This CVPR paper is the Open Access version, provided by the Computer Vision Foundation.
Except for this watermark, it is identical to the version available on IEEE Xplore.

LCNN: Lookup-based Convolutional Neural Network

Hessam Bagherinezhad[1,2] Mohammad Rastegari[2,3] Ali Farhadi[1,2,3]

[1]University of Washington [2]XNOR.AI [3]Allen Institute for AI
{hessam, mohammad, ali}@xnor.ai

Abstract

Porting state of the art deep learning algorithms to resource constrained compute platforms (e.g. VR, AR, wearables) is extremely challenging. We propose a fast, compact, and accurate model for convolutional neural networks that enables efficient learning and inference. We introduce LCNN, a lookup-based convolutional neural network that encodes convolutions by few lookups to a dictionary that is trained to cover the space of weights in CNNs. Training LCNN involves jointly learning a dictionary and a small set of linear combinations. The size of the dictionary naturally traces a spectrum of trade-offs between efficiency and accuracy. Our experimental results on ImageNet challenge show that LCNN can offer $3.2\times$ *speedup while achieving* 55.1% *top-1 accuracy using AlexNet architecture. Our fastest LCNN offers* $37.6\times$ *speed up over AlexNet while maintaining* 44.3% *top-1 accuracy. LCNN not only offers dramatic speed ups at inference, but it also enables efficient training. In this paper, we show the benefits of LCNN in few-shot learning and few-iteration learning, two crucial aspects of on-device training of deep learning models.*

1. Introduction

In recent years convolutional neural networks (CNN) have played major roles in improving the state of the art across a wide range of problems in computer vision, including image classification [25, 37, 39, 18], object detection [11, 10, 36], segmentation [34, 32], etc. These models are very expensive in terms of computation and memory. For example, AlexNet[25] has 61M parameters and performs 1.5B high precision operations to classify a single image. These numbers are even higher for deeper networks, *e.g.,*VGG [37]. The computational burden of learning and inference for these models is significantly higher than what most compute platforms can afford.

Recent advancements in virtual reality (VR by Oculus) [33], augmented reality (AR by HoloLens) [14], and smart wearable devices increase the demand for getting our state of the art deep learning algorithm on these portable compute platforms. Porting deep learning methods to these platforms is challenging mainly due to the gap between what these platforms can offer and what our deep learning methods require. More efficient approaches to deep neural networks is the key to this challenge.

Recent work on efficient deep learning have focused on model compression and reducing the computational precision of operations in neural networks [3, 15, 35]. CNNs suffer from over-parametrization [7] and often encode highly correlated parameters [22], resulting in inefficient computation and memory usage[7]. Our key insight is to leverage the correlation between the parameters and represent the space of parameters by a compact set of weight vectors, called dictionary. In this paper, we introduce LCNN, a lookup-based convolutional neural network that encodes convolutions by few lookups to a dictionary that is trained to cover the space of weights in CNNs. Training LCNN involves jointly learning a dictionary and a small set of linear combinations. The size of the dictionary naturally traces a spectrum of trade-offs between efficiency and accuracy. Our experimental results using AlexNet on ImageNet challenge show that LCNN can offer $3.2\times$ speedup while achieving 55.1% top-1 accuracy. Our fastest LCNN offers $37.6\times$ speed up over CNN while maintaining 44.3% top-1 accuracy. In the ResNet-18, the most accurate LCNN offers $5\times$ speedup with 62.2% accuracy and the fastest LCNN offers $29.2\times$ speedup with 51.8% accuracy

In addition, LCNN enables efficient training; almost all the work in efficient deep learning have focused on efficient inference on resource constrained platforms [35]. Training on these platforms is even more challenging and requires addressing two major problems: i. **few-shot learning:** the settings of on-device training dictates that there won't be enough training examples for new categories. In fact, most training needs to be done with very few training examples; ii. **few-iteration learning:** the constraints in computation and power require the training to be light and quick. This imposes hard constraints on the number of iterations in training.LCNN offers solutions for both of these problems in deep on-device training.

Few-shot learning, the problem of learning novel categories from few examples (sometimes even one example), have been extensively studies in machine learning and computer vision[9]. The topic is, however, relatively new for deep learning[17], where the main challenge is to avoid overfitting. The number of parameters are significantly higher than what can be learned from few examples. LCNN, by virtue of having fewer parameters to learn (only around 7% of parameters of typical networks), offers a simple solution to this challenge. Our dictionary can be learned offline from training data where enough training examples per category exists. When facing new categories, all we need to learn is the set of sparse reconstruction weights. Our experimental evaluations show significant gain in few-shot learning; 6.3% in one training example per category.

Few-iteration learning is the problem of getting highest possible accuracy in few iterations that a resource constrained platform can offer. In a typical CNN, training often involves hundreds of thousands of iterations. This number is even higher for recent deeper architectures. LCNN offers a solution: dictionaries in LCNN are architecture agnostic and can be transferred across architectures or layers. This allows us to train a dictionary using a shallow network and transfer it to a deeper one. As before, all we need to learn are the few reconstruction weights; dictionaries don't need to be trained again. Our experimental evaluations on ImageNet challenge show that using LCNN we can train an 18-layer ResNet with a pre-trained dictionary from a 10-layer ResNet and achieve 16.2% higher top-1 accuracy on $10K$ iterations.

In this paper, we 1) introduce LCNN; 2) show state of the art efficient inference in CNNs using LCNN; 3) demonstrate possibilities of training deep CNNs using as few as one example per category 4) show results for few iteration learning .

2. Related Work

A wide range of methods have been proposed to address efficient training and inference in deep neural networks. Here, we briefly study these methods under the topics that are related to our approach.

Weight compression: Several attempts have been made to reduce the number of parameters of deep neural networks. Most of such methods [13, 46, 3, 15, 38] are based on compressing the fully connected layers, which contain most of the weights. These methods do not achieve much improvement on speed. In [21], a small DNN architecture is proposed which is fully connected free and has 50x fewer parameters in compare to AlexNet [26]. However, their model is slower than AlexNet. Recently [16, 15] reduced the number of parameters by pruning. All of these approaches update a pre-trained CNN, whereas we propose to train a compact structure that enables faster inference.

