Compressing Convolutional Neural Networks in the Frequency Domain

Wenlin Chen
Department of Computer
Science and Engineering
Washington University in St.
Louis
wenlinchen@wustl.edu

James Wilson
University of Edinburgh
jwilson1020@gmail.com

Stephen Tyree
NVIDIA
Santa Clara, CA, USA
styree@nvidia.com

Kilian Q. Weinberger
Department of Computer
Science
Cornell University
kqw4@cornell.edu

Yixin Chen
Department of Computer
Science and Engineering
Washington University in St.
Louis
chen@cse.wustl.edu

ABSTRACT

Convolutional neural networks (CNN) are increasingly used in many areas of computer vision. They are particularly attractive because of their ability to "absorb" great quantities of labeled data through millions of parameters. However, as model sizes increase, so do the storage and memory requirements of the classifiers, hindering many applications such as image and speech recognition on mobile phones and other devices. In this paper, we present a novel network architecture, Frequency-Sensitive Hashed Nets (FreshNets), which exploits inherent redundancy in both convolutional layers and fully-connected layers of a deep learning model, leading to dramatic savings in memory and storage consumption. Based on the key observation that the weights of learned convolutional filters are typically smooth and low-frequency, we first convert filter weights to the frequency domain with a discrete cosine transform (DCT) and use a low-cost hash function to randomly group frequency parameters into hash buckets. All parameters assigned the same hash bucket share a single value learned with standard back-propagation. To further reduce model size, we allocate fewer hash buckets to high-frequency components, which are generally less important. We evaluate FreshNets on eight data sets, and show that it leads to better compressed performance than several relevant baselines.

CCS Concepts

•Information systems → Data mining; •Computing methodologies → Supervised learning; Neural networks;

Keywords

Model compression; convolutional neural networks; hashing

1. INTRODUCTION

In the recent years convolutional neural networks (CNN) have led to impressive results in image classification [15, 30, 48, 63], object detection [19, 58], image retrieval [43], image caption generation [26, 37, 58], face verification [47, 55], video understanding [27] and audio classification [35]. Problems that seemed impossibly hard only five years ago can now be solved at better than human accuracy [24]. Although CNNs have been known for a quarter of a century [17], only recently have their superb generalization abilities been accepted widely across the machine learning and computer vision communities. This broad acceptance coincides with the release of very large collections of labeled data [12]. Deep networks and CNNs are particularly well suited to learn from large quantities of data, in part because they can have arbitrarily many parameters. As data sets grow, so do model sizes. In 2012, the first winner of the ImageNet competition that used a CNN had already 240MB of parameters and the most recent winning model, in 2014, required 567MB [50].

Independently, there has been another parallel shift of computing from servers and workstations to mobile platforms. As of January 2014 there have already been more web searches through smart phones than computers[1]. Today speech recognition is primarily used on cell phones with intelligent assistants such as Apple's Siri, Google Now or Microsoft's Cortana. As this trend continues, we are expecting machine learning applications to also shift increasingly towards mobile devices. However, the disjunction of deep learning with ever increasing model sizes and mobile computing reveals an inherent dilemma. Mobile devices have tight memory and storage limitations. For example, even the most recent iPhone 6 only features 1GB of RAM, most of which must be used by the operating system or the application itself. In addition, developers must make their apps compatible with the most limited phone still in circulation, often restricting models to just a few megabytes of parameters. In addition to memory limitations, there are also low-power restrictions. Loading 500MB of parameters for a deep network from SSD drive to memory, to maybe just classify a single image, requires a significant amount of energy.

In response, there has been a recent interest in reducing the model sizes of deep networks. Denil et al. [13] use low-rank decomposi-

---

[1] http://tinyurl.com/omd58sq

tion of the weight matrices to reduce the effective number of parameters in the network. Bucilua et al. [5] and Ba et al. [2] show that complex models can be compressed into 1-layer neural networks. Independently, the model size of neural networks can be reduced effectively through reduced bit precision [9]. Meanwhile, the conventional wisdom that fully connected layers contain more parameters is no longer true. More and more parameters are shifting towards convolutional layers. For example, *GoogleNet* [54] contains 11 layers involving convolutions[2] with only one single fully connected layer on the top. As a result, 85% of the parameters lie in the convolutional layers. A recent work proposed by Long *et al.* [36] advocates fully convolutional training which replaces the fully connected layers with the convolutional layer. All of these make the compression of the convolutional layers a must.

In this paper we propose a novel approach for neural network compression targeted especially for CNNs. We build on recent work by Chen et al. [6], who show that weights of fully connected networks can be effectively compressed with the hashing trick [60]. Due to the nature of local pixel correlation in images (*i.e.* spatial locality), filters in CNNs tend to be smooth. We transform these filters into frequency domain with the discrete cosine transform (DCT) [42]. In frequency space, the filters are naturally dominated by low frequency components. Our compression takes this smoothness property into account and randomly hashes the frequency components of all CNN filters at a given layer into one common set of hash buckets. All components inside one hash bucket share the same value. As lower frequency components are more pronounced than higher frequencies, we allow collisions only between similar frequencies and allocate fewer hash buckets for the high frequencies (which are less important).

Our approach has several compelling properties: 1. The number of parameters in the CNN is *independent* of the number of convolutional filters; 2. During testing we only need to add a low-cost hash function and the inverse DCT transformation to any existing CNN code for filter reconstruction; 3. During training, the hashed weights can be learned with simple back-propagation [3]—the gradient of a hash bucket value is the sum of gradients of all hashed frequency components in that bucket.

We evaluate our compression scheme on eight deep learning image benchmark data sets and compare against four competitive baselines. Although all compression schemes lead to lower test accuracy as the compression increases, our FreshNets method is by far the most effective compression method and yields the lowest generalization error rates on almost all tested tasks.

The rest of the paper is organized as follows. Section 2 introduces background on feature hashing and discrete cosine transformation. We describe FreshNets in Section 3 and review the literature in Section 5. Experimental results are presented in Section 6. Section 7 draws the conclusion.

2. BACKGROUND

2.1 Feature Hashing

Feature Hashing (*a.k.a* the hashing trick) [11, 49, 60] has been previously studied as a technique for reducing model storage size. In general, it can be regarded as a dimensionality reduction method that maps an input vector $\boldsymbol{x} \in \mathbb{R}^d$ to a much smaller feature space via a mapping $\phi \colon \mathbb{R}^d \to \mathbb{R}^k$ where $k \ll d$. The mapping ϕ is a composite of two approximately uniform auxiliary hash functions $h \colon \mathbb{N} \to \{1, \ldots, k\}$ and $\xi \colon \mathbb{N} \to \{-1, +1\}$. The jth element of the

---

[2] We count the *inception* layers as convolutional layers as they consist of multi-scale convolutional filters.

