Pruning Filters and Classes: Towards On-Device Customization of Convolutional Neural Networks

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

ABSTRACT

In recent years, we have witnessed more and more mobile applications based on deep learning. Widely used as they may be, those applications provide little flexibility to cater to the diversified needs of different groups of users. For users facing a classification problem, it is natural that some classes are more important to them, while the rest are not. We thus propose a lightweight method that allows users to prune the unneeded classes together with associated filters from convolutional neural networks (CNNs). Such customization can result in substantial reduction in computational costs at test time. Early results have shown that after pruning the Network-in-Network (NIN) model on CIFAR-10 dataset[12] down to a 5-class classifier, we can trade a 3% loss in accuracy for a 1.63× gain in energy consumption and a 1.24× improvement in latency when experimenting on an off-the-shelf smartphone, while the procedure incurs with very little overhead. After pruning, the custom-tailored model can still achieve a higher classification accuracy than the unmodified classifier because of a smaller problem space that more accurately reflects users' needs.

Keywords

Deep learning; pruning; customization

1. INTRODUCTION

In today's deep learning app development workflow, developers usually train one model on the servers and ship the same model to all end users. It is intrinsically hard for developers to cater to the diversified needs of different users. To satisfy the needs of all the users, the model has to be a universal one that includes different capabilities needed by different groups of users. In that case, the model will inevitably include some of the unnecessary capabilities which a particular user does not need. In the case of classification, for example, the model might be able to identify more classes than what a user would need it to. Evidently, an increase in the number of classes that the model could classify is accompanied by an increase in the model size and computational costs. Therefore, resources are wasted for a user when s/he is running inference using a model more powerful than needed.

EMDL'17, June 23, 2017, Niagara Falls, NY, USA

DOI: http://dx.doi.org/10.1145/3089801.3089806

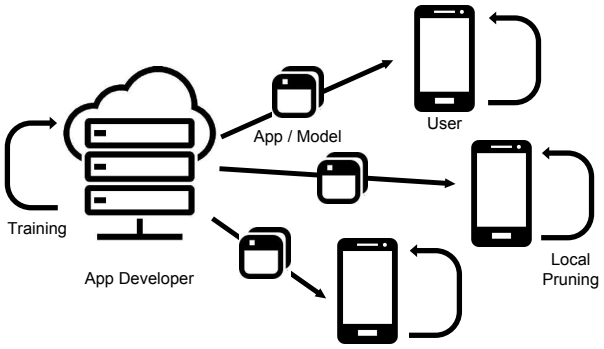


Figure 1: Once the neural network models are trained and shipped to users, each user will run a local program to prune the model according to their needs.

To tackle the problem, one might think of training different, smaller models for different users. However, the extremely high computational costs of training a model prevents that from being feasible. A more desired approach would be to *prune* unneeded classes and the parts associated with those classes from the existing model. The procedure should be so lightweight that users could run it on their own mobile devices. We will later refer to it as "on-device customization". An on-device customization method would enable a new workflow of application development, as shown in Figure 1. In the new workflow, developers still train one model, albeit a comprehensive one that is endowed with all the possible capabilities. Then the users use a lightweight program that prunes the model locally on the phone to the extent that only the capabilities that the users want are left. After that, the user could run faster and more energy efficient inference using the smaller pruned model.

In recent years, large bodies of work have emerged on pruning neural network models. However, we are yet to see a method that could be facilitated by a lightweight program. So far, many researchers have focused on pruning individual parameters [10][7][4][5]. In particular, Han *et al.* achieved impressive memory savings on widely used models [4][5]. Other researchers proposed filter-level pruning methods with the aim to reduce computational requirements in convolution operations [13][11]. There have been proposals to include different levels of pruning on mobile devices and leverage the cloud to achieve a trade-off between energy and accuracy [6]. But none of the aforementioned methods carry out pruning on the mobile device, as they all require tens of epochs of fine tuning before the model regains accuracy. In DeepX and SparseSep, the authors used matrix decomposition based methods

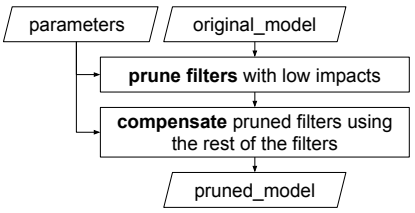


Figure 2: Overview of our method.

to approximate weight tensors [9][2], similar to methods proposed in [8] and [3]. This method can be applied without retraining given some loss in accuracy, but it is unclear to us whether it could help us in the task of pruning classes.

