Less is More: Towards Compact CNNs

Hao Zhou[1], Jose M. Alvarez[2] and Fatih Porikli[2,3]

[1] University of Maryland, College Park
[2] Data61/CSIRO
[3] Australian National University
hzhou@cs.umd.edu, jose.alvarez@data61.csiro.au
fatih.porikli@anu.edu.au

Abstract. To attain a favorable performance on large-scale datasets, convolutional neural networks (CNNs) are usually designed to have very high capacity involving millions of parameters. In this work, we aim at optimizing the number of neurons in a network, thus the number of parameters. We show that, by incorporating sparse constraints into the objective function, it is possible to decimate the number of neurons during the training stage. As a result, the number of parameters and the memory footprint of the neural network are also reduced, which is also desirable at the test time. We evaluated our method on several well-known CNN structures including AlexNet, and VGG over different datasets including ImageNet. Extensive experimental results demonstrate that our method leads to compact networks. Taking first fully connected layer as an example, our compact CNN contains only 30% of the original neurons without any degradation of the top-1 classification accuracy.

Keywords: convolutional neural network, neuron reduction, sparsity

1 Introduction

Last few years have witnessed the success of deep convolutional neural networks (CNN) in many computer vision applications. One important reason is the emergence of large annotated datasets and the development of high-performance computing hardware facilitating the training of high capacity CNNs with an exceptionally large number of parameters.

When defining the structure of a network, large networks are often preferred, and strong regularizers [1] tend to be applied to give the network as much discriminative power as possible. As such, the state-of-the-art CNNs nowadays contain hundreds of millions of parameters [2]. Most of these parameters come from one or two layers that host a large number of neurons. Take AlexNet [3] as an example. The first and the second fully connected layers, which have 4096 neurons each, contain around 89% of all the parameters. Having to use such a large number of parameters leads to high memory requirements when mapping and running the network on computational platforms. Consequently, using deep CNNs obligates significant hardware resources, which hinders their practicability for mobile computing devices that have limited memory specifications.

Table 1. Neuron reduction in the first fully connected layer, the total parameter compression, reduced memory, the top-1 validation error rate (red: error rate without sparse constraints).

Network	compression (%)		memory reduced[a]	Top-1 error (%)
	neurons[b]	parameters[c]		
LeNet	97.80	92.00	1.52 (MB)	0.63 (0.72)
CIFAR-10 quick	73.44	33.42	0.19 (MB)	25.25 (24.75)
AlexNet	73.73	65.42	152.14 (MB)	46.10 (45.57)
VGG-13	76.21	61.29	311.06 (MB)	39.26 (37.50)

[a] Supposing a single type is used to store the weight
[b] Results on number of neurons in the first fully connected layer
[c] Results on the number of parameters of the whole network

Several recent studies have focused on reducing the size of the network concluding that there is substantial redundancy in the number of parameters of a CNN. For instance, [4,5] represent the filters in a CNN using a linear combination of basis filters. However, these methods can only apply to an already trained network. Other works have investigated directly reducing the number of parameters in a CNN by using sparse filters instead of its original full-size filters. To the best of our knowledge, existing deep learning toolboxes [6,7,8] do not support sparse data structures yet. Therefore, special structures need to be implemented. Moreover, FFT is shown to be very efficient to compute convolutions on GPUs [9]. However, a sparse matrix is usually no longer sparse in the Fourier domain, which limits the applicability (i.e., reduction of memory footprint) of these methods.

To address the shortcomings of the aforementioned works, we propose an efficient method to decimate the number of neurons. Our approach has four key advantages: (1) Neurons are assessed and reduced during the training phase. Therefore, no pre-trained network is required. (2) The reduction of parameters do not rely on the sparsity of neurons; thus, our method can be directly included in popular deep learning toolboxes. (3) The number of filters in Fourier domain is proportional to the number of neurons in spatial domain. Therefore, our method can also reduce the number of parameters in the Fourier domain. (4) Reducing the number of neurons of a layer directly condenses the data dimensionality when the output of an internal layer is utilized as image features [10,11].

Our method consists of imposing sparse constraints on the neurons of a CNN in the objective function during the training process. Furthermore, to minimize the extra computational cost of adding these constraints, we solve the sparse constrained optimization problem using forward-backward splitting method [12,13]. The main advantage of forward-backward splitting is the bypass of the sparse constraint evaluations during the standard back-propagation step, making the implementation very practical. We also investigate the importance of rectified linear units in sparse constrained CNNs. Our experiments show that rectified

linear units help to reduce the number of neurons leading to even more compact networks.

We conduct a comprehensive set of experiments to validate our method using four well-known models (i.e., LeNet, CIFAR-10 quick, AlexNet and VGG) on three public datasets including ImageNet. Table 1 summarizes a representative set of our results when constraints are applied to the first fully connected layer of each model, which has the largest number of neurons and parameters. As shown, even with a large portion of neurons removed and, therefore, a significant reduction in the memory footprint of the original model; the resulting networks still perform as good as the original ones. These results are also convenient for deployment in mobile platforms where computational resources are relevant. For instance, using single float type (4 bytes) to store the network, the amount of memory saved for AlexNet in Table 1 is 152 MB.

To reiterate, the contribution of this paper is threefold. First, we remove neurons of a CNN during the training stage. Second, we analyze the importance of rectified linear units for sparse constraints. Finally, our experimental results on four well-known CNN architectures demonstrate a significant reduction in the number of neurons and the memory footprint of the testing phase without affecting the classification accuracy.

