containing $\tilde{S}_f \tilde{V}_f^\top$ for each of the F output features. Given this decomposition, we can compute the output of the convolutional layer by first transforming the input signal, $I \in \mathbb{R}^{C \times N \times M}$ into a different basis using the color transform matrix: $\tilde{I} = W_C \otimes I$ where $\tilde{I} \in \mathbb{R}^{C' \times N \times M}$.

After the color transformation (left part of the Figure 1(a)), each of the f filters in W_{mono} is monochromatic in the sense that it only acts upon one of the C' color channels (right part of the Figure 1(a)). The fact that this approximation can be made without hurting performance indicates that the structure inherent in first layer weights inherently has sparsity between the input and output maps, when projected into the new color basis. If the color transformation is computed once at the outset, then the number of operations performed is significantly reduced. Table 1 shows the number of operations required for the standard and monochromatic versions.

4.2 Biclustering Approximations

The number of input and output feature planes is large for all layers beyond the first. As described in Section **??** we cluster input and output features and then, for each input-output pair,approximate the resulting sub-tensor separately. We explore two different approximations for each sub-tensor: (i) SVD (see Section 3.1.2), and (ii) outer product decomposition (see Section 3.1.3).

Using these approximations on the convolutional weights, the target output can be computed with significantly fewer operations. The number of operations required is a function of both the number of input clusters, G, and output clusters H. Moreover, let K denote the rank of the approximated tensors for the outer product decomposition. For the SVD decomposition, let K_1 denote the input mapping rank, and let K_2 denote the output mapping rank. The two different approximation methods have the following number of operations:

Approximation technique	Number of operations
No approximation	$XYCFNM\Delta^{-2}$
Monochromatic	$C'CNM + XYFNM\Delta^{-2}$
Biclustering + outer product decomposition	$GHK(NM\frac{C}{G} + XYNM\Delta^{-2} + \frac{F}{H}NM\Delta^{-2})$
Biclustering + SVD	$GHNM(\frac{C}{G}K_1 + K_1XYK_2\Delta^{-2} + K_2\frac{F}{H})$

Table 1: Number of operations required for verious approximation methods.

4.3 Reconstruction metric

The previous sections described a series of tensor approximations that exploit the redundancy of learnt convolutional layers. Approximation equations 1, 4, 6 minimize L_2 reconstruction error, which doesn't provide any guarantee that the network using approximated weights $\widetilde{W}$ will keep the same label prediction performance as the original network.

One may ask whether there exists a better criterion to guide the approximation than the L_2 norm. We propose here a simple modification of the metric of the form

$$\|W\|_\alpha^2 := \sum_p \alpha_p^2 W(p)^2 \ , \tag{7}$$

where the sum runs over the tensor coordinates and $\alpha_p \geq 0$ are weights. Since (7) is a reweighted Euclidiean metric, we can still apply the previous approximation algorithms, by simply computing $W' = \alpha . * W$, where $.*$ denotes element-wise multiplication, then computing the approximation $\widetilde{W'}$ on W' using the standard L_2 norm, and finally outputing

$$\widetilde{W} = \alpha^{-1} . * \widetilde{W'} \ .$$

The natural question is then how to choose the weights α_p. For that purpose, we seek to emphasize coordinates more prone to produce prediction errors over coordinates whose effect is less harmful for the overall system.

We can obtain such measurements as follows. Let $\Theta = \{W_1, \dots, W_S\}$ denote the set of all parameters of the S-layer network, and let $U(I; \Theta)$ denote the output after the softmax layer of input image I. We consider a given input training set $(I_1, \dots, I_N)$ with known labels $(y_1, \dots, y_N)$. For each pair (I_n, y_n), the compute the forward propagation pass $U(I_n, \Theta)$, and define as $\{\beta_n\}$ the indices of the h largest values of $U(I_n, \Theta)$ different from y_n. Then, for a given layer s, we compute

$$d_{n,l,s} = \nabla_{W_s} \left(U(I_n, \Theta) - \delta(i - l) \right) \ , \ n \leq N \ , \ l \in \{\beta_n\} \ , \ s \leq S \ , \tag{8}$$