In this paper, we report early results from a method that enables on-device customization of neural network models without the requirement for retraining. The key idea of the proposed method follows two steps: pruning and compensation. We 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 utilize the remaining filters to compensate for the pruned filters. Aside from a few parameters that have to be pre-calculated before shipping the model, the whole procedure incurs little overhead. We have developed the method and tested it on the Network-in-Network(NIN) model with CIFAR-10 dataset implemented on an off-the-shelf smartphone using TensorFlow mobile support [1]. Since the proposed method only prunes filters related with certain unneeded classes, it could potentially be applied in parallel to the aforementioned pruning methods as a client-side counterpart.

The rest of the paper is organized in the following manner. Section 2 gives an overview of our method. Section 3 and Section 4 outline our pruning method and compensation method respectively. For both methods, we first introduce them on a conceptual level and then demonstrate how we could realize the concept through modifications of the existing model. Section 5 describes some of the implementation details. It also goes provides a description of the pre-computation steps and the overall procedure. Section 6 evaluates the method through classification accuracy tests as well as energy and latency measurements on the smartphone.

2. OVERVIEW

The program for on-device customization can be represented as a function `f` that takes in the original model together some parameters, and returns a smaller model: `pruned_model = f( original_model, parameters)`. The parameters mentioned here are a list of pre-computed values needed for pruning and compensation. This function requires a relatively small footprint, can run on mobile devices and does not require retraining and fine-tuning of the original model.

Figure 2 shows an overview of `f`. The function operates in 2 major steps. It first identifies and removes filters in convolution layers that are less important in distinguishing between the target subset of classes. Those filters are determined to be low-impact filters and we could thus tolerate some amount of errors in calculating them. In the next step, we use a linear combination of the rest of the filters to approximate and replace the filters we removed. Both steps require some pre-calculated parameters to finish.

3. PRUNING FILTERS

3.1 Selecting Filters with Low Impacts

To determine which filters to prune, researchers have proposed to use criteria such as variances in the channel activation [13] and the sum of absolute weights [11]. In our approach, we make the assumption that different filters represent features, and that some are important to the target classes while others are not. We propose a gradient based method that aims to identify the *impact* of a particular filter on given classes. In Section 6, we will present a comparison between different pruning criteria.

Suppose that we are pruning an arbitrary convolution layer. Filter j of that layer produces a feature vector $\boldsymbol{x}_{ij}$ [1] given the ith sample. $P(y_i)$ is the probability (SoftMax output) corresponding to the class y_i that the sample belongs to. Then the impact $\delta_j y$ of that filter j has on class y is defined as

$$\delta_{jy} = \sum_{\{i|y_i=y\}} || \frac{\partial P(y_i)}{\partial \boldsymbol{x}_{ij}} \cdot \boldsymbol{x}_{ij} ||$$

This definition intuitively reflects the effect on $P(y_i)$ if we scale up the filter j by an infinitely small amount. For each sample, there will be a different impact value. We average all the values among samples belonging to the same class to obtain the final impact on that class.

Our filter selection procedure is based on the impact. Let Y_{target} be the set that contains the classes that we want in the pruned model. We set a threshold $\delta_{threshold}$, and prune filter j if j satisfies the following

$$\sum_{\{y \in \mathsf{Y}_{target}\}} \delta_{jy} < \delta_{threshold} \quad (1)$$

The threshold can be pre-computed given the target percentage of removal ΔN. If we want to remove $\Delta N = 50\%$ of the filters in a layer, we calculate $\sum_{\{y \in \mathsf{Y}_{target}\}} \delta_{jy}$ for each j, and set the $50th$ percentile value as the threshold $\delta_{threshold}$.

3.2 Implementing Filter Pruning

The implementation of pruning is straight forward. We simply need to remove the weights corresponding to the pruned channels from the previous layer and the current layer. Suppose the weight tensor in a convolution layer is $\boldsymbol{W}$ with shape $C \times H \times W \times N$ (N filter banks that has one $H \times W$ filter for each of the C input channels). If we prune ΔC percent of channels from previous layer and ΔN percent from the current layer, the new weight tensor will be of shape $(1 - \Delta C)C \times H \times W \times (1 - \Delta N)N$.