2 Related Work

Deep CNNs demand large memory and excessive computational times. This motivated studies to simplify the structure of CNNs. However, so far only one pioneering work explored the redundancy in the number of neurons [1].

We organize the related work in three categories; network distillation, approximating parameters or neurons by memory efficient structures, and parameter pruning in large networks.

Network distillation: The work in [14] is among the first papers that tried to mimic a complicated model with a simple one. The key idea is to train a powerful ensemble model and use it to label a large amount of unlabeled data. Then a neural network is trained on these data so as to perform similarly to the ensemble model. Following the idea of [14], [15,16,17] presenting training of a simple neural network based on the soft output of a complicated one. Their results show that a much simpler network can imitate a complicated network. However, these methods require a two-step training process and the simple network must be trained after the complicated one.

Memory efficient structures: Most of the studies in this category are inspired by the work of Denil et al. [18] that demonstrated redundancy in the parameters of neural networks. It proposed to learn only 5% of the parameters and predicted the rest by choosing an appropriate dictionary. Inspired by this, [4,5,19] proposed to represent the filters in convolutional layers by a linear combination of basis filters. As a result, convolving with the original filters is equivalent to convolving with the base filters followed by a linear combination of the output of these convolutions. These methods focus on speeding up the test-

ing phase of CNNs. In [20], the use of the CP-decomposition to approximate the filter in each layer as a sequence of four convolutional layers with small kernels is investigated in order to speed up testing time on CPU mode of a network. In [21] several schemes to quantize the parameters in CNNs are presented. All these methods require a trained network and then fine tuning for the new structures. [22] applied Tensor Train (TT) format to approximate the fully connected layers and train the TT format CNN from scratch. They can achieve a high compression rate with little loss of accuracy. However, it is not clear whether TT format can be applied to convolutional layers.

Parameter pruning: A straightforward way to reduce the size of CNNs is directly removing some of the unimportant parameters. Back to 1990, LeCun et al. [23] introduced computing the second derivatives of the objective function with respect to the parameters as the saliency, and removing the parameters with low saliency value. Later, Hassibi and Stork [24] followed their work and proposed an optimal brain surgeon, which achieved better results. These two methods, nevertheless, require computation of the second derivatives, which is computationally costly for large-scale neural networks [25]. In [26], directly adding sparse constraints on the parameters was discussed. This method, different from ours, cannot reduce the number of neurons.

Another approach [27] combines the memory efficient structures and parameter pruning. First, unimportant parameters are removed, and then, after a fine-tuning process, the remaining parameters are quantized for further compression. This work is complementary to ours and can be used to further compress the network trained using our method.

Directly related to our proposed method is [1]. Given a pretrained CNN, it proposes a theoretically sound approach to combine similar neurons together. Yet, their method is mainly for pruning a trained network whereas our method directly removes neurons during training. Although [1] is *data-free* when pruning, it requires a pretrained network, which is data dependent. Moreover, our results show that, without degrading the performance, we can remove more neurons than the suggested cut-off value reported in [1].

3 Sparse Constrained Convolutional Neural Networks

Notation: In the following discussion, if not otherwise specified, $|\cdot|$ is the ℓ_1 norm, $||\mathbf{x}|| = \sqrt{\sum_i x_i^2}$ is the ℓ_2 norm for a vector (Frobenius norm for a matrix). We use $\mathbf{W}$ and $\mathbf{b}$ to denote all the parameters in filters and bias terms in a CNN, respectively. $\mathbf{w}_l$ and $\mathbf{b}_l$ represent the filter and bias parameters in the l-th layer. $\mathbf{w}_l$ is a tensor whose size is $w \times h \times m \times n$, where w and h are the width and the height of a 2D filter, m represents the number of channels for the input feature and n is the number of channels for the output feature. $\mathbf{b}_l$ is a n dimensional vector, i.e. each output feature of this layer has one bias term. $\mathbf{w}_{lj}$ represents a $w \times h \times m$ filter that creates the j-th channel of the output feature for layer l, b_{lj} is its corresponding bias term. $\mathbf{w}_{lj}$ and b_{lj} together form a neuron. We use $\hat{\mathbf{W}}$,

$\hat{\mathbf{w}}_l$ and $\hat{\mathbf{w}}_{lj}$ to represent the augmented filters (they contain the corresponding bias term).

3.1 Training a Sparse Constrained CNN

Let $\{\mathbf{X}, \mathbf{Y}^*\}$ be the training samples and corresponding ground-truth label. Then, a CNN can be represented as a function $\mathbf{Y} = f(\hat{\mathbf{W}}, \mathbf{X})$, where $\mathbf{Y}$ is the output of the network. $\hat{\mathbf{W}}$ is learned through minimizing an objective function:

$$\min_{\hat{\mathbf{W}}} \psi(f(\hat{\mathbf{W}}, \mathbf{X}), \mathbf{Y}^*). \tag{1}$$

We use $\psi(\hat{\mathbf{W}})$ to represent the objective function for simplicity. The objective function $\psi(\hat{\mathbf{W}})$ is usually defined as the average cross entropy of the ground truth labels with respect to the output of the network for each training image. Equation (1) is usually solved using a gradient descend based method such as back-propagation [28].