where $\delta(i-l)$ is the dirac distribution centered at l. In other words, for each input we back-propagate the difference between the current prediction and the h "most dangerous" mistakes. The resulting weights α_p are then obtained by computing the average energy in the tensors $d_{n,l,s}$:

$$\alpha_p = \Big(\sum_{n,l} d_{n,l,s}(p)^2 \Big)^{1/2} \ .$$

An even better metric can be obtained by considering the Mahalanobis distance defined from the covariance of d:

$$\|W\|_{maha}^2 = w \Sigma^{-1} w^T \ ,$$

where w is the vector containing all the coordinates of W, and Σ is the covariance of $(d_{n,l,s})_{n,l}$. However, we do not report results using this metric, since it requires inverting a matrix of size equal to the number of parameters, which can be prohibitively expensive in large networks.

Figure 2 compares the relationship between reconstruction error and prediction error using the unweighted and reweighted distance metrics, measured on 4096 samples of Imagenet for a range of different approximation hyperparameters. As expected, the reweighted L_2 distance correlates more strongly with performance loss. Consequently, optimizing the reweighted distance reduces the performance drop, for a given ℓ_2 reconstruction error.

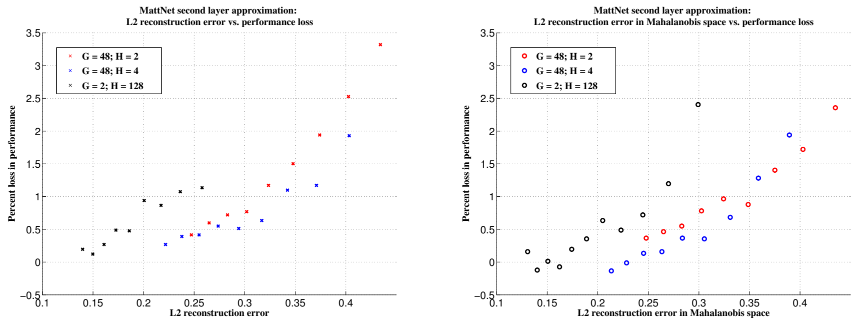

Figure 2: ℓ_2 reconstruction error of approximated weights in the original space (left) and the reweighted space (designed to match output error, see 7) (right), versus decrease in peformance for a range of different approximation hyperparameters. Markers with the same color use the same settings for G, H but vary the approximation rank. The reweighted space makes the correlation between l_2 and classification error more linear (e.g. the red circles are well approximated by a line, but no so for the red crosses). Furthermore, for a given l_2 error, the performance loss is lower for the reweighted space.

5 Experiments

We use two different convolutional architectures, each trained on the ImageNet 2012 dataset [4]: PierreNet [5], and MattNet [6]. We evaluate these networks on both CPU and GPU platforms.

We now present results showing the performance of the approximations decribed in section 4 in terms of prediction accuracy, speedup gains and reduction in memory overhead.

5.1 Speedup

Table 2 shows the time breakdown of forward propagation for each layer in MattNet. The majority of time is spent in the first and second layer convolution operations, so restrict our attention to these layers, although our approximations could easily applied to convolutions in upper layers also.

We implemented several different approximation routines for the CPU and as well as the GPU in an effort to achieve empirical speedups. Both the baseline and approximation CPU code is implemented in C++ using Eigen3 library [7] compiled with Intel MKL. We also use Intel's implementation of openmp, and multithreading. The baseline gives comparable performance to highly optimized MATLAB convolution routines and all of our CPU speedup results are computed relative to this. We used Alex Krizhevsky's CUDA convolution routines [1] as a baseline for GPU comparisons. The approximation versions are written in CUDA. All GPU code was run on a standard nVidia Titan card.

We have found that in practice it is often difficult to achieve speedups close to the theoretical gains based on the number of arithmetic operations. Moreover, different computer architectures and convolutional network architectures afford different optimization strategies making most implementations highly specific. However, regardless of implementation details, all of the approximations we present reduce both the number of operations and number of weights required to compute the output by at least a factor of two. While we present both theoretical and empirical speedup results, the empirical aspects serves largely as a proof of concept. Our aim is to illustrate that the reduction in floating point operations and memory accesses that are afforded by the approximations can in fact translate into speedups in practice. Our speedup results on both CPU and GPU are promising, especially given that our baselines for comparison comprise of highly optimized code. However, we believe that with extensive engineering efforts, speedups can be much closer to theoretically optimal levels.

