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 redundancy present within the convolutional filters to derive approximations that significantly reduce the required computation. Using large state-of-the-art models, we demonstrate speedups on both CPU and GPU 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.

We consider convolutional neural networks (CNNs) used for computer vision tasks, since they are large and widely used in commercial applications. These networks typically require a huge number of parameters ($\sim 10^7$ in [**?**]) to produce state-of-the-art results. This redundancy seems necessary in order to overcome a highly non-convex optimization [1, **?**], but as a byproduct the resulting network wastes computing resources. In this paper we show that this redundancy can be exploited with linear compression techniques, resulting in significant speedups for the evaluation of *trained* large scale networks, with minimal compromise to performance. In particular, we concentrate in the lower convolutional layers, which typically dominate the evaluation cost.

We follow a relatively simple strategy: we start by compressing each convolutional layer by finding an appropriate low-rank approximation, and then we fine-tune the upper layers until the prediction performance is restored. We consider several elementary tensor decompositions based on singular value decompositions, as well as vector quantization to also exploit nonlinear approximation.

In summary, our main contributions are the following: We present a collection of generic methods to exploit the redundancy inherent in deep CNNs. We report experiments on state-of-the-art Imagenet CNNs, showing $2-3\times$ empirical speedups on convolutional layers with less than 1% drop in performance and a reduction of parameters in fully connected layers by a factor of $XXX\times$.

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 which we denote 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

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

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.

If X is a tensor, $\|X\|$ denotes its operator norm, and $\|X\|_F$ denotes its Frobenius norm. If v is a vector, $\|v\|$ denotes its Euclidean norm.

2 Related Work

Vanhoucke *et al.* [2] 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 CNNs 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.* [3] 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.* [1] 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.

Finally, a recent preprint [**?**] also exploits low-rank decompositions of convolutional tensors to speed up the evaluation of CNNs, applied to scene text character recognition. This work was developed simultaneously with ours, and provides further evidence that such techniques can be applied to a variety of architectures and tasks.

3 Convolutional Tensor Compression

In this section we describe techniques for compressing 4 dimensional convolutional weight tensors and fully connected weight matrices into a representation that permits efficient computation and storage. Section 3.1 describes how to construct a good approximation criteria. Section 3.2 describes techniques for low-rank tensor approximations. Sections 3.3 and 3.4 describe how to apply these techniques to approximate weights of a convolutional neural network.

3.1 Approximation Metric

The goal is to find an approximation $\tilde{W}$ of a convolutional tensor W that facilitates more efficient computation while maintaining the prediction performance of the network. A natural choice for

an approximation criterion is to minimize $\|\tilde{W} - W\|_F$. This criterion yields efficient compression schemes using elementary linear algebra, and also controls the operator norm of each linear convolutional layer. However, this criterion assumes that all directions in the space of weights equally affect prediction performance. We now present two methods of improving this criterion while keeping the same efficient approximation algorithms.

Mahalanobis distance metric: The first distance metric we propose seeks 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 $\Theta = \{W_1, \ldots, W_S\}$ denote Let $\Theta = \{W_1, \ldots, W_S\}$ 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 $(I_1, \ldots, I_N)$ with known labels $(y_1, \ldots, y_N)$. 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

$$d_{n,l,s} = \nabla_{W_s} \left(U(I_n, \Theta) - \delta(i - l) \right) \ , \ n \leq N \ , \ l \in \{\beta_n\} \ , \ s \leq S \ , \tag{1}$$

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 Mahalanobis distance is defined from the covariance of d: $\|W\|^2_{maha} = w\Sigma^{-1}w^T$, where w is the vector containing all the coordinates of W, and Σ is the covariance of $(d_{n,l,s})_{n,l}$. 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. Instead we use an approximation that considers only the diagonal of the covariance matrix. In particular, we propose the following, approximate, Mahalanobis distance metric:

$$\|W\|_{\widetilde{maha}} := \sum_p \alpha_p W(p) \ , \text{ where } \alpha_p = \Big(\sum_{n,l} d_{n,l,s}(p)^2 \Big)^{1/2} \tag{2}$$

where the sum runs over the tensor coordinates. Since (2) is a reweighted Euclidiean metric, we can simply compute $W' = \alpha \ . * W$, where $.*$ denotes element-wise multiplication, then compute the approximation $\widetilde{W'}$ on W' using the standard L_2 norm, and finally output $\widetilde{W} = \alpha^{-1} . * \widetilde{W'}$.