Our goal is adding sparse constraints on the neurons of a CNN. Therefore, the optimization problem of (1) can be written as:

$$\min_{\hat{\mathbf{W}}} \psi(\hat{\mathbf{W}}) + g(\hat{\mathbf{W}}), \tag{2}$$

where $g(\hat{\mathbf{W}})$ represents the set of constraints added to $\hat{\mathbf{W}}$. Given this new optimization problem, the k-th iteration of a standard back-propagation can be defined as:

$$\hat{\mathbf{W}}^k = \hat{\mathbf{W}}^{k-1} - \tau \frac{\partial \psi(\hat{\mathbf{W}})}{\partial \hat{\mathbf{W}}}|_{\hat{\mathbf{W}}=\hat{\mathbf{W}}^{k-1}} - \tau \frac{\partial g(\hat{\mathbf{W}})}{\partial \hat{\mathbf{W}}}|_{\hat{\mathbf{W}}=\hat{\mathbf{W}}^{k-1}}, \tag{3}$$

where $\hat{\mathbf{W}}^k$ represents the parameters learned at k-th iteration and τ is the learning rate. Based on (3), a new term $\frac{\partial g(\hat{\mathbf{W}})}{\partial \hat{\mathbf{W}}}|_{\hat{\mathbf{W}}=\hat{\mathbf{W}}^{k-1}}$ must be added to the gradient of each constrained layer during back-propagation. In those cases where $g(\hat{\mathbf{W}})$ is non-differentiable at some points, sub-gradient methods of $g(\hat{\mathbf{W}})$ are usually needed. However, these methods have three main problems. First, iterates of the sub-gradient at the points of non-differentiability hardly ever occur [12]. Second, sub-gradient methods usually cannot generate an accurate sparse solution [29]. Finally, sub-gradients of some sparse constraints are difficult to choose due to their complex form or they may not be unique [13]. To avoid these problems, and in particular with l_1 constrained optimization problems, [30] and [26] proposed to use proximal mapping. Following their idea, in the next section, we apply proximal operator to our problem.

3.2 Forward-Backward Splitting

Our proposal to solve the problem (2) and therefore, train a constrained CNN, consists of using forward-backward splitting algorithm [12,13]. Forward-backward

splitting provides a way to solve non-differentiable and constrained large-scale optimization problem of the generic form:

$$\min_{\mathbf{z}} f(\mathbf{z}) + h(\mathbf{z}), \tag{4}$$

where $\mathbf{z} \in \mathbb{R}^N$, $f(\mathbf{z})$ is differentiable and $h(\mathbf{z})$ is an arbitrary convex function [12,13]. The algorithm consists of two stages: First, a forward gradient descent on $f(\mathbf{z})$. Then, a backward gradient step evaluating the proximal operator of $h(\mathbf{z})$. Using this algorithm has two main advantages. First, it is usually easy to estimate the proximal operator of $h(\mathbf{z})$ or even having a closed form solution. Second, backward analysis has an important effect on the convergence of the method when $f(\mathbf{z})$ is convex [13].

Though there is no guarantee about convergence when $f(\mathbf{z})$ is non-convex, forward-backward splitting method usually works quite well for non-convex optimization problems [13]. By treating $\psi(\hat{\mathbf{W}})$ in Equation (2) as $f(\mathbf{z})$, the forward gradient descent can be computed exactly as the standard back-propagation algorithm in training CNNs. As a result, using forward-backward splitting method to solve sparse constrained CNNs has two steps in one iteration. Algorithm 1 shows how to apply this method to optimize Equation (2), where τ^k is the learning rate of forward step at k-th iteration.

Algorithm 1 Forward-backward splitting for sparse constrained CNNs

1: **while** Not reaching maximum number of iterations **do**
2: One step back-propagation for $\psi(\hat{\mathbf{W}})$ to get $\hat{\mathbf{W}}^{k*}$
3: $\hat{\mathbf{W}}^{k+1} = \arg\min_{\hat{\mathbf{W}}} g(\hat{\mathbf{W}}) + \frac{1}{2\tau^k}||\hat{\mathbf{W}} - \hat{\mathbf{W}}^{k*}||^2$
4: **end while**

In practice, we define one step in line 2 of Algorithm 1 as one epoch instead of one iteration of the stochastic gradient descent algorithm. There are two main reasons for this. First, to minimize the computational training overhead of the algorithm as we need to estimate fewer proximal operators of $g(\hat{\mathbf{W}})$. Second, the gradient of $\psi(\hat{\mathbf{W}})$ at each iteration is an approximation to the exact gradient which is noisy [30]. Computing the gradient after certain number of iterations would make the learned parameters more stable [29].

4 Sparse Constraints

Our goal is removing neurons $\hat{\mathbf{w}}_{lj}$, each of which is a tensor. To this end, we consider two sparse constraints for $g(\hat{\mathbf{W}})$ in Equation (2): tensor low rank constraints [31] and group sparsity [32].

4.1 Tensor Low Rank Constraints

Although the low-rank constraints for 2D matrices and their approximations have been extensively studied, as far as we know, there are few works considering

the low-rank constraints for higher dimensional tensors. In [31], the authors proposed to minimize the average rank of different unfolding of a tensor matrix. To relax the problem to convex, they proposed to approximate the average rank using the average of trace norms, which is called tensor trace norm, for different unfolding [31]. We use this formulation as our tensor low rank constraints.