Low Rank Assumption: Approximating the weights of convolutional layers with low-rank tensor expansion has been explored by [22, 7]. They only demonstrated speedup in the case of large convolutions. [8] uses SVD for tensor decomposition to reduce the computation in the lower layers on a pre-trained CNN. [47] minimizes the reconstruction error of the nonlinear responses in a CNN, subject to a low-rank constraint which helps to reduce the complexity of filters. Notably, all of these methods are a post processing on the weights of a trained CNN, and none of them train a lower rank network from scratch.

Low Precision Networks: A fixed-point implementation of 8-bit integer was compared with 32-bit floating point activations in [41, 20]. Several network quantization methods are proposed by [13, 1, 29, 29, 19]. Most recently, binary networks has shown to achieve relatively strong result on ImageNet [35]. They have trained a network that computes the output with mostly binary operations, except for the first and the last layer. [5] uses the real-valued version of the weights as a key reference for the binarization process. [4] is an extension of [5], where both weights and activations are binarized. [23] retrains a previously trained neural network with binary weights and binary inputs. Our approach is orthogonal to this line of work. In fact, any of these methods can be applied in our model to reduce the precision.

Sparse convolutions: Recently, several attempts have been made to sparsify the weights of convolutional layers [31, 45, 44]. [31] shows how to reduce the redundancy in parameters of a CNN using a sparse decomposition. [45] proposed a framework to simultaneously speed up the computation and reduce the storage of CNNs. [44] proposed a Structured Sparsity Learning (SSL) method to regularize the structures (i.e., filters, channels, filter shapes, and layer depth) of CNNs. Only in [44] a sparse CNN is trained from scratch which makes it more similar to our approach. However, our method provides a rich set of dictionary that enables implementing convolution with lookup operations.

Few-Shot Learning: The problem of learning novel categories has been studied in [40, 2, 30]. Learning from few examples per category explored by [17]. [9, 42, 24] proposed a method to learn from one training example per category, known as one-shot learning. Learning without any training example, zero-shot learning, is studied by [27, 28].

3. Our Approach

Overview: In a CNN, each convolutional layer consists of n cubic weight filters of size $m \times k_w \times k_h$, where m and n are the number of input and output channels, respectively, and k_w and k_h are the width and the height of the filter. Therefore, the weights in each convolutional layer is composed of nk_wk_h vectors of length m. These vectors are shown to have redundant information[7]. To avoid this re-

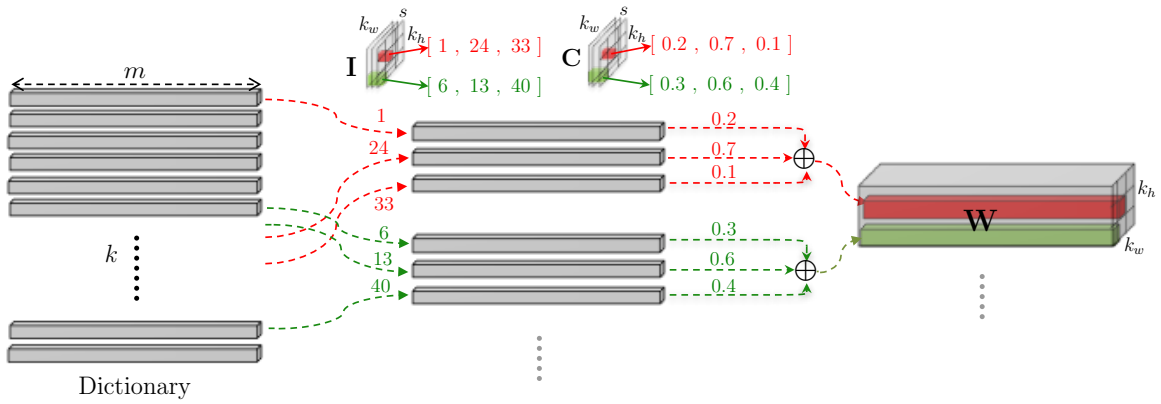


Figure 1. This figure demonstrates the procedure for constructing a weight filter in LCNN. A vector in the weight filter (the long colorful cube in the gray tensor $\mathbf{W}$) is formed by a linear combination of few vectors, which are looked up from the dictionary $\mathbf{D}$. Lookup indices and their coefficients are stored in tensors $\mathbf{I}$ and $\mathbf{C}$.

dundancy, we build a relatively small set of vectors for each layer, to which we refer as dictionary, and enforce each vector in the weight filter to be a linear combination of a few elements from this set. Figure 1 shows an overview of our model. The gray matrix at the left of the figure is the dictionary. The dashed lines show how we lookup a few vectors from the dictionary and linearly combine them to build up a weight filter. Using this structure, we devise a fast inference algorithm for CNNs. We then show that the dictionaries provide a strong prior on the visual data and enables us to learn from few examples. Finally, we show that the dictionaries can be transferred across different network architectures. This allows us to speedup the training of a deep network by transferring the dictionaries from a shallower model.

3.1. LCNN

A convolutional layer in a CNN consists of four parts: 1) the input tensor $\mathbf{X} \in \mathbb{R}^{m \times w \times h}$; where m, w and h are the number of input channels, the width and the height, respectively, 2) a set of n weight filters, where each filter is a tensor $\mathbf{W} \in \mathbb{R}^{m \times k_w \times k_h}$, where k_w and k_h are the width and the height of the filter, 3) a scalar bias term $b \in \mathbb{R}$ for each filter, and 4) the output tensor $\mathbf{Y} \in \mathbb{R}^{n \times w' \times h'}$; where each channel $\mathbf{Y}_{[i,:,:]} \in \mathbb{R}^{w' \times h'}$ is computed by $\mathbf{W} * \mathbf{X} + b$. Here $*$ denotes the discrete convolution operation[1].