---

k-dimensional hashed input is defined as

$$\phi_j(\boldsymbol{x}) = \sum_{i:h(i)=j} \xi(i)\, x_i.$$

As shown in [60], a key property of feature hashing is its preservation of inner product operations, where inner products after hashing produce the correct pre-hash inner product in expectation:

$$\mathbb{E}[\phi(\boldsymbol{x})^\top \phi(\boldsymbol{y})]_\phi = \boldsymbol{x}^\top \boldsymbol{y}.$$

This property holds because of the bias correcting sign factor $\xi(i)$. With feature hashing, models are directly learned in the much smaller space $\mathbb{R}^k$, which not only speeds up training and evaluation but also significantly conserves memory. For example, a linear classifier in the original space could occupy $O(d)$ memory for model parameters, but when learned in the hashed space only requires $O(k)$ parameters. The information loss induced by hash collision is much less severe for sparse feature vectors and can be counteracted through multiple hashing [49] or larger hash tables [60].

2.2 Discrete Cosine Transform (DCT)

Methods built on the DCT [1, 42] are widely used for compressing images and movies, including forming the standard technique for JPEG [41, 59]. DCT expresses a function as a weighted combination of sinusoids of different phases/ frequencies where the weight of each sinusoid reflects the magnitude of the corresponding frequency in the input. When employed with sufficient numerical precision and without quantization or other compression operations, the DCT and inverse DCT (projecting frequency inputs back to the spatial domain) are lossless. Compression is made possible in images by local smoothness of pixels (*e.g.* a blue sky) which can be well represented regionally by fewer non-zero frequency components. Though highly related to the discrete Fourier transformation (DFT), DCT is often preferable for compression tasks because of its *spectral compaction* property where weights for most images tend to be concentrated in a few low-frequency components of the DCT [42]. Further, the DCT transformation yields a real-valued representation, unlike the DFT whose representation has imaginary components. Given an input matrix $\boldsymbol{V} \in \mathbb{R}^{d \times d}$, the corresponding matrix $\boldsymbol{\mathcal{V}} \in \mathbb{R}^{d \times d}$ in frequency domain after DCT is defined as:

$$\mathcal{V}_{j_1 j_2} = s_{j_1} s_{j_2} \sum_{i_1=0}^{d-1} \sum_{i_2=0}^{d-1} c(i_1, i_2, j_1, j_2)\, V_{i_1 i_2}, \qquad (1)$$

where

$$c(i_1, i_2, j_1, j_2) = \cos\left[\frac{\pi}{d}\left(i_1 + \frac{1}{2}\right) j_1\right] \cos\left[\frac{\pi}{d}\left(i_2 + \frac{1}{2}\right) j_2\right]$$

is the cosine basis function, and $s_j = \sqrt{\frac{1}{d}}$ when $j = 0$ and $s_j = \sqrt{\frac{2}{d}}$ otherwise. We use the shorthand f_{dct} to denote the DCT operation in Eq. (1), *i.e.* $\boldsymbol{\mathcal{V}} = f_{dct}(\boldsymbol{V})$. The inverse DCT converts $\boldsymbol{\mathcal{V}}$ from the frequency domain back to the spatial domain, reconstructing $\boldsymbol{V}$ without loss:

$$V_{i_1 i_2} = \sum_{j_1=0}^{d-1} \sum_{j_2=0}^{d-1} s_{j_1} s_{j_2}\, c(i_1, i_2, j_1, j_2)\, \mathcal{V}_{j_1 j_2}. \qquad (2)$$

We denote the inverse DCT function in Eq. (2) as f_{dct}^{-1}, *i.e.* $\boldsymbol{V} = f_{dct}^{-1}(\boldsymbol{\mathcal{V}})$.

3. Frequency-Sensitive Hashed Nets

Here we present FreshNets, a method for using weight sharing to reduce the model size (and memory demands) of convolutional
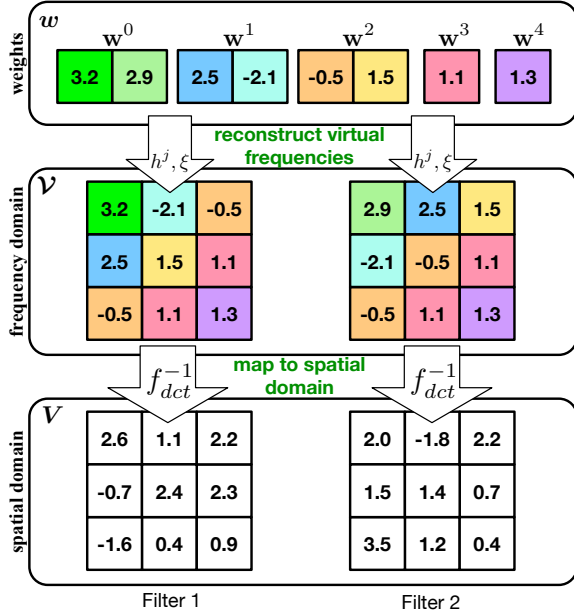
Figure 1: A schematic illustration of FreshNets. Two spatial filters are re-constructed from the frequency weights in vector w. The frequency weights are accessed with two hash functions and then transformed to the spatial domain. The vector w is partitioned into sub-vectors w^j shared by all entries with similar frequency (corresponding to index sum $j = j_1 + j_2$). Colors indicate which hash bucket was accessed.

neural networks. Similar to the work of Chen et al. [6], we achieve smaller models by randomly forcing weights throughout the network to share identical values. Unlike previous work, we implement the weight sharing and gradient updates of convolutional filters in the *frequency domain*. These sharing constraints are made prior to training, and we learn frequency weights under the sharing assignments. Since the assignments are made with a hash function, they incur no additional storage.

We begin by deriving the equivalent filter representation after the DCT, and describe an efficient random weight-sharing scheme in the frequency space implemented using the hashing trick. Next we show how to learn the parameters in the frequency domain with standard back-propagation. At last, we describe a scheme to take advantage of filter smoothness by allocating more shared weights to low frequency components.

