Deep Fried Convnets

Zichao Yang[1]   Marcin Moczulski[2]   Misha Denil[2,5]
Nando de Freitas[2,5,6]   Alex Smola[1,4]   Le Song[3]   Ziyu Wang[2,5]

[1]Carnegie Mellon University, [2]University of Oxford, [3]Georgia Institute of Technology
[4]Google, [5]Google DeepMind, [6]Canadian Institute for Advanced Research
zichaoy@cs.cmu.edu, marcin.moczulski@stcatz.ox.ac.uk,
mdenil@google.com, nandodefreitas@google.com,
alex@smola.org, lsong@cc.gatech.edu, ziyu@google.com

Abstract

The fully- connected layers of deep convolutional neural networks typically contain over 90% of the network parameters. Reducing the number of parameters while preserving predictive performance is critically important for training big models in distributed systems and for deployment in embedded devices.

In this paper, we introduce a novel Adaptive Fastfood transform to reparameterize the matrix-vector multiplication of fully connected layers. Reparameterizing a fully connected layer with d inputs and n outputs with the Adaptive Fastfood transform reduces the storage and computational costs costs from $\mathcal{O}(nd)$ to $\mathcal{O}(n)$ and $\mathcal{O}(n \log d)$ respectively. Using the Adaptive Fastfood transform in convolutional networks results in what we call a deep fried convnet. These convnets are end-to-end trainable, and enable us to attain substantial reductions in the number of parameters without affecting prediction accuracy on the MNIST and ImageNet datasets.

1. Introduction

In recent years we have witnessed an explosion of applications of convolutional neural networks with millions and billions of parameters. Reducing this vast number of parameters would improve the efficiency of training in distributed architectures. It would also allow for the deployment of state-of-the-art convolutional neural networks on embedded mobile applications. These train and test time considerations are both of great importance.

A standard convolutional network is composed of two types of layers, each with very different properties. Convolutional layers, which contain a small fraction of the network parameters, represent most of the computational effort. In contrast, fully connected layers contain the vast majority of the parameters but are comparatively cheap to evaluate [21].

This imbalance between memory and computation suggests that the efficiency of these two types of layers should be addressed in different ways. [12] and [18] both describe methods for minimizing computational cost of evaluating a network at test time by approximating the learned convolutional filters with separable approximations. These approaches realize speed gains at test time but do not address the issue of training, since the approximations are made after the network has been fully trained. Additionally, neither approach achieves a substantial reduction in the number of parameters, since they both work with approximations of the convolutional layers, which represent only a small portion of the total number of parameters. Many other works have addressed the computational efficiency of convolutional networks in more specialized settings [13, 24].

In contrast to the above approaches, [11] shows that there is significant redundancy in the parameterization of several deep learning models, and exploits this to reduce the number of parameters. More specifically, their method represents the parameter matrix as a product of two low rank factors, and the training algorithm fixes one factor (called static parameters) and only updates the other factor (called dynamic parameters). [33] uses low-rank matrix factorization to reduce the size of the fully connected layers at train time. They demonstrate large improvements in reducing the number of parameters of the output softmax layer, but only modest improvements for the hidden fully connected layers. [37] implements low-rank factorizations using the SVD after training the full model. In contrast, the methods advanced in [33] and this paper apply both at train and test time.

In this paper we show how the number of parameters required to represent a deep convolutional neural network can be substantially reduced without sacrificing predictive performance. Our approach works by replacing the fully

connected layers of the network with an Adaptive Fastfood transform, which is a generalization of the Fastfood transform for approximating kernels [23].

Convolutional neural networks with Adaptive Fastfood transforms, which we refer to as deep fried convnets, are end-to-end trainable and achieve the same predictive performance as standard convolutional networks on ImageNet using approximately half the number of parameters.

Several works have considered kernel methods in deep learning [16, 7, 10, 27]. The Doubly Stochastic Gradients method of [10] showed that effective use of randomization can allow kernel methods to scale to extremely large data sets. However, the approach used fixed convolutional features, and cannot jointly learn the kernel classifier and convolutional filters. [27] showed how to learn a kernel function in an unsupervised manner.

There have been other attempts to replace the fully connected layers. The Network in Network architecture of [25] achieves state of the art results on several deep learning benchmarks by replacing the fully connected layers with global average pooling. A similar approach was used by [35] to win the ILSVRC 2014 object detection competition [32].

Although the global average pooling approach achieves impressive results, it has two significant drawbacks. First, feature transfer is more difficult with this approach. It is very common in practice to take a convolutional network trained on ImageNet and re-train the top layer on a different data set, re-using the features learned from ImageNet for the new task (potentially with fine-tuning), and this is difficult with global average pooling. This deficiency is noted by [35], and motivates them to add an extra linear layer to the top of their network to enable them to more easily adapt and fine tune their network to other label sets. The second drawback of global average pooling is computation. Convolutional layers are much more expensive to evaluate than fully connected layers, so replacing fully connected layers with more convolutions can decrease model size but comes at the cost of increased evaluation time.

