Sparse Convolutional Neural Networks*

Baoyuan Liu[1], Min Wang[1], Hassan Foroosh[1], Marshall Tappen[3], and Marianna Penksy[2]

[1]Computational Imaging Lab, Computer Science, University of Central Florida, Orlando, FL, USA
[2]Department of Mathematics, University of Central Florida, Orlando, FL, USA
[3]Amazon.com, Seattle, WA 98109

{bliu, mwang, foroosh}@cs.ucf.edu, tappenm@amazon.com, Marianna.Pensky@ucf.edu

Abstract

Deep neural networks have achieved remarkable performance in both image classification and object detection problems, at the cost of a large number of parameters and computational complexity. In this work, we show how to reduce the redundancy in these parameters using a sparse decomposition. Maximum sparsity is obtained by exploiting both inter-channel and intra-channel redundancy, with a fine-tuning step that minimize the recognition loss caused by maximizing sparsity. This procedure zeros out more than 90% of parameters, with a drop of accuracy that is less than 1% on the ILSVRC2012 dataset. We also propose an efficient sparse matrix multiplication algorithm on CPU for Sparse Convolutional Neural Networks (SCNN) models. Our CPU implementation demonstrates much higher efficiency than the off-the-shelf sparse matrix libraries, with a significant speedup realized over the original dense network. In addition, we apply the SCNN model to the object detection problem, in conjunction with a cascade model and sparse fully connected layers, to achieve significant speedups.

1. Introduction

In this paper, we show how expressing the filtering steps in a convolutional neural network using sparse decomposition can dramatically cut down the cost of computation, while maintaining the accuracy of the system. Deep neural networks have achieved remarkable performance in both image classification and object detection problems [14][8]. Results of ImageNet LSVRC [2] competitions in recent years have demonstrated a strong correlation between the network size and the classification accuracy. The ILSRVR 2014 submission from VGG [20] builds a network with up to 16 convolutional layers that reduces the top-5 classification error to 7.4%, at the expense of approximately one month of network training with 4 high-end GPUs.

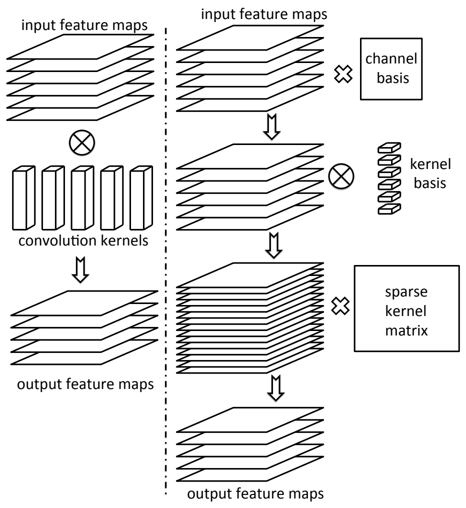


Figure 1: Overview of our sparse convolutional neural network. Left: the operation of convolution layer for classical CNN, which convolves large amount of convolutional kernels with the input feature maps. Right: our proposed SCNN model. We apply two-stage decompositions over the channels and the convolutional kernels, obtaining a remarkably(more than 90%) sparse kernel matrix and converting the operation of convolutional layer to spare matrix multiplication.

The structure of these networks makes it reasonable to conjecture that there exists heavy redundancy in these huge networks. Due to the highly non-convex property of neural networks, over-parameterization, together with random initialization, is necessary to overcome the negative impact of local minimum in network training. Additionally, the fact that no independence constraint is imposed among the con-

*This work was supported in part by the National Science Foundation under grants IIS-1212948, IIS-091686, DMS-1106564 and DMS-1407475.

volutional kernels for each layer in the training phase also indicates high potential for redundancy.

In this paper, we show that this redundancy makes it possible to notably reduce the amount of computation required to process images, by sparse decompositions of the convolutional kernels. As Figure 1 illustrate, two-stage decompositions are applied to explore the inter-channel and intra-channel redundancy of convolution kernels. We first perform an initial decomposition based on the reconstruction error of kernel weights, then fine-tune the network while imposing the sparsity constraint. In the fine-tuning phase, we optimize the network training error, the sparsity of convolutional kernels, as well as the number of convolutional bases simultaneously, by minimizing a sparse group-lasso object function. Surprisingly high sparsity can be achieved in our model. We are able to zero out more than 90% of the convolutional kernel parameters of the network in [14] with relatively small number of bases while keeping the drop of accuracy to less than 1%.

In our Sparse Convolutional Neural Networks (SCNN) model, each sparse convolutional layer can be performed with a few convolution kernels followed by a sparse matrix multiplication. It could be assumed that the sparse matrix formulation naturally leads to highly efficient computation. However, computing sparse matrix multiplication can involve severe overhead that makes it difficult to actually achieve attractive acceleration. Thus, we also propose an efficient sparse matrix multiplication algorithm. Based on the fact that the sparse convolutional kernels are fixed after training, we avoid the necessity of indirect and discontinuous memory access by encoding the structure of the input sparse matrix into our program as the index of registers. Our CPU-based implementation demonstrates much higher efficiency than off-the-shelf sparse matrix libraries and a significant speedup over the original dense networks is realized. While convolutional network systems are dominated by GPU-based approaches, advances in CPU-based systems are useful because they can be deployed in commodity clusters that do not have specialized GPU nodes.