The tensor trace norm of a neuron $\hat{\mathbf{w}}_{lj}$ is $||\hat{\mathbf{w}}_{lj}||_{tr} = \frac{1}{n}\sum_{i=1}^{n}||\hat{\mathbf{w}}_{lj(i)}||_{tr}$, where n is the order of tensor $\hat{\mathbf{w}}_{lj}$, and $\hat{\mathbf{w}}_{lj(i)}$ is the result of unfolding the tensor $\hat{\mathbf{w}}_{lj}$ along the i-th mode. Under this definition, function $g(\hat{\mathbf{W}})$ can be defined as:

$$g(\hat{\mathbf{W}}) = \lambda \sum_{(j,l)\in\Omega} \frac{1}{n}\sum_{i=1}^{n}||\hat{\mathbf{w}}_{lj(i)}||_{tr}, \tag{5}$$

where Ω is a set containing all the neurons to be constrained and λ is the weight for the sparse constraint.

As a result, the backward step in the forward-backward splitting is given by:

$$\hat{\mathbf{w}}_{lj}^{k} = \arg\min_{\hat{\mathbf{w}}_{lj}} \frac{1}{n}\sum_{i=1}^{n}||\hat{\mathbf{w}}_{lj(i)}||_{tr} + \frac{1}{2\tau\lambda n}\sum_{i=1}^{n}||\hat{\mathbf{w}}_{lj(i)}^{k} - \hat{\mathbf{w}}_{lj(i)}^{k*}||^2. \tag{6}$$

This problem can be solved using the Low Rank Tensor Completion (LRTC) algorithm proposed in [31].[1]

4.2 Group Sparse Constraints

These are defined as $l_{2,1}$ regularizer. Applying $l_{2,1}$ to our objective function, we have:

$$g(\hat{\mathbf{W}}) = \lambda \sum_{(j,l)\in\Omega} ||\hat{\mathbf{w}}_{lj}||, \tag{7}$$

where λ and Ω are the same as in Section 4.1. $||.||$ is defined in Section 3. According to [32], backward step in forward backward splitting method at k-th iteration now becomes:

$$\hat{\mathbf{w}}_{lj}^{k} = \max\{||\hat{\mathbf{w}}_{lj}^{k*}|| - \tau\lambda, 0\}\frac{\hat{\mathbf{w}}_{lj}^{k*}}{||\hat{\mathbf{w}}_{lj}^{k*}||}, \tag{8}$$

where τ is the learning rate and $\hat{\mathbf{w}}_{lj}^{k*}$ is the optimized neuron from the forward step in forward backward splitting.

5 Importance of Rectified Linear Units in Sparse Constrained CNNs

Convolutional layers in a CNN are usually followed by a nonlinear activation function. Rectified Linear Units (ReLU) [3] has been heavily used in CNNs for

[1] Please refer to the supplementary material for the specific details of the algorithm.

computer vision tasks. Besides the advantages of ReLU discussed in [33,34,35], we show that $\hat{\mathbf{w}}_{lj} = \mathbf{0}$ is a local minimum in sparse constrained CNNs, of which the non-linear function is ReLU. This can explain our findings that ReLU can help removing more neurons and inspires us to set momentum to 0 for neurons which reach their sparse local minimum during training as discussed in Section 6.1.

ReLU function was defined as $ReLU(x) = max(0, x)$ which is non-differentiable at 0. This brings us difficulty to analyze the local minimum of sparse constrained CNNs. Based on the observation that, in practical implementations, the gradient of ReLU function at 0 is set to 0 [6,7,8]. We consider the following practical definition of ReLU:

$$ReLU(x) = \begin{cases} x & \text{if } x > \epsilon \\ 0 & \text{if } x \leq \epsilon. \end{cases} \tag{9}$$

Where ϵ is chosen such that for any real number x that a computer can represent, if $x > 0$, then $x > \epsilon$; if $x \leq 0$, then $x < \epsilon$. The non-differentiable point of ReLU function is now at ϵ which will never appear in practice. Under this definition, ReLU function is differentiable and continuous at point 0, the gradient of $ReLU(x)$ at 0 is now 0. As a result, this practical definition of ReLU is consistent with the implementations of ReLU function in practice [6,7,8].

By fixing all other neurons, it is not difficult to show that a particular neuron $\hat{\mathbf{w}}_{lj} = \mathbf{0}$ lies in a flat region of $\psi(\hat{\mathbf{w}}_{lj})$ under the above definition of ReLU. Moreover, $g(\hat{\mathbf{w}}_{lj})$ contains a sparse constrains, so $\hat{\mathbf{w}}_{lj} = \mathbf{0}$ is the local minimum of the objective function $\psi(\hat{\mathbf{w}}_{lj}) + g(\hat{\mathbf{w}}_{lj})$.[2]

The fact that $\hat{\mathbf{w}}_{lj} = \mathbf{0}$ is a local minimum for sparse constrained CNNs using ReLU as nonlinear activation function can explain the improvement in the number of neurons removed as discussed in Section 6.1. Importantly, we find that using momentum during the optimization may push a zero neuron away from being $\mathbf{0}$ since momentum memorizes the gradient of this neuron in previous steps. As a consequence, using momentum would lead to more non-zero neurons without performance improvement. In practice, once a neuron reaches $\mathbf{0}$, we set its momentum to zero, forcing the neuron to maintain its value. As we will demonstrate in Section 6.1 this results in more zero neurons without affecting the performance.