5.1.1 First Layer

The first convolutional layer in MattNet has 3 input channels, 96 output channels and 7x7 filters. We approximated the weights in this layer using the monochromatic approximation described in section 4. The monochromatic approximation works well if the color components span a small number of

[1]https://code.google.com/p/cuda-convnet/

Layer	Time per batch (sec)	Fraction
Conv1	2.8317 ± 0.1030	21.97%
MaxPool	0.1059 ± 0.0154	0.82%
LRNormal	0.1918 ± 0.0162	1.49%
Conv2	4.2626 ± 0.0740	33.07%
MaxPool	0.0705 ± 0.0029	0.55%
LRNormal	0.0772 ± 0.0027	0.60%
Conv3	1.8689 ± 0.0577	14.50%
MaxPool	0.0532 ± 0.0018	0.41%
Conv4	1.5261 ± 0.0386	11.84%
Conv5	1.4222 ± 0.0416	11.03%
MaxPool	0.0102 ± 0.0006	0.08%
FC	0.3777 ± 0.0233	2.93%
FC	0.0709 ± 0.0038	0.55%
FC	0.0168 ± 0.0018	0.13%
Softmax	0.0028 ± 0.0015	0.02%
Total	12.8885	

Layer	Time per batch (sec)	Fraction
Conv1	0.0604 ± 0.0112	5.14%
MaxPool	0.0072 ± 0.0040	0.61%
LRNormal	0.0041 ± 0.0043	0.35%
Conv2	0.4663 ± 0.0072	39.68%
MaxPool	0.0032 ± 0.0000	0.27%
LRNormal	0.0015 ± 0.0003	0.13%
Conv3	0.2219 ± 0.0014	18.88%
MaxPool	0.0016 ± 0.0000	0.14%
Conv4	0.1991 ± 0.0001	16.94%
Conv5	0.1958 ± 0.0002	16.66%
MaxPool	0.0005 ± 0.0001	0.04%
FC	0.0077 ± 0.0013	0.66%
FC	0.0017 ± 0.0001	0.14%
FC	0.0007 ± 0.0002	0.06%
Softmax	0.0038 ± 0.0098	0.32%
Total	1.1752	

Table 2: Evaluation time in seconds per layer of MattNet on CPU (left) and GPU (right) with batch size of 128. Results are averaged over 8 runs.

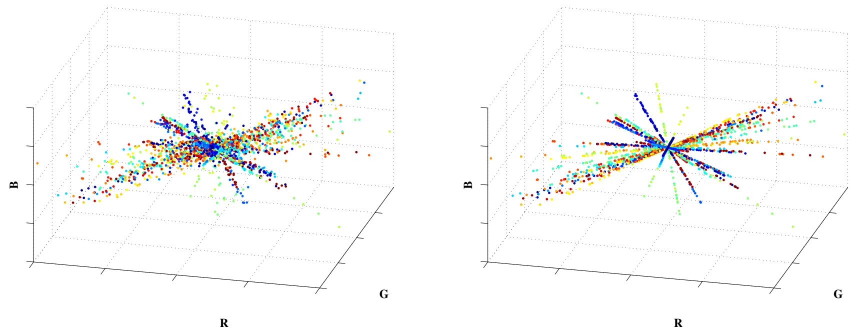

Figure 3: Visualization of the 1st layer filters in MattNet. Each component of the 96 7x7 filters is plotted in RGB space. Points are colored based on the output filter they belong to. Hence, there are 96 colors and 7^2 points of each color. (**Left**) Shows the original filters and (**Right**) shows the filters after the monochromatic approximation, where each filter has been projected down to a line in colorspace.

one dimensional subspaces. Figure 5.1.1 illustrates that by plotting the RGB components of the original filters and the filters after projecting onto one dimensional subspaces. Figure 5.1.1 gives an alternate presentation of this approximation, showing the original first layer filters beside their monochromatic approximations. Interestingly, we notice that the approximated filters often appear to be cleaned up versions of the original filters. This leads up to believe that the approximation techniques presented ere might be viable strategies of cleaning up or denoising weights after training, potentially to improve generalization performance.