For each layer, we define a matrix $\mathbf{D} \in \mathbb{R}^{k \times m}$ as the shared dictionary of vectors. This is illustrated in figure 1, on the left side. This matrix contains k row vectors of length m. The size of the dictionary, k, might vary for different layers of the network, but it should always be smaller than nk_wk_h, the total number of vectors in all weight filters of a layer. Along with the dictionary $\mathbf{D}$, we have a tensor for lookup indices $\mathbf{I} \in \mathbb{N}_{\leq k}^{s \times k_w \times k_h}$, and a tensor for lookup coefficients $\mathbf{C} \in \mathbb{R}^{s \times k_w \times k_h}$ for each layer. For a pair (r, c), $\mathbf{I}_{[:,r,c]}$ is a vector of length s whose entries are indices of the rows of the dictionary, which form the linear components of $\mathbf{W}_{[:,r,c]}$. The entries of the vector $\mathbf{C}_{[:,r,c]}$ specify the linear coefficients with which the components should be combined to make $\mathbf{W}_{[:,r,c]}$ (illustrated by a long colorful cube inside the gray cub in Figure 1-right). We set s, the number of components in a weight filter vector, to be a small number. The weight tensor can be constructed as follows:

$$\mathbf{W}_{[:,r,c]} = \sum_{t=1}^{s} \mathbf{C}_{[t,r,c]} \cdot \mathbf{D}_{[\mathbf{I}_{[t,r,c]},:]} \quad \forall r, c \tag{1}$$

This procedure is illustrated in Figure 1. In LCNN, instead of storing the weight tensors $\mathbf{W}$ for convolutional layers, we store $\mathbf{D}$, $\mathbf{I}$ and $\mathbf{C}$, the building blocks of the weight tensors. As a result, we can reduce the number of parameters in a convolutional layer by reducing k, the dictionary size, and s, the number of components in the linear combinations. In the next section, we will discuss how LCNN uses this representation to speedup the inference.

3.1.1 Fast Convolution using a Shared Dictionary

A forward pass in a convolutional layer consists of n convolutions between the input $\mathbf{X}$ and each of the weight filters $\mathbf{W}$. We can write a convolution between an $m \times k_w \times k_h$ weight filter and the input $\mathbf{X}$ as a sum of k_wk_h separate (1×1)-convolutions:

$$\mathbf{X} * \mathbf{W} = \sum_{r,c}^{k_h, k_w} \text{shift}_{r,c}(\mathbf{X} * \mathbf{W}_{[:,r,c]}) \tag{2}$$

, where $\text{shift}_{r,c}$ is the matrix shift function along rows and columns with zero padding relative to the filter size. Now

[1]The (:) notation is borrowed from NumPy for selecting all entries in a dimension.

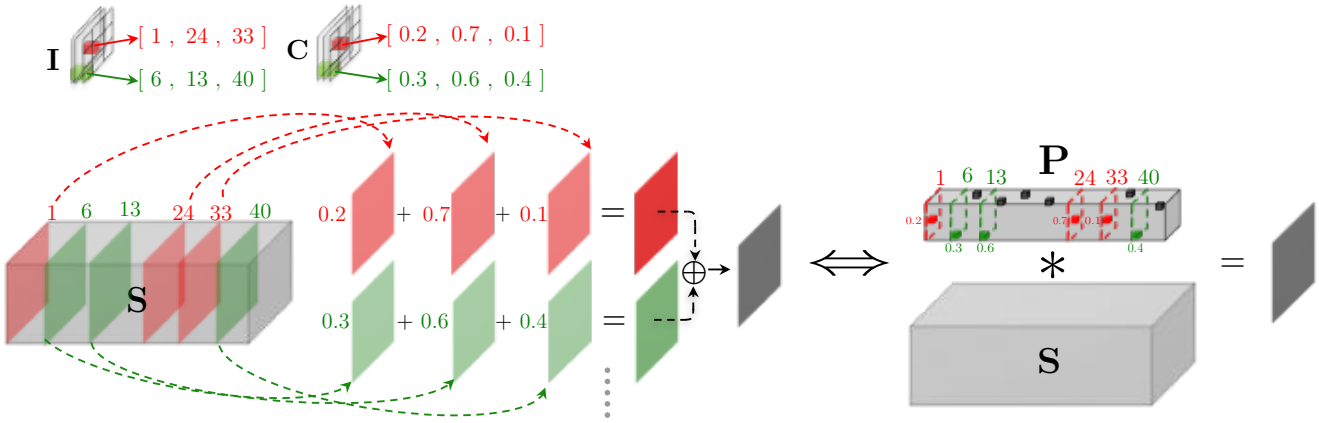


Figure 2. $\mathbf{S}$ is the output of convolving the dictionary with the input tensor. **The left side** of this figure illustrates the inference time forward pass. The convolution between the input and a weight filter is carried out by lookups over the channels of $\mathbf{S}$ and a few linear combinations. Direct learning of tensors $\mathbf{I}$ and $\mathbf{C}$ reduces to an intractable discrete optimization. **The right side** of this figure shows an equivalent computation for training based on sparse convolutions. Parameters $\mathbf{P}$ can be trained using SGD. The tiny cubes in $\mathbf{P}$ denote the non-zero entries.

we use the LCNN representation of weights (equation 1) to rewrite each 1×1 convolution:

$$\begin{aligned} \mathbf{X} * \mathbf{W} &= \sum_{r,c} \text{shift}_{r,c}(\mathbf{X} * (\sum_{t=1}^{s} \mathbf{C}_{[t,r,c]} \cdot \mathbf{D}_{[\mathbf{I}_{[t,r,c]},:]})) \\ &= \sum_{r,c} \text{shift}_{r,c}(\sum_{t=1}^{s} \mathbf{C}_{[t,r,c]} (\mathbf{X} * \mathbf{D}_{[\mathbf{I}_{[t,r,c]},:]})) \end{aligned} \tag{3}$$