6 Experiments

We test our method on four well-known convolutional neural networks on three well-known datasets: LeNet on MNIST [28], CIFAR10-quick [36] on CIFAR-10 [37] and AlexNet [3] and VGG [2] on ILSVRC-2012 [38]. We use LeNet and CIFAR10-quick provided by Matconvnet [8] and AlexNet and VGG provided by [39] to carry out all our experiments.

All the structures of the networks are provided by Matconvnet [8] or [39], the only change we made is to add ReLU function after each convolutional layer

[2] Please find the detailed discussion in supplementary material.

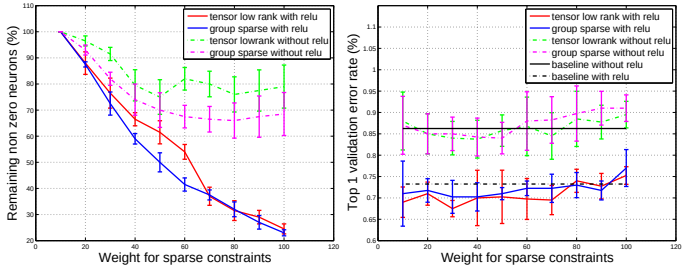


Fig. 1. (a) Percentage of nonzero neurons on the second convolutional layer for LeNet under different weights for sparse constraints with and without adding ReLU layer. (b) Corresponding top 1 validation error rate. Baseline in (b) shows the top 1 validation error rate without adding any sparse constraints. Error bars represent the standard deviation over the four experiments.

in LeNet, which we will discuss in detail later. For all our experiments, data augmentation and pre-processing strategies are provided by [8] and [39].

6.1 LeNet on MNIST

MNIST is a well-known dataset for machine learning and computer vision. It contains $60,000$ handwritten digits for training and $10,000$ for testing. All these digits are images of 28×28 with a single channel. All the data will be subtracted by the mean as suggested in [8]. The average top 1 validation error rate of LeNet on MNIST adding ReLU layer is 0.73%. As training a LeNet on MNIST is fast, we use this experiment as a sanity check. Results are computed as the average over four runs of each experiment using different random seeds.

ReLU helps removing more neurons: The LeNet structure provided by Matconvnet has two convolutional layers and two fully connected layers. The two convolutional layers are followed by a max pooling layer respectively. Under this structure, the sparse solution may not be a local minimum. Based on the discussion in Section 5, we add a ReLU layer after each of these two convolutional layers so that $\hat{\mathbf{w}}_{lj} = \mathbf{0}$ is a local minimum. We compare these two structures by using different weights for sparse constraints added to the second convolutional layer and show the results in Figure 1.

As shown in Figure 1, adding ReLU improves the performance of LeNet no matter whether we add sparse constraints or not. Comparing these two structures, adding ReLU layer always leads to more zero neurons compared with the original structure. What is more interesting is that, with ReLU layer, the top 1 validation error rate of LeNet with sparse constraints is more close to, if not better than, the performance without sparse constraints. This may be explained by the fact the $\hat{\mathbf{w}}_{lj} = \mathbf{0}$ is a local minimum of the structure with ReLU and this local minimum is usually a good one. In the following discussion, when LeNet is

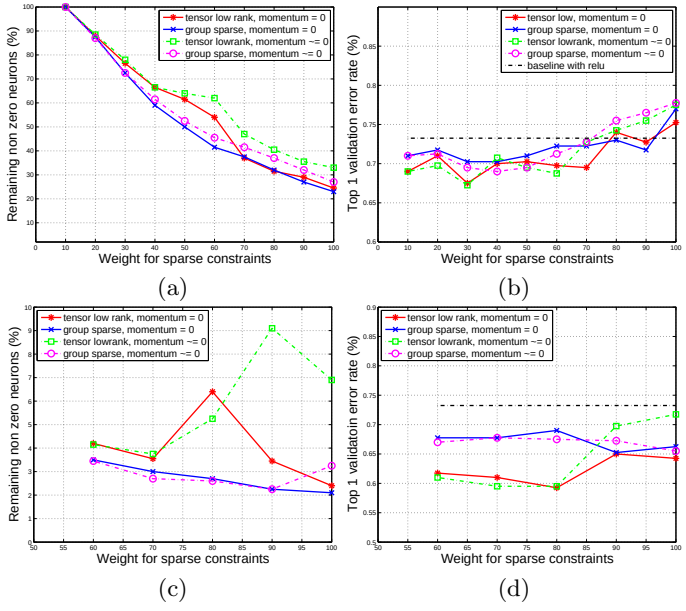


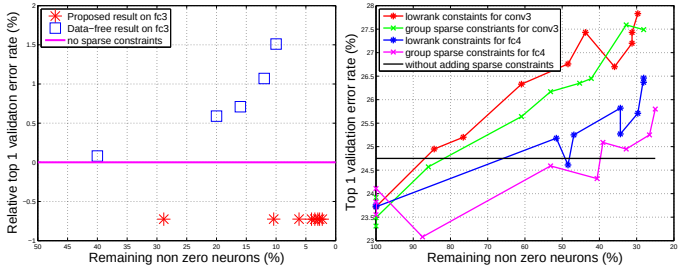
Fig. 2. Comparison of the proposed results with setting momentum to be zero. (a) and (c) show the percentage of non zero neurons for second convolutional layer and first fully connected layer under different weights for sparse constraints respectively. (b) and (d) show their corresponding top 1 validation error rate.

mentioned, we mean LeNet with ReLU function after each of its convolutional layers.