3.1 Filters in spatial and frequency domain

Let the matrix $V^{k\ell} \in \mathbb{R}^{d \times d}$ denote the weight matrix of the $d \times d$ convolutional filter that connects the kth input plane to the ℓth output plane. (For notational convenience we assume square filters and only consider the filters in a single layer of the network.) The weights of all filters in a convolutional layer can be denoted by a 4-dimensional tensor $V \in \mathbb{R}^{m \times n \times d \times d}$ where m and n are the number of input planes and output planes, respectively, resulting in a total of $m \times n \times d^2$ parameters. Convolutional filters can be represented equivalently in either the spatial or frequency domain, mapping between the two via the DCT and its inverse. We denote the filter in frequency domain as

$$\mathcal{V}^{k\ell} = f_{dct}(V^{k\ell}) \in \mathbb{R}^{d \times d} \qquad (3)$$

and recover the original spatial representation through

$$V^{k\ell} = f_{dct}^{-1}(\mathcal{V}^{k\ell}) \in \mathbb{R}^{d \times d}, \qquad (4)$$

as defined in Eq. (1) and (2), respectively. The tensor of all filters is denoted $\mathcal{V} \in \mathbb{R}^{m \times n \times d \times d}$.

3.2 Random Weight Sharing by Hashing

After the lossless conversion via DCT, filters in frequency domain $\mathcal{V}^{k\ell}$ remain the same size as equivalent filters in the spatial domain $V^{k\ell}$. We propose to use weight sharing to reduce the number of parameters in the frequency domain. With filters in a frequency representation, $\mathcal{V}^{k\ell}$, we would like to reduce the number of model parameters to exactly K values stored in a weight vector $w \in \mathbb{R}^K$, where $K \ll m \times n \times d^2$. To achieve this, we randomly assign a value from w to each filter frequency weight in $\mathcal{V}$. A naïve implementation of this random weight sharing would introduce an auxiliary matrix for $\mathcal{V}$ to track the weight assignments, using to significant additional memory. To address this problem, Chen et al. [6] advocate use of the hashing trick to (pseudo-)randomly assign shared parameters. Using the hashing trick, we tie each filter weight $\mathcal{V}^{k\ell}_{j_1 j_2}$ to an element of w indexed by the output of a hash function $h(\cdot)$:

$$\mathcal{V}^{k\ell}_{j_1,j_2} = \xi(k,\ell,j_1,j_2)\, w_{h(k,\ell,j_1,j_2)}, \qquad (5)$$

where $h(k,\ell,j_1,j_2) \in \{1, \cdots, K\}$, and $\xi(k,\ell,j_1,j_2) \in \{\pm 1\}$ is a sign factor computed by a second hash function $\xi(\cdot)$ to preserve inner-products in expectation as described in Section 2. With the mapping in Eq. (5), we can implement shared parameter assignments with no additional storage cost. For a schematic illustration, see Figure 1. The figure also incorporates a frequency sensitive hashing scheme discussed later in this section. Note that the same K weights in w are shared across all filters in the convolutional layer. This way, we compress the whole $\mathcal{V}^{k\ell}$ which contains mnd^2 parameters into a K-dimensional weight vector w. In other words, adding more convolutional filters does not change the model size, as all filter values are "recycled" from already existing filters. We can arbitrarily control the number of effective parameters in each layer simply by adjusting K.

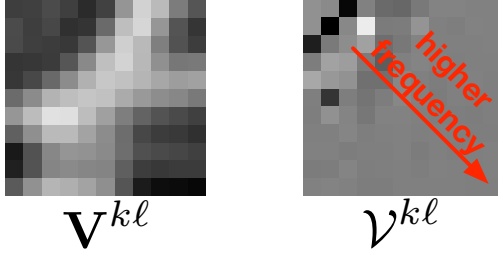## 3.3 Gradients over Shared Frequency Weights

Typical convolutional neural networks learn filters in the spatial domain. As our shared weights are stored in the frequency domain, we derive the gradient with respect to the filter parameters in the frequency space. Following Eq. (2), we express the gradient of parameters in the spatial domain with respect to their counterparts in the frequency domain:

$$\frac{\partial V^{k\ell}_{i_1 i_2}}{\partial \mathcal{V}^{k\ell}_{j_1 j_2}} = s_{j_1} s_{j_2}\, c(i_1, i_2, j_1, j_2). \qquad (6)$$

Let $\mathcal{L}$ be the loss function adopted for training. Using standard back-propagation, we can derive the gradient with respect to the filter parameters in the spatial domain, $\frac{\partial \mathcal{L}}{\partial V^{k\ell}_{i_1 i_2}}$. By the chain rule with Eq. (6), we express the gradient of $\mathcal{L}$ in the frequency domain:

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathcal{V}^{k\ell}_{j_1 j_2}} &= \sum_{i_1=0}^{d-1} \sum_{i_2=0}^{d-1} \frac{\partial \mathcal{L}}{\partial V^{k\ell}_{i_1 i_2}} \frac{\partial V^{k\ell}_{i_1 i_2}}{\partial \mathcal{V}^{k\ell}_{j_1 j_2}} \\
&= s_{j_1} s_{j_2} \sum_{i_1=0}^{d-1} \sum_{i_2=0}^{d-1} c(i_1, i_2, j_1, j_2)\, \frac{\partial \mathcal{L}}{\partial V^{k\ell}_{i_1 i_2}}.
\end{aligned}
\qquad (7)
$$

Comparing with Eq. (1), we see that the gradient in the frequency

Figure 2: An example of a filter in spatial (left) and frequency domain (right).

Figure 3: This figure illustrates the frequency domain of all filters. Cells in the same frequency region share the same color. Each frequency region is an antidiagonal stripe across all $m \times n$ filters.

domain is merely the DCT of the gradient in the spatial domain:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\mathcal{V}}^{k\ell}} = f_{dct}\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{V}^{k\ell}}\right). \tag{8}$$

In a nutshell, there is a straightforward procedure for learning filter parameters $\boldsymbol{\mathcal{V}}^{k\ell}$ in the frequency domain. In the feedforward phase, we reconstruct spatial domain weights $\boldsymbol{V}^{k\ell}$ with the inverse DCT transformation $f_{dct}^{-1}(\boldsymbol{\mathcal{V}}^{k\ell})$. During back-propagation, after computing the traditional gradient $\frac{\partial \mathcal{L}}{\partial \boldsymbol{V}^{k\ell}}$ in the spatial domain, we use the inverse DCT to compute the gradient $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\mathcal{V}}^{k\ell}}$ in the frequency domain (8). These steps can be efficiently implemented using off-the-shelf DCT software.