Contribution Relative to Previous Work

As will be discussed in Section 2, below, previous work, such as [4][12], have used low-rank approximations to express the network computations in terms of a smaller number of basis filter. The approach presented here gains additional efficiency by expressing the filtering steps using a sparse decomposition, in addition to low-rank approximations. As will be shown in Section 6.2, this results in a combination of efficiency and accuracy that cannot be matched by just a low-rank decomposition.

2. Related Work

Several attempts have been made to study the redundancy of deep neural networks. Denil et al. [3] reduce the number of parameters in general neural network with low rank matrix factorization. They obtain 95% parameter reduction of MLP network on MNIST. Both Jaderberg et al. [12] and Denton et al. [4] use the idea of tensor low-rank expansions technique to speedup convolutional neural networks. Jaderberg et al. [12] obtain 4.5x speedup with less than 1% drop in accuracy of a 4 layer CNN trained on a scene character classification dataset. Denton et al. [4] achieve $2\times$ speedup on the first two convolutional layers of CNN trained on ILSVRC dataset. Notably, both [12] and [4] only demonstrate speedups on relatively large convolutional kernel size. None of them show that their method can work on kernels as small as 3×3, which are extensively used in state-of-the-art CNN models.

There are also several works that try to optimize the speed of CNN from other perspectives. Vanhoucke et al. [22] studies CPU based general neural network speed optimization. They discuss the usage of SIMD instructions, alignment of memory, as well as fixed point quantization of the network. Mathieu et al. [17] proposes to utilize FFT to perform convolution in Fourier domain. They achieve 2x speedup on Alex net. Their method prefers a relatively large kernel size due to the overhead of FFT. Farabet et al. [6] implement a large scale CNN based on FPGA infrastructure that can perform embedded real-time recognition tasks.

Previous works on sparse matrix computation focus on the sparse matrix dense vector multiplication (SpMV) problem. The sparse matrix is stored with various formats, such as CSR [1] and ESB [15], for efficiency. Blocking is adopted in register [11] and cache [18] level to improve the spatial locality of sparse matrix. To further reduce bandwidth requirement, various techniques including matrix reordering [19], value and index compression [23] are proposed. We refer readers to [9] for a more comprehensive review.

3. Our Method

3.1. Sparse Convolutional Neural Networks

Consider the input feature maps $\mathbf{I}$ in $\mathbb{R}^{h\times w\times m}$, where h, w and m are the height, width and number of channels of the input feature maps, and the convolutional kernel $\mathbf{K}$ in $\mathbb{R}^{s\times s\times m\times n}$, where s is size of the convolutional kernel and n is the number of output channels. We assume the convolution is performed with no padding zeros and stride equals to 1. Then, the output feature maps of a convolutional layer $\mathbf{O} \in \mathbb{R}^{(h-s+1)\times(w-s+1)\times n} = \mathbf{K} * \mathbf{I}$ are given by

$$\mathbf{O}(y,x,j) = \sum_{i=1}^{m} \sum_{u,v=1}^{s} \mathbf{K}(u,v,i,j)\mathbf{I}(y+u-1,x+v-1,i) \quad (1)$$

Our objective is to replace computationally expensive convolutional operation $\mathbf{O} = \mathbf{K} * \mathbf{I}$ in formula (1) by its fast sparsified version which is based on multiplication of sparse matrices.

For this purpose, we first transform the tensor $\mathbf{I}$ to $\mathbf{J} \in \mathbb{R}^{h\times w\times m}$ and convolutional kernel $\mathbf{K}$ to $\mathbf{R} \in \mathbb{R}^{s\times s\times m\times n}$ using a matrix $\mathbf{P} \in \mathbb{R}^{m\times m}$ obtaining $\mathbf{O} \approx \mathbf{R} * \mathbf{J}$ where

$$\begin{aligned} \mathbf{K}(u,v,i,j) &\approx \sum_{k=1}^{m} \mathbf{R}(u,v,k,j)\mathbf{P}(k,i) \\ \mathbf{J}(y,x,i) &= \sum_{k=1}^{m} \mathbf{P}(i,k)\mathbf{I}(y,x,k) \end{aligned} \tag{2}$$

Next, for every channel $i = 1, \cdots, m$, we decompose tensor $\mathbf{R}(\cdot,\cdot,i,\cdot) \in \mathbb{R}^{s\times s\times n}$ into the product of matrix $\mathcal{S}_i \in \mathbb{R}^{q_i\times n}$ and tensor $\mathcal{Q}_i \in \mathbb{R}^{s\times s\times q_i}$, where q_i is the number of bases:

$$\begin{aligned} \mathbf{R}(u,v,i,j) &\approx \sum_{k=1}^{q_i} \mathcal{S}_i(k,j)\mathcal{Q}_i(u,v,k) \\ \mathcal{T}_i(y,x,k) &= \sum_{u,v=1}^{s} \mathcal{Q}_i(u,v,k)\mathbf{J}(y+u-1, x+v-1, i). \end{aligned} \tag{3}$$

so that

$$\mathbf{O}(y,x,j) \approx \sum_{i=1}^{m}\sum_{k=1}^{q_i} \mathcal{S}_i(k,j)\mathcal{T}_i(y,x,k) \tag{4}$$

Note that if we represent the tensor $\mathbf{O}$ and $\mathcal{T}_i$ as matrices by combining the first two dimensions, and concatenate both $\mathcal{S}_i$ and $\mathcal{T}_i$ along the dimension q_i, formula (4) can be implemented by a single matrix multiplication.