Momentum for sparse local minimum: Momentum plays a significant role in training CNNs. It can be treated as a memory of the gradient computed in previous iterations and has been proved to accelerate the convergence of SGD. However, this memory of gradient effect may push a neuron in its sparse local minimum away, leading to results with more non-zero neurons with no improvement in performance. To avoid this problem we can directly set the momentum of the neurons to be zero for those reached sparse local minimum.

In Figure 2, we compare the number of non-zero neurons with and without setting the momentum to be zero for LeNet on MNIST dataset. We add the sparse constraints on the second convolutional layer and the first fully connected layer since they have most of the filters. As shown in Figure 2, the performance does not drop when momentum is set to be zero for local sparse minimum. Additionally, we sporadically achieve better performance. Moreover, for first fully connected layer, setting momentum to be zero leads to results with more zero neurons under large sparse weights. [3]

[3] For a clear comparison, we only show results under large weights for the first fully connected layer in the figure.



(a) (b)

Fig. 3. a) **MNIST** Comparison between of single layer results of the proposed method and the data-free method presented in [1] using LeNet on MNIST. b) **CIFAR-10** Top 1 validation error rate versus percentage of remaining non zero neurons on conv3 and fc4 using CIFAR10-quick.

From Figure 1 and Figure 2, we find that the top-1 error rates of using tensor low-rank constraints are a little better than using group sparse constraints. Group sparse constraints, however, can give more zero neurons which can result in more compact networks. These differences are very small, so we conclude that both of these methods can help removing neurons without hurting the performance of the network. In the following discussion, only results from group sparse constraints are shown when necessary.

Compression for LeNet: Figure 3 (a) shows the comparison of the proposed method with [1]. Since [1] only adds sparse constraints on first fully connected layer of LeNet (fc3), we only show our results using group sparse constraints on fc3. The compression rate and top 1 validation error are averaged over 4 results with different random seeds. Since the two networks may be trained differently, we show the relative top 1 validation error rate, which is defined as top 1 validation error rate of the proposed method minus that without sparse constraints. A smaller value means better performance. As shown in this figure, our method outperforms the one proposed in [1] in both compression and accuracy. Please, note that [1] predicts the cut-off number of neurons for fc3 to be 440 and a drop of performance of 1.07%. The proposed method, on the contrary, can remove 489 neurons while maintaining the performance of the original one.

To improve the compression rate, we add group sparse constraints to the layers containing most of the parameters in LeNet: the second convolutional layer (conv2) and the first fully connected layer (fc3). We compare two strategies. First, adding sparse constraints in a layer by layer fashion and second, jointly constraining both layers. Empirically we found that the first strategy performs better. Thus, we first add sparse constraints on fc3 and, after the number of zero-neurons in this layer is stable, we add sparse constraints on conv2. We report the results of using group sparse constraints. The weight used for fc3 is set to 100, and the weights for conv2 are set to $60, 80, 100$. Table 3 summarizes the average results over the four runs of the experiment. As shown, our method not only reduces significantly the number of neurons, which leads to significant reduction

Table 2. Results of adding group sparse constraints on three layers.

τ			Non-zero Neurons			Number of	Top-1 error (%)
conv1	conv2	fc3	conv1	conv2	fc3	Parameters	
100	80	90	7	23	20	11820	0.72
100	120	80	7	13	21	7079	0.76
120	120	90	6	14	20	6980	0.81

in the number of parameters in these two layers but also compresses the total number of parameters of the network for more than 90%, leading to a memory footprint reduction larger than 1 MB.

We further try to add group sparse constraints on all three layers of LeNet to check whether our method can work on more layers. To introduce more redundancy on conv1 and conv2, we initialize the number of non zero neurons for conv1 and conv2 to be 100. Similar to adding sparse constraints on two layers, we add sparse constraints layer by layer. We show some of our results in Table 2. To compare, the best compression result of adding sparse constraints on conv2 and fc3 leads to a model with 13062 parameters (third row of LeNet in Table 2). We find that by adding sparse constraints on three layers, a more compact network can be achieved though we initialize the network with more neurons.

6.2 CIFAR-10 quick on CIFAR-10

CIFAR-10 [37] is a database consisting of $50,000$ training and $10,000$ testing RGB images with a resolution of 32×32 pixels split into 10 classes. As suggested in [8], data is standardized using zero-mean unit length normalization followed by a whitening process. Furthermore, we use random flips as data augmentation. For a fair comparison, we train the original network, CIFAR10-quick, without any sparse constraints using the same training set-up and achieve a top 1 validation error rate of 24.75%.

Since the third convolutional layer (conv3) and the first fully connected layer (fc4) contain most of the parameters, we add sparse constraints on these two layers independently. Figure 3 (b) shows the top 1 validation error rate versus the percentage of remaining non-zero neurons. As shown, 20% and 70% of neurons for conv3 and fc4 can be removed without a noticeable drop in performance.