In parallel or after the first (technical report) version of this work, several researchers have attempted to create sparse networks by applying pruning or sparsity regularizers [8, 4, 26, 14]. These approaches however require training the original full model and, consequently, do not enjoy the efficient training time benefits of the techniques proposed in this paper. Since then, hashing methods have also been advanced to reduce the number of parameters [6, 3]. Hashes have irregular memory access patterns and, consequently, good performance on large GPU-based platforms is yet to be demonstrated. Finally, distillation [15, 31] also offers a way of compressing neural networks, as a post-processing step.

2. The Adaptive Fastfood Transform

Large dense matrices are the main building block of fully connected neural network layers. In propagating the signal from the l-th layer with d activations $\mathbf{h}_l$ to the $l + 1$-th layer with n activations $\mathbf{h}_{l+1}$, we have to compute

$$\mathbf{h}_{l+1} = \mathbf{W}\mathbf{h}_l. \tag{1}$$

The storage and computational costs of this matrix multiplication step are both $\mathcal{O}(nd)$. The storage cost in particular can be prohibitive for many applications.

Our proposed solution is to reparameterize the matrix of parameters $\mathbf{W} \in \mathbb{R}^{n \times d}$ with an Adaptive Fastfood transform, as follows

$$\mathbf{h}_{l+1} = (\mathbf{SHG\Pi HB})\,\mathbf{h}_l = \widehat{\mathbf{W}}\mathbf{h}_l. \tag{2}$$

In Section 3, we will provide background and intuitions behind this design. For now it suffices to state that the storage requirements of this reparameterization are $\mathcal{O}(n)$ and the computational cost is $\mathcal{O}(n \log d)$. We will also show in the experimental section that these theoretical savings are mirrored in practice by significant reductions in the number of parameters without increased prediction errors.

To understand these claims, we need to describe the component modules of the Adaptive Fastfood transform. For simplicity of presentation, let us first assume that $\mathbf{W} \in \mathbb{R}^{d \times d}$. Adaptive Fastfood has three types of module:

- $\mathbf{S}, \mathbf{G}$ and $\mathbf{B}$ are diagonal matrices of parameters. In the original non-adaptive Fastfood formulation they are random matrices, as described further in Section 3. The computational and storage costs are trivially $\mathcal{O}(d)$.

- $\mathbf{\Pi} \in \{0, 1\}^{d \times d}$ is a random permutation matrix. It can be implemented as a lookup table, so the storage and computational costs are also $\mathcal{O}(d)$.

- $\mathbf{H}$ denotes the Walsh-Hadamard matrix, which is defined recursively as

$$\mathbf{H}_2 := \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \text{ and } \mathbf{H}_{2d} := \begin{bmatrix} \mathbf{H}_d & \mathbf{H}_d \\ \mathbf{H}_d & -\mathbf{H}_d \end{bmatrix}.$$

The Fast Hadamard Transform, a variant of Fast Fourier Transform, enables us to compute $\mathbf{H}_d\mathbf{h}_l$ in $\mathcal{O}(d \log d)$ time.

In summary, the overall storage cost of the Adaptive Fastfood transform is $\mathcal{O}(d)$, while the computational cost is $\mathcal{O}(d \log d)$. These are substantial theoretical improvements over the $\mathcal{O}(d^2)$ costs of ordinary fully connected layers.

When the number of output units n is larger than the number of inputs d, we can perform n/d Adaptive Fastfood transforms and stack them to attain the desired size. In doing so, the computational and storage costs become $\mathcal{O}(n \log d)$ and $\mathcal{O}(n)$ respectively, as opposed to the more substantial $\mathcal{O}(nd)$ costs for linear modules. The number of outputs can also be refined with pruning.

2.1. Learning Fastfood by backpropagation

The parameters of the Adaptive Fastfood transform $(\mathbf{S}, \mathbf{G}$ and $\mathbf{B})$ can be learned by standard error derivative backpropagation. Moreover, the backward pass can also be computed efficiently using the Fast Hadamard Transform.

In particular, let us consider learning the l-th layer of the network, $\mathbf{h}_{l+1} = \mathbf{SHG\Pi HB h}_l$.

For simplicity, let us again assume that $\mathbf{W} \in \mathbb{R}^{d \times d}$ and that $\mathbf{h}_l \in \mathbb{R}^d$. Using backpropagation, assume we already have $\frac{\partial E}{\partial \mathbf{h}_{l+1}}$, where E is the objective function, then

$$\frac{\partial E}{\partial \mathbf{S}} = \mathrm{diag}\left\{ \frac{\partial E}{\partial \mathbf{h}_{l+1}} (\mathbf{HG\Pi HB h}_l)^\top \right\}. \qquad (3)$$

Since $\mathbf{S}$ is a diagonal matrix, we only need to calculate the derivative with respect to the diagonal entries and this step requires only $\mathcal{O}(d)$ operations.

