Pruning ConvNets Online for Efficient Specialist Models

Jia Guo and Miodrag Potkonjak
University of California, Los Angeles
{jia, miodrag}@cs.ucla.edu

Abstract

Convolutional neural networks (CNNs) excel in various computer vision related tasks but are extremely computationally intensive and power hungry to run on mobile and embedded devices. Recent pruning techniques can reduce the computation and memory requirements of CNNs, but a costly retraining step is needed to restore the classification accuracy of the pruned model. In this paper, we present evidence that when only a subset of the classes need to be classified, we could prune a model and achieve reasonable classification accuracy without retraining. The resulting specialist model will require less energy and time to run than the original full model. To compensate for the pruning, we take advantage of the redundancy among filters and class-specific features. We show that even simple methods such as replacing channels with mean or with the most correlated channel can boost the accuracy of the pruned model to reasonable levels.

1. Introduction

In recently years, CNNs kept outperforming traditional machine learning models in computer vision [11][13][16][17][8]. However, the exceptionally high computation and memory requirements of these models hinder their massive deployment on mobile and embedded devices.

To tackle the problem, researchers proposed various ways to prune convolutional neural networks in order to reduce the computation and memory requirements of the models. A big body of works have focused on pruning parameters from fully connected layers [12][7][6]. Although large amount of parameters are needed to compute fully connected layers, it is the convolution layers that are dominating the computation [5]. Realizing this, researchers started to propose filter-level pruning methods with the aim to reduce computation requirements in convolutional operations [14][1][5]. Nevertheless, the procedures in all of the methods above follow similar patterns: examine which parameters/filters have less impact on the final outputs, prune the parameters/filters, retrain the network to regain classification. The functionality of the model remain unchanged.

We would like to approach the problem from a different angle. Instead of trying to prune the model and then retrain to regain all of its functionality, we explore the possibility of reducing the size together with some of the of functionalities of the original models in order to create specialist models. The notion *specialist model* also appeared in an earlier work that reported an attempt to classify a Google internal dataset that contains 15,000 classes [9]. In their problem setting, the purpose of creating specialist models is to assist the generalist model on the classification of specific classes, as a single model would not have a capacity large enough for all the classes. We believe that the specialist models alone would be meaningful in many use cases. To better illustrate the use cases, we will use a fictional example of a classifier that recognizes handwritten postal codes. A generic handwritten digit classifier is built for recognizing 10 digits. But for a postal office that is only responsible for delivery of postal code 10000, 10001 and 10002, all it needs is a classifier that recognizes digits 0, 1 and 2. The ability to recognize digit 3 through 9 is completely redundant for them. Therefore, it would be in the best interest of this poster office to prune the model such that it only distinguishes 3 digits in order to improve the energy efficiency of the recognition task. Granted that handwritten digit recognition is a lightweight application and that there is little gain in further pruning the network, the idea itself can be applied to a wider range of applications.

In this paper, we propose a method that create specialist models online without the need for retraining. The method removes the parts of the network that are tied closely to the non-essential classes. The idea is to first identify and remove the parts of the network that do not contribute as much in distinguishing between the target subset of classes. Then we use the mean or the most correlated filters to compensate for the filters we removed. We tested the method on off-the-shelf mobile devices and report effectiveness of the method.

The rest of the paper is organized in the following manner. Section 2 reviews related work. Section 3 lists the notations that we will use later to describe the proposed method.

Section 5 outlines the method on a conceptual level. Section 5 explains how the method can be implemented by directly modifying existing models in an online fashion. Section 6 evaluates the method through classification accuracy tests as well as energy and wall-clock time measurements.

2. Related Work

Researchers have proposed various methods to speed up the computation of deep neural networks. One straight forward approach is to reduce the precision of computation. Notable examples include weight quantization (weight sharing) [4] and binary neural networks [15]. Other researchers proposed to take advantage of the redundancies in the weight by adopting matrix/tensor decomposition techniques [2][10].