To obtain the best compression results, we jointly constraint conv3 and fc4 as we did with LeNet. To this end, we first add the constraints on fc4, and then, we include the same ones on conv3. We run the experiment for more epochs compared to the default values in Matconvnet [8]. For a fair comparison, we train the baseline (i.e., CIFAR-10 quick without sparse constraints) for the same number of epochs. As a result, we obtain a top 1 validation error of 22.33%. We used this result to compute relative top 1 validation error rate. The weight of group sparse constraints for fc4 is fixed to 280 while the three different weights for conv3 are $220, 240, 280$. A summary of results is listed in the second part of Table 3. Through this experiment, it can be seen that even for this simple network,

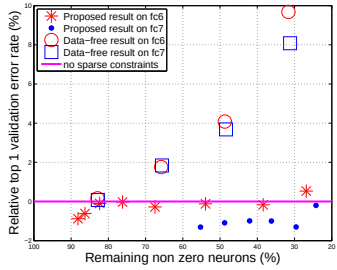


Fig. 4. ImageNet. Comparison between single layer results on fc6 and fc7 and the data-free method in [1] using AlexNet on ImageNet. Data-free results are from [1].

we can remove a large number of neurons which leads to a great compression in the number of parameters. Considering that there are only 64 neurons on each of these two layers, the compression results are significant.

6.3 AlexNet and VGG on ImageNet

ImageNet [40] is a dataset with over 15 million labeled images split into 22,000 categories. AlexNet [3] was proposed to be trained on ILSVRC-2012 [38] which is a subset of ImageNet with 1.2 million training images and 50,000 validation images. We use the implementation of AlexNet and VGG-13 provided by [39] in order to test ImageNet on our cluster. Random flipping is applied to augment the data. Quantitative results are reported using a single crop in the center of the image. For comparison, we consider the network trained without adding any sparse constraints. The top 1 validation error for this baseline is 45.57% and 37.50% for AlexNet and VGG-13 respectively. For the rest of experiments we only report results using group sparse constraints.

Figure 4 shows the top 1 validation error rate versus the percentage of non-zero neurons for AlexNet. Group sparse constraints are added on first and second fully connected layers (fc6 and fc7) independently. Results from [1] are copied from their paper and shown in this figure for comparison. Similar to Section 6.1, we show the relative top 1 validation error rates due to the training of two methods may be different. This figure clearly shows that compared with [1], the proposed method can remove a large number of neurons without decreasing the performance. For instance, the best compression results of the proposed method can eliminate 76.76% of all the parameters of AlexNet with a negligible drop in performance (0.57% in top 1 validation error). The best performance model in [1], on the other hand, can only remove 34.89% of the parameters with top 1 validation error rate decreased by 2.24%. A representative set of compression results obtained using sparse constraints on two layers is shown in the third part of Table 3.

We test the proposed method on VGG-13 on the first fully connected layer as it contains most of the parameters of the network. Table 4 summarizes the outcomes of the experiment for different group sparsity weights. As shown, for

Table 3. Results of adding group sparse constraints on two layers. The best compression results within 1% decrease in top 1 error rate is shown in bold.

	τ		Neurons pruned(%)		Top-1 error (%)		parameter	memory
	conv2	fc3	conv2	fc3	absolute	relative	reduction (%)	reduced (MB)
LeNet	60	100	45.5	97.75	0.73	0.00	95.35	1.57
	80	100	56.5	97.75	0.77	0.04	96.31	1.58
	100	100	63.0	97.75	0.76	0.03	**96.79**	**1.59**
	τ		Neurons pruned (%)		Top 1 error (%)		parameters	memory
	conv3	fc4	conv3	fc4	absolute	relative	reduction(%)	reduced (KB)
cifar10-quick	220	280	31.25	70.31	22.21	-0.12	47.17	268.24
	240	280	46.88	71.86	22.73	0.4	**55.15**	**313.62**
	280	280	54.69	70.31	23.78	1.45	58.56	333.01
	τ		Neurons pruned (%)		Top 1 error (%)		parameters	memory
	fc6	fc7	fc6	fc7	absolute	relative	reduction (%)	reduced(MB)
AlexNet	40	35	48.46	56.49	44.58	-0.98	55.15	128.26
	45	30	77.05	60.21	46.14	0.57	**76.76**	**178.52**
	45	35	73.39	65.80	45.88	0.31	74.88	174.14

Table 4. Some compression results of proposed method on fc1 for vgg-B. Neuron: compression of neurons in the fc1. Parameter: compression of total parameters.

layer	τ	compression %		memory	top 1 error (%)	
		neurons	parameters	reduced (MB)	absolute	relative
fc1	5	39.04	35.08	178.02	38.30	0.80
fc1	10	49.27	44.28	224.67	38.54	1.04
fc1	20	76.21	61.30	311.06	39.26	1.76

this state-of-the-art network structure, our method can reduce nearly half of the parameters and significantly reduce the memory footprint at the expenses of a slight drop in performance.

7 Conclusion

We proposed an algorithm to significantly reduce of the number of neurons in a convolutional neural network by adding sparse constraints during the training step. The forward-backward splitting method is applied to solve the sparse constrained problem. We also analyze the benefits of using rectified linear units as non-linear activation function to remove a larger number of neurons.

Experiments using four popular CNNs including AlexNet and VGG-B demonstrate the capacity of the proposed method to reduce the number of neurons, therefore, the number of parameters and memory footprint, with a negligible loss in performance.

Acknowledgment The authors thank NVIDIA for generous hardware donations.