Note that $\boldsymbol{\mathcal{V}}^{k\ell}$ are virtual parameters and we still need to compute the gradient over the real weight vector $\boldsymbol{w} \in \mathbb{R}^K$. We compute gradient for each shared weight w_i by simply summing over the gradient at each filter parameter where the weight is assigned, *i.e.* all $\mathcal{V}_{j_1 j_2}^{k\ell}$ where $i = h(k, \ell, j_1, j_2)$, up to the sign factor $\xi(k, \ell, j_1, j_2)$:

$$\frac{\partial \mathcal{L}}{\partial w_i} = \sum_{k=0}^{m} \sum_{\ell=0}^{n} \sum_{j_1=0}^{d-1} \sum_{j_2=0}^{d-1} \frac{\partial \mathcal{L}}{\partial \mathcal{V}_{j_1 j_2}^{k\ell}} \frac{\partial \mathcal{V}_{j_1 j_2}^{k\ell}}{\partial w_h} \tag{9}$$

$$= \sum_{\substack{k,\ell,j_1,j_2: \\ i=h(k,\ell,j_1,j_2)}} \xi(k,\ell,j_1,j_2) \left[f_{dct}\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{V}^{k\ell}}\right) \right]_{j_1 j_2} \tag{10}$$
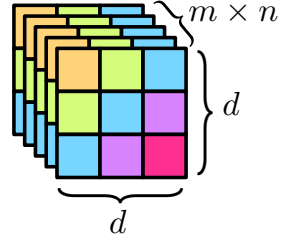
where $[\boldsymbol{A}]_{j_1 j_2}$ denotes the (j_1, j_2) entry in matrix $\boldsymbol{A}$.

3.4 Frequency Sensitive Hashing

Figure 2 shows a filter in spatial (left) and frequency (right) domains. In the spatial domain CNN filters are smooth [30] due to the local pixel smoothness in natural images. In the frequency domain this corresponds to components with large magnitudes in the low frequencies, depicted in the upper left half of $\boldsymbol{\mathcal{V}}^{k\ell}$ in Figure 2, with small indices (j_1, j_2). Correspondingly, the high frequencies, in the bottom right half of $\boldsymbol{\mathcal{V}}^{k\ell}$, with large indices (j_1, j_2), have magnitudes near zero.

As stated earlier, the filters in the spatial domain $\boldsymbol{V}^{k\ell}$ are typically "smooth" and in the frequency domain most weight intensities are in the low-frequency regions. The low frequency regions correspond to entries with small indices (j_1, j_2) and the low frequency entries typically have much larger norms than higher frequency values (entries with larger indices (j_1, j_2).

As components of different frequency regions tend to be of different magnitudes (and thereby varying importance to the spatial structure of the filter), we want to avoid collisions between high and low frequency components. Therefore, we assign separate hash spaces to different frequency regions. As shown in Figure 3, each frequency region owns a separate hash space. In particular, we partition the K values of $\boldsymbol{w}$ into sub-vectors $\boldsymbol{w}^0, \dots, \boldsymbol{w}^{2d-2}$ of sizes

$K_0, \dots, K_{2d-2}$, where $\sum_j K_j = K$. This partitioning allows parameters with the same frequency, corresponding to their index sum $j = j_1 + j_2$, to be hashed into a corresponding dedicated hash space $\boldsymbol{w}^j$, as shown in Figure 4. We rewrite Eq. (5) with the new frequency sensitive shared weight assignments:

$$\mathcal{V}_{j_1,j_2}^{k\ell} = \xi(k,\ell,j_1,j_2) \, w_{h^j(k,\ell,j_1,j_2)}^j \tag{11}$$

where $h^j(\cdot)$ maps an input key to a natural number in $\{1, \cdots, K_j\}$ and $j = j_1 + j_2$. With frequency-sensitive hashing, the gradient with respect to the real weight is the same as the one in Eq (10) except the hash function h is replaced with a new hash function g as follows:

$$\frac{\partial \mathcal{L}}{\partial w_i} = \sum_{\substack{k,\ell,j_1,j_2: \\ i=g(k,\ell,j_1,j_2)}} \xi(k,\ell,j_1,j_2) \left[f_{dct}\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{V}^{k\ell}}\right) \right]_{j_1 j_2} \tag{12}$$

Here, the hash function g is defined as

$$g(k,\ell,j_1,j_2) = h^j(k,\ell,j_1,j_2) + A_{j-1} \tag{13}$$

where $j = j_1 + j_2$ is the index of a frequency region, and $A_j = \sum_{p=0}^{j} K_p$ is a cumulative sum of the size of the first j hash spaces. The new hash function allows each hash space to be independent from each other. Though there seem to be $2d - 1$ number of hash functions, all h^js can be implemented by a single hash function $\hat{h}$ that takes 5 input arguments with the last argument being j, as follows:

$$h^j(k,\ell,j_1,j_2) = \hat{h}(k,\ell,j_1,j_2,j)$$

.

We define a compression rate $r_j \in (0, 1]$ for each frequency region j and assign $K_j = r_j N_j$ where N_j is the number of virtual parameters in the jth frequency regions. A smaller r_j induces more collisions during hashing, leading to increased weight sharing. Since lower frequency components tend to be of higher importance, making collisions more hurtful, we commonly assign larger r_j (fewer collisions) to low-frequency regions. Intuitively, given a size budget for the whole convolutional layer, we want to squeeze the hash space of high frequency region to save space for low frequency regions. These compression rates can either be assigned by hand or determined programmatically by cross-validation, as demonstrated in Section 6.

4. OVERVIEW

In a nutshell, we summarize the training procedure of FreshNets as follows.
Initialization phase:
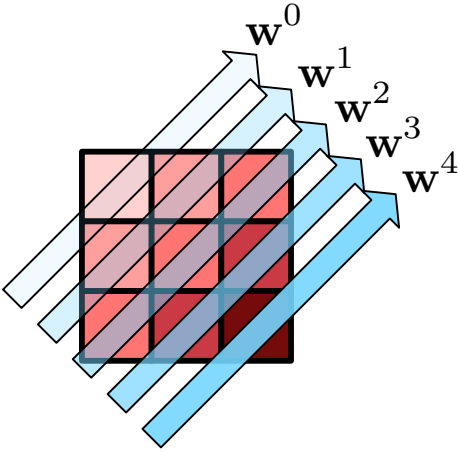
Figure 4: This figure illustrates the frequency domain of a filter. The redder a cell is, the higher frequency. Based on the index sum $j = j_1 + j_2$, the real weight vector w is divided into $2d - 1$ subvectors w^j, each of which takes charge of a frequency region.

- Given the filter size d and an overall budget for the number of parameters K, determine the budget for each frequency region K_j such that $\sum_{j=0}^{2d-2} K_j = K$.

- Randomly initialize weight vector w which consists of $2d-1$ number of sub-vectors w^j.