Proceeding in this way, denote the partial products by

$$
\begin{aligned}
\mathbf{h}_S &= \mathbf{HG\Pi HB h}_l \\
\mathbf{h}_{H1} &= \mathbf{G\Pi HB h}_l \\
\mathbf{h}_G &= \mathbf{\Pi HB h}_l \\
\mathbf{h}_\Pi &= \mathbf{HB h}_l \\
\mathbf{h}_{H2} &= \mathbf{B h}_l.
\end{aligned}
\qquad (4)
$$

Then the gradients with respect to different parameters in the Fastfood layer can be computed recursively as follows:

$$
\begin{aligned}
\frac{\partial E}{\partial \mathbf{h}_S} &= \mathbf{S}^\top \frac{\partial E}{\partial \mathbf{h}_{l+1}} & \frac{\partial E}{\partial \mathbf{h}_{H1}} &= \mathbf{H}^\top \frac{\partial E}{\partial \mathbf{h}_S} \\
\frac{\partial E}{\partial \mathbf{G}} &= \mathrm{diag}\left\{ \frac{\partial E}{\partial \mathbf{h}_{H1}} \mathbf{h}_G^\top \right\} & \frac{\partial E}{\partial \mathbf{h}_G} &= \mathbf{G}^\top \frac{\partial E}{\partial \mathbf{h}_{H1}} \\
\frac{\partial E}{\partial \mathbf{h}_\Pi} &= \mathbf{\Pi}^\top \frac{\partial E}{\partial \mathbf{h}_G} & \frac{\partial E}{\partial \mathbf{h}_{H2}} &= \mathbf{H}^\top \frac{\partial E}{\partial \mathbf{h}_\Pi} \\
\frac{\partial E}{\partial \mathbf{B}} &= \mathrm{diag}\left\{ \frac{\partial E}{\partial \mathbf{h}_{H2}} \mathbf{h}_l^\top \right\} & \frac{\partial E}{\partial \mathbf{h}_l} &= \mathbf{B}^\top \frac{\partial E}{\partial \mathbf{h}_{H2}}.
\end{aligned}
\qquad (5)
$$

Note that the operations in $\frac{\partial E}{\partial \mathbf{h}_{H1}}$ and $\frac{\partial E}{\partial \mathbf{h}_{H2}}$ are simply applications of the Hadamard transform, since $\mathbf{H}^\top = \mathbf{H}$, and consequently can be computed in $\mathcal{O}(d \log d)$ time. The operation in $\frac{\partial E}{\partial \mathbf{h}_\Pi}$ is an application of a permutation (the transpose of permutation matrix is a permutation matrix) and can be computed in $\mathcal{O}(d)$ time. All other operations are diagonal matrix multiplications.

3. Intuitions behind Adaptive Fastfood

The proposed Adaptive Fastfood transform may be understood either as a trainable type of structured random projection or as an approximation to the feature space of a learned kernel. Both views not only shed light on Adaptive Fastfood and competing techniques, but also open up room to innovate new techniques to reduce computation and memory in neural networks.

3.1. A view from structured random projections

Adaptive Fastfood is based on the Fastfood transform [23], in which the diagonal matrices $\mathbf{S}$, $\mathbf{G}$ and $\mathbf{B}$ have random entries. In the experiments, we will compare the performance of the existing random and proposed adaptive versions of Fastfood when used to replace fully connected layers in convolutional neural networks.

The intriguing idea of constructing neural networks with random weights has been reasonably explored in the neural networks field [34, 19]. This idea is related to random projections, which have been deeply studied in theoretical computer science [28]. In a random projection, the basic operation is of the form

$$\mathbf{y} = \mathbf{W}\mathbf{x}, \qquad (6)$$

where $\mathbf{W}$ is a random matrix, either Gaussian [17] or binary [1]. Importantly, the embeddings generated by these random projections approximately preserve metric information, as formalized by many variants of the celebrated Johnson-Lindenstrauss Lemma.

The one shortcoming of random projections is that the cost of storing the matrix $\mathbf{W}$ is $\mathcal{O}(nd)$. Using a sparse random matrix $\mathbf{W}$ by itself to reduce this cost is often not a viable option because the variance of the estimates of $\|\mathbf{W}\mathbf{x}\|$ can be very high for some inputs, for example when $\mathbf{x}$ is also sparse. To see this, consider the extreme case of a very sparse input $\mathbf{x}$, then many of the products with $\mathbf{W}$ will be zero and hence not help improve the estimates of metric properties of the embedding space.

One popular option for reducing the storage and computational costs of random projections is to adopt random hash functions to replace the random matrix multiplication. For example, the count-sketch algorithm [5] uses pairwise independent hash functions to carry this job very effectively in many applications [9]. This technique is often referred to as the hashing trick [36] in the machine learning literature. Hashes have irregular memory access patterns, so it is not clear how to get good performance on GPUs when following this approach, as pointed out in [6].