Here, we shall search for matrices $\mathbf{P}$, $\mathcal{Q}_i$ and $\mathcal{S}_i$, $i = 1, \cdots, m$, such that q_i are much smaller than s^2, matrices $\mathcal{S}_i$ have a large number of zero elements and columns, while our new sparse convolutional kernel $\mathbf{R}$ provides output that is close to the one obtained with the original kernel $\mathbf{K}$.

3.2. Computational Complexity

We analyze the theoretical complexity of our method by measuring the number of multiplications. The multiplications required by original convolution is given by:

$$m\,n\,s^2(h-s+1)(w-s+1) \tag{5}$$

Our method reduces the complexity by sparsifying the the convolutional kernel, while introducing overhead of two matrix decompositions:

$$\left(\gamma mn + \sum_{i=1}^{n} q_i\right) s^2(h-s+1)(w-s+1) + m^2hw \tag{6}$$

where γ is the proportion of non-zeros of the sparse matrix. The decomposition overhead is small when (1) average of q_i is much smaller than γm and (2) m is much smaller than $\gamma n s^2$.

3.3. Learning Parameters

The parameters of Sparse Convolutional Neural Networks are learned in two phases: initial decomposition and fine-tuning. The factorization in both formula (2) and formula (3) can be treated as matrix factorization problem if we transform the convolutional kernel from tensor to matrix along the decomposition dimension. Sparse matrix decomposition algorithm is an intuitive choice for initialization. However, there are a couple of concerns regarding current sparse dictionary learning algorithms: (a) They are not convex and therefore cannot achieve a global optimum solution. (b) The learning process is a tradeoff between accuracy and sparsity, hence cannot provide exact decomposition. Due to these concerns, we choose the following decomposition methods and compare their performance in Section 6.4:

- Decompose $\mathbf{K}$ and $\mathbf{R}$ using the sparse dictionary learning algorithm in [16], with $\mathbf{P}$, $\mathcal{Q}_i$ the bases
- Decompose $\mathbf{K}$ and $\mathbf{R}$ using Principal Component Analysis (PCA), with $\mathbf{P}$, $\mathcal{Q}_i$ the principal components
- Initialize $\mathbf{P}$, $\mathcal{Q}_i$ as identity matrices, and keep K and R unchanged. In this way, sparsity is achieved solely by fine-tuning.

As initial decompositions, the above methods can only obtain very limited sparsity, but provide meaningful starting point with very low or no reconstruction error. Further fine-tuning is critical for obtaining a highly sparse network. In the fine-tuning phase, we impose sparsity constraints over the network parameters, while continuing to train the whole network. The underlying reason of why fine-tuning can increase sparsity is that the original network is trained without any sparsity constraint. Since the network is known to be over-parameterized to avoid the sensitivity to initialization, it should have some freedom of modification without compromising the classification accuracy. The initialization phase can only obtain sparsity based on the reconstruction error, while the fine-tuning process can fully exploit the sparsity potential of our model by adopting the network loss as the direct error measurement. Our experiments justify this point by showing a significant increase in sparsity due to fine-tuning.

Formally, we minimize the following objective function in the fine-tuning phase

$$\begin{aligned} &\underset{\mathbf{P},\mathcal{Q}_i,\mathcal{S}_i}{\text{minimize}}\ \mathbf{L}_{net} + \lambda_1 \sum_{i=1}^{m} \|\mathcal{S}_i\|_1 + \lambda_2 \sum_{i=1}^{m}\sum_{j=1}^{q_i} \|\mathcal{S}_i(j,\cdot)\|_2 \\ &\text{s.t. } \|\mathbf{P}(\cdot,j)\|_2 \le 1, j = 1,\dots,m \\ &\qquad \|\mathcal{Q}_i(\cdot,\cdot,k)\|_2 \le 1, i = 1,\dots,m, k = 1,\dots,q_i \end{aligned} \tag{7}$$

where $\mathbf{L}_{net}$ stands for the logistic loss function of the output layer of the network[14], and $\|\cdot\|_1$ and $\|\cdot\|_2$ denote the element-wise l_1 and l_2 norms of a matrix. The second term in (7) imposes lasso constraint over each of the elements of $\mathcal{S}_i$, and the third term treats each row of $\mathcal{S}_i$ as a group variable in a spirit of group-lasso formulation. The effective columns of each $\mathcal{Q}_i$ that need to be calculated is equal to the number of non-zero rows of corresponding $\mathcal{S}_i$. In this way, we can further reduce the overhead of convolving $\mathcal{Q}_i$ over the feature maps.

3.4. Comparison between our method and low-rank decomposition

Like [12] [4], our model uses a low-rank decomposition, but goes beyond previous work by encouraging sparsity in the weights used to express the filter to further reduce the redundancy in convolutional layers. We impose both sparse constraints and low-rank constraints with a combination of an l_1 norm and group lasso penalty

Consider the decomposing matrix $\mathbf{M} \in \mathbb{R}^{m\times n}$ as the multiplication of $\mathbf{S} \in \mathbb{R}^{m\times n}$ and $\mathbf{P} \in \mathbb{R}^{n\times n}$. Using a sparse decomposition, the speedup is proportional to the percentage of non-zero elements in S, while the reduction in complexity using a decomposition is proportional to the percentage of non-zero columns. The sparsity constraint is able to target specific entries in S, while a low-rank constraint must eliminate entire columns. This makes it possible for our approach to target redundancy more precisely. As the results in Section 6.2 will show, this leads to remarkable performance improvement using the multiplication algorithm.