Feedforward phase:

- Construct $\boldsymbol{\mathcal{V}}^{k\ell}$ according to Eq (11).

- Use inverse DCT to convert $\boldsymbol{\mathcal{V}}^{k\ell}$ to its spatial domain $\boldsymbol{V}^{k\ell}$ according to (4).

- Use $\boldsymbol{V}^{k\ell}$ to perform convolution on the inputs, and generate outputs.

Backpropagation phase:

- Compute the gradient *w.r.t.* the convolutional filters $\boldsymbol{V}^{k\ell}$ in the spatial domain using normal backpropagation.

- Compute the gradient *w.r.t.* $\boldsymbol{\mathcal{V}}^{k\ell}$ using DCT according to (8).

- Compute the gradient *w.r.t.* the real weight vector w according to Eq (12) and update w using any gradient descent methods such as SGD with momentum [53], Adagrad [16] and RMSprop [56].

All the above training operations can be implemented in existing CNN packages by modifying the computational procedure for back propagation.

5. RELATED WORK

Several recent studies have confirmed that there is significant redundancy in the parameters learned in deep neural networks. Recent work by Denil et al. [13] learns parameters in fully-connected layers after decomposition into two low-rank matrices, *i.e.* $\boldsymbol{W} = \boldsymbol{AB}$ where $\boldsymbol{W} \in \mathbb{R}^{m \times n}$, $\boldsymbol{A} \in \mathbb{R}^{m \times k}$ and $\boldsymbol{B} \in \mathbb{R}^{k \times n}$. In this way, the original $O(mn)$ parameters could be stored with $O(k(m+n))$ storage, where $k \ll \min(m, n)$.

Several works apply related approaches to speed up the evaluation time with convolutional neural networks. Two works propose to approximate convolutional filters by a weighted linear combination of basis filters [25, 45]. In this setting, the convolution operation only needs to be performed with the small set of basis filters instead of all the filters. The desired output feature maps are computed by matrix multiplication as the weighted sum of these basis convolutions.

Further speedup can be achieved by learning rank-one basis filters so that the convolution operations are very cheap to compute [14, 33]. Based on this idea, Denton *et al.* [14] advocate decomposing the four-dimensional tensor of the filter weights into a sum of different rank-one, four-dimensional tensors and show some encouraging results. In addition, they adopt bi-clustering to group filters such that each subgroup can be better approximated by rank-one tensors. Courbariaux *et al.* [10] introduce *BinaryConnect* that enforces weights in neural networks to take on binary values. This replaces many multiply-accumulate operations by simple accumulations, leading to less power-hungry and fast computation.

There is a distinctive difference between FreshNets and the above works. In each of the above works, evaluation time is the main focus, with any resulting storage reduction achieved merely as a side effect. However, with the trend toward architectures with fewer fully connected layers and additional convolutional layers [54], compression of filters is of increasing importance.

Another technique for speeding up convolutional neural network evaluation is computing convolutions in the Fourier frequency domain, as convolution in the spatial domain is equivalent to (comparatively lower-cost) element-wise multiplication in the frequency domain [38, 57]. Unlike FreshNets, for a filter of size $d \times d$ and an image of size $n \times n$ where $n > d$, Mathieu et al. [38] convert the filter to its frequency domain of size $n \times n$ by oversampling the frequencies, which is necessary for doing element-wise multiplication with a larger image but also increases the memory overhead at test time. Training in the Fourier frequency domain may be advantageous for similar reasons, particularly when convolutions are being performed over large 3-D volumes [4].

Other works focus entirely on compressing the fully-connected layers of CNNs. A branch of these works propose to post-process a trained convolutional net. Leveraging the similarity between connections, Gong *et al.* [20] learns a convolutional net in advance, and then applies *kmeans* clustering on the weight values for quantization. These clusters form a smaller-size codebook for all the weights. With similar spirit, another work by Han *et al.* [23] recursively train a neural network and prune unimportant connections based on their weight magnitude. Han *et al.* [22] further combine the techniques of pruning and quantization to achieve more compression.

A few recent works [52, 61] also adopt the quantization techniques. However, with quantization, each connection still needs to store the index to the cluster it belongs to, which limits its potential for compression. Other works focus on matrix or tensor decomposition for compression. For example, Yang *et al.* [62] adopt the *fastfood* transformation [32] for compressing the fully connected layers, and Kim *et al.* [28] investigate general tensor decomposition with rank selection for compressing the entire network.

Most relevant to this work is HashedNets [6] which compresses the fully connected layers of deep neural networks. This method uses the hashing trick to efficiently implement parameter sharing prior to learning, achieving notable compression with less loss of accuracy than the competing baselines which relied on low-rank decomposition or learning in randomly sparse architectures.

Another recent work by Rippel *et al.* [46] proposes to learn weight

parameters in the frequency domain for faster convergence of learning, which is a fundamentally different goal compared with Fresh-Nets. In addition, they adopts discrete Fourier transformation (DFT) [8] while FreshNets uses DCT.

6. EXPERIMENTAL RESULTS

In this section, we conduct several comprehensive experiments on several benchmark datasets to evaluate the compression performance of FreshNets.

6.1 Datasets

We experiment with eight benchmark datasets: CIFAR10 [29], CIFAR100 [29], SVHN [39] and five challenging variants of MNIST [31, 34, 44]. The CIFAR10 dataset contains 60000 images of 32×32 pixels with three color channels. Images are selected from ten classes with each class consisting of 6000 unique instances.

The CIFAR100 dataset also contains 60000 32×32 images, but is more challenging since the images are selected from 100 classes (each class has 600 images). For both CIFAR datasets, 50000 images are designated for training and the remaining 10000 images for testing. To improve accuracy on CIFAR100, we augment by horizontal reflection and cropping [30], resulting in 0.8M training images.

The SVHN dataset is a large collection of digits (10 classes) cropped from real-world scenes, consisting of 73257 training images, 26032 testing images and 531131 less difficult images for additional training.

In our experiments, we use all available training images, for a total of 604388 training samples. For the MNIST variants [31], each variation either reduces the training size (MNIST-07) or amends the original digits by rotation (ROT), background superimposition (BG-RAND and BG-IMG), or a combination thereof (BG-ROT). We pre-process all datasets with whitening (except CIFAR100 and SVHN which were prohibitively large).

6.2 Baselines

We compare the proposed FreshNets with four baseline methods: HashedNets [6], low-rank decomposition (LRD) [13], filter dropping (DropFilt) and frequency dropping (DropFreq). HashedNets was first proposed to compress fully-connected layers in deep neural networks via the hashing trick. In this baseline, we apply the hashing trick directly to the convolutional layer by hashing filter weights in the spatial domain. This induces random weight sharing across all filters in a single convolutional layer.