Ailon and Chazelle [2] introduced an alternative approach that is not only very efficient, but also preserves most of the desirable theoretical properties of random projections. Their idea was to replace the random matrix by a transform that mimics the properties of random matrices, but which can be stored efficiently. In particular, they proposed the following PHD transform:

$$\mathbf{y} = \mathbf{PHD}\mathbf{x}, \qquad (7)$$

where $\mathbf{P}$ is a sparse $n \times d$ random matrix with Gaussian entries, $\mathbf{H}$ is a Hadamard matrix and $\mathbf{D}$ is a diagonal matrix with $\{+1, -1\}$ entries drawn independently with probability $1/2$. The inclusion of the Hadamard transform avoids

the problems of using a sparse random matrix by itself, but it is still efficient to compute.

We can think of the original Fastfood transform

$$\mathbf{y} = \mathbf{SHG\Pi HBx} \qquad (8)$$

as an alternative to this. Fastfood reduces the computation and storage of random projections to $\mathcal{O}(n \log d)$ and $\mathcal{O}(n)$ respectively. In the original formulation $\mathbf{S}, \mathbf{G}$ and $\mathbf{B}$ are diagonal random matrices, which are computed once and then stored.

In contrast, in our proposed Adaptive Fastfood transform, the diagonal matrices are learned by backpropagation. By adapting $\mathbf{B}$, we are effectively implementing Automatic Relevance Determination on features. The matrix $\mathbf{G}$ controls the bandwidth of the kernel and its spectral incoherence. Finally, $\mathbf{S}$ represents different kernel types. For example, for the RBF kernel $\mathbf{S}$ follows Chi-squared distribution. By adapting $\mathbf{S}$, we learn the correct kernel type.

While we have introduced Fastfood in this section, it was originally proposed as a fast way of computing random features to approximate kernels. We expand on this perspective in the following section.

3.2. A view from kernels

There is a nice duality between inner products of features and kernels. This duality can be used to design neural network modules using kernels and vice-versa.

For computational reasons, we often want to determine the features associated with a kernel. Working with features is preferable when the kernel matrix $\mathbf{K}$ is dense and large. (Storing this matrix requires $O(m^2)$ space, and computing it takes $O(m^2 d)$ operations, where m is the number of data points and d is the dimension.) We might also want to design statistical methods using kernels and then map these designs to features that can be used as modules in neural networks. Unfortunately, one of the difficulties with this line of attack is that deriving features from kernels is far from trivial in general.

An important fact, noted in [30], is that infinite kernel expansions can be approximated in an unbiased manner using randomly drawn features. For shift-invariant kernels this relies on a classical result from harmonic analysis, known as Bochner's Lemma, which states that a continuous shift-invariant kernel $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x} - \mathbf{x}')$ on $\mathbb{R}^d$ is positive definite if and only if k is the Fourier transform of a non-negative measure $\mu(\mathbf{w})$. This measure, known as the spectral density, in turn implies the existence of a probability density $p(\mathbf{w}) = \mu(\mathbf{w})/\alpha$ such that

$$k(\mathbf{x}, \mathbf{x}') = \int \alpha e^{-i\mathbf{w}^\top (\mathbf{x} - \mathbf{x}')} p(\mathbf{w}) d\mathbf{w}$$
$$= \alpha \mathbb{E}_{\mathbf{w}} [\cos(\mathbf{w}^\top \mathbf{x}) \cos(\mathbf{w}^\top \mathbf{x}') + \sin(\mathbf{w}^\top \mathbf{x}) \sin(\mathbf{w}^\top \mathbf{x}')],$$

where the imaginary part is dropped since both the kernel and distribution are real.

We can apply Monte Carlo methods to approximate the above expectation, and hence approximate the kernel $k(\mathbf{x}, \mathbf{x}')$ with an inner product of stacked cosine and sine features. Specifically, suppose we sample n vectors $i.i.d.$ from $p(\mathbf{w})$ and collect them in a matrix $\mathbf{W} = (\mathbf{w}_1, \dots \mathbf{w}_n)^\top$. The kernel can then be approximated as the inner-product of the following random features:

$$\boldsymbol{\phi}_{\mathrm{rbf}}(\mathbf{Wx}) = \sqrt{\alpha/n} \, (\cos(\mathbf{Wx}), \sin(\mathbf{Wx}))^\top . \quad (9)$$

That is, $\boldsymbol{\phi}(\mathbf{Wx})$ is the neural network module, consisting of a linear layer $\mathbf{Wx}$ and entry-wise nonlinearities (cosine and sine in the above equation), that corresponds to a particular implicit kernel function.