4. COMPENSATION

4.1 Compensating with Linear Combination of Filters

Once the filters are pruned, traditional methods will retrain network for at least 20-40 epochs of retraining for a pruned model to regain accuracy [11]. It is certainly inefficient, if not infeasible, to run the retraining step on mobile devices. To avoid the retraining step, we need an alternative way of compensating for the errors caused by pruning the model. Fortunately, the filters that we remove do not have significant impacts the classes that we want, and we could tolerate some errors in these filters. We thus propose to approximate the output of the filter by linear combinations of the

[1]We use feature vector to denote a feature map that is flattened into a vector.

rest of the filters. The ides can appear similar to the proposed methods in [14], although we are only approximating the filters rather than the input image itself.

To formally state our goal, approximating the output of filter j with a linear combination of the rest of the filter is expressed as follows

$$\hat{\boldsymbol{x}_j} = \sum_{k=1}^{N} z_k \beta_{kj} \boldsymbol{x}_k$$

here $z_k \in \{0, 1\}$ represents whether or not filter k is to be pruned ($z_k = 0$ means that filter k will be pruned), N is the total number of filters, and β_{kj} is the coefficient in the linear combination. We can solve for the best set of coefficients by minimizing the Mean Squared Error (MSE) over the subset of samples belonging to target classes

$$\underset{\beta_{jk}}{\operatorname{argmin}} \sum_{\{i|y_i \in \mathsf{Y}_{target}\}} \sum_{j=1}^{N} (1 - z_j) \cdot || \sum_{k=1}^{N} z_k \beta_{jk} \boldsymbol{x}_{ik} - \boldsymbol{x}_{ij} ||_2^2 \tag{2}$$

Intuitively, the expression represents the difference between $\boldsymbol{x}_j$ and $\hat{\boldsymbol{x}_j}$. Further we only take into consideration samples from the classes that we care about. This minimization can be easily solved by taking the derivatives of the MSE. The seemingly complex process can actually be pre-computed. We will talk more about it in Section 5.2.

Algorithm 1 Pruning and Compensating a Model

Input: (i) Y_{target}: the target subset of classes (ii) ΔN: the target removal rate (iii) the original model. $\boldsymbol{W}_l$ of shape $C_l \times H_l \times W_l \times N_l$ is the weight tensor of the lth convolution layer (iv) the pre-computed parameters

Output: (i)the pruned model. $\boldsymbol{W'}_l$ represents new the new weight tensor

1: z_j, β_{kj} = 1
2: **for** each convolution layer l **do**
3: ▷ update weights using z and β from the last layer
4: **for** $k := 1 : C_l$ **do**
5: $\boldsymbol{W}'(k, :, :, :) = \boldsymbol{W}(k, :, :, :) +$
6: $\sum_{j=1}^{N} z_j \beta_{kj} \cdot \boldsymbol{W}(j, :, :, :)$ ▷ Equation 3
7: **end for**
8: Remove $\boldsymbol{W}'_l(j, :, :, :)$ if $z_j = 0$
9: ▷ update z and β for the next layer
10: **for** $j := 1 : N_l$ **do**
11: $\delta_j := \sum_{\{y \in \mathsf{Y}_{target}\}} \delta_{jy}$ ▷ using the pre-computed impacts δ_{jy}
12: **end for**
13: $\delta_{threshold} :=$ ΔNth percentile of $\{\delta_j\}$
14: **for** $j := 1 : N_l$ **do**
15: $z_j := \delta_j < \delta_{threshold}$
16: **end for**
17: Obtain β_{kj} by solving Equation 2 ▷ using the pre-computed pairwise products
18: Remove $\boldsymbol{W}'_l(:, :, :, j)$ if $z_j = 0$
19: **end for**