We evaluated MattNet on 8K validation images from the ImageNet12 dataset using various monochromatic approximations for the first layer weights. The only parameter in the approximation is C', the number of color channels used for the intermediate representation. As expected, the network performance begins to degrade as C' decreases. Table 3 shows how the classification performance degrades for increasing C'.

Theoretically achievable speedups can measured in terms of the number of floating point operations required to compute the output of the convolution. For the monochromatic approximation this theoretical speedup with respect to MattNet is given in table 3. The majority of the operations result from the convolution part of the computation. In comparison, the number of operations required for the color transformation is negligible. Thus, the theoretically achievable speedup decreases only slightly as the number of color components used is increased.

In practice, we found it difficult to optimize the monochromatic convolution routines as for large C'. Less work can be shared amongst output filters when C' is large making it challenging to parallelize. Figure 5 shows the empirical speedups we acheived on CPU (left) and GPU (right) and the corresponding network performance for various numbers of colors used in the monochromatic approximation. Our CPU and GPU implementations achieve empirical speedups of $\sim 1.8\times$ relative to the baseline with a drop in classification performance of less than 1.2%.

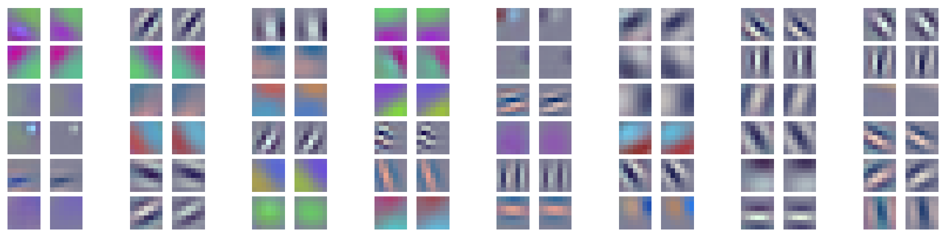
Figure 4: Original and approximate versions (using 12 colors) of 1st later filters from MattNet.

Number of colors	Increase in test error	Theoretical speedup
6	10.1929%	2.95×
8	4.0528%	2.94×
12	2.1973%	2.91×
16	1.1475%	2.88×
24	1.7212%	2.82×
48	0.5249%	2.66×
96	0.5738%	2.39×

Table 3: Performance of MattNet when first layer weights are replaced with monochromatic approximation and the corresponding theoretical speedup. Classification error on 8K validation images tends to increase as the approximation becomes harsher (i.e. fewer colors are used). Theoretical speedups vary only slightly as the number of colors used increases since the color transformation contributes relatively little to the total number of operations.

5.1.2 Second Layer

The second convolutional layer in MattNet has 96 input channels, 256 output channels and 5x5 filters. We approximated the weights using the biclustering operation, followed by the outer product decomposition approximation described in section 4. We evaluated the original and approximate convolution operation in this layer using 8K validation images from the ImageNet12 dataset. We explored various configurations of the approximations by varying the number of input clusters G, the number of output clusters H and the rank of the approximation.

We can measure the theoretically achievable speedups for a particular approximation in terms of the number of floating point operations required to compute the output of the convolution. Figure 6 plots the theoretically achievable speedups against the drop in classification performance for various configurations. For a given setting of input and output clusters numbers, the performance tends to degrade as the rank is decreased.

In practice we found it difficult to achieve speedups close to theoretically optimal levels. This is unsuprising given the amount of engineering effort that has gone into the baseline routines. However, we achieved promising results and present speedups of $\sim 1.7\times$ relative to the baseline with less than a 1% drop in performance. Figure 7 shows our empirical speedups on CPU (left) and GPU (right) and the corresponding network performance for various approximation configurations. For the CPU implementation we used the biclustering with SVD decomposition approximation. For the GPU implementation we using the biclustering with outer product decomposition approximation. While we were unable to attain speedups on the order of those theoretically achievable, we are confident that further engineering efforts should decrease the margin.

5.2 Reduction in memory overhead

In many commercial applications memory conservation and storage are a central concern. This mainly applies to embedded systems (e.g. smartphones), where available memory is limited, and users are reluctant to download large files. In these cases, being able to compress the neural network is crucial for the viability of the product.