4. Sparse Matrix Multiplication Algorithm

While the sparsity penalties can target redundancy more precisely, the performance benefits can only be realized with a sparse matrix multiplication algorithm that does not incur overhead that overwhelms the benefit of the sparse matrix. In this section, we show how the multiplication can be implemented efficiently.

4.1. Motivation

To avoid extra storage and calculation of zero values, the non-zero elements in a sparse matrix are typically stored continuously, with their locations indexed in some specific structure. This leads to indirect jumping memory access when traversing the matrix, which is much slower than the continuous direct access used in the dense case. In addition, the irregular pattern of input matrices also makes it difficult to fully utilize the capacity of Single Instruction Multiple Data (SIMD) micro architectures, which is the key in high-performance dense matrix algorithms.

We propose an efficient, sparse-dense matrix multiplication algorithm for executing the sparse convolutional kernels. Our idea is based on the following two key observations:

- Once the network is fully trained, the convolutional kernels are constant while the input feature maps vary with input images. Therefore, the location of non-zero elements are known and can be encoded directly in the compiled multiplication code.
- Only the convolutional kernels are treated as sparse matrices. The input feature maps do possess moderate sparsity for some layers, but will be treated as a dense matrix.

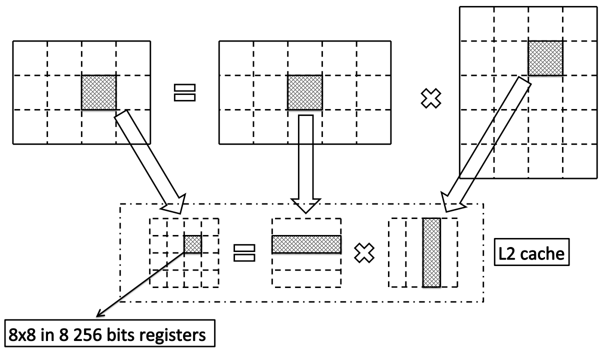


Figure 2: Matrix Multiplication Algorithm in OpenBLAS. The input matrices are first divided into blocks which can fit in L2 cache. Each block is then divided into 8 element wide strips. The multiplication outputs of two strips are held in 8 AVX registers during calculation.

We implemented our method on x86_64 CPU microarchitecture with the Advanced Vector Extension (AVX), which is available on both Intel and AMD's CPUs after 2011, although we expect that this approach could be extended to GPU architectures. Our method is based on OpenBLAS[24], which is an open source dense linear algebra library. In the following sections, we first briefly describe the matrix multiplication algorithm in OpenBLAS, then introduce our method on sparse matrices.

4.2. Dense Matrix Multiplication in OpenBLAS

There are two main considerations on designing an efficient matrix multiplication algorithm: (a) Taking advantage of SIMD instructions for higher computing throughput; (b) Maximally utilizing the caches to reduce memory access latency. In latest OpenBLAS library, matrix multiplication is implemented with AVX instruction sets, in which 8 float numbers can be stored in one 256bit AVX register. Namely, 8 pairs of float point numbers can be multiplied and added simultaneously in one cycle of each CPU core. To maximally reduce the memory latency, the input matrices are first divided into blocks that can fit into the L2 cache of CPU. Each block of one input matrix is then multiplied with each block of the other input matrix in the following way: One block is divided to 8-elements wide row strips and the other block is divided to 8-elements wide column strips. Then every two strips are multiplied to generate an 8×8 tiny square which can be stored in 8 AVX registers.

Figure 2 gives a graphical illustration of the matrix multiplication algorithm in OpenBLAS.

4.3. Sparse Matrix Multiplication

We focus on the sparse-dense matrix multiplication problem $\mathbf{C} = \mathbf{A} \times \mathbf{B}$. $\mathbf{A} \in \mathbb{R}^{m \times k}$ is a dense matrix and $\mathbf{B} \in \mathbb{R}^{k \times n}$ is a fixed sparse matrix. $\mathbf{A}$ and $\mathbf{B}$ are divided into blocks and strips in the same way as the dense case described above. Now, we consider the multiplication of one row strip and one column strip $\bar{\mathbf{C}} = \bar{\mathbf{A}} \times \bar{\mathbf{B}}, \bar{\mathbf{A}} \in \mathbb{R}^{8 \times k}, \bar{\mathbf{B}} \in \mathbb{R}^{k \times 8}, \bar{\mathbf{C}} \in \mathbb{R}^{8 \times 8}$. For any matrix $\mathbf{M}$, let $\mathbf{m}_{i,*}$ be the i^{th} row of $\mathbf{M}$ and $\mathbf{m}_{*,j}$ be the j^{th} column of $\mathbf{M}$. The matrix multiplication can be represented as

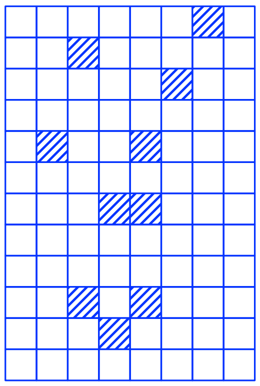
$$\bar{\mathbf{c}}_{*,j} = \sum_{i=1}^{k} \bar{\mathbf{a}}_{*,i} \bar{b}_{i,j}, 1 \le j \le 8 \tag{8}$$

In which every $\bar{\mathbf{c}}_{*,j}$ and $\bar{\mathbf{a}}_{*,i}$ are held in one AVX vector. Since $\bar{\mathbf{B}}$ is sparse in our case, we need to have the knowledge of the locations of non-zero elements and skip the zeros. To avoid indirect memory access, we propose to encode the structure of $\bar{\mathbf{B}}$ into the program. For each non-zero value $\bar{b}_{i,j}$, i indicates which $\bar{\mathbf{a}}_{*,i}$ to multiply with and j indicates which $\bar{\mathbf{c}}_{*,j}$ to save to. Since they both correspond to single AVX registers, we can simply encode i and j into our program as the index of registers. Figure 3 shows a toy example of how our method generates code from given sparse matrix.