Additionally, we compare against low-rank decomposition of the convolutional filters [13]. Following the method in [14], we unfold the four-dimensional filter tensor to form a two dimensional matrix on which we apply the low-rank decomposition. The parameters of the decomposition are fine-tuned via back-propagation. DropFreq learns parameters in the DCT frequency domain but sets high frequency components to 0 to meet the compression requirement. DropFilt compresses simply by reducing the number of filters in each convolutional layer.

The experimental environment is an off-the-shelve desktop with two 8-core Intel(R) Xeon(R) processors of 2.67 GHz and $128GB$ RAM. All methods were implemented using Torch7 [7] and run on NVIDIA GTX TITAN graphics cards with 2688 cores and 6GB of global memory. Model parameters are stored and updated as 32 bit floating-point values.[3] We further hold out 20% data from the training set as the validation set for early stopping. In particular, if the

validation error does not decrease for a number of epochs, the training would stop and output the model with the lowest validation error. Hyperparameters are selected for all algorithms with Bayesian optimization [40, 51] and hand tuning on the validation set. We use the open source Bayesian Optimization MATLAB implementation "bayesopt.m" from Gardner *et al.* [18].[4] All our reported results are based on the test error performance.

6.3 Comprehensive evaluation

We adopt the neural network architecture shown in Table 1 for all datasets. The architecture is a deep convolutional neural network consisting of five convolutional layers (with 5×5 filters) and one fully-connected layer. Before convolution, input feature maps are zero-padded such that output maps remain the same size as the (un-padded) input maps after convolution. Max-pooling is performed after convolutions in layers 2, 4 and 5 with filter size 2×2 and stride 2, reducing both input map dimensions by half. Rectified linear units are adopted as the activation function throughout. The output of the network is a softmax function over labels.

In this architecture, the convolutional layers hold the majority of parameters (1.2 million in convolutional layer *v.s.* 40 thousand in the fully connected layer with 10 output classes). During training, we optimize parameters using mini-batch gradient descent with batch size 64 and momentum 0.9. We use 20 percent of the training set as a validation set for early stopping. For FreshNets, we use a frequency-sensitive compression scheme which increases weight sharing among higher frequency components.[5] For all baselines, we apply HashedNets [6] to the fully connected layer at the corresponding level of compression. All error results are reported on the test set.

Table 2(a) and (b) show the comprehensive evaluation of all methods under compression ratios $1/16$ and $1/64$, respectively. We exclude DropFilt and DropFreq in Table 2(b) because neither supports $1/64$ compression in this architecture for all layers. For all methods, the fully connected layer (top layer) is compressed by HashedNets [6] at the corresponding compression rate. In this way, the final size of the entire network respects the specified compression ratio. For reference, we also show the error rate of a standard convolutional neural network (CNN, columns 2 and 8) with the fully-connected layer compressed by HashedNets and *no compression* in the convolutional layers. Excluding this reference, we highlight the method with best test error on each dataset in **bold**.

We discern several general trends. In Table 2(a), we observe the performance of the DropFilt and DropFreq at $1/16$ compression. At this compression rate, DropFilt corresponds to a network $1/16$ filters at each layer: 2, 4, 4, 8, 16 at layers $1-5$ respectively. This architecture yields particularly poor test accuracy, including essentially random predictions on three datasets. DropFreq, which at $1/16$ compression parameterizes each filter in the original network by only 1 or 2 low-frequency values in the DCT frequency space, performs with similarly poor accuracy. Low rank decomposition (LRD) and HashedNets each yield similar performance at both $1/16$ and $1/64$ compression. Neither explicitly considers the smoothness inherent in learned convolutional filters, instead compressing the filters in the spatial domain. Our method, FreshNets, consistently outperforms all baselines, particularly at the higher compression rate as shown in Table 2(b).

Using the same model in Table 1, Figure 5 shows more com-

---

[3]The compression rates of all methods could be further improved by learning and storing parameters in lower precision [9, 21].

[4]http://tinyurl.com/bayesopt

[5]We evaluate several frequency-sensitive schemes later in this section, but for this comprehensive evaluation we set frequency compression rates by a rescaled beta distribution with $\alpha = 0.25$ and $\beta = 2.5$ for all layers.

Layer	Operation	Input dim.	Inputs	Outputs	C size	MP size	Parameters
1	C,RL	32×32	3	32	5×5		$2K$
2	C,MP,DO,RL	32×32	32	64	5×5	$2\times2(2)$	$51K$
3	C,RL	16×16	64	64	5×5		$102K$
4	C,MP,DO,RL	16×16	64	128	5×5	$2\times2(2)$	$205K$
5	C,MP,DO,RL	8×8	128	256	5×5	$2\times2(2)$	$819K$
6	FC,Softmax	–	4096	10/100			$40/400K$

Table 1: **Network architecture. C: Convolution. RL: ReLu. MP: Max-pooling. DO: Dropout. FC: Fully-connected. The number of parameters in the fully-connected layer is specific to 32×32 input images and varies with the number of classes, either 10 or 100 depending on the dataset.**

	(a) Compression$=1/16$						(b) Compression$=1/64$			
	CNN	DropFilt	DropFreq	LRD	HashedNets	FreshNets	CNN	LRD	HashedNets	FreshNets
CIFAR10	14.91	54.87	30.45	23.23	24.70	**21.42**	14.37	34.35	43.08	**30.79**
CIFAR100	33.66	81.17	55.93	51.88	48.64	**47.49**	33.76	66.44	67.06	**62.33**
SVHN	3.71	30.93	14.96	10.67	9.00	**8.01**	3.69	22.32	23.31	**18.37**
MNIST-07	0.80	4.90	2.20	1.18	1.10	**0.94**	0.85	1.95	1.77	**1.24**
ROT	3.42	29.74	8.39	4.79	5.53	**3.87**	3.32	9.90	10.10	**6.60**
BG-ROT	11.42	88.88	56.63	20.19	**16.15**	18.43	11.28	35.64	32.40	**27.91**
BG-RAND	2.17	90.10	8.83	2.94	2.80	**2.63**	1.77	4.57	5.10	**3.62**
BG-IMG	2.61	89.41	27.89	4.35	**3.26**	3.97	2.38	7.23	**6.68**	8.04