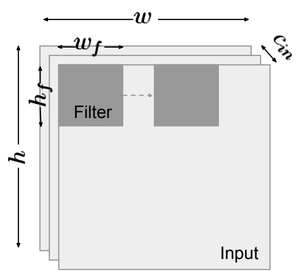
Our work is more concerned with a third cateogiry of methods that involves pruning neural networks. Le Cun *et al.* were one for the first researchers to introduce the idea of pruning neural networks, where he pruned parameters that analytically has less effects when perturbed [12]. Hassibi *et al.* proposed to use second order derivative to determine which parameters to prune [7]. Han *et al.* proposed to remove weights with magnitudes smaller than a threshold, and retrain the network to regain accuracy. [6]. Specifically for reducing the size convolutional neural networks, various researchers have proposed methods that prune the model at higher levels than individual neurons. Polyak *et al.* used the variance in activation to estimate the importance of feature maps [14]. In comparison, Li *et al.* used the sum of absolute weights of each filter as the criteria [5]. Anwar *et al.* pruned models at different levels and used particle filter to decide the best filters to prune [1].

In an earlier work, we proposed to prune CNNs and compensate using a linear combination of existing filters [3]. In this paper we show that we can achieve reasonable results using even simpler forms of compensation, such as the mean of the feature map and a single correlated channel.

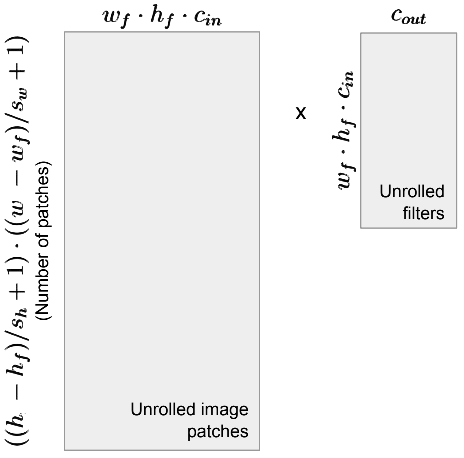
3. Notations

In almost all of today's machine learning frameworks, the computation of convolution operation is unrolled into a matrix multiplication, as shown in Figure 1. Let h_f and w_f be the height and width of the filter in a convolution layer with c_{out} number of filters and with inputs of shape $h \times w \times c_{in}$. Then before computing the convolution, the image patches with shape $h_f \times w_f$ for each channel c_{in} are unrolled into vectors, which make up for the rows in the matrix on the left in Figure 1(b). The weights for each of the c_{out} output channels are unrolled to form the columns in the matrix on the right.

When we involve parameters or variables from convolution layers in this paper, we are implicitly referring to those



(a)



$w_f \cdot h_f \cdot c_{in}$
c_{out}
$((h - h_f)/s_h + 1) \cdot ((w - w_f)/s_w + 1)$
(Number of patches)
x
$w_f \cdot h_f \cdot c_{in}$
Unrolled filters
Unrolled image patches

(b)

Figure 1. The unrolled convolution operation.

from the unrolled matrix multiplication form of convolution. In most cases, we are dealing with only one channel, the jth channel of the input. In these cases, the input feature map of ith sample is:

$$\boldsymbol{X}_{ij} = \begin{bmatrix} \boldsymbol{x}_{ij1}^{\mathsf{T}} \\ \boldsymbol{x}_{ij2}^{\mathsf{T}} \\ \vdots \\ \boldsymbol{x}_{ijH}^{\mathsf{T}} \end{bmatrix}$$

$\boldsymbol{X}_{ij}$ is of height $H = ((h - h_f)/s_h + 1) \cdot ((w - w_f)/s_w + 1)$ and width $W = w_f \cdot h_f$, where s_h and s_w are strides. We use $\boldsymbol{w}_{jk}$ to represent the unrolled weights vector for the jth input channel and kth output channel.

When it comes to pruning, we use $Y_{target} = \{y_1, y_2, \dots, y_{M'}\}$ to represent the subset of M' classes that the specialist model targets.

4. Method

4.1. Intuition

Recent works in visualizing convolution neural networks have shed light on the role that different layers in the network play [18]. It is now widely accepted that different filters in a convolution layer represent different features. The filters in earlier layers represent lower level features, while the filters in later layers represent higher level features.

Given the interpretation above, we have the following assumption: while some features might be universal, others are associated closely with certain classes. If we are to target a certain subset of classes and remove the rest, we only need the features that are associated with the target classes. We could remove the rest without severely damaging the core feature extraction abilities of the model. The errors introduced by pruning can be relatively easily compensated.