Approximating a given kernel function with random features requires the specification of a sampling distribution $p(\mathbf{w})$. Such distributions have been derived for many popular kernels. For example, if we want the implicit kernel to be a squared exponential kernel,

$$k(\mathbf{x}, \mathbf{x}') = \exp \left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\ell^2} \right), \qquad (10)$$

we know that the distribution $p(\mathbf{w})$ must be Gaussian: $\mathbf{w} \sim \mathcal{N}(0, \mathrm{diag}(\ell^2)^{-1})$. In other words, if we draw the rows of $\mathbf{W}$ from this Gaussian distribution and use equation (9) to implement a neural module, we are implicitly approximating a squared exponential kernel.

As another example of the mapping between kernels and random features, [7, 29] introduced the rotationally invariant arc-cosine kernel

$$k(\mathbf{x}, \mathbf{x}') = \frac{1}{\pi} \|\mathbf{x}\| \|\mathbf{x}'\| (\sin(\theta) + (\pi - \theta) \cos(\theta)), \quad (11)$$

where θ is the angle between $\mathbf{x}$ and $\mathbf{x}'$. Then by choosing $\mathbf{W}$ to be a random Gaussian matrix, they showed that this kernel can be approximated with Rectified Linear Unit (ReLU) features:

$$\boldsymbol{\phi}_{\mathrm{relu}}(\mathbf{Wx}) = \sqrt{1/n} \max(0, \mathbf{Wx}). \qquad (12)$$

The Fastfood transform was introduced to replace $\mathbf{Wx}$ in Equation 9 with $\mathbf{SHG\Pi HBx}$, thus decreasing the computational and storage costs.

4. Deep Fried Convolutional Networks

We propose to greatly reduce the number of parameters of the fully connected layers by replacing them with an Adaptive Fastfood transform followed by a nonlinearity. We call this new architecture a deep fried convolutional network. An illustration of this architecture is shown in Figure 1.
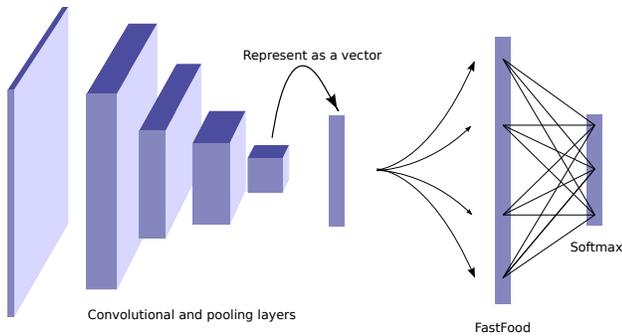
Figure 1. The structure of a deep fried convolutional network. The convolution and pooling layers are identical to those in a standard convnet. However, the fully connected layers are replaced with the Adaptive Fastfood transform.

MNIST (joint)	Error	Params
Fastfood 1024 (ND)	0.83%	38,821
Adaptive Fastfood 1024 (ND)	0.86%	38,821
Fastfood 2048 (ND)	0.90%	52,124
Adaptive Fastfood 2048 (ND)	0.92%	52,124
Fastfood 1024	**0.71**%	38,821
Adaptive Fastfood 1024	0.72%	38,821
Fastfood 2048	**0.71**%	52,124
Adaptive Fastfood 2048	0.73%	52,124
Reference Model	**0.87**%	430,500

Table 1. MNIST jointly trained layers: comparison between a reference convolutional network with one fully connected layer (followed by a densely connected softmax layer) and two deep fried networks on the MNIST dataset. Numbers indicate the number of features used in the Fastfood transform. The results tagged with *(ND)* were obtained wtihout dropout.

In principle, we could also apply the Adaptive Fastfood transform to the softmax classifier. However, reducing the memory cost of this layer is already well studied; for example, [33] show that low-rank matrix factorization can be applied during training to reduce the size of the softmax layer substantially. Importantly, they also show that training a low rank factorization for the internal layers performs poorly, which agrees with the results of [11]. For this reason, we focus our attention on reducing the size of the internal layers.

5. MNIST Experiment

The first problem we study is the classical MNIST optical character recognition task. This simple task serves as an easy proof of concept for our method, and contrasting the results in this section with our later experiments gives insights into the behavior of the Adaptive Fastfood transform at different scales.

As a reference model we use the Caffe implementation of the LeNet convolutional network.[1] It achieves an error rate of 0.87% on the MNIST dataset.

We jointly train all layers of the deep fried network (including convolutional layers) from scratch. We compare both the adaptive and non-adaptive Fastfood transforms using 1024 and 2048 features. For the non-adaptive transforms we report the best performance achieved by varying the standard deviation of the random Gaussian matrix over the set $\{0.001, 0.005, 0.01, 0.05\}$, and for the adaptive variant we learn these parameters by backpropagation as described in Section 2.1.

The results of the MNIST experiment are shown in Table 1. Because the width of the deep fried network is substantially larger than the reference model, we also experimented with adding dropout in the model, which increased performance in the deep fried case. Deep fried networks are

able to obtain high accuracy using only a small fraction of of parameters of the original network (11 times reduction in the best case). Interestingly, we see no benefit from adaptation in this experiment, with the more powerful adaptive models performing equivalently or worse than their non-adaptive counterparts; however, this should be contrasted with the ImageNet results reported in the following sections.