Equation 3 suggests that instead of reconstructing the weight tensor $\mathbf{W}$ and convolving with the input, we can convolve the input with all of the dictionary vectors, and then compute the output according to $\mathbf{I}$ and $\mathbf{C}$. Since the dictionary $\mathbf{D}$ is shared among all weight filters in a layer, we can pre-compute the convolution between the input tensor $\mathbf{X}$ and all the dictionary vectors. Let $\mathbf{S} \in \mathbb{R}^{k \times w \times h}$ be the output of convolving the input $\mathbf{X}$ with all of the dictionary vectors $\mathbf{D}$, i.e.,

$$\mathbf{S}_{[i,:,:]} = \mathbf{X} * \mathbf{D}_{[i,:]} \quad \forall 1 \leq i \leq k \tag{4}$$

Once the values of $\mathbf{S}$ are computed, we can reconstruct the output of convolution by *lookups* over the channels of $\mathbf{S}$ according to $\mathbf{I}$, then *scale* them by the values in $\mathbf{C}$:

$$\mathbf{X} * \mathbf{W} = \sum_{r,c}^{k_h,k_w} \text{shift}_{r,c}(\sum_{t=1}^{s} \mathbf{C}_{[t,r,c]} \mathbf{S}_{[\mathbf{I}_{[t,r,c]},:,:]}) \tag{5}$$

This is shown in Figure 2 (left). Reducing the size of the dictionary k lowers the cost of computing $\mathbf{S}$ and makes the forward pass faster. Since $\mathbf{S}$ is computed by a dense matrix multiplication, we are still able to use OpenBlas [43] for fast matrix multiplication. In addition, by pushing the value of s to be small, we can reduce the number of lookups and floating point operations.

3.1.2 Training LCNN

So far we have discussed how LCNN represents a weight filter by linear combinations of a subset of elements in a shared dictionary. We have also shown that how LCNN performs convolutions efficiently in two stages: 1- *Small convolutions*: convolving the input with a set of 1×1 filters (equation 4). 2- *Lookup and scale*: few lookups over the channels of a tensor followed by a linear combination (equation 5) . Now, we explain how one can jointly train the dictionary and the lookup parameters, $\mathbf{I}$ and $\mathbf{C}$. Direct training of the proposed lookup based convolution leads to a combinatorial optimization problem, where we need to find the optimal values for the integer tensor $\mathbf{I}$. To get around this, we reformulate the lookup and scale stage (equation 5) using a standard convolution with sparsity constraints.

Let $\mathbf{T} \in \mathbb{R}^{k \times k_w \times k_h}$ be a one hot tensor, where $\mathbf{T}_{[t,r,c]} = 1$ and all other entries are zero. It is easy to observe that convolving the tensor $\mathbf{S}$ with $\mathbf{T}$ will result in $\text{shift}_{r,c}(\mathbf{S}_{[t,:,:]})$. We use this observation to convert the lookup and scale stage (equation 5) to a standard convolution. Lookups and scales can be expressed by a convolution between the tensor $\mathbf{S}$ and a sparse tensor $\mathbf{P}$, where $\mathbf{P} \in \mathbb{R}^{k \times w \times h}$, and $\mathbf{P}_{[:,r,c]}$ is a s-sparse vector (*i.e.* it has only s non-zero entries) for all spatial positions (r, c). Positions of the non-zero entries in $\mathbf{P}$ are determined by the index tensor $\mathbf{I}$ and their values are determined by the coefficient tensor $\mathbf{C}$. Formally, tensor $\mathbf{P}$ can be expressed by $\mathbf{I}$ and $\mathbf{C}$:

$$\mathbf{P}_{j,r,c} = \begin{cases} \mathbf{C}_{t,r,c}, & \exists t : \mathbf{I}_{t,r,c} = j \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

Note that this conversion is reversible, *i.e.,*we can create $\mathbf{I}$ and $\mathbf{C}$ from the position and the values of the non-zero

entries in $\mathbf{P}$. With this conversion, the lookup and scale stage (equation 5) becomes:

$$\sum_{rc} \text{shift}_{(r,c)}(\sum_{t=1}^{s} \mathbf{C}_{[t,r,c]} \mathbf{S}_{[\mathbf{I}_{[t,r,c]},:,:]}) = \mathbf{S} * \mathbf{P} \tag{7}$$

This is illustrated in Figure 2-right. Now, instead of directly training $\mathbf{I}$ and $\mathbf{C}$, we can train the tensor $\mathbf{P}$ with ℓ_0-norm constraints ($\|\mathbf{P}_{[:,r,c]}\|_{\ell_0} = s$) and then construct $\mathbf{I}$ and $\mathbf{C}$ from $\mathbf{P}$. However, ℓ_0-norm is a non-continuous function with zero gradients everywhere. As a workaround, we relax it to ℓ_1-norm. At each iteration of training, to enforce the sparsity constraint for $\mathbf{P}_{[:,r,c]}$, we sort all the entries by their absolute values and keep the top s entries and zero out the rest. During training, in addition to the classification loss L we also minimize $\sum_{[r,c]} \|\mathbf{P}_{[:,r,c]}\|_{\ell_1} = \|\mathbf{P}\|_{\ell_1}$, by adding a term $\lambda\|\mathbf{P}\|_{\ell_1}$ to the loss function. The gradient with respect to the values in $\mathbf{P}$ is computed by:

$$\frac{\partial(L + \lambda \|\mathbf{P}\|_{\ell_1})}{\partial \mathbf{P}} = \frac{\partial L}{\partial \mathbf{P}} + \lambda \, \text{sign}(\mathbf{P}) \tag{8}$$

where $\frac{\partial L}{\partial \mathbf{P}}$ is the gradient that is computed through a standard back-propagation. λ is a hyperparameter that adjusts the trade-off between the CNN loss function and the ℓ_1 regularizer. We can also allow s, the sparsity factor, to be different at each spatial position (r, c), and be determined automatically at training time. This can be achieved by applying a threshold function,

$$\delta(x) = \begin{cases} x, & |x| > \epsilon \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

over the values in $\mathbf{P}$ during training. We also backpropagate through this threshold function to compute the gradients with respect to $\mathbf{P}$. The derivative of the threshold function is 1 everywhere except at $|x| < \epsilon$, which is 0. Hence, if any of the entries of $\mathbf{P}$ becomes 0 at some iteration, they stay 0 forever. Using the threshold function, we let each vector to be a combination of arbitrary vectors. At the end of the training, the sparsity parameter s at each spatial position (r, c) is determined by the number of non-zero values in $\mathbf{P}[:, r, c]$.