Table 2: **Test error rates (in %) with compression factors $1/16$ and $1/64$. Convolutional layers were compressed by the indicated methods (DropFilt, DropFreq, LRD, HashedNets, and FreshNets), with no *convolutional layer* compression applied to CNN. The *fully connected* layer is compressed by HashNets for *all methods*, including CNN.**
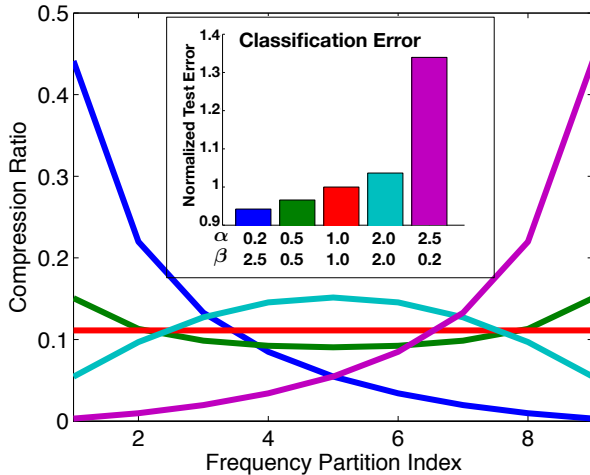

Figure 6: **Results with different frequency sensitive compression schemes, each adopting a different beta distribution as the compression rate for each frequency. The inner figure shows normalized test error of each scheme on CIFAR10 with the beta distribution hyper-parameters. The outer figure depicts the corresponding beta distributions. The setting $\alpha = 0.2, \beta = 2.5$ (blue line), which compresses low frequencies the least and high frequencies the most, yields lowest error.**

plete curves of test errors with multiple compression factors on the CIFAR10 and ROT datasets.

6.4 Varying compression by frequency

As mentioned in Section 3.4, we allow a higher collision rate in the high frequency components than in the low frequency components for each filter.

To demonstrate the utility of this scheme, we evaluate several hash compression schemes. Systematically, we set the compression rate of the jth frequency band r_j with a parameterized function, *i.e.* $r_j = f(j)$. In this experiment, we use the beta distribution:

$$f(j; \alpha, \beta) = Zx^{\alpha-1}(1-x)^{\beta-1},$$

where $x = \frac{j+1}{2k-1}$ is a real number between 0 and 1, k is the filter size, and Z is a normalizing factor such that the resulting distribution of parameters meets the target parameter budget K, *i.e.* when:

$$\sum_{j=0}^{2k-2} r_j N_j = K.$$

We adjust α and β to control the compression rate for each frequency region. As shown in Figure 6, we have multiple pairs of α and β, each of which results in a different compression scheme. For example, if $\alpha = 0.25$ and $\beta = 2.5$, the compression rate monotonically decreases as a function of component frequency, meaning more parameter sharing among high frequency components (blue curve in Figure 6).

To quickly evaluate the performance of each scheme, we use a simple four-layer FreshNets where the first two layers are DCT-hashed convolutional layers (with 5×5 filters) containing 32 and 64 feature maps respectively, and the last two layers are fully connected layers.

We test FreshNets on CIFAR10 with each of the compression schemes shown in Figure 6. In each, weight sharing is limited to be within groups of similar frequencies, as described in Section 3.4, however number of unique weights shared within each group is
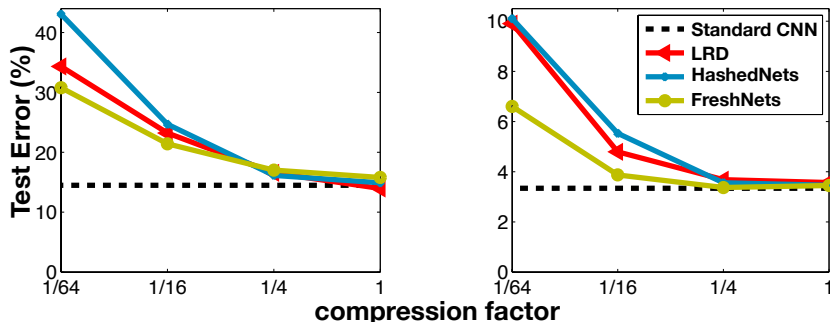
Figure 5: Test error rates at varying compression levels for datasets CIFAR10 **(left) and** ROT **(right).**

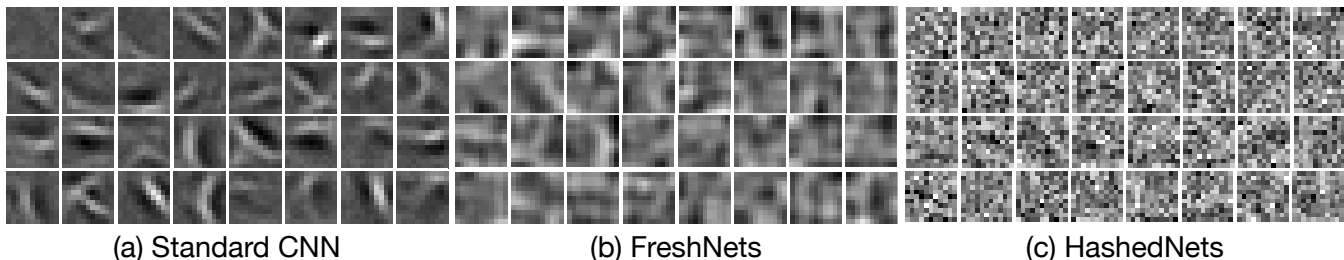(a) Standard CNN   (b) FreshNets   (c) HashedNets

Figure 7: Visualization of filters learning on MNIST **in (a) an uncompressed CNN, (b) a CNN compressed with FreshNets, and (c) a CNN compressed with HashedNets (compression rate** $1/16$ **in both (b) and (c)). FreshNets preserves the smoothness of the filters, whereas HashedNets does not.**

varied. We denote the compression scheme with $\alpha, \beta = 1$ (red curve) as a *frequency-oblivious scheme* since it produces a uniform compression independent of frequency.

In the inset bar plot in Figure 6, we report test error normalized by the test error of the frequency-oblivious scheme and averaged over compression rates $1, 1/2, 1/4, 1/16, 1/64$, and $1/256$. We can see that the proposed scheme with fewer shared weights allocated to high frequency components (represented by the blue curve) outperforms all other compression schemes. An inverse scheme where the high frequency regions have the lowest collision rate (purple curve) performs the worst. These empirical results fit our assumption that the low frequency components of a filter are more important than the high frequency components.

6.5 Filter visualization

We investigate the smoothness of the learned convolutional filters in Figure 7 by visualizing the filter weights (first layer) of (a) a standard, uncompressed CNN, (b) FreshNets, and (c) HashedNets (with weight sharing in the spatial domain). For this experiment, we again apply a four layer network with two convolutional layers but adopt larger filters (11×11) for better visualization.