6. Imagenet Experiments

We now examine how deep fried networks behave in a more realistic setting with a much larger dataset and many more classes. Specifically, we use the ImageNet ILSVRC-2012 dataset which has 1.2M training examples and 50K validation examples distributed across 1000 classes.

We use the the Caffe ImageNet model[2] as the reference model in these experiments [20]. This model is a modified version of AlexNet [22], and achieves 42.6% top-1 error on the ILSVRC-2012 validation set. The initial layers of this model are a cascade of convolution and pooling layers with interspersed normalization. The last several layers of the network take the form of an MLP and follow a 9216–4096–4096–1000 architecture. The final layer is a logistic regression layer with 1000 output classes. All layers of this network use the ReLU nonlinearity, and dropout is used in the fully connected layers to prevent overfitting.

There are total of 58,649,184 parameters in the reference model, of which 58,621,952 are in the fully connected layers and only 27,232 are in the convolutional layers. The parameters of fully connected layer take up 99.9% of the total number of parameters. We show that the Adaptive Fastfood transform can be used to substantially reduce the number of parameters in this model.

---

[1] https://github.com/BVLC/caffe/blob/master/examples/mnist/lenet.prototxt

[2] https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet

ImageNet (fixed)	Error	Params
Dai et al. [10]	44.50%	163M
Fastfood 16	50.09%	16.4M
Fastfood 32	50.53%	32.8M
Adaptive Fastfood 16	45.30%	16.4M
Adaptive Fastfood 32	**43.77%**	32.8M
MLP	**47.76%**	58.6M

Table 2. Imagenet fixed convolutional layers: MLP indicates that we re-train 9216–4096–4096–1000 MLP (as in the original network) with the convolutional weights pretrained and fixed. Our method is *Fastfood 16* and *Fastfood 32*, using 16,384 and 32,768 Fastfood features respectively. [10] report results of max-voting of 10 transformations of the test set.

6.1. Fixed feature extractor

Previous work on applying kernel methods to ImageNet has focused on building models on features extracted from the convolutional layers of a pre-trained network [10]. This setting is less general than training a network from scratch but does mirror the common use case where a convolutional network is first trained on ImageNet and used as a feature extractor for a different task.

In order to compare our Adaptive Fastfood transform directly to this previous work, we extract features from the final convolutional layer of a pre-trained reference model and train an Adaptive Fastfood transform classifier using these features. Although the reference model uses two fully connected layers, we investigate replacing these with only a single Fastfood transform. We experiment with two sizes for this transform: *Fastfood 16* and *Fastfood 32* using 16,384 and 32,768 Fastfood features respectively. Since the Fastfood transform is a composite module, we can apply dropout between any of its layers. In the experiments reported here, we applied dropout after the $\mathbf{\Pi}$ matrix and after the $\mathbf{S}$ matrix. We also applied dropout to the last convolutional layer (that is, before the $\mathbf{B}$ matrix).

We also train an MLP with the same structure as the top layers of the reference model for comparison. In this setting it is important to compare against the re-trained MLP rather than the jointly trained reference model, as training on features extracted from fixed convolutional layers typically leads to lower performance than joint training [38].

The results of the fixed feature experiment are shown in Table 2. Following [38] and [10] we observe that training on ImageNet activations produces significantly lower performance than of the original, jointly trained network. Nonetheless, deep fried networks are able to outperform both the re-trained MLP model as well as the results in [10] while using fewer parameters.

In contrast with our MNIST experiment, here we find that the Adaptive Fastfood transform provides a significant performance boost over the non-adaptive version, improving top-1 performance by 4.5-6.5%.

6.2. Jointly trained model

Finally, we train a deep fried network from scratch on ImageNet. With 16,384 features in the Fastfood layer we lose less than 0.3% top-1 validation performance, but the number of parameters in the network is reduced from 58.7M to 16.4M which corresponds to a factor of 3.6x. By further increasing the number of features to 32,768, we are able to perform 0.6% better than the reference model while using approximately half as many parameters. Results from this experiment are shown in Table 3.

ImageNet (joint)	Error	Params
Fastfood 16	46.88%	16.4M
Fastfood 32	46.63%	32.8M
Adaptive Fastfood 16	42.90%	16.4M
Adaptive Fastfood 32	**41.93%**	32.8M
Reference Model	**42.59%**	58.7M

Table 3. Imagenet jointly trained layers. Our method is *Fastfood 16* and *Fastfood 32*, using 16,384 and 32,768 Fastfood features respectively. *Reference Model* shows the accuracy of the jointly trained Caffe reference model.