Thus, the whole method can be implemented online without the need of retraining. The following subsections will explain our methods in more details.

4.2. Pruning

While pruning filters accords with our interpretation of filters as features, there are other merits in this way of pruning convolution operations. Pruning entire filters in a convolution layer is effectively creating another convolution layer with a different (smaller) set of weights. It can be easily implemented on any existing machine learning frameworks. Thus pruning filters is a great fit to our proposal.

We now introduce our approach to pruning filters. We introduce our way of determining which filters to prune, the *Filter Sensitivity Analysis* method. Then we show how we apply the method to determine which filters to prune.

4.2.1 Filter Sensitivity Analysis

To determine which filters to prune, researchers have proposed to use criteria such as variances in the channel activation [14] and the sum of absolute weights [5]. For our particular use case, we need a method that could tie the criterion closely to particular classes. In other words, we want to prune filters that are less useful to the subset of the classes that are targeted in the specialist model. To achieve that objective, we propose *Filter Sensitivity Analysis*.

In traditional sensitivity analysis methods, usually, only one variable is at play. In the case of analyzing the impact of a channel on certain classes, we have $h \times w$ number of variables to examine. Apparently, we need to find a way to represent the feature map as a whole.

Our proposed method approaches this problem by assigning a weight variable $\omega_j = 1$ for each channel j of a layer and shifting our attention to the weight variable instead of the feature map itself. Suppose a sample produces an feature map $\boldsymbol{X}_j$ at channel j (now effectively $\omega_j \boldsymbol{X}_j$ after we added the weight), and p_y is the score/probability corresponding to the class y that the sample belongs to, then the *impact* $\mathcal{I}_j y$ of that channel j has on class y is defined as

$$\mathcal{I}_{jy} = \frac{\partial p_y}{\partial \omega_j}$$

In this paper, the impact is obtained by applying a small perturbation $\Delta\omega_f$ to the weight. We measure the difference in the output of the score Δp_y and calculate the impact by $\mathcal{I}_{jy} = \Delta P_y / \Delta\omega_f$. We average the impact values for every single sample to obtain the final impact.

To intuitively interpret the method, we can view ω_j as a variable that controls how much we want to strengthen or weaken the feature that the channel j represents. If we strengthen that feature by increasing ω_j, the more the class score changes, the more the feature is associated with the class. Thus the impact serves as a good indicator of whether we should remove a filter: if the channel that the filter produces has a high impact value on / is not closely associated with the class we need, then we should prune that channel.

4.2.2 Selecting Which Filters to Prune

Given the measure of a filter's impact on a particular class, we now describe the filter selection process. We set a threshold $I_{threshold}$, and prune filter j if j satisfies

$$\max\{\mathcal{I}_{jy} | y \in Y_{target}\} < \mathcal{I}_{threshold}$$

. The threshold can be pre-computed given the target percentage of removal Δc. If we want to remove $\Delta c = 50\%$ of, then we calculate $\max\{\mathcal{I}_{jy} | y \in Y_{target}\}$ for each j, and set the $50th$ percentile as the threshold $\mathcal{I}_{threshold}$.

4.3. Compensation

Once the filters are pruned, traditional methods will resort to retraining the network to regain accuracy. Early works reported 2x more time spent on retraining than training [6]. Util recently it still took 20-40 epochs of retraining for a pruned specialist model to regain accuracy [5]. If the desired objective is to create a model that classifies on a subset of labels for every ad hoc situation, a method that requires retraining is equivalent to training a new model every time. It is certainly inefficient and most likely infeasible. Therefore, an *online* method that requires no retraining is desired in this problem setting. This is where the process of what we call *compensation* comes into play. The objective of compensation is to restore activation of the filters that are pruned away from the model. The closer the restore values are to the original ones, the closer the more accurate our network will be.

We adopted two ways of compensation. The first way is to simply use the mean of feature map. The second is to use the feature map of the most similar filter. We choose between the two methods by using a simple heuristic.

4.3.1 Compensation with Mean

Since we are compensating for only a subset of classes, when we are calculating the mean of feature maps, we should only select the the samples that belong the the subset of classes Y_{target}. The mean of activation for jth channel $\bar{\boldsymbol{X}}_j$ is calculated by:

$$\bar{\boldsymbol{X}}_j = \frac{1}{|\{i|y_i \in Y_{target}\}|} \sum_{\{i|y_i \in Y_{target}\}} \boldsymbol{X}_{ij}$$

Once we obtain $\bar{\boldsymbol{X}}_j$, we use it as the feature map for channel j whatever the input data is.