5. Application to Object Detection

We now apply SCNN to the object detection problem. Girshick et al. [8] first proposed to use CNN to solve the object detection problem. They warp each candidate window generated by the Selective Search[21] method to a fixed size and use CNN to generate high level discriminative features, with whick linear SVM with hard negative mining is adopted to train object detection model. He et al. [10] significantly improves the speed of [8] by utilizing a Spatial Pyramid Pooling (SPP) scheme, in which the convolutional layers only need to be performed once (single scale) or several times (multiple scales) on the whole image and only the fully connected layers are performed on each candidate window. Due to large amount of candidate windows in each image ($\sim$2000), the total running time of fully connected layers in [10] is comparable to or dominant over the one of convolutional layers.

We apply SCNN to accelerate the convolutional layers of SPP model. Although our method can accelerate the convolutional layers by a high factor, the total running time would not reduce much due to the relatively time consuming fully connected layers. In this section, we propose two schemes to reduce the complexity of fully connected layers to achieve an overall high efficiency.



(a) An example sparse matrix $\mathbf{B}$. The shadowed squares represent non-zero elements and the blank squares represent zero elements.

```
Input:
    A: $8 \times 12$ dense matrix
    B: $12 \times 8$ sparse matrix
Output:
    $\mathbf{C} = \mathbf{A} \times \mathbf{B}$
Operations:
    $\mathbf{c}_7 += \mathbf{a}_1 \times b_{1,7}$
    $\mathbf{c}_3 += \mathbf{a}_2 \times b_{2,3}$
    $\mathbf{c}_6 += \mathbf{a}_3 \times b_{3,6}$
    $\mathbf{c}_2 += \mathbf{a}_5 \times b_{5,2}$
    $\mathbf{c}_5 += \mathbf{a}_5 \times b_{5,5}$
    $\mathbf{c}_4 += \mathbf{a}_7 \times b_{7,4}$
    $\mathbf{c}_5 += \mathbf{a}_7 \times b_{7,5}$
    $\mathbf{c}_3 += \mathbf{a}_{10} \times b_{10,3}$
    $\mathbf{c}_5 += \mathbf{a}_{10} \times b_{10,5}$
    $\mathbf{c}_4 += \mathbf{a}_{11} \times b_{11,4}$
```

(b) Generated Pseudo code for calculating $\mathbf{C} = \mathbf{A} \times \mathbf{B}$. $\mathbf{c_i}$ is the i^{th} column of $\mathbf{C}$ and $\mathbf{a_j}$ is the j^{th} column of $\mathbf{A}$. $b_{i,j}$ is the element of $\mathbf{B}$ at i^{th} column and j^{th} row

Figure 3: An example that illustrates how our algorithm generates code for multiplying a dense matrix and a sparse matrix

First, we propose to apply a cascade scheme over the network in a similar fashion to [7]. Since the output of the last convolutional layer is already highly discriminative, directly applying a linear SVM classifier over it achieves good performance [8][10]. Therefore, we can use the last convolutional layer as the first stage of our cascade model to prune large portion of candidate windows, and then use the output of the final fully connected layer as a second stage classifier to generate the final detection results. The decision of detection threshold of first stage is a tradeoff between efficiency and accuracy. A lower threshold retains high recall so that the overall accuracy is not affected while a higher threshold removes more candidates to achieve higher efficiency. In our case, we found that a threshold with a corresponding precision equals 0.05 is a balanced tradeoff. Previous work on cascade models were applied only to single class detection. Multiple class detection, in general, achieve less gain, since the number of candidates rapidly increases with the number of classes. However in our model we can still remove considerably large portion of candidates with multiple classes.

Second, we decompose the fully connected layer in a similar fashion as we did with the convolutional layers. The operation of fully connected layer can be represented by matrix-vector multiplication followed by the neuron function. We decompose the weight matrix into the product of one sparse matrix and one dense matrix as $\mathbf{M} \approx \mathbf{PS}$, where $\mathbf{M} \in \mathbb{R}^{m \times n}$, $\mathbf{P} \in \mathbb{R}^{m \times k}$ and $\mathbf{S} \in \mathbb{R}^{k \times n}$. We choose to enforce sparsity on $\mathbf{P}$ if $m > n$ and on $\mathbf{S}$ otherwise, so that the dense matrix has a smaller dimension. We impose

layer	conv1	conv2	conv3	conv4	conv5
kernel size	11	5	3	3	3
input channels	3	96	256	384	384
output channels	96	256	384	384	256
complexity%	15.8	33.6	22.5	16.8	11.2
sparsity%	0.927	0.950	0.951	0.942	0.938
average q_i	29	7.91	5.23	4.32	3.95
theoretical speedup	2.61	7.14	16.12	12.42	10.77
Actual speedup	2.47	4.52	6.88	5.18	3.92

Table 1: Sparsity, Average number of bases ,theoretical and actual speedup corresponding to each convolutional layer for our SCNN model. q_i is the average number of bases in each channel. Results demonstrates that our highly sparse model could lead to remarkably acceleration for computation in both theory and practice.

both l_1 norm and group lasso cost function so that k is also reduced.