All three networks are trained on MNIST, and both FreshNets and HashedNets have $1/16$ compression on the first convolutional layer. When plotting, we scale the values in each filter matrix to the range $[0, 255]$. Therefore, white and black pixels stand for large positive and negative weights, respectively. We observe that, although they are more blurry due to the compression, the filter weights of FreshNets are still smooth while weights in HashedNets appear more chaotic.

7. CONCLUSIONS

In this paper we present FreshNets, a method for learning convo-

lutional neural networks with dramatically compressed model storage. We introduce negligible efficiency overhead (since most running time is still spent on convolutional operations) but obtain a dramatically smaller convolutional neural network. Harnessing the hashing trick for parameter-free random weight sharing and leveraging the smoothness inherent in convolutional filters, FreshNets compresses parameters in a frequency-sensitive fashion such that significant model parameters (*e.g.* low-frequency components) are better preserved. As such, FreshNets preserves prediction accuracy better than competing baselines at high compression rates.

We believe that the proposed compression techniques will have broad impacts on real-world applications, such as better image and speech recognition on mobile phones and other devices. In the future, we will investigate more on the frequency structure of FreshNets and explore additional hashing schemes. Moreover, we will investigate its integration with other implementation schemes such as low precision representation of weights.

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] N. Ahmed, T. Natarajan, and K. R. Rao. Discrete cosine transform. *IEEE transactions on Computers*, (1):90–93, 1974.
[2] J. Ba and R. Caruana. Do deep nets really need to be deep? In *Advances in neural information processing systems*, pages 2654–2662, 2014.

[3] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., 1995.

[4] T. Brosch and R. Tam. Efficient training of convolutional deep belief networks in the frequency domain for application to high-resolution 2d and 3d images. *Neural Computation*, 27(1):211–227, 2015.

[5] C. Bucilua, R. Caruana, and A. Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541. ACM, 2006.

[6] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, 2015.

[7] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*, volume 6. MIT press Cambridge, 2001.

[9] M. Courbariaux, Y. Bengio, and J.-P. David. Low precision storage for deep learning. *arXiv preprint arXiv:1412.7024*, 2014.

[10] M. Courbariaux, Y. Bengio, and J.-P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pages 3105–3113, 2015.

[11] A. Dasgupta, R. Kumar, and T. Sarlós. A sparse johnson: Lindenstrauss transform. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 341–350. ACM, 2010.

[12] 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, IEEE Conference on*, pages 248–255. IEEE, 2009.

[13] 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.

[14] E. L. 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*, pages 1269–1277, 2014.

[15] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. *arXiv preprint arXiv:1310.1531*, 2013.

[16] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[17] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[18] J. Gardner, M. Kusner, K. Weinberger, J. Cunningham, et al. Bayesian optimization with inequality constraints. In *International Conference on Machine Learning*, 2014.

[19] 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*, pages 580–587, 2014.

[20] Y. Gong, L. Liu, M. Yang, and L. Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[21] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. *arXiv preprint arXiv:1502.02551*, 2015.

[22] S. Han, H. Mao, and W. J. Dally. A deep neural network compression pipeline: Pruning, quantization, huffman encoding. *arXiv preprint arXiv:1510.00149*, 2015.

[23] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.

[24] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852*, 2015.

[25] M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[26] A. Karpathy and L. Fei-Fei. Deep visual-semantic alignments for generating image descriptions. *arXiv preprint arXiv:1412.2306*, 2014.

[27] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[28] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[29] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images, 2009.

[30] 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.

[31] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th international conference on Machine learning*, pages 473–480. ACM, 2007.

[32] Q. Le, T. Sarlós, and A. Smola. Fastfood-computing hilbert space expansions in loglinear time. In *Proceedings of the 30th International Conference on Machine Learning*, pages 244–252, 2013.

[33] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

[34] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[35] H. Lee, P. Pham, Y. Largman, and A. Y. Ng. Unsupervised feature learning for audio classification using convolutional deep belief networks. In *Advances in neural information processing systems*, pages 1096–1104, 2009.

[36] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[37] J. Mao, W. Xu, Y. Yang, J. Wang, Z. Huang, and A. Yuille. Deep captioning with multimodal recurrent neural networks (m-rnn). *arXiv preprint arXiv:1412.6632*, 2014.

[38] M. Mathieu, M. Henaff, and Y. LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

[39] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, page 4. Granada, Spain, 2011.

[40] M. Pelikan. Bayesian optimization algorithm. In *Hierarchical Bayesian optimization algorithm*, pages 31–48. Springer, 2005.

[41] M. Rabbani and R. Joshi. An overview of the jpeg 2000 still image compression standard. *Signal processing: Image communication*, 17(1):3–48, 2002.

[42] K. R. Rao and P. Yip. *Discrete cosine transform: algorithms, advantages, applications*. Academic press, 2014.

[43] A. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 806–813, 2014.

[44] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 833–840, 2011.

[45] R. Rigamonti, A. Sironi, V. Lepetit, and P. Fua. Learning separable filters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2754–2761, 2013.

[46] O. Rippel, J. Snoek, and R. P. Adams. Spectral representations for convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 2440–2448, 2015.

[47] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. *arXiv preprint arXiv:1503.03832*, 2015.

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

[49] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, and S. Vishwanathan. Hash kernels for structured data. *Journal of Machine Learning Research*, 10:2615–2637, Dec. 2009.

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

[51] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[52] G. Soulié, V. Gripon, and M. Robert. Compression of deep neural networks on the fly. *arXiv preprint arXiv:1509.08745*, 2015.

[53] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th international conference on machine learning (ICML-13)*, pages 1139–1147, 2013.

[54] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[55] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.

[56] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4:2, 2012.

[57] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun. Fast convolutional nets with fbfft: A gpu performance evaluation. *arXiv preprint arXiv:1412.7580*, 2014.

[58] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. *arXiv preprint arXiv:1411.4555*, 2014.

[59] G. K. Wallace. The jpeg still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991.

[60] K. Q. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1113–1120. ACM, 2009.

[61] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng. Quantized convolutional neural networks for mobile devices. *arXiv preprint arXiv:1512.06473*, 2015.

[62] Z. Yang, M. Moczulski, M. Denil, N. de Freitas, A. Smola, L. Song, and Z. Wang. Deep fried convnets. *arXiv preprint arXiv:1412.7149*, 2014.

[63] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *Computer vision–ECCV 2014*, pages 818–833. Springer, 2014.