4.2 Implementing Compensation by Weight Modification

The linear nature of convolution allows us to implement our compensation method through direct modifications of the weight tensors. Suppose filter j is pruned, then the weight tensor of the *next* convolution layer should be modified for compensation. Specifically, let $\boldsymbol{W}$ of shape $C \times H \times W \times N$ be the weight tensor of the next convolution layer. We use $\boldsymbol{W}(j, :, :, :)$ to represent the weights for the jth input channel (the channel pruned). For any filter k that is not pruned, its new weight tensor should be its original weight tensor plus the weights for the pruned channels multiplied by the corresponding coefficients

$$\boldsymbol{W}'(k, :, :, :) = \boldsymbol{W}(k, :, :, :) + \sum_{j=1}^{N} z_j \beta_{kj} \cdot \boldsymbol{W}(j, :, :, :) \tag{3}$$

The added weights effectively represent separate convolution operations for the purpose of compensating the pruned channels. But since the convolution operation is linear, we can just add them to the existing weights. Once the pruning described in Section 3.2 is done, the weight tensor will only contain the channels with updated weights $\boldsymbol{W}'$.

5. IMPLEMENTATION

In the following, we describe some of the implementation details of our method. Further, we show the pre-computation needed for our method followed by an algorithm that describes the overall procedure.

5.1 TensorFlow Models

TensorFlow uses a protocol buffer class called GraphDef to serialize the model structure and weights into language independent binaries. The current version of TensorFlow Android support requires loading protocol buffer based models and invoking Java Native Interface (JNI) for running the C-based TensorFlow inference library. The `original_model` and the `pruned_model` that we mentioned in Section 2 are in reality serialized GraphDef protocol buffers. The function we implemented directly operates on the GraphDef models to carry out pruning and compensation.

5.2 Pre-computation

There are mainly two sets of pre-computed parameters that we need for successful pruning and compensation:

Impacts δ. As noted in Section 3.1, the impacts of each filter on each of the output classes have to be pre-computed. The computation can be implemented effectively using TensorFlow `gradient` and `gather_nd` operation and one forward and one backward pass over all the training data for each layer.

Pairwise Products of Channels. In Section 4.1, we mentioned that the coefficients β_{kj} in the linear combinations can be solved through taking the derivative of the MSE. The result is a set of linear equations whose parameters solely involve the sum of pairwise products of channels $\sum_i \boldsymbol{x}_{ik} \boldsymbol{x}_{ij}$ of each classes. This can be computed with one forward pass over all training data for each layer.

The computation can be done before the model is shipped to the users, and the parameters can be sent over to the users upon request. The pre-computed data totals about 20MB for the NIN model that we experiment with in this paper. Depending on which classes to remove, we only need the relevant portion of the data

5.3 Overall Procedure

Algorithm 1 describes the overall procedure. The main source of computation is from line 17, where one matrix inversion and one multiplication are involved for solving the linear equations for each one of the pruned filters. Combining the cost of transmitting the pre-computed parameters, the total overhead of our method should be within reasonable limits.

Table 1: Comparison between the original 10-Class NIN model and pruned 5-class NIN models that on average achieves 91.2% classification accuracy, 38.5% in energy savings and 19.5% in latency improvements.

Name	Output Size	#Filters Before	#Params	#FLOPs Before	#Filters 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	111	4.36e+07	30.7%
CCCP2	32×32	96	1.55e+04	3.15e+07	67	1.52e+07	51.7%
CONV2	16×16	192	4.66e+05	2.36e+08	135	1.16e+08	50.8%
CCCP3	16×16	192	3.71e+04	1.89e+07	133	9.19e+06	51.4%
CCCP4	16×16	192	3.71e+04	1.89e+07	135	9.19e+06	51.4%
CONV3	8×8	192	3.34e+05	4.25e+07	134	2.08e+07	51.1%
CCCP5	8×8	192	3.71e+04	4.72e+06	134	2.30e+06	51.3%
CCCP6	8×8	10	1.93e+03	2.46e+05	5	8.58e+04	65.1%
Total				4.45e+08		2.46e+08	44.7%

6. EVALUATION

In the evaluation section, we report early results from experiments done on the Network-in-Network (NIN) model [12]. Since our method targets convolution operations, we chose a model that does not include any fully connected layers. We've seen some very encouraging results from these early experiments, and it would certainly be interesting to investigate how this method applies on more complex models and larger datasets.