Data covariance distance metric: One can view the Frobenius norm of W as $\|W\|^2_F = \mathbb{E}_{x \sim \mathcal{N}(0,I)} \|Wx\|^2_F$. Another alternative, which has also been considered in [**?**], is to replace the isotropic covariance assumption by the empirical covariance of the input of the layer. If $W \in \mathbb{R}^{C \times X \times Y \times F}$ is a convolutional layer, and $\widehat{\Sigma} \in \mathbb{R}^{CXY \times CXY}$ is the empirical estimate of the input data covariance, it can be efficiently computed as

$$\|W\|_{data} = \|\widehat{\Sigma}^{1/2} W_F\|_F \ , \tag{3}$$

where W_F is the matrix obtained by folding the first three dimensions of W.

3.2 Low-rank Tensor Approximations

A particularly simple strategy to exploit the redundancy present in trained convolutional network weights is to linearly compress the tensors, which amounts to finding low-rank approximations.

3.2.1 Matrix Decomposition

Matrices are 2-tensors which can be linearly compressed using the Singular Value Decomposition. If $W \in \mathbb{R}^{m \times k}$ is a real matrix, the SVD is defined as $W = USV^\top$, 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 the singular values 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 entries of S, resulting in the approximation $\tilde{W} = \tilde{U}\tilde{S}\tilde{V}^\top$, where $\tilde{U} \in \mathbb{R}^{m \times t}, \tilde{S} \in \mathbb{R}^{t \times t}, \tilde{V} \in \mathbb{R}^{t \times k}$ Then, for $I \in \mathbb{R}^{n \times m}$, the approximation error $\|I\tilde{W} - IW\|_F$ satisfies $\|I\tilde{W} - IW\|_F \leq s_{t+1}\|I\|_F$, 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 $O(nmk)$.

3.2.2 Higher Order Tensor Approximations

SVD can be used to approximate a tensor $W \in \mathbb{R}^{m \times n \times k}$ by first folding all but two dimensions together to convert it into a 2-tensor, and then considering the SVD of the resulting matrix. For example, we can approximate $W_m \in \mathbb{R}^{m \times (nk)}$ as $\tilde{W}_m \approx \tilde{U}\tilde{S}\tilde{V}^\top$. W can be compressed even further by applying SVD to $\tilde{V}$. We refer to this approximation as the SVD decomposition and use K_1 and K_2 to denote the rank used in the first and second application of SVD respectively.

Alternatively, we can approximate a 3-tensor, $W_S \in \mathbb{R}^{m \times n \times k}$, by a rank 1 3-tensor by finding a decomposition that minimizes

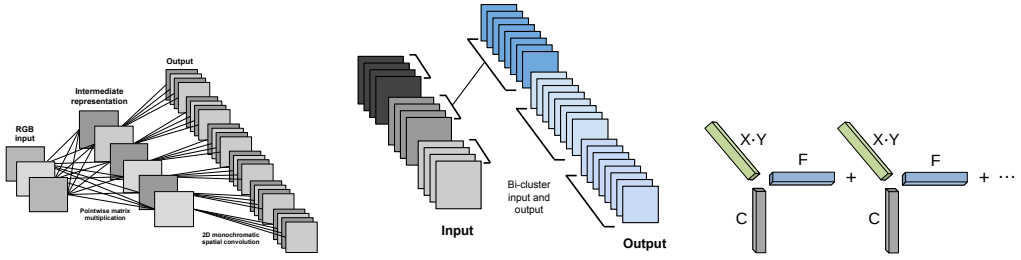
$$\|W - \alpha \otimes \beta \otimes \gamma\|_F \ , \tag{4}$$

where $\alpha \in \mathbb{R}^m$, $\beta \in \mathbb{R}^n$, $\gamma \in \mathbb{R}^k$ and $\otimes$ denotes the outer product operation. Problem (4) is solved efficiently by performing alternate least squares on α, β and γ respectively, although more efficient algorithms can also be considered [**?**].