4.3.2 Compensation with Correlated Filters

The first step of this method is to find the most correlated channel for all the channels to be pruned. To obtain the correlation measurement, we unroll the feature map of every channel into a vector. Let $\mathcal{C}_i(j', j)$ represent the Pearson correlation of the vectors of channel j' and j given input sample i. The correlation between channel j' and j is calculated by averaging the correlation of individual samples that belong to the target subset of classes:

$$\mathcal{C}(j', j) = \frac{1}{|\{i|y_i \in Y_{target}\}|} \sum_{\{i|y_i \in Y_{target}\}} \mathcal{C}_i(j', j)$$

To compensate for the channel j that is to be pruned, we choose a channel j' that bears the highest correlation value $\mathcal{C}_(j', j)$.

When we compensate channel j using the feature map of another channel j', we are effectively calculating the convolution on channel j' twice: the first time using its own weights $\boldsymbol{w}_{j'k}$, the second time using the weights for the pruned channel $\boldsymbol{w}_{jk}$. Inevitably there are errors caused by such replacement. However, there is no easy way of modifying the new feature map such that the values are closer to the original feature map. By easy, we mean that it could be implemented using operations existing in convolution neural networks. One workaround is to adjust the weights such that the next convolution layer will produce closer results.

Consulting the notation in Section 3, let $\boldsymbol{X}_{j'}$ represent the feature map of the replacement channel that compensates $\boldsymbol{X}_j$, and $\boldsymbol{w}'_{jk}$ is used to represent the new set of weights for output channel k that we want to analytically solve for. We set our objective to be minimizing the mean square error after we use the new feature map to calculate the convolution:

$$\underset{\boldsymbol{w}'_{jk}}{\text{argmin}}\, L(\boldsymbol{w}'_{jk}) = \sum_{\{i|y_i \in Y_{target}\}} \sum_{h=1}^{H} (\boldsymbol{x}_{ijh}^{\mathsf{T}} \boldsymbol{w}_{jk} - \boldsymbol{x}_{ij'h}{}^{\mathsf{T}} \boldsymbol{w}'_{jk})^2$$

By setting the derivative of the loss function to 0,

$$\begin{aligned}&\frac{\partial L(\boldsymbol{w}'_{jk})}{\partial \boldsymbol{w}'_{jk}} \\ &= -2 \sum_{\{i|y_i \in Y_{target}\}} \sum_{h=1}^{H} (\boldsymbol{x}_{ijh}^{\mathsf{T}} \boldsymbol{w}_{jk} - \boldsymbol{x}_{ij'h}{}^{\mathsf{T}} \boldsymbol{w}'_{jk}) \boldsymbol{x}_{ij'h} \\ &= 0\end{aligned}$$

We will have W equations for the vector $\boldsymbol{w}'_{jk}$ with W elements. We could solve for the new weights $\boldsymbol{w}'_{jk}$ using techniques that solve standard linear equations.

4.3.3 Selecting How to Compensate

To select whether to compensate a channel with its own mean feature map or with the most correlated channel, we use a simple heuristic. If the correlation is larger than an empirically determined threshold, then we use the latter method. Otherwise, we use the former method.

5. Implementation

Till now we have elaborated our method on a conceptual level. In this section, we will explain how we can implement the method by making modifications to an existing network model. Thus, we are essentially presenting a function f that takes in the original model together some parameters, and returns a smaller model:

$$pruned\_model = f(original\_model, parameters)$$

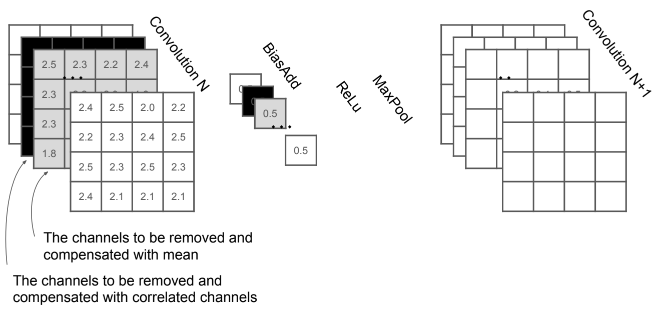
This function requires a relatively small footprint, can run online on mobile devices and does not require retraining and fine-tuning of the original model.