Nearly all of the parameters of the deep fried network reside in the final softmax regression layer, which still uses a dense linear transformation, and accounts for more than 99% of the parameters of the network. This is a side effect of the large number of classes in ImageNet. For a data set with fewer classes the advantage of deep fried convolutional networks would be even greater. Moreover, as shown by [11, 33], the last layer often contains considerable redundancy. We also note that any of the techniques from [8, 6] could be applied to the final layer of a deep fried network to further reduce memory consumption at test time. We illustrate this with low-rank matrix factorization in the following section.

7. Comparison with Post Processing

In this section we provide a comparison to some existing works on reducing the number of parameters in a convolutional neural network. The techniques we compare against here are *post-processing* techniques, which start from a full trained model and attempt to compress it, whereas our method trains the compressed network from scratch.

Matrix factorization is the most common method for compressing neural networks, and has proven to be very effective. Given the weight matrix of fully connected layers $\mathbf{W} \in \mathbb{R}^{d \times n}$, we factorize it as

$$\mathbf{W} = \mathbf{USV}^\top,$$

where $\mathbf{U} \in \mathbb{R}^{d \times d}$ and $\mathbf{V} \in \mathbb{R}^{n \times n}$ and $\mathbf{S}$ is a $d \times n$ diagonal matrix. In order to reduce the parameters, we truncate all but the k largest singular values, leading to the approximation $\mathbf{W} \approx \tilde{\mathbf{U}}\tilde{\mathbf{V}}^\top$, where $\tilde{\mathbf{U}} \in \mathbb{R}^{d \times k}$ and $\tilde{\mathbf{V}} \in \mathbb{R}^{n \times k}$

and $\mathbf{S}$ has been absorbed into the other two factors. If k is sufficiently small then storing $\tilde{\mathbf{U}}$ and $\tilde{\mathbf{V}}$ is less expensive than storing $\mathbf{W}$ directly, and this parameterization is still learnable.

It has been shown that training a factorized representation directly leads to poor performance [11] (although it does work when applied only to the final logistic regression layer [33]). However, first training a full model, then preforming an SVD of the weight matrices followed by a fine tuning phase preserves much of the performance of the original model [37]. We compare our deep fried approach to SVD followed by fine tuning, and show that our approach achieves better performance per parameter in spite of training a compressed parameterization from scratch. We also compare against a post-processed version of our model, where we train a deep fried convnet and then apply SVD plus fine-tuning to the final softmax layer, which further reduces the number of parameters.

Results of these post-processing experiments are shown in Table 4. For the SVD decomposition of each of the three fully connected layers in the reference model we set $k = \min(d, n)/2$ in SVD-half and $k = \min(d, n)/4$ in SVD-quarter. SVD-half-F and SVD-quarter-F mean that the model has been fine tuned after the decomposition.

There is 1% drop in accuracy for SVD-half and 3.5% drop for SVD-quarter. Even though the increase in the error for the SVD can be mitigated by finetuning (the drop decreases to 0.1% for SVD-half-F and 1.3% for SVD-quarter-F), deep fried convnets still perform better both in terms of the accuracy and the number of parameters.

Applying a rank 600 SVD followed by fine tuning to the final softmax layer of the Adaptive Fastfood 32 model removes an additional 12.5M parameters at the expense of ~0.7% top-1 error.

For reference, we also include the results of Collins and Kohli [8], who pre-train a full network and use a sparsity regularizer during fine-tuning to encourage connections in the fully connected layers to be zero. They are able to achieve a significant reduction in the number of parameters this way, however the performance of their compressed network suffers when compared to the reference model. Another drawback of this method is that using sparse weight matrices requires additional overhead to store the indexes of the non-zero values. The index storage takes up space and using sparse representation is better than using a dense matrix only when number of nonzero entries is small.

8. Conclusion

Many methods have been advanced to reduce the size of convolutional networks at test time. In contrast to this trend, the Adaptive Fastfood transform introduced in this paper is end-to-end differentiable and hence it enables us to attain reductions in the number of parameters even at train time.

Model	Error	Params	Ratio
Collins and Kohli [8]	44.40%	—	—
SVD-half	43.61%	46.6M	0.8
SVD-half-F	42.73%	46.6M	0.8
Adaptive Fastfood 32	**41.93%**	32.8M	0.55
SVD-quarter	46.12%	23.4M	0.5
SVD-quarter-F	43.81%	23.4M	0.5
Adaptive Fastfood 16	42.90%	16.4M	0.28
Ada. Fastfood 32 (F-600)	42.61%	20.3M	0.35
Reference Model	42.59%	58.7M	1

Table 4. Comparison with other methods. The result of [8] is based on the the Caffe AlexNet model (similar but not identical to the Caffe reference model) and achieves ~4x reduction in memory usage, (slightly better than Fastfood 16 but with a noted drop in performance). SVD-half: 9216-2048-4096-2048-4096-500-1000 structure. SVD-quarter: 9216-1024-4096-1024-4096-250-1000 structure. F means after fine tuning.

Deep fried convnets capitalize on the proposed Adaptive Fastfood transform to achieve a substantial reduction in the number of parameters without sacrificing predictive performance on MNIST and ImageNet. They also compare favorably against simple test-time low-rank matrix factorization schemes.