6. Experimental Results

6.1. Setup

We trained our model on the ImageNet LSVRC 2012 [2] dataset. We start from a pre-trained Caffe[13] reference CNN model, which is almost identical to the model described in [14]. The model consists of 5 convolutional layers and two fully connected layers, interlaced with subsampling layers, local normalizing layers, max pooling layers, rectified linear unit layers and dropout layers. The first convolutional layer has relatively large 11×11 kernels and only 3 input channels; the second convolutional layer has 5×5 kernels; The third, fourth and fifth convolutional layers have very small 3×3 kernels. The difference of kernel sizes as well as the number of input kernels affects the possible sparsity that can be achieved.

All 5 convolutional layers are optimized simultaneously according to Equation 7 using stochastic gradient decent with momentum. The base learning rate is initially set to 0.001, while sparsifying the network parameters. To stabilize the training process, we adopt a thresholding function that sets parameters smaller than $1e^{-4}$ to zero during training. Once the training process converges, we remove the sparsity constraint, but keep the thresholding function. Finally, we gradually decrease the base learning rate to fine-tune the network for best accuracy.

6.2. Results on ILSVRC12

Table 1 shows the results on ILSVRC12. For all 5 convolutional layers, we obtain more than 90% sparsity. The average number of bases in each layer is significantly smaller than s^2(square of kernel size), which corresponds to the full rank decomposition. The theoretical speedup are the ratio between the running time of our SCNN layer, and the original convolutional layer. The final column of Table 1 shows the actual acceleration factor achieved. Because of the overhead in sparse matrix multiplication, it is expected that actual performance improvements will not match theoretical results.

layer	conv1	conv2	conv3	conv4	conv5
kernel size	7	5	3	3	3
sparsity%	0.840	0.956	0.893	0.904	0.890
average q_i	21	9.06	6.76	6.86	6.98
theoretical speedup	2.62	7.06	8.03	8.78	7.29
low-rank[4]	2.4	2.5	-	-	-

Table 2: Comparison between a model, similar to[10], trained with sparsity and the speedup factors reported in [4].

The speedup factor in conv1 is not as significant as other layers due to limited redundancy that is caused by large kernel size and the small number of input and output channels. The number of bases in conv1, although substantially reduced, still accounts for a large portion of running time because of small number of output channels. This issue is also present in previous low-rank approaches [12]. We experimented with decomposing the basis filters with combination of separable filters, but the bases showed too much variation to be expressed with separable filters.

6.3. Comparison with Only Using Low-Rank Approximations

To more directly compare with previous work, we also trained a modified version of the model used in [10], which is similar to that used in [4]. The primary difference lies in a spatial pyramid layer that will not affect the sparsity properties of the convolutional layers. Table 2 shows the comparison of theoretical speedup factors between [4] and our method trained based on [10]. We have a similar speedup as [4] in conv1 due to the above mentioned reason, while obtaining much higher speedup for conv2. It should be noted that [4] did not attempt to accelerate the later layers, likely due to the tiny size of filters (3×3), while our method is able to achieve significant speedup.

6.4. Comparison of Initialization Methods

Figure 4 shows a comparison of the performance of different initialization methods that we adopted. Both PCA and sparse coding obtain more than 90% sparsity, while random initialization shows inferior sparsity by a large margin. Notably, all the methods obtain very limited sparsity with only initialization. These results demonstrate the importance of both initialization and fine-tuning. All of the methods show very small amount of accuracy drop during the fine-tuning process. The accuracy numbers shown in Figure 4 are generally lower than final accuracy since the learning rate we set during sparsifying the network is higher than final learning rate for faster convergence. Among all the three methods, the accuracy of PCA is slightly better than the others. The sparse coding method, although seems to make

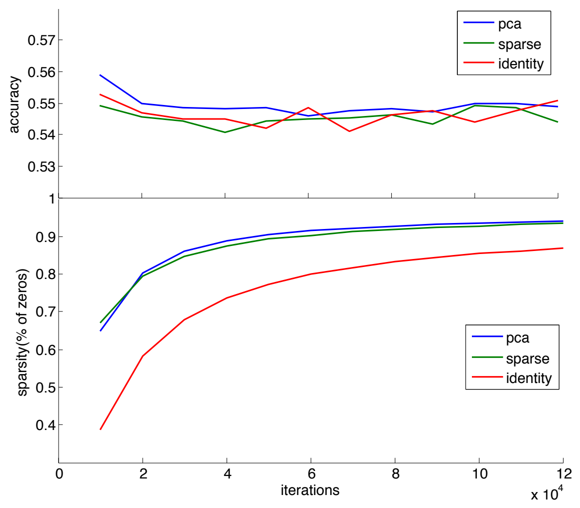


Figure 4: Comparison of initial decomposition methods. We show the variation of both the accuracy and the average sparsity of our sparse CNN during the training process.

more sense theoretically, is inferior practically. We argue that the main reason is its non-convexness and non-accurate reconstruction. Although PCA is also a non-convex problem, global unique optimum can be obtained with SVD.