Similar to the previous section, we will introduce the procedures for pruning and compensation in separate subsections in the respective order.

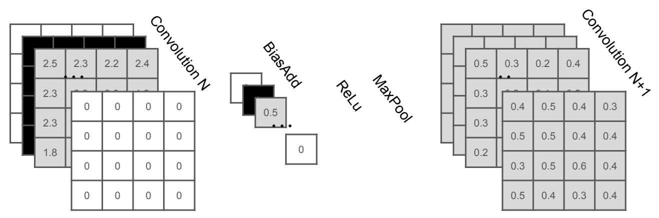
To illustrate the procedure of pruning and compensation, we will focus on a dummy convolution layer called Convolution N, as shown in Figure 2. After computing convolution on its input, the convolution N layer adds bias to the results, and runs them through a ReLu layer and a MaxPool layer before the next convolution layer Convolution $N+1$ takes them as its input.

5.1. Pruning

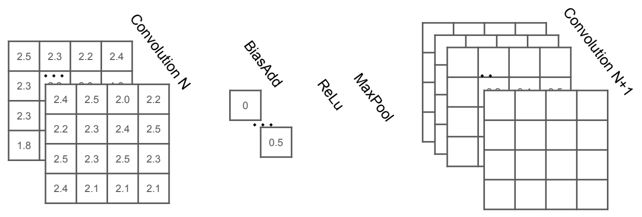
The implementation of pruning is straight forward. We simply need to remove the weights that are corresponding to the pruned channels from the previous layer and the current layer. Suppose the weight in the current convolution layer is $\boldsymbol{w}$ with shape $h_f \times w_f \times c_{in} \times c_{out}$ (for every channel among the c_{out} output channels, there is a $h_f \times w_f$ filter for


Convolution N
BiasAdd
ReLu
MaxPool
Convolution N+1
The channels to be removed and compensated with mean
The channels to be removed and compensated with correlated channels

(a) Before Pruning



(b) Compensating with Mean



(c) After Pruning

Figure 2. Implementation of pruning and compensation for convolution N.

every channel among c_{in} input channels). If we prune Δc_{in} percent of channels from previous layer and Δc_{out} percent from the current layer, which makes $c'_{in} = (1 - \Delta c_{in})c_{in}$ and $c'_{out} = (1 - \Delta c_{out})c_{out}$. Then the new weights will be in shape $h_f \times w_f \times c'_{in} \times c'_{out}$.

5.2. Compensation

5.2.1 Compensation with Mean

Implementing the compensation with the mean feature map is actually more tricky than it might appear. The key difficulty is that when we run the pruned specialist model, the network structure has already changed. Not only will the channels be removed, the weights that are originally used for these channels in the next convolution layer will no long exist as well. There is no way of placing the compensation around the layer where the channels are pruned.

To deal with the problem, we place the compensation at the next convolution layer. The idea is to look at the *contributions* of those pruned channels on the next convolution layer and try to compensate for their contributions. We first fill the feature maps of the channels to be removed with their mean values, and set the rest to 0. Then we run inference to obtain the feature maps at the next convolution layer. The values in those feature maps consists only of the contribution of the pruned channels. We save the values as a constant and add these constants to the next layer at run time.

5.2.2 Compensation with Correlated Channels

As we explained earlier, when replacing a pruned channel j with another channel j', we are effectively computing a convolution on the replace channel j' using weights for the pruned channel j. Since any output channel k is the sum of convolutions on all input channels, we simply add the adjusted weights of channel j that we mentioned in Section 4.3.2 to the existing weights of channel j':

$$\boldsymbol{w}'_{j'k} = \boldsymbol{w}_{j'k} + \boldsymbol{w}'_{jk}$$

If a channel is used to compensate for multiple pruned channels, we can add all the adjusted weights together.