This easily extends to a rank K approximation using a greedy algorithm: Given a tensor M, we compute (α, β, γ) using (4), and we update $W_S \leftarrow W_S - \alpha \otimes \beta \otimes \gamma$. Repeating this operation K times results in

$$\tilde{W}_S = \sum_{k=1}^{K} \alpha_k \otimes \beta_k \otimes \gamma_k \ . \tag{5}$$

We refer to this approximation as the outer product decomposition ans use K to denote the rank of the approximation.



(a) (b) (c)

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. **(c)** The weight tensors for each input-output pair in (b) is approximated by a sum of rank 1 tensors using techniques described in 3.2.2

3.3 Monochromatic Convolution 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 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 4.1.1 which shows the original first layer convolutional weights 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 $W_f \in \mathbb{R}^{C \times (XY)}$, where the spatial dimensions of the filter corresponding to the output feature have been combined, and find the SVD, $W_f = U_f S_f V_f^\top$, where $U_f \in \mathbb{R}^{C \times C}$, $S_f \in \mathbb{R}^{C \times XY}$, and $V_f \in \mathbb{R}^{XY \times XY}$. We then take the rank 1 approximation of W_f:

$$\tilde{W}_f = \tilde{U}_f \tilde{S}_f \tilde{V}_f^\top \ , \tag{6}$$

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 various approximation methods.

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 using a single 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 3.4. Then, for each of the $\frac{F}{C'}$ output features f that is assigned to cluster c_f, we can approximate W_f with

$$\tilde{W}_f = U_{c_f}\tilde{S}_f\tilde{V}_f^\top \tag{7}$$

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.

This monochromatic approximation is illustrated in the left panel of Figure 1(c). Table 1 shows the number of operations required for the standard and monochromatic versions.

3.4 Biclustering Approximations

We exploit the redundancy within the 4-D weight tensors in the higher convolutional layers by clustering the filters, such that each cluster can be accurately approximated by a low-rank factorization. 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 $W_F \in \mathbb{R}^{(CXY)\times F}$, producing 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, thus is be easier to fit with a low-rank approximation.

In order to 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 3.3) using a modified version of the k-means algorithm 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.

After the input and output clusters have been obtained, we find a low-rank approximation of each sub-tensor using either the SVD decomposition or the outer product decomposition as described in 3.2.2 Using these approximations, the target output can be computed with significantly fewer operations. The number of operations required is a function the number of input clusters, G, the output clusters H and the rank of the sub-tensor approximations (K_1, K_2 for the SVD decomposition; K for the outer product decomposition. The number of operations required for each approximation is described in table 1.

4 Experiments

We use the convolutional architecture of [4], trained on the ImageNet 2012 dataset [5]. We evaluate these networks on both CPU and GPU platforms.

We present results showing the performance of the approximations described in section **??** in terms of prediction accuracy, speedup gains and reduction in memory overhead.

4.1 Speedup

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

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 on CPU (left) and GPU (right) with batch size of 128. Results are averaged over 8 runs.

We implemented several different CPU and GPU approximation routines in an effort to achieve empirical speedups. Both the baseline and approximation CPU code is implemented in C++ using Eigen3 library [6] 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.

4.1.1 First Layer

The first convolutional layer has 3 input channels, 96 output channels and 7x7 filters. We approximated the weights in this layer using the monochromatic approximation described in section **??**. The monochromatic approximation works well if the color components span a small number of one dimensional subspaces. Figure 4.1.1 illustrates that by plotting the RGB components of the original filters and the filters after projecting onto one dimensional subspaces. Figure 4.1.1 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 here might be viable strategies of cleaning up or denoising weights after training, potentially improving generalization performance.

We evaluated the network on 20K 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.