6.5. Bases Visualization

Figure 6 shows the average number of non-zero elements in our sparse convolution kernels corresponding to each basis of our decomposition over both input channels and kernels inside each channel. The bases are sorted according their eigenvalues of PCA initialization. High correlation can be found between the sparseness and the eigenvalues of PCA, which justifies the importance of PCA initialization. For a significant portion of the bases, the sparse kernels are almost all zero. The all zero bases are equivalent to the ones that can be eliminated by low-rank decomposition. For relatively large-size kernels, like conv1 and conv2, the percentage of all zero bases is significant enough to achieve moderate level of speedup. However, for small-size kernels like conv3, conv4 and conv5, the percentage of all zero bases is very limited. Even a speedup factor of 2 will significantly sacrifice the accuracy with low-rank decomposition. The advantages of our method are clearly shown by the sparsity obtained for the non-zero bases, shown in Figure 6.

To justify the necessity of fine-tuning, we compare the convolution kernels in the original model and the ones that are reconstructed from our fine-tuned sparse model in Figure 5. We only show the kernels of conv1, conv2 and conv3 layers due to limited space. We ignore the kernels that are all zeros, which is very common from conv3 to conv5. We also measure the average similarity between the original and reconstructed kernels by first deduct the mean values from both kernels, and then calculate the cosine similarity measurement, which is defined as $\text{Sim}_{\cos}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|}$. From

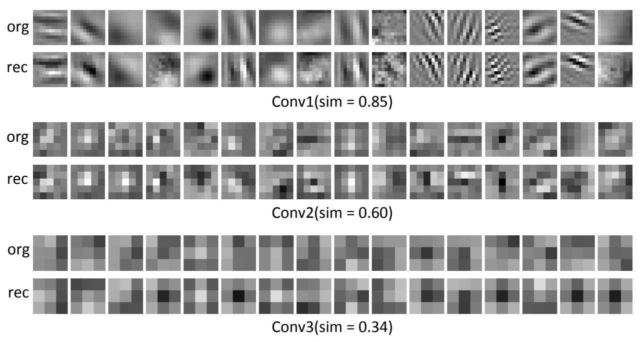


Figure 5: Comparison between the original convolution kernels and the convolution kernels reconstructed from our sparse kernels. Here we show randomly sampled kernels conv1, conv2 and conv3 layers. conv4 and conv5 are very similar to conv3. For each layer, the first row shows the original kernels and the second row shows the reconstructed ones. The average cosine similarity between them are displayed under.

Figure 5, we can see that the average similarity decreases rapidly from conv1 to conv3. The kernels in conv3 look very different from the original ones. Considering how little the accuracy drops in our model, this justifies our argument that the original network is extremely over-parameterized and significant regularization can be imposed without affecting the performance. Thus, one can hardly exploit the full sparsity potential of the network by only attempting to approximate the original kernel instead of network loss based fine-tuning adopted in our method.

6.6. Evaluation of Sparse Matrix Multiplication Algorithm

We first analyze the performance of our sparse dense matrix multiplication algorithm with a randomly generated matrix. We randomly generate one 1024×1024 dense matrix and one sparse 1024×1024 matrix, and measure the running time of multiplying them with our algorithm. Our experiments are performed on an Intel i7-3930k CPU. We measure the single-thread performance in this case for simplicity. The multi-thread performance should be consistent. Figure 7 shows the result of our evaluation. The following conclusions can be drawn from this figure: (a)The arithmetic time is strictly proportional to the density of the input matrix, and very close to the theoretical limit; (b) The I/O time decreases with the increase in sparsity, but in a sublinear speed. The time of loading from memory to cache and storing results are constant regardless of the variation of sparsity, and the time of loading the dense matrix from cache to CPU depends on the percentage of consecutive 8 zeros in the sparse matrix; (c) The I/O time is increasingly dominant when the sparsity increases, when the density is less than 10%, the I/O operations takes more than 80% of the total running time. (d) Note that the sum of arithmetic time and I/O time is significantly higher than the total time.

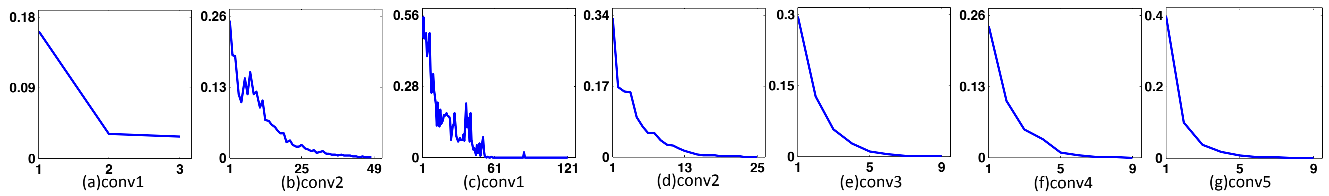


Figure 6: Average ratio of non-zero elements in our sparse convolution kernels corresponding to the bases of decompositions over channels and filters. (a) (b) show the bases over channels and (c) to (g) show the bases over filters.The bases in each figure are sorted in descending order of their eigenvalues in PCA initialization.