Although the focus of our work is to speedup convolutional layers where most of the computations are, our lookup based convolution model can also be applied on fully connected (FC) layers. An FC layer that goes from m inputs to n outputs can be viewed as a convolutional layer with input tensor $m \times 1 \times 1$ and n weight filters, each of size $m \times 1 \times 1$. We take the same approach to speedup fully connected layers.

After training, we convert $\mathbf{P}$ to the indices and the coefficients tensors $\mathbf{I}$ and $\mathbf{C}$ for each layer. At test time, we follow equation 5 to efficiently compute the output of each convolutional layer.

3.2. Few-shot learning

The shared dictionary in LCNN allows a neural network to learn from very few training examples on novel categories, which is known as few-shot learning[17]. A good model for few-shot learning should have two properties: a) strong priors on the data, and b) few trainable parameters. LCNN has both of these properties. An LCNN trained on a large dataset of images (e.g. ImageNet [6]) will have a rich dictionary $\mathbf{D}$ at each convolutional layer. This dictionary provides a powerful prior on visual data. At the time of fine-tuning for a new set of categories with few training examples, we only update the coefficients in $\mathbf{C}$. This reduces the number of trainable parameters significantly.

In a standard CNN, to use a pre-trained network to classify a set of novel categories, we need to reinitialize the classification layer randomly. This introduces a large number of parameters, on which we don't have any prior, and they should be trained solely by a few examples. LCNN, in contrast, can use the dictionary of the classification layer of the pre-trained model, and therefore only needs to learn $\mathbf{I}$ and $\mathbf{C}$ from scratch, which form a much smaller set of parameters. Furthermore, for all other layers, we only fine-tune the coefficients $\mathbf{C}$, *i.e.,*only update the non-zero entries of $\mathbf{P}$. Note that the dictionary $\mathbf{D}$ is fixed across all layers during the training with few examples.

3.3. Few-iteration learning

Training very deep neural networks are computationally expensive and require hundreds of thousands of iterations. This is mainly due to the complexity of these models. In order to constrain the complexity, we should limit the number of learnable parameters in the network. LCNN has a suitable setting that allows us to limit the number of learnable parameters without changing the architecture. This can be done by transferring the shared dictionaries $\mathbf{D}$ from a shallower network to a deeper one.

Not only we can share a dictionary $\mathbf{D}$ across layers, but we can also share it across different network architectures of different depths. A dictionary $\mathbf{D} \in \mathbb{R}^{m \times k}$ can be used in any convolutional layer with input channel size m in any CNN architecture. For example, we can train our dictionaries on a shallow CNN and reuse in a deeper CNN with the same channel size. On the deeper CNN we only need to train the indices and coefficients tensors $\mathbf{I}$ and $\mathbf{C}$.

4. Experiments

We evaluate the accuracy and the efficiency of LCNN under different settings. We first evaluate the accuracy and speedup of our model for the task of object classification, evaluated on the standard image classification challenge of ImageNet, ILSRVC2012 [6]. We then evaluate the accuracy of our model under few-shot setting. We show that

	AlexNet		
Model	**speedup**	**top-1**	**top-5**
CNN	1.0×	56.6	80.2
Wen *et al*. [44]	3.1×[2]	**55.4**	N/A
XNOR-Net[35]	8.0×[3]	44.2	69.2
LCNN-fast	**37.6**×	44.3	68.7
LCNN-accurate	3.2×	**55.1**	**78.1**

Table 1. Comparison of different efficient methods on AlexNet. The accuracies are classification accuracy on the validation set of ILSVRC2012.

	ResNet-18		
Model	**speedup**	**top-1**	**top-5**
CNN	1.0×	69.3	90.0
XNOR-Net[35]	10.6×	51.2	73.2
LCNN-fast	**29.2**×	51.8	76.8
LCNN-accurate	5×	**62.2**	**84.6**

Table 2. Comparison of LCNN and XNOR-Net on ResNet-18. The accuracies are classification accuracy on the validation set of ILSVRC2012.

given a set of novel categories with as small as 1 training example per category, our model is able to learn a classifier that is both faster and more accurate than the CNN baseline. Finally we show that the dictionaries trained in LCNN are generalizable and can be transferred to other networks. This leads to a higher accuracy in small number of iterations compared to standard CNN.

4.1. Implementation Details

We follow the common way of initializing the convolutional layers by Gaussian distributions introduced in [12], including for the sparse tensor $\mathbf{P}$. We set the threshold in equation 9 for each layer in such a way that we maintain the same initial sparsity across all the layers. That is, we set the threshold of each layer to be $\epsilon = c \cdot \sigma$, where c is constant across layers and σ is the standard deviation of Gaussian initializer for that layer. We use $c = 0.01$ for AlexNet and $c = 0.001$ for ResNet. Similarly, to maintain the same level of sparsity across layers we need a λ (equation 8) that is proportional to the standard deviation of the Gaussian initializers. We use $\lambda = \lambda' \epsilon$, where λ' is constant across layers and ϵ is the threshold value for that layer. We try $\lambda' \in \{0.1, 0.2, 0.3\}$ for both AlexNet and ResNet to get different sparsities in $\mathbf{P}$.