We first present a case study to provide a high-level understanding of our pruning process. Then we present detailed evaluations on classification accuracy, energy consumption, and latency respectively. We also discuss how our proposed pruning criteria compares to existing methods. Note that

6.1 Case Study

To clearly illustrate our approach, we present a pruned 5-class NIN model as a case study. The architecture of the model is shown in the Table 1. The structure can be roughly described as groups of 5×5 convolution layers (CONV) followed by 1×1 convolution layers (which the authors called *cascaded cross-channel parametric pooling(CCCP)* layer). The last CCCP layer outputs a total of 10 channels, each corresponding to one class in the CIFAR-10 dataset. The score for each class is produced from a global average pooling layer on the lat CCCP layer.

Table 1 further demonstrates how the model is pruned. During our experiments, we found out that the first convolution layer produces universal features, and is indispensable to later layers. Thus we empirically decide to leave the first layer intact while pruning the rest of the layers, from which we uniformly remove $\Delta N = 30\%$ of the filters. As we discussed in Section 3.2, that corresponds to a $(1-(1-\Delta C)(1-\Delta N))$ reduction in the number of parameters and amount of computation except for the first and last layer. We prune 5 out of 10 filters, corresponding to the 5 pruned classes in the last layer.

6.2 Classification Accuracy

Figure 3(a) shows the classification accuracy with the model pruned at different levels. The y-axis shows classification accuracy, and the x-axis shows the percentage of filters pruned (except the first convolution layer and the last layer). In particular, the 0% shows the prediction accuracy using highest-scoring predictions from the original model after the unneeded classes are removed. For each experiment done with a different number of target classes, 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. As we can see from the figure, even if we aggressively prune half of the filters, 2 and 3-class models on average still achieve a higher classification accuracy than 10-class classifiers. 2-class models achieve an average accuracy of 95.5%, compared to the 98.5% of the unpruned model. When pruning 30% of the filters from 5-class models, the accuracy changed from 94.2% to 91.2%. For 10-classes models, apparently, the accuracy is always below the original accuracy if we do any pruning. If we prune 10% of the filters, we lose 1.4% in accuracy.

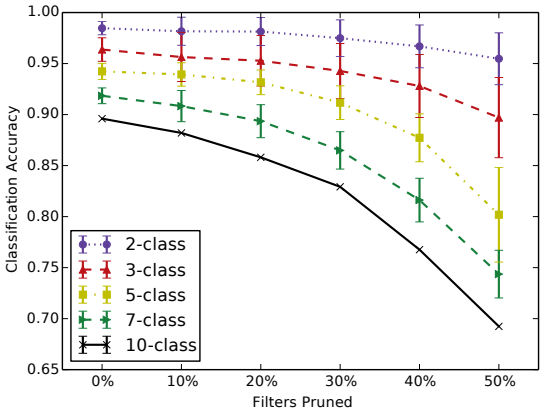
6.3 Energy Consumption and Latency

We now report energy consumption and latency measurements obtained from running inference on a Google Nexus 4 Android Smartphone. The energy is measured using a Monsoon power monitor. To calculate the the amount of energy spent on inference, we separately measured the energy spent on loading data and idle power and deduct them from the total energy consumption.

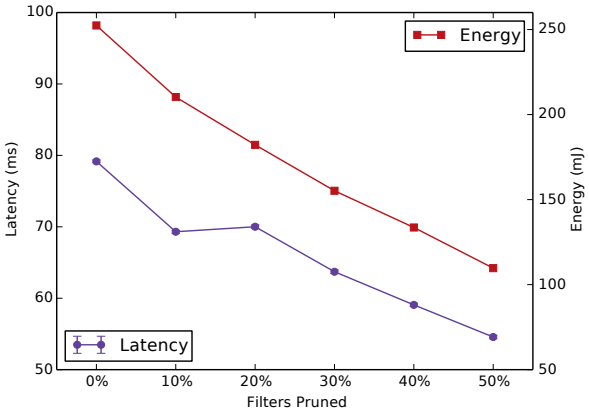
Figure 3(b) shows the average energy consumption and latency running inference on one data sample. Note that the only difference between models targeting different subset of classes (say a 2-class model and a 7-class model), is the last CCCP layer. But since the last layer only contributes to 0.05% of the total number of FLOPs, we ignore the differences and consider that the models take the same amount of energy and time to run. For 2-class classifiers, we can aggressively prune 50% of the filters while losing only 3% in accuracy. That brings $2.30\times$ in energy savings and $1.45\times$ in latency improvements. For 5-class classifiers, we can prune 30% of the filters while loosing 3% in accuracy (while still achieve a higher accuracy than the original 10-class classifier). We will, in turn, get $1.63\times$ in energy savings and $1.24\times$ in latency improvements.