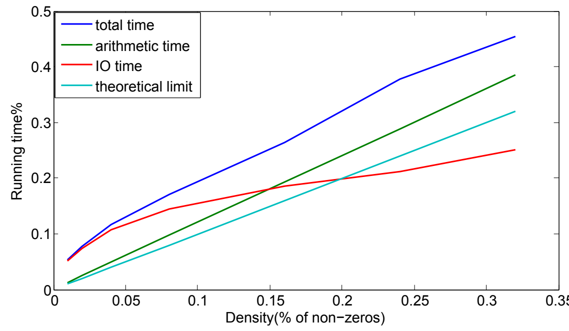


Figure 7: Running time analysis of our sparse-dense matrix multiplication algorithm. The horizontal axis stands for the percentage of non-zero elements in the input sparse matrix, and the vertical axis is the relative running time comparing to the dense matrix multiplication code in OpenBLAS. The arithmetic time is the running time of multiplication and addition, and the I/O time includes loading the input matrix from memory to cache, loading from cache to CPU and writing result to memory. The theoretical time is the best possible speedup one can achieve, which is identical to the density of the input matrix.

layer	conv1	conv2	conv3	conv4	conv5
Channel Decomp	0.06	0.14	-	-	-
Basis Convolution	0.78	0.41	0.21	0.30	0.44
Matrix Mult	0.16	0.45	0.79	0.70	0.56
Original	2.47	4.52	6.88	5.18	3.92

Table 3: Running time analysis and comparison with original dense networks. All the numbers for each layer are normalized with the layer's total running time with our method. The last row is the speed-up factor of our method.

This is due to the parallelism introduced by the pipeline strategy of CPU. For this reason, the arithmetic operations and I/Os can be executed simultaneously, thus providing a significantly better efficiency.

6.7. Running Time Analysis

Table 3 shows the actual speed of our code as well as the proportion of each component. Significant speedups are achieved for all 5 layers. The sparse matrix multiplication time is dominant for the last three layers while the basis convolution time takes a large portion for the first two layers, which is consistent with theoretical analysis. The gap between actual speedup and theoretical one comes mainly from two factors: (1) Overhead of sparse matrix multiplication; (2) Low efficiency of basis convolution. Caffe implements convolution as matrix multiplication, which is relatively inefficient for small number of filters as in our case. We implement a faster version but is still not fully optimized. In addition, joint cache optimization with both convolution and sparse matrix multiplication will further improve the efficiency.

	fc7(1s)	fc7bb(1s)	fc7(5s)	fc7bb(5s)
spp[10]	52.47	54.19	54.75	57.19
ours	50.16	52.64	52.58	55.13

Table 4: Mean average precision of object detection with our method compared with [10]. "bb" stands for bounding box regression, "1s" means 1 scale and "5s" means 5 scales. Our sparse model is inferior to [10] by approximately 2%.

6.8. Results on Object Detection

We used the fine-tuned model in Table 2 to perform object detection on PASCAL VOC2007 [5] dataset. The accuracy of our method compared with the original one in [10] is shown in Table 4. We are not able to reproduce the accuracy using the code published by [10], so we put the number we get instead. Our method is approximately 2% worse than the original SPP model, while obtaining several times faster speed. We speculate that the higher accuracy drop than classification problem is probably due to the fact that the convolutional layer is trained on ILSVRC dataset, while only the fully connected layer are fine-tuned to adapt to the PASCAL data as in [10]. Thus, although we are sparsifying the convolutional layers with fine-tuning, the loss of the network is not equivalent to the loss of the detection problem. A whole network based fine-tuning should further reduce the accuracy drop of our method.

In our cascade model, we set the thresholds of first stage so that the precision for each class equals 0.05. Approximately 80% of candidate windows are pruned for each image, thus bringing approximately 5× speedup of fully connected layers with almost no drop in accuracy. In addition, 85% and 68% sparsity are achieved for the first and second fully connected layer, further providing over 2× speedup. The running time of fully connected layer is thus reduced to be much smaller than the convolutional layers.

References

[1] R. Barrett, M. W. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*, volume 43. Siam, 1994.

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[3] M. Denil, B. Shakibi, L. Dinh, N. de Freitas, et al. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*, pages 2148–2156, 2013.

[4] E. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, 2014.

[5] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[6] C. Farabet, Y. LeCun, K. Kavukcuoglu, E. Culurciello, B. Martini, P. Akselrod, and S. Talay. Large-scale fpga-based convolutional networks. *Machine Learning on Very Large Data Sets*, 2011.

[7] P. F. Felzenszwalb, R. B. Girshick, and D. McAllester. Cascade object detection with deformable part models. In *Computer vision and pattern recognition (CVPR), 2010 IEEE conference on*, pages 2241–2248. IEEE, 2010.

[8] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[9] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *The Journal of Supercomputing*, 50(1):36–77, 2009.

[10] K. He, X. Zhang, S. Ren, and J. Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. In *Computer Vision–ECCV 2014*, pages 346–361. Springer, 2014.

[11] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, 2004.

[12] M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. In *Proc. BMVC*, 2014.

[13] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[15] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 273–282. ACM, 2013.

[16] J. Mairal, F. Bach, J. Ponce, and G. Sapiro. Online dictionary learning for sparse coding. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 689–696. ACM, 2009.

[17] M. Mathieu, M. Henaff, and Y. LeCun. Fast training of convolutional networks through ffts. In *International Conference on Learning Representations (ICLR2014)*. CBLS, April 2014.

[18] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):297–311, 2007.

[19] L. Oliker, X. Li, P. Husbands, and R. Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *Siam Review*, 44(3):373–393, 2002.

[20] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[21] J. R. Uijlings, K. E. van de Sande, T. Gevers, and A. W. Smeulders. Selective search for object recognition. *International journal of computer vision*, 104(2):154–171, 2013.

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

[23] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 307–316. ACM, 2006.

[24] Z. Xianyi, W. Qian, and Z. Chothia. Openblas. *URL: http://xianyi. github. io/OpenBLAS*, 2012.