References

1. Srinivas, S., Babu, R.V.: Data-free parameter pruning for deep neural networks. In: BMVC. (2015) 1, 3, 4, 11, 13
2. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. CoRR **abs/1409.1556** (2014) 1, 8
3. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: NIPS. (2012) 1, 7, 8, 13
4. Denton, E.L., Zaremba, W., Bruna, J., LeCun, Y., Fergus, R.: Exploiting linear structure within convolutional networks for efficient evaluation. In: NIPS. (2014) 2, 3
5. Jaderberg, M., Vedaldi, A., Zisserman, A.: Speeding up convolutional neural networks with low rank expansions. In: BMVC. (2014) 2, 3
6. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093 (2014) 2, 8
7. Collobert, R., Kavukcuoglu, K., Farabet, C.: Torch7: A matlab-like environment for machine learning. In: BigLearn, NIPS Workshop. (2011) 2, 8
8. Vedaldi, A., Lenc, K.: Matconvnet – convolutional neural networks for matlab. In: ACM MM 2, 8, 9, 12
9. Mathieu, M., Henaff, M., LeCun, Y.: Fast training of convolutional networks through ffts. CoRR **abs/1312.5851** (2013) 2
10. Girshick, R., Donahue, J., Darrell, T., Malik, J.: Rich feature hierarchies for accurate object detection and semantic segmentation. In: CVPR. (2014) 2
11. Chen, L., Papandreou, G., Kokkinos, I., Murphy, K., Yuille, A.L.: Semantic image segmentation with deep convolutional nets and fully connected crfs. In: ICLR. (2014) 2
12. Duchi, J., Singer, Y.: Efficient online and batch learning using forward backward splitting. Journal of Machine Learning Research **10** (2009) 2, 5, 6
13. Goldstein, T., Studer, C., Baraniuk, R.G.: A field guide to forward-backward splitting with a FASTA implementation. arXiv eprint **abs/1411.3406** (2014) 2, 5, 6
14. Bucila, C., Caruana, R., Niculescu-Mizil, A.: Model compression. In: ACM SIGKDD. (2006) 3
15. Ba, J., Caruana, R.: Do deep nets really need to be deep? In: NIPS. (2014) 3
16. Hinton, G.E., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network. In: NIPS 2014 Deep Learning Workshop. (2014) 3
17. Romero, A., Ballas, N., Kahou, S.E., Chassang, A., Gatta, C., Bengio, Y.: Fitnets: Hints for thin deep nets. In: ICLR. (2015) 3
18. Denil, M., Shakibi, B., Dinh, L., Ranzato, M., Freitasa, N.D.: Predicting parameters in deep learning. In: NIPS. (2013) 3
19. Liu, B., Wang, M., Foroosh, H., Tappen, M., Penksy, M.: Sparse convolutional neural networks. In: CVPR. (2015) 3
20. Lebedev, V., Ganin, Y., Rakhuba, M., Oseledets, I.V., Lempitsky, V.S.: Speeding-up convolutional neural networks using fine-tuned cp-decomposition. In: ICLR. (2015) 4
21. Gong, Y., Liu, L., Yang, M., Bourdev, L.D.: Compressing deep convolutional networks using vector quantization. CoRR **abs/1412.6115** (2014) 4
22. Novikov, A., Podoprikhin, D., Osokin, A., Vetrov, D.P.: Tensorizing neural networks. In: NIPS. (2015) 4

23. LeCun, Y., Denker, J.S., Solla, S.A.: Optimal brain damage. In: NIPS. (1990) 4
24. Hassibi, B., Stork, D.G.: Second order derivatives for network pruning: Optimal brain surgeon. In: NIPS. (1993) 4
25. Han, S., Pool, J., Tran, J., Dally, W.J.: Learning both weights and connections for efficient neural networks. In: NIPS. (2015) 4
26. Collins, M.D., Kohli, P.: Memory bounded deep convolutional networks. CoRR **abs/1412.1442** (2014) 4, 5
27. Han, S., Mao, H., Dally, W.J.: Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In: ICLR. (2016) 4
28. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11) (1998) 5, 8
29. Yu, D., Seide, F., Li, G., Deng, L.: Exploiting sparseness in deep neural networks for large vocabulary speech recognition. In: ICASSP. (2012) 5, 6
30. Tsuruoka, Y., Tsujii, J., Ananiadou, S.: Stochastic gradient descent training for l1-regularized log-linear models with cumulative penalty. In: ACL-IJCNLP. (2009) 5, 6
31. Liu, J., Musialski, P., Wonka, P., Ye, J.: Tensor completion for estimating missing values in visual data. IEEE Transactions on PAMI **35**(1) (2013) 6, 7
32. Deng, W., Yin, W., Zhang, Y.: Group sparse optimization by alternating direction method. In: SPIE. Volume 8858. (2013) 6, 7
33. Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: ICML. (2010) 8
34. Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. In: AISTATS. (2011) 8
35. Zeiler, M.D., Ranzato, M., Monga, R., Mao, M., Yang, K., Le, Q., Nguyen, P., Senior, A., Vanhoucke, V., Dean, J., Hinton, G.: On rectified linear units for speech processing. In: ICASSP. (2013) 8
36. Snoek, J., Larochelle, H., Adams, R.P.: Practical bayesian optimization of machine learning algorithms. In: NIPS. (2012) 8
37. Krizhevsky, A.: Learning multiple layers of features from tiny images. Technical report, Technical report, Department of Computer Science, University of Toronto (2009) 8, 12
38. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L.: Imagenet large scale visual recognition challenge. IJCV (2015) 8, 13
39. Chintala, S.: soumith/imagenet-multigpu.torch. url=https://github.com/soumith/imagenet-multiGPU.torch (2015) 8, 9, 13
40. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: CVPR. (2009) 13