6.4 Different Pruning Criteria

Figure 4 compares different pruning criteria with the proposed impact based criterion. "Magnitude" refers to pruning filters with the lowest absolute weights, a method used in [11], and "random" refers to pruning filters randomly. Similarly, all the accuracy measurements are the average of 10 repeated experiments. For a fair comparison, the 10 subsets of classes are the same for all these criteria. Apparently random pruning results in inferior performance. Our method can achieve better results when less filters are pruned, or when more classes are pruned, compared with the magnitude based method. One intuitive explanations would be that our impact based criteria can clearly identify the the irrelevant filters. However, when more filters are pruned, we are inevitably touching relevant filters (filters that is important to at least one class). In that case, it is questionable whether our sum of impact criteria described by Equation 1 would be the best measure of relevance.



(a) Classification Accuracy



(b) Energy and Latency

Figure 3: Performance evaluations of the method using NIN on CIFAR-10 dataset with different percentage of filters pruned [2].

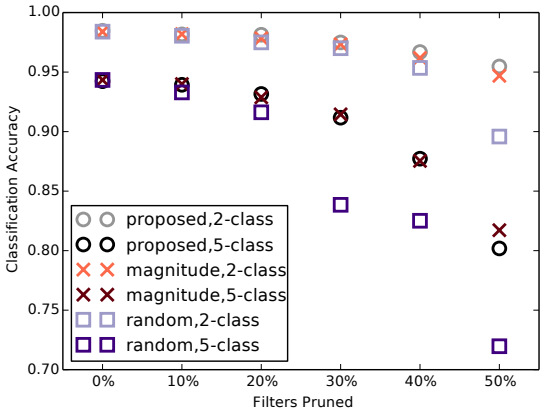


Figure 4: Comparison among different pruning criteria.

7. CONCLUSION

In this paper, we propose a lightweight customization method that allows users to prune the unneeded classes and filters from CNNs. The method could be seen as a client-side counterpart to the existing pruning methods which do not address the customization needs from users. In the proposed method, we first identify and remove the low-impact filters. Then we use a linear combination of the remaining filters to replace the pruned filters. The whole procedure can be efficiently run on-device with little overhead. Pruning unneeded classes not only brings about more targeted and accurate classification but also reduces computation costs. We observe a substantial reduction in energy consumption and latency from early experiments running the NIN model on CIFAR-10 dataset on an off-the-shelf smartphone.

8. ACKNOWLEDGMENTS

This work was supported in part by the NSF under award CNS-1059435 and award CNS-1513306.

[2]The percentage is calculated without the first and the last convolution layer

9. REFERENCES

[1] M. Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] S. Bhattacharya and N. D. Lane. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *SenSys*, pages 176–189, 2016.

[3] E. L. Denton et al. Exploiting linear structure within convolutional networks for efficient evaluation. In *NIPS*, pages 1269–1277, 2014.

[4] S. Han et al. Learning both weights and connections for efficient neural network. In *NIPS*, pages 1135–1143, 2015.

[5] S. Han et al. EIE: Efficient inference engine on compressed deep neural network. In *ISCA*, 2016.

[6] S. Han et al. MCDNN: an approximation-based execution framework for deep stream processing under resource constraints. In *MobiSys*, pages 123–136, 2016.

[7] B. Hassibi and D. G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *NIPS*, pages 164–171, 1992.

[8] M. Jaderberg et al. Speeding up convolutional neural networks with low rank expansions. In *BMVC*, 2014.

[9] N. D. Lane et al. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *IPSN*, pages 23:1–23:12, 2016.

[10] Y. LeCun et al. Optimal brain damage. In *NIPS*, pages 598–605, 1989.

[11] H. Li et al. Pruning filters for efficient convnets. In *ICLR*, 2017.

[12] M. Lin et al. Network in network. In *ICLR*, 2014.

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

[14] R. Rigamonti et al. Learning separable filters. In *CVPR*, pages 2754–2761, 2013.