Our experiments have also cast some light on the issue of random versus adaptive weights. The structured random transformations developed in the kernel literature perform very well on MNIST without any learning; however, when moving to ImageNet, the benefit of adaptation becomes clear, as it allows us to achieve substantially better performance. This is an important point which illustrates the importance of learning which would not have been visible from experiments only on small data sets.

The Fastfood transform allows for a theoretical reduction in computation from $\mathcal{O}(nd)$ to $\mathcal{O}(n \log d)$. However, the computation in convolutional neural networks is dominated by the convolutions, and hence deep fried convnets are not necessarily faster in practice.

It is clear looking at out results on ImageNet in Table 2 that the remaining parameters are mostly in the output softmax layer. The comparative experiment in Section 7 showed that the matrix of parameters in the softmax can be easily compressed using the SVD, but many other methods could be used to achieve this. One avenue for future research involves replacing the softmax matrix, at train and test times, using the abundant set of techniques that have been proposed to solve this problem, including low-rank decomposition, Adaptive Fastfood, and pruning.

The development of GPU optimized Fastfood transforms that can be used to replace linear layers in arbitrary neural models would also be of great value to the entire research community given the ubiquity of fully connected layers.

References

[1] D. Achlioptas. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66(4):671–687, 2003.

[2] N. Ailon and B. Chazelle. The Fast Johnson Lindenstrauss Transform and approximate nearest neighbors. *SIAM Journal on Computing*, 39(1):302–322, 2009.

[3] A. H. Bakhtiary, À. Lapedriza, and D. Masip. Speeding up neural networks for large scale classification using WTA hashing. *ArXiv*, 1504.07488, 2015.

[4] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra. Weight uncertainty in neural networks. In *ICML*, 2015.

[5] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

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

[7] Y. Cho and L. K. Saul. Kernel methods for deep learning. In *NIPS*, pages 342–350, 2009.

[8] M. D. Collins and P. Kohli. Memory bounded deep convolutional networks. Technical report, University of Wisconsin-Madison, 2014.

[9] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. *Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches*. Foundations and Trends on databases. Now Publishers, 2012.

[10] B. Dai, B. Xie, N. He, Y. Liang, A. Raj, M. Balcan, and L. Song. Scalable kernel methods via doubly stochastic gradients. In *NIPS*, 2014.

[11] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas. Predicting parameters in deep learning. In *NIPS*, pages 2148–2156, 2013.

[12] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *NIPS*, pages 1269–1277, 2014.

[13] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello. Hardware accelerated convolutional neural networks for synthetic vision systems. In *ISCAS*, pages 257–260, 2010.

[14] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. *ArXiv*, 1506.02626, 2015.

[15] G. E. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *ArXiv*, 1503.02531, 2015.

[16] P.-S. Huang, H. Avron, T. N. Sainath, V. Sindhwani, and B. Ramabhadran. Kernel methods match deep neural networks on timit. In *ICASSP*, 2014.

[17] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.

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

[19] H. Jaeger and H. Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667):78–80, 2004.

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

[21] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. Technical report, Google, 2014.

[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1106–1114, 2012.

[23] Q. Le, T. Sarlós, and A. Smola. Fastfood – approximating kernel expansions in loglinear time. In *ICML*, 2013.

[24] H. Li, R. Zhao, and X. Wang. Highly efficient forward and backward propagation of convolutional neural networks for pixelwise classification. Technical report, Chinese University of Hong Kong, 2014.

[25] M. Lin, Q. Chen, and S. Yan. Network in Network. In *ICLR*, 2014.

[26] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky. Sparse convolutional neural networks. In *CVPR*, 2015.

[27] J. Mairal, P. Koniusz, Z. Harchaoui, and C. Schmid. Convolutional kernel networks. In *NIPS*, pages 2627–2635. 2014.

[28] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[29] G. Pandey and A. Dukkipati. Learning by stretching deep networks. In *ICML*, pages 1719–1727, 2014.

[30] A. Rahimi and B. Recht. Random features for large-scale kernel machines. In *NIPS*, pages 1177–1184, 2007.

[31] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio. FitNets: Hints for thin deep nets. In *ICLR*, 2015.

[32] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet large scale visual recognition challenge. *ArXiv*, 1409.0575, 2014.

[33] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *ICASSP*, pages 6655–6659, 2013.

[34] A. Saxe, P. W. Koh, Z. Chen, M. Bhand, B. Suresh, and A. Ng. On random weights and unsupervised feature learning. In *ICML*, pages 1089–1096, 2011.

[35] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. Technical report, Google, 2014.

[36] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In *ICML*, pages 1113–1120, 2009.

[37] J. Xue, J. Li, and Y. Gong. Restructuring of deep neural network acoustic models with singular value decomposition. In *Interspeech*, pages 2365–2369, 2013.

[38] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *NIPS*, pages 3320–3328. 2014.