5.3. Deployment and Computation Reduction

Throughout the paper we have been stressing one of the key advantages of this pruning and compensation method: it can be run online. Running online involves doing calculations with some pre-computed values. There are mainly four sets of values that we need to pre-compute:

1. The impacts each filter has on different classes $\mathcal{I}_{jy}$, described in Section 4.2.1;
2. The mean of feature maps $\bar{\boldsymbol{X}}_j$, described in Section 4.3.1;
3. The pairwise correlations $\mathcal{C}(j', j)$, described in Section 4.3.2;
4. The element wise product among pairs of channels feature maps used to compute the adjust weight $\boldsymbol{w}'_{jk}$ in Section 4.3.2.

These values can be pre-computed before the deployment of the model. When any user wants to prune a model, the procedure involves:

- Using 1 to determine which filters to prune;
- Using 3 to determine how pruned channels can be replaced by other channels, and 4 to recreate a new set of weights that accounts for the replacements;
- Using 2 to create the constant bias that are for the channels to be compensated by their mean values.

As we mentioned in Section 5.1. For a pruned convolution layer, the number of weights reduced from $h_f \times w_f \times c_{in} \times c_{out}$ to $h_f \times w_f \times (1 - \Delta c_{in}) c_{in} \times (1 - \Delta c_{out}) c_{out}$. That amounts to a $(1 - (1 - \Delta c_{in})(1 - \Delta c_{out}))$ reduction in the number of parameters and amount of computation.

6. Evaluation

6.1. Case Study

To clearly illustrate our approach, we present a 5-class pruned NIN model that achieves 38.9% computation savings as a case study. The NIN model, which is trained to classify the CIFAR-10 dataset, consists of 3 sets of *MLP convolution layers*. Each set consists of a 5×5 convolution layer (CONV) followed by 2 *cascaded cross channel parametric pooling(CCCP)* layer, which are effectively 1×1 convolution layers. The last CCCP layer of the last set outputs a total of 10 channels, each corresponding to one of the 10 classes. They use a global average pooling on that layer to produce the score for each class and then use a softmax layer to generate probabilities.

Table 1 shows the comparison between the original NIN model and the pruned specialist model. Through our experiments, we found out that the first few layers produce universal features, and are indispensable to the extraction of higher level features in later layers. Pruning those layers will significantly reduce the classification accuracy of the whole network. Thus we empirically decide to leave the first two layers intact while pruning the rest of the layers. The first two layers contribute to roughly 20% of the total amount of computation. For the rest of the layers, we set the target to be removing $\Delta c = 30\%$ of the filters. For the layer `CCCP2`, the first layer whose filters are removed, such removal amounts to 30% reduction in computation. For the rest of the layers (except the last layer), the computation is reduced by $1 - (1 - 30\%) \cdot (1 - 30\%) = 51\%$. For the last layer, since we have 5 classes, we remove 5 channels from a total of 10 channels. The computation reduction can thus be calculated as $1 - (1 - 30\%) \cdot (1 - 50\%) = 65\%$. The total amount of reduction adds up to 38.9%.

6.2. Classification Accuracy

The case study subsection above should serve as a clear illustration of the approach that we take to conduct experiments. We further repeat the approach with different sets of parameters and report the accuracy results in this subsection.

Table 2 shows the classification accuracy on CIFAR-10 dataset with the NIN model pruned at different levels. The values with dark grey background are equal of higher than the accuracy of the original 10-class model. Those with light gray background are within 6% of the standard accuracy. The columns represent the level of pruning measured in FLOPs, with values ranging from reducing 57.8% to "None", which represent the original network. The rows represent networks pruned for subsets of classes of different sizes. For each level of pruning (except for the original network), we randomly pick 10 different combinations of classes to make up for a subset and test the pruned network against the test set to obtain the mean and the 95% confidence interval of the classification accuracy. For the original network, the results are obtained after removing unneeded classes from the SoftMax output.

To make more sense of the different levels for pruning, we have to use Section 6.1 as a reference. The different level of pruning is achieved by removing different portions of all the channels from layer CCCP2 to CCCP6. As shown in the case study, 38.9% is achieved by removing 30% from the aforementioned list of channels. Similarly, the 57.8%, 48.6%, 27.9% and 14.3% reduction rates are achieved by reducing 50%, 40%, 20% and 10% of the channels respectively. These reduction rates are referred to as *levels of pruning* throughout the paper. As described in the subsection above, CCCP6, the last convolution layer of the network, has different channel counts given the size of the target subset of classes. For models with the same level of pruning but target different number of classes, only the last layer will be different. Since the last layer only contributes to 0.05% of the total number of FLOPs, we ignore the differences and consider models with the same level of pruning share the same computation requirements.