The dictionary size k, the regularizer coefficient λ, and threshold value ϵ are the three important hyperparameters for gaining speedup. The larger the dictionary is, the more accurate (but slower) the model becomes. The size of the the dictionary for the first layer does not need to be very large as it's representing a 3-dimensional space. We observed that for the first layer, a dictionary size as small as 3 vectors is sufficient for both AlexNet and ResNet. In contrast, fully connected layers of AlexNet are of higher dimensionality and a relatively large dictionary is needed to cover the input space. We found dictionary sizes 512 and 1024 to be proper for fully connected layers. In AlexNet we use the same dictionary size across other layers, which we vary from 100 to 500 for different experiments. In ResNet, aside from the very first layer, all the other convolutional layers are grouped into 4 types of ResNet blocks. The dimensionality of input is equal between same ResNet block types, and is doubled for consecutive different block types. In a similar way we set the dictionary size for different ResNet blocks: equal between the same block types, and doubles for different consecutive block types. We vary the dictionary size of the first block from 16 to 128 in different experiments.

[2]They have not reported the overall speedup on AlexNet, but only per layer speedup. 3.1× is the weighted average of their per layer speedups.

[3]XNOR-Net gets 32× layer-wise speedup on a 32 bit machine. However, since they haven't binarized the first and the last layer (which has 9.64% of the computation), their overall speedup is 8.0×.

4.2. Image Classification

In this section we evaluate the efficiency and the accuracy of LCNN for the task of image classification. Our proposed lookup based convolution is general and can be applied on any CNN architecture. We use AlexNet [25] and ResNet [18] architectures in our experiments. We use ImageNet challenge ILSVRC2012 [6] to evaluate the accuracy of our model. We report standard top-1 and top-5 classification accuracy on 1K categories of objects in natural scenes. To evaluate the efficiency, we compare the number of floating point operations as a representation for speedup. The speed and the accuracy of our model depend on two hyperparameters: 1) k, the dictionary size and 2) λ, which controls the sparsity of $\mathbf{P}$; *i.e.,*the average number of dictionary components in the linear combination . One can

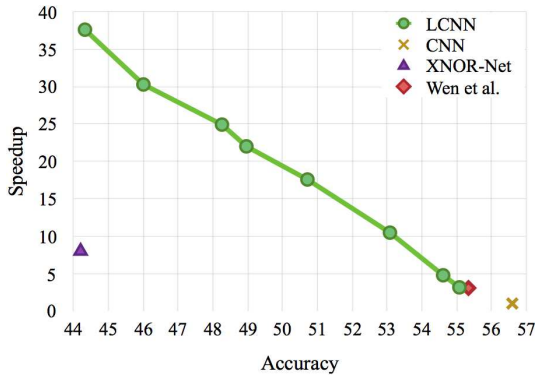
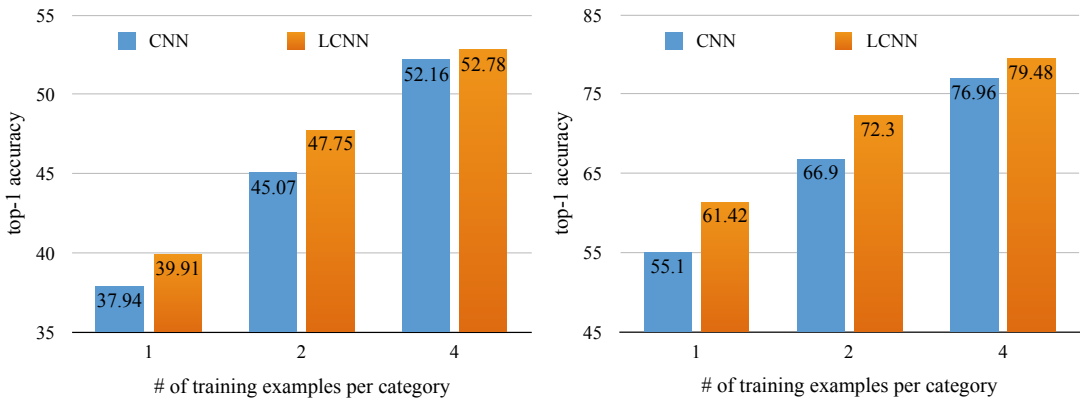


Figure 3. Accuracy vs. speedup. By tuning the dictionary size, LCNN achieves a spectrum of speedups.



(a) cats, sofas and bicycles excluded

(b) 10 random categories excluded

Figure 4. Comparison between the performance of LCNN and CNN baseline on few-shot learning, for $\{1, 2, 4\}$ examples per category. In (a) all cats (7 categories), sofas (1 category) and bicycles (2 categories) are held out for few-shot learning. In (b), 10 random categories are held out for few-shot learning. We repeat sampling the 10 random categories 5 times to avoid over-fitting to a specific sampling.

set a trade-off between the accuracy and the efficiency of LCNN by adjusting these two parameters. We compare our model with several baselines: 1- XNOR-Net [35], which reduces the precision of weights and outputs to 1-bit, and therefore multiplications can be replaced by binary operations. In XNOR-Net, all the layers are binarized except the first and the last layer (in AlexNet, they contain 9.64% of the computation). 2- Wen *et al*. [44], which speeds up the convolutions by sparsifying the weight filters.

Table 1 compares the top-1 and top-5 classification accuracy of LCNN with baselines on AlexNet architecture. It shows that with small enough dictionaries and sparse linear combinations, LCNN offers $37.6\times$ speedup with the accuracy of XNOR-Net. On the other hand, if we set the dictionaries to be large enough, LCNN can be as accurate as slower models like Wen *et al*. In LCNN-fast, the dictionary size of the mid-layer convolutions is 30 and for the fully connected layers is 512. In LCNN-accurate, the mid-layer convolutions have a dictionary of size 500 and the size of dictionary in fully connected layers is 1024. The regularizer constant (Section 4.1) λ' for LCNN-fast and LCNN-accurate is 0.3 and 0.1, respectively.

Depending on the dictionary size and λ', LCNN can achieve various speedups and accuracies. Figure 3 shows different accuracies vs. speedups that our model can achieve. The accuracy is computed by top-1 measure and the speedup is relative to the original CNN model. It is interesting to see that the trend is nearly linear. The best fitted line has a slope of -3.08, *i.e.,*for each one percent accuracy that we sacrifice in top-1, we gain 3.08 more speedup.