In addition to requiring fewer operations, our approximations require significantly fewer parameters when compared to the original model. Table 4 shows the number of parameters for various approximation methods as a function of hyperparameters for the approximation techniques. Table 5 shows the empirical reduction of parameters for MattNet and the corresponding network performance.

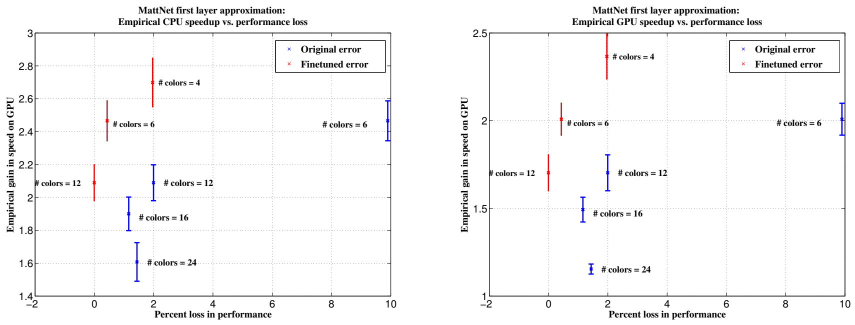

Figure 5: Empirical speedups on (**Left**) CPU and (**Right**) GPU for the first layer of MattNet.

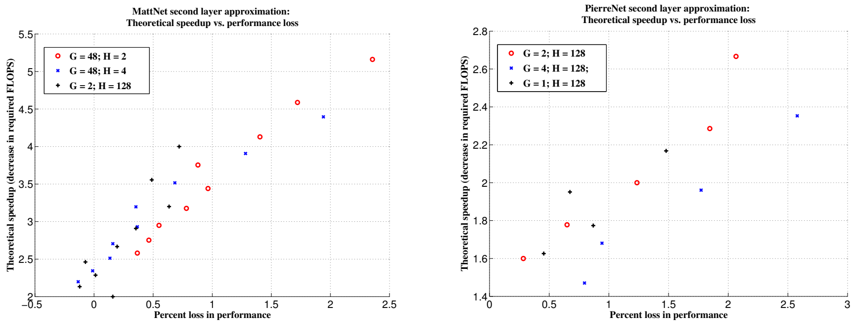

Figure 6: Theoretically achievable speedups with various biclustering approximations applied to (**Left**) MattNet and (**Right**) PierreNet.

6 Discussion

In this paper we have presented techniques that can speed up the bottleneck convolution operations in the first layers of a convolutional network by a factor $1.6 - 2$, with a moderate loss of performance ($\sim 1\%$). The empirical speedups achieved are still some way short of the theoretical gains, thus further improvements might be expected with further engineering effort. Moreover, the techniques are orthogonal to other approaches for efficient evaluation, such as quantization or working in the Fourier domain. Hence, they can potentially be used together to obtain further gains.

Given the widespread use of neural networks in large scale industrial settings, the speed gains we demonstrate have significant economic value. Companies such as Google or Facebook process 10^8 images/day, requiring many thousands of machines. Our speed gains mean that the number of machines dedicated to running convolutional networks could be roughly halved, so saving millions of dollars, as well as a significant reduction in environmental impact.

We also show that our methods reduce the memory footprint of weights in the first two layers by factor of $2-3\times$. When applied to the whole network, this would translate into a significant savings, which would facilitate mobile deployment of convolutional networks.

Our approximations are applied to fully trained networks. The small performance drops that result could well be mitigated by further training the network after applying the approximation. By alternating these two steps, we could potentially achieve further gains in speed-up for a modest performance drop.

Another aspect of our technique is regularization. It seems that approximated filters look cleaner, and that sporadically we get better test error (e.g. bottom left of Figure 6(left)). We would like to experiment with low-rank projections during training as a regularization technique. Effectively it decreases number of learnable parameters, so it might improve generalization, a major issue with large convolutional networks.