Note that the accuracies of some of the pruned specialist models are the same as or even higher than the original 10-class classifier. The reasons behind it stem from our problem setting of "pruning classes". Given our definition of the problem, the test set only contains samples of the chosen subset of the classes, and thus the problem space is smaller.

One problem that we observe in our experiments is that models for different subsets of classes with the same level of pruning vary in prediction accuracy. This problem is especially prominent when we prune larger portions of the network. For example, pruning half of the filters for some combination of two classes produces an accuracy of merely 0.624, while for other combinations it produces an accuracy as high as 0.923. We believe it has to do with the fact that different subsets of classes have a stronger dependency on different subsets of filters at each layer. Some classes depend on more features than others and thus need more channels at each layer. Some combination of classes could share similar sets of features and are more "compatible" with each other in a combination. In our approach, we prune uniform percentage of filters across all layers and for all subset of classes. That approach is clearly problematic. We have yet to devise a method that could compare the normalized impact of one filter (in any layer) on each of the class. We believe such a method could drastically improve the accu-

Name	Output Size	#Maps Before	#Params	#FLOPs Before	#Maps After	#FLOPs After	FLOPs Pruned
CONV1	32×32	192	1.92e+04	2.95e+07	192	2.95e+07	0%
CCCP1	32×32	160	3.09e+04	6.29e+07	160	6.29e+07	0%
CCCP2	32×32	96	1.55e+04	3.15e+07	67	2.20e+07	30.2%
CONV2	16×16	192	4.66e+05	2.36e+08	134	1.15e+08	51.3%
CCCP3	16×16	192	3.71e+04	1.89e+07	135	9.26e+06	51.0%
CCCP4	16×16	192	3.71e+04	1.89e+07	136	9.40e+06	50.3%
CONV3	8×8	192	3.34e+05	4.25e+07	136	2.13e+07	49.9%
CCCP5	8×8	192	3.71e+04	4.72e+06	134	2.33e+06	50.6%
CCCP6	8×8	10	1.93e+03	2.46e+05	5	8.58e+04	65.1%
Total				4.45e+08		2.72e+08	38.9%

Table 1. Comparison between the original 10-Class NIN model and a pruned 5-class NIN model.

FLOPs Pruned / Num. Classes	57.8%	48.6%	38.9%	27.9%	14.3%	None
2	84.1 ± 6.7	90.8 ± 4.0	95.2 ± 2.4	96.1 ± 3.1	97.7 ± 1.5	98.5 ± 0.6
3	72.7 ± 7.0	84.0 ± 4.5	91.1 ± 2.6	93.9 ± 1.9	95.4 ± 1.9	96.4 ± 1.2
4	62.0 ± 9.0	76.0 ± 7.6	85.8 ± 4.4	90.8 ± 2.7	93.3 ± 2.1	95.1 ± 1.0
5	59.8 ± 2.5	74.9 ± 3.2	83.8 ± 2.6	89.3 ± 1.7	92.4 ± 1.2	94.2 ± 0.8
6	55.1 ± 3.4	72.5 ± 2.9	82.7 ± 1.6	88.7 ± 0.8	92.0 ± 0.7	92.9 ± 0.9
7	47.0 ± 5.6	68.9 ± 2.5	79.5 ± 1.8	86.0 ± 1.5	89.6 ± 1.2	91.8 ± 0.8
8	47.4 ± 2.9	67.7 ± 1.2	78.7 ± 1.3	86.3 ± 0.7	89.8 ± 0.7	90.7 ± 0.6
9	36.3 ± 2.2	65.8 ± 1.5	75.4 ± 1.2	83.7 ± 0.7	88.0 ± 0.5	90.3 ± 0.6
10	28.7	62.6	74.3	82.7	87.2	89.6

Table 2. Classification accuracy on CIFAR-10 dataset using NIN with different levels of pruning.

racy of the pruned networks.

6.3. Energy Consumption and Wall-Clock Time

Using the mobile support from TensorFlow, we are able to port the model to off-the-shelf smartphones. We conduct energy consumption and wall-clock-time measurement on a Google Nexus 4 Android Smartphone, and report results in this section. The energy is measured using a Monsoon power monitor. The wall-clock time is obtained by checking system time before and after running the inference.