The number of FLOPS required to compute the output of the monochromatic convolution is reduced by a factor of $2-3\times$, with the larger gain resulting for small C'. 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 4 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

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

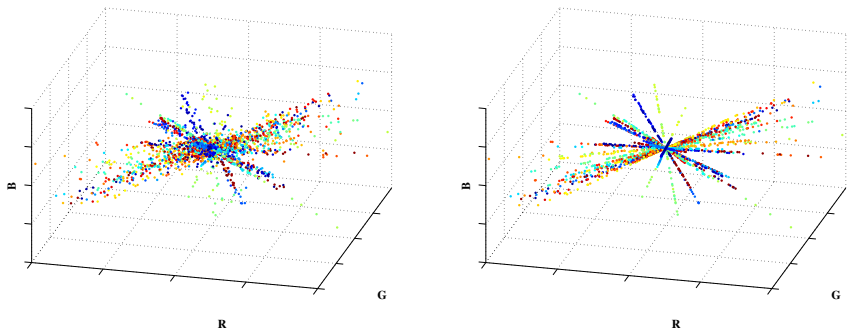


Figure 2: Visualization of the 1st layer filters in MattNet. Each component of the 96 7x7 filters is plotted in RGB space. Points are colored based on the output filter they belong to. Hence, there are 96 colors and 7^2 points of each color. (**Left**) Shows the original filters and (**Right**) shows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

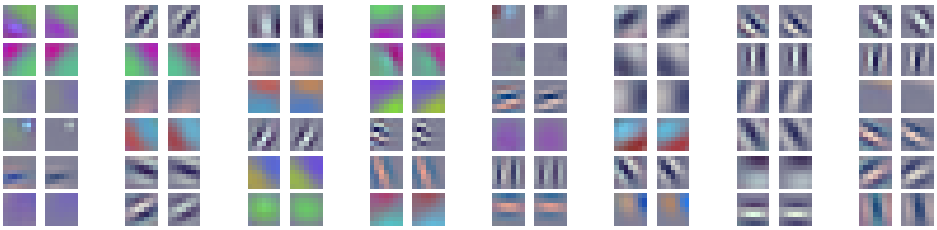


Figure 3: Original and approximate versions (using 12 colors) of 1st later filters from MattNet.

and GPU implementations achieve empirical speedups of $2 - 2.5\times$ relative to the baseline with less than 1% drop in classification performance.

4.1.2 Second Layer

The second convolutional layer has 96 input channels, 256 output channels and 5x5 filters. We approximated the weights using the techniques described in section 3.4. outer product decomposition described in section **??**. We evaluated the original and approximate convolution operation in this layer using 20K 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 (denoted by $K1$ and $K2$ for the SVD decomposition and K for the outer product decomposition).

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 5 plots the theoretically achievable speedups against the drop in classification performance for various configurations of the biclustering with outer product decomposition technique. 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. However, we achieved promising results and present speedups of $2 - 2.5\times$ relative to the baseline with less than a 1% drop in performance. Figure 6 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.

4.2 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.

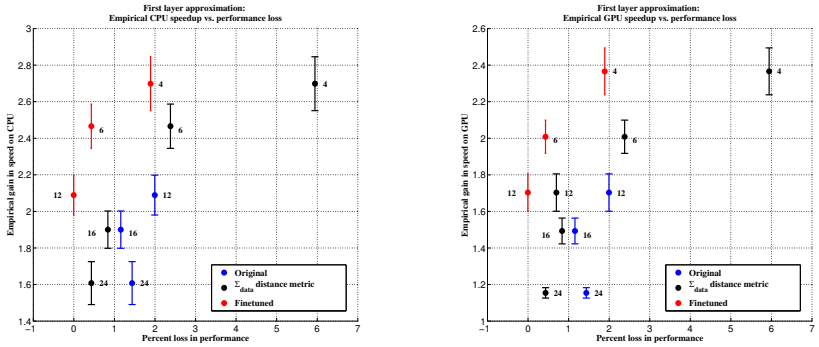


Figure 4: Empirical speedups on (**Left**) CPU and (**Right**) GPU for the first layer. The number beside each point indicates the number of colors, C', used in the approximation.