We also evaluate the performance of LCNN on ResNet-18 architecture. ResNet-18 is a compact architecture, which has $5\times$ fewer parameters in compare to AlexNet while it achieves 12.7% higher top-1 accuracy. That makes it a much more challenging architecture for further compression. Yet we show that we can gain large speedups with a few points drop in the accuracy. Table 2 compares the accuracy of LCNN, XNOR-Net [35], and the original model (CNN). LCNN-fast is getting the same accuracy as XNOR-Net while getting a much larger speedup. Moreover, LCNN-accurate is getting a much higher accuracy yet maintaining a relatively large speedup. LCNN-fast has dictionaries of size 16, 32, 64, and 128 for different block types. LCNN-accuracte has larger dictionaries: 128, 256, 512 and 1024 for different block types.

4.3. Few-shot Learning

In this section we evaluate the performance of LCNN on the task of few-shot learning. To evaluate the performance of LCNN on this task, we split the categories of ImageNet challenge ILSVRC2012 into two sets: i) base categories, a set of 990 categories which we use for pre-training, and ii) novel categories, a set of 10 categories that we use for few-shot learning.We do experiments under 1, 2, and 4 samples per category. We take two strategies for splitting the categories. One is random splitting, where we randomly split the dataset into 990 and 10 categories. We repeat the random splitting 5 times and report the average over all. The other strategy is to hold out all cats (7 categories), bicycles (2 categories) and sofa (1 category) for few-shot learning, and use the other 990 categories for pre-training. With this strategy we make sure that base and novel categories do not share similar objects, like different breeds of cats. For each split, we repeat the random sampling of 1, 2, and 4 training images per category 20 times, and get the average over all. Repeating the random sampling of the few examples is crucial for any few-shot learning experiment, since a model can easily overfit to a specific sampling of images.

We compare the performance of CNN and LCNN on few-shot learning in Figure 4. We first train an original AlexNet and an LCNN AlexNet on all training images of base categories (990 categories, 1000 images per category). We then replace the 990-way classification layer with a randomly initialized 10-way linear classifier. In CNN, this produces 10×4096 randomly initialized weights, on which we don't have any prior. These parameters need to be trained merely from the few examples. In LCNN, however, we transfer the dictionary trained in the 990-way classification layer to the new 10-way classifier. This reduces the number of randomly initialized parameters by at least a factor of 4. We use AlexNet LCNN-accurate model (same as the one in Table 1) for few-shot learning. At the time of fine-tuning for few-shot categories, we keep the dictionaries in all layers fixed and only fine-tune the sparse $\mathbf{P}$ tensor. This reduces the total number of parameters that need to be fine-tuned by a factor of $14\times$. We use different learning rates η and η' for the randomly initialized classification layer (which needs to be fully trained) and the previous pre-trained layers (which only need to be fine-tuned). We tried $\eta' = \eta$, $\eta' = \frac{\eta}{10}$, $\eta' = \frac{\eta}{100}$ and $\eta' = 0$ for both CNN and LCNN, then picked the best for each configuration.

Figure 4 shows the top-1 accuracies of our model and the baseline in the two splitting strategies of our few-shot learning experiment. In Figure 4 (a) we are holding out all cat, sofa, and bicycle categories (10 categories in total) for few-shot learning. LCNN is beating the baseline consistently in $\{1, 2, 4\}$ examples per category. Figure 4 (b) shows the comparison in the random splitting strategy. We repeat randomly splitting the categories into 990 and 10 categories 5 times, and report the average over all. Here LCNN gets a larger improvement in the top-1 accuracy compared to the baseline for $\{1, 2, 4\}$ images per category.

4.4. Few-iteration Learning

In section 3.3 we discussed that the dictionaries in LCNN can be transferred from a shallower network to a deeper one. As a result, one can train fewer parameters–only $\mathbf{I}$ and $\mathbf{C}$– in the deeper network with few iterations obtaining a higher test accuracy compared to a standard CNN. In this experiment we train a ResNet with 1 block of each type, 10 layers total. We then transfer the dictionaries of each layer to its corresponding layer of ResNet-18 (with 18 layers). After transfer, we keep the dictionaries fixed. We show that we get higher accuracy in small number of iterations compared to standard CNN. Figure 5 illustrates the learning curves on top-1 accuracy for both LCNN and standard CNN. The test accuracy of LCNN is 16.2% higher than CNN at iteration $10K$. The solid lines denote the training accuracy and the dashed lines denote the test accuracy.

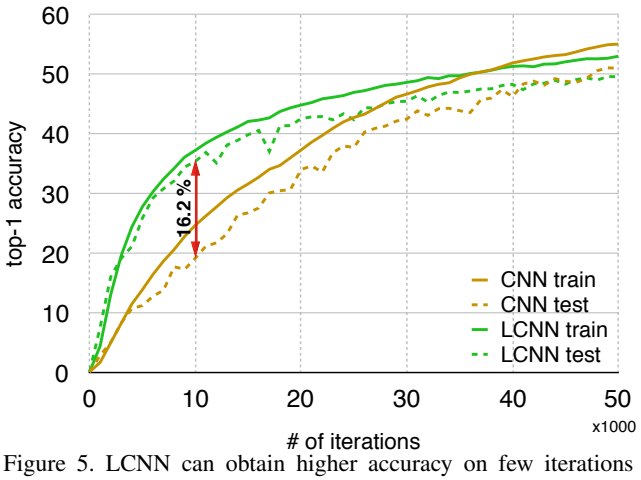


Figure 5. LCNN can obtain higher accuracy on few iterations by transferring the dictionaries $\mathbf{D}$ from a shallower architecture. This figure illustrates the learning curves on top-1 accuracy for both LCNN and standard CNN. The accuracy of LCNN is 16.2% higher than CNN at iteration $10K$.