Table 3 lists the energy measurement results. The average energy consumption of running inference on one sample as well as the percentage of energy saved are reported for different levels of pruning. To obtain the energy consumption per sample, we first measure the total amount of energy of running inference on all the samples with a certain model. Then, we measure the energy used for loading data by running a dummy neural network that stay idle for the duration of running the inference. The energy shown in the table is the difference between the two measurements stated above.

In the case where we prune 57.8% of the FLOPs, we achieve 41.73% in energy savings. For a 14.3% rate of pruning, the energy savings drop to less than 2%. The reason that we are not getting exactly proportional savings is that in compensating with mean we created another ”add” layer that involves additional arithmetic and memory operations.

Table 4 shows the wall-clock time improvements. We are not getting significant improvements in terms of wall-clock time. The smallest model grants around 25% savings in delay. Yet for less pruned models, the savings from pruning filters gets amortized by the additional add operation, and model takes the same or even more time to run compared to the original one.

7. Conclusion

In this paper, we propose an online method of pruning CNNs to create new specialist models that only classifies a subset of the classes for better energy efficiency in different use cases. In this method, we first prune filters that are of less importance to the target subset of classes using the proposed filter sensitivity analysis method. Then we compensate for the filters pruned using either the mean feature map or the feature map of the most correlated filter. We test our method on the Network in Network [13] model on the

FLOPs Pruned	57.8%	48.6%	38.9%	27.9%	14.3%	None
Energy (mJ)	157.9	187.9	209.6	237.3	266.0	271.0
Energy Saved	41.73%	30.67%	22.66%	12.44%	1.845%	-

Table 3. Average inference energy consumption per sample.

FLOPs Pruned	57.8%	48.6%	38.9%	27.9%	14.3%	None
Wall-Clock Time (ms)	60.89±0.22	72.97±0.26	78.00±0.26	86.54±0.26	86.75±0.22	81.08±0.21
Time Saved	24.90%	10.00%	3.799%	-6.734%	-6.993%	-

Table 4. Average inference wall-clock time per sample.

CIFAR-10 dataset. In the extreme case of pruning the model down to a binary classifier, we could get 22.66% in energy savings and achieve 95.2% in accuracy with some variance. It is 5.6% better than the original 10-class classifier while 3.3% worse than the full model tuned for 2 classes.

8. Acknowledgments

This work was supported in part by the NSF under award CNS-1059435 and award CNS-1513306. We would like to thank anonymous reviews for their comments and suggestions.

References

[1] S. Anwar, K. Hwang, and W. Sung. Structured pruning of deep convolutional neural networks. *arXiv preprint arXiv:1512.08571*, 2015. 1, 2

[2] 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 (NIPS)*, pages 1269–1277, 2014. 2

[3] J. Guo and M. Potkonjak. Pruning filters and classes: Towards on-device customization of convolutional neural networks. *Proceedings of the 1st International Workshop on Embedded and Mobile Deep Learning*, 2017. to appear. 2

[4] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *International Symposium on Computer Architecture (ISCA)*, 2016. 2

[5] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *6th International Conference on Learning Representations (ICLR)*, 2016. 1, 2, 3

[6] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1135–1143, 2015. 1, 2, 3

[7] B. Hassibi and D. G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems (NIPS)*, pages 164–171, 1992. 1, 2

[8] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. 1

[9] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. 1

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

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1106–1114, 2012. 1

[12] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems (NIPS)*, pages 598–605, 1989. 1, 2

[13] M. Lin, Q. Chen, and S. Yan. Network in network. In *2nd International Conference on Learning Representations (ICLR)*, 2014. 1, 7

[14] A. Polyak and L. Wolf. Channel-level acceleration of deep face representations. *IEEE Access*, 3:2163–2175, 2015. 1, 2, 3

[15] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision(ECCV)*, pages 525–542, 2016. 2

[16] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *3th International Conference on Learning Representations (ICLR)*, 2015. 1

[17] 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 (CVPR)*, pages 1–9, 2015. 1

[18] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson. Understanding neural networks through deep visualization. In *ICML Deep Learning Workshop*, 2015. 3