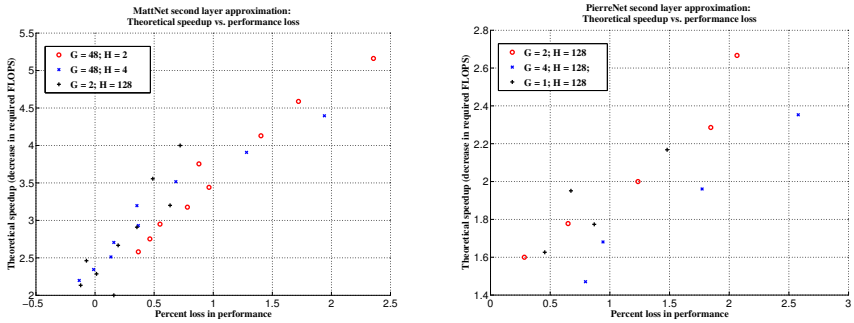


Figure 5: Theoretically achievable speedups with various biclustering approximations applied to (**Left**) MattNet and (**Right**) PierreNet.

In addition to requiring fewer operations, our approximations require significantly fewer parameters when compared to the original model. Since the majority of parameters come from the fully connected layers, we include these layers in our analysis of memory overhead. Table **??** shows the number of parameters for various approximation methods as a function of hyperparameters for the approximation techniques. Table 3 shows the empirical reduction of parameters and the corresponding network performance.

5 Discussion

In this paper we have presented techniques that can speed up the bottleneck convolution operations in the first layers of a CNN by a factor $2-3\times$, with negligible loss of performance ($< 1\%$). 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

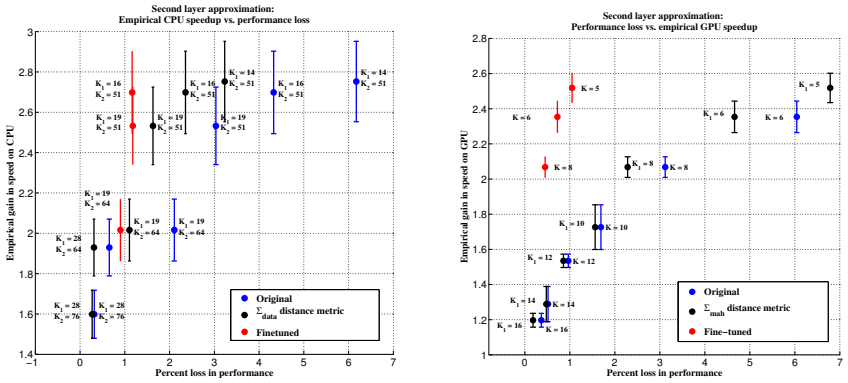


Figure 6: Empirical speedups for second convolutional layer. (**Left**) Speedups on CPU using biclustered ($G = 2$ and $h = 2$) with SVD approximation.(**Right**) peedups on GPU using biclustered ($G = 48$ and $h = 2$) with outer product decomposition approximation.

Approximation method	**Number of parameters**	**Approximation hyperparameters**	**Reduction in weights**	**Increase in test error**
No approximation	$C \cdot X \cdot Y \cdot F$			
Layer 1: Monochromatic	$C \cdot C' + X \cdot Y \cdot F$	$C' = 48$	2.9109×	0.5249%
Layer 2: Biclustering + outer product decomposition	$G \cdot H \cdot K(\frac{C}{G} + X \cdot Y + \frac{F}{H})$	$G = 48$ $H = 2$ $K = 11$	3.7537×	0.8789%
Layer 2: Biclustering + SVD	$G \cdot H \cdot (\frac{C}{G} \cdot K_1 + K_1 \cdot X \cdot Y \cdot K_2 + K_2 \cdot \frac{F}{H})$	$G = 2$ $H = 2$ $K_1 = 28$ $K_2 = 76$	2.0386×	0.8057%

Table 3: Number of parameters expressed as a function of hyperparameters for various approximation methods and empirical reduction in parameters with corresponding network performance.

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 5(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.

Acknowledgments

References

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

[2] 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)

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

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

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

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

[7] 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)

[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)

[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