5. Conclusion

With recent advancements in virtual reality, augmented reality, and smart wearable devices, the need for getting the state of the art deep learning algorithms onto these resource constrained compute platforms increases. Porting state of the art deep learning algorithms to resource constrained compute platforms is extremely challenging. We introduce LCNN, a lookup-based convolutional neural network that encodes convolutions by few lookups to a dictionary that is trained to cover the space of weights in CNNs. Training LCNN involves jointly learning a dictionary and a small set of linear combinations. The size of the dictionary naturally traces a spectrum of trade-offs between efficiency and accuracy.

LCCN enables efficient inference; our experimental results on ImageNet challenge show that LCNN can offer $3.2\times$ speedup while achieving 55.1% top-1 accuracy using AlexNet architecture. Our fastest LCNN offers $37.6\times$ speed up over AlexNet while maintaining 44.3% top-1 accuracy. LCNN not only offers dramatic speed ups at inference, but it also enables efficient training. On-device training of deep learning methods requires algorithms that can handle few-shot and few-iteration constrains. LCNN can simply deal with these problems because our dictionaries are architecture agnostic and transferable across layers and architectures, enabling us to only learn few linear combination weights. Our future work involves exploring low-precision dictionaries as well as compact data structures for the dictionaries.

Acknowledgments: This work is in part supported by ONR N00014-13-1-0720, NSF IIS-1338054, NSF-1652052, NRI-1637479, Allen Distinguished Investigator Award, and the Allen Institute for Artificial Intelligence.

References

[1] S. Anwar, K. Hwang, and W. Sung. Fixed point optimization of deep convolutional neural networks for object recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 1131–1135. IEEE, 2015. 2

[2] H. Azizpour, A. Sharif Razavian, J. Sullivan, A. Maki, and S. Carlsson. From generic to specific deep representations for visual recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 36–45, 2015. 2

[3] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen. Compressing neural networks with the hashing trick. In *ICML*, 2015. 1, 2

[4] M. Courbariaux and Y. Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, 2016. 2

[5] 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. 2

[6] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR09*, 2009. 5, 6

[7] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas. Predicting parameters in deep learning. In *NIPS*, 2013. 1, 2

[8] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *NIPS*, 2014. 2

[9] L. Fei-Fei, R. Fergus, and P. Perona. One-shot learning of object categories. *IEEE transactions on pattern analysis and machine intelligence*, 28(4):594–611, 2006. 2

[10] R. Girshick. Fast r-cnn. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1440–1448, 2015. 1

[11] 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. 1

[12] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010. 6

[13] Y. Gong, L. Liu, M. Yang, and L. Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014. 2

[14] M. Gottmer. Merging reality and virtuality with microsoft hololens. 2015. 1

[15] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *ICLR*, 2015. 1, 2

[16] 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. 2

[17] B. Hariharan and R. Girshick. Low-shot visual object recognition. *arXiv preprint arXiv:1606.02819*, 2016. 2, 5

[18] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, 2015. 1, 6

[19] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061*, 2016. 2

[20] K. Hwang and W. Sung. Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pages 1–6. IEEE, 2014. 2

[21] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡1mb model size. *CoRR*, abs/1602.07360, 2016. 2

[22] M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. In *British Machine Vision Conference (BMVC)*, 2014. 1, 2

[23] M. Kim and P. Smaragdis. Bitwise neural networks. *arXiv preprint arXiv:1601.06071*, 2016. 2

[24] G. Koch, R. Zemel, and R. Salakhutdinov. Siamese neural networks for one-shot image recognition. In *ICML Deep Learning Workshop*, 2015. 2

[25] 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. 1, 6

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012. 2

[27] C. H. Lampert, H. Nickisch, and S. Harmeling. Attribute-based classification for zero-shot visual object categorization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(3):453–465, 2014. 2

[28] J. Lei Ba, K. Swersky, S. Fidler, et al. Predicting deep zero-shot convolutional neural networks using textual descriptions. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4247–4255, 2015. 2

[29] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio. Neural networks with few multiplications. *arXiv preprint arXiv:1510.03009*, 2015. 2

[30] E. Littwin and L. Wolf. The multiverse loss for robust transfer learning. *arXiv preprint arXiv:1511.09033*, 2015. 2

[31] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky. Sparse convolutional neural networks. In *CVPR*, 2015. 2

[32] 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. 1

[33] V. Oculus. Oculus rift-virtual reality headset for 3d gaming. *URL: http://www. oculusvr. com*, 2012. 1

[34] P. O. Pinheiro, R. Collobert, and P. Dollar. Learning to segment object candidates. In *Advances in Neural Information Processing Systems*, pages 1990–1998, 2015. 1

[35] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016. 1, 2, 6, 7

[36] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems*, pages 91–99, 2015. 1

[37] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 1

[38] S. Srinivas and R. V. Babu. Data-free parameter pruning for deep neural networks. In *British Machine Vision Conference (BMVC)*, 2015. 2

[39] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015. 1

[40] S. Thrun. Is learning the n-th thing any easier than learning the first? *Advances in neural information processing systems*, pages 640–646, 1996. 2

[41] V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011. 2

[42] O. Vinyals, C. Blundell, T. Lillicrap, K. Kavukcuoglu, and D. Wierstra. Matching networks for one shot learning. *arXiv preprint arXiv:1606.04080*, 2016. 2

[43] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi. Augem: automatically generate high performance dense linear algebra kernels on x86 cpus. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013. 4

[44] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. In *NIPS*, 2016. 2, 6, 7

[45] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng. Quantized convolutional neural networks for mobile devices. *arXiv preprint*, 2015. 2

[46] Z. Yang, M. Moczulski, M. Denil, N. de Freitas, A. Smola, L. Song, and Z. Wang. Deep fried convnets. In *ICCV*, 2015. 2

[47] X. Zhang, J. Zou, X. Ming, K. He, and J. Sun. Efficient and accurate approximations of nonlinear convolutional networks. In *CVPR*, 2015. 2