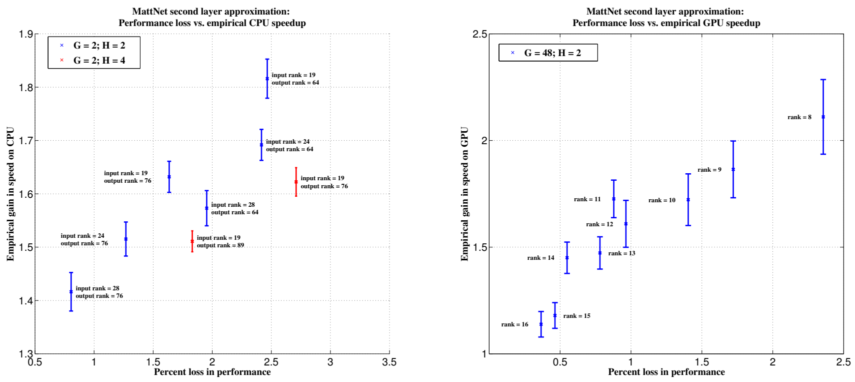

Figure 7: Empirical speedups on (**Left**) CPU and (**Right**) GPU for the second layer of MattNet.

Approximation method	Number of parameters
No approximation	$C \cdot X \cdot Y \cdot F$
Monochromatic	$C \cdot C' + X \cdot Y \cdot F$
Biclustering + outer product decomposition	$G \cdot H \cdot K(\frac{C}{G} + X \cdot Y + \frac{F}{H})$
Biclustering + singular value decomposition	$G \cdot H \cdot (\frac{C}{G} \cdot K_1 + K_1 \cdot X \cdot Y \cdot K_2 + K_2 \cdot \frac{F}{H})$

Table 4: Number of parameters expressed as a function of hyperparameters for various approximation methods.

Acknowledgments

References

[1] Vanhoucke, V., Senior, A., Mao, M.Z.: Improving the speed of neural networks on cpus. In: Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop. (2011)

[2] Mathieu, M., Henaff, M., LeCun, Y.: Fast training of convolutional networks through ffts. arXiv preprint arXiv:1312.5851 (2013)

[3] Denil, M., Shakibi, B., Dinh, L., Ranzato, M., de Freitas, N.: Predicting parameters in deep learning. arXiv preprint arXiv:1306.0543 (2013)

[4] Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: ImageNet: A Large-Scale Hierarchical Image Database. In: CVPR09. (2009)

[5] Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., LeCun, Y.: Overfeat: Integrated recognition, localization and detection using convolutional networks. arXiv preprint arXiv:1312.6229 (2013)

[6] Zeiler, M.D., Fergus, R.: Visualizing and understanding convolutional neural networks. arXiv preprint arXiv:1311.2901 (2013)

[7] Guennebaud, G., Jacob, B., et al.: Eigen v3. http://eigen.tuxfamily.org (2010)

[8] Zeiler, M.D., Taylor, G.W., Fergus, R.: Adaptive deconvolutional networks for mid and high level feature learning. In: Computer Vision (ICCV), 2011 IEEE International Conference on, IEEE (2011) 2018–2025

[9] Le, Q.V., Ngiam, J., Chen, Z., Chia, D., Koh, P.W., Ng, A.Y.: Tiled convolutional neural networks. In: Advances in Neural Information Processing Systems. (2010)

[10] Le, Q.V., Ranzato, M., Monga, R., Devin, M., Chen, K., Corrado, G.S., Dean, J., Ng, A.Y.: Building high-level features using large scale unsupervised learning. arXiv preprint arXiv:1112.6209 (2011)

[11] Lowe, D.G.: Object recognition from local scale-invariant features. In: Computer vision, 1999. The proceedings of the seventh IEEE international conference on. Volume 2., Ieee (1999) 1150–1157

[12] Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.R.: Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv:1207.0580 (2012)

Approximation method	Approximation hyperparameters	Reduction in weights	Increase in test error
Layer 1: Monochromatic	$C' = 48$	2.9109×	0.5249%
Layer 2: Biclustering +outer product decomposition	$G = 48$ $H = 2$ $K = 11$	3.7537×	0.8789%
Layer 2: Biclustering + singular value decomposition	$G = 2$ $H = 2$ $K_1 = 28$ $K_2 = 76$	2.0386×	0.8057%

Table 5: Empirical reduction in parameters for first two layers of MattNet with corresponding network performance.

[13] Krizhevsky, A., Sutskever, I., Hinton, G.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems 25. (2012) 1106–1114