Pruning in Deep Compression

Jinze Chen

January 19, 2019

1 Introduction

People design pruned neural networks mainly for 2 reasons: one for the deployment in resource constrained devices, *e.g.*, mobile phones or embedded gadgets and the other to reduce the effect subjected to over-parameterization, thus this approach may even increase the inference accuracy in certain networks.

For the latter one, this process resembles the biological phenomena in mammalian brain, where the number of neuron synapses has reached the peak in early childhood, followed by gradual pruning during its development. However, we don't concern much about increasing the accuracy in this article. Having a new sparse network that functions not so strayed from the original one is our ultimate goal.

2 Related work

The main problem is the principle to find the appropriate mask in the whole network. Different choices may make different trade-offs between computational resources and accuracy, *e.g.*, weight-wise pruning may achieve better compression rate while layer-wise pruning is much easier to be performed. There are also some attempts to reconstruct a similar network but with fewer parameters or layers [**?**], which we'll examine later in **??**.

In summary, less significant compositions may be masked safely without affecting the network performance. Based on this principle, Han et al. [**?**] proposed an iterative pruning method to remove the redundancy in deep models. Their main insight is that small-weight connectivity below a threshold should be discarded. In practice, this can be aided by applying l_1 or l_2 regularization to push connectivity values becoming smaller. The major weakness of this strategy is the loss of universality and flexibility, thus seems to be less practical in the real applications.

In order to avoid these weaknesses, some attention has been focused on the group-wise sparsity. Lebedev and Lempitsky [**?**] explored group-sparse convolution by introducing the group-sparsity regularization to the loss function, then some entire groups of weights would shrink to zeros, thus can be removed. Similarly, Wen et al. [**?**] proposed the Structured Sparsity Learning (SSL) method to regularize filter, channel, filter shape and depth structures. In spite of their success, the original network structure has been destroyed. As a result, some dedicated libraries are needed for an efficient inference speed-up.

Some filter level pruning strategies have been explored too. The core is to evaluate neuron importance, which has been widely studied in the community [**?**,**?**,**?**,**?**,**?**]. A simplest possible method is based on the magnitude of weights. Li et al. [**?**] measured the importance of each filter by calculating its absolute weight sum. Another practical criterion is to measure the sparsity of activations after the ReLU function. Hu et al. [**?**] believed that if most outputs of some neurons are zero, these activations should be expected to be redundant. They compute the Average Percentage of Zeros (APoZ) of each filter as its importance score. These two criteria are simple and straightforward, but not directly related to the final loss. Inspired by this observation, Molchanov et al. [**?**] adopted Taylor expansion to approximate the influence to loss function induced by removing each filter.

3 Different methodologies

3.1 Neuron Pruning for Compressing Deep Networks using Maxout Architectures [?]

This paper presents an approach for reducing the size of deep neural networks by pruning entire neurons. It also can be combined with subsequent weight pruning.

The idea of the proposed approach is to use the maxout units and their model selection abilities for pruning entire neurons from an architecture without expensive processing. Thus, reducing the size and the memory consumption of a deep network. It's assumed redundancies exist in a deep neural network. The process is shown below:

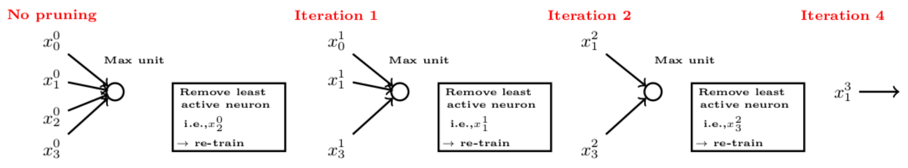

No pruning
Iteration 1
Iteration 2
Iteration 4
x_0^0
x_1^0
x_2^0
x_3^0
Max unit
Remove least active neuron i.e.,x_2^0 → re-train
x_0^1
x_1^1
x_3^1
Max unit
Remove least active neuron i.e.,x_1^1 → re-train
x_1^2
x_3^2
Max unit
Remove least active neuron i.e.,x_3^2 → re-train
x_1^3

Fig. 2: Neuron pruning process for $k = 4$ inputs per maxout unit. x_a^b represents a neuron with $'a'$ the neuron index and $'b'$ the iteration.

First, a CNN with a maxout layer is trained. This maxout layer performs a max function among k adjacent neurons, reducing the amount of weights connecting with the next layer by a factor of k. So, placing this maxout layer after the one with the highest number of weights would be advisable. Second, by counting the number of times neurons become the maximal value in each maxout unit when computing a forward pass over the training dataset, the least active neurons of each maxout unit are removed from the network. Their effects are negligible with respect to other neurons. Third, the remaining neurons of the CNN are re-trained. After re-training, the process is repeated;

In weight pruning, the weights get pruned by thresholding them. This is one simple approach, and it can be replaced by other criterions like relevance measure using Hessian matrix.

This approach reduced the network size by up to 74% in LeNet-5 and 61% in VGG16 without affecting the network's performance, only applying the neuron pruning.

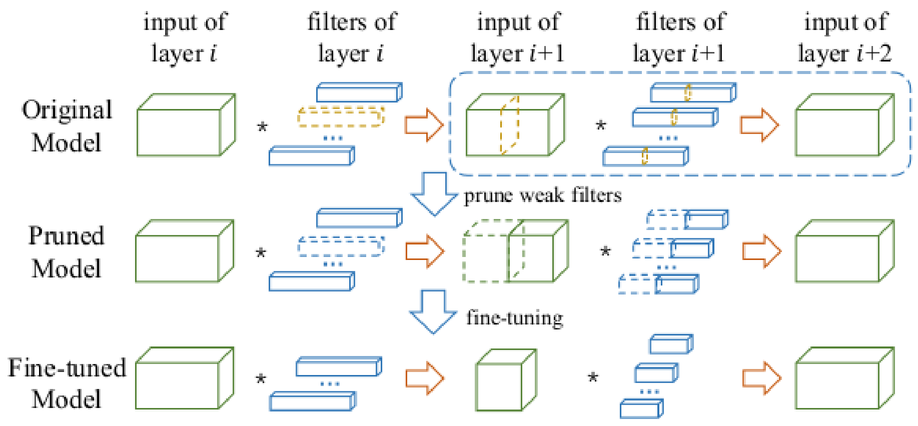
The main insight in this paper is considering redundancies in maxout architecture. Because this can be combined with other weight pruning methods, it's not bad to combine this approach in certain networks.

3.2 ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression [?]

This paper proposed a filter level pruning framework, called ThiNet, to simultaneously accelerate and compress CNN models in both training and inference stages.

The idea is to establish filter pruning as an optimization problem, *i.e.*, to find a subset of channels in layer (i+1)'s (not i's) input to approximate the output in layer i+1. Instead of considering which filter to discard, the author finds the most important filters to be preserved.

Given a pre-trained model, it would be pruned layer by layer with a predefined compression rate. The process is shown below:



1. A training set is randomly sampled.

2. Using a greedy algorithm for channel selection, *i.e.*, to minimize

$$\arg\min_{T} \sum_{i=1}^{M} \left(\hat{y}_i - \sum_{j \in S} \hat{x}_{i,j} \right)^2 \quad s.t. \quad |S| = C \times r, S \in \{1, 2, \ldots, C\}$$

3. Minimize the reconstruction error by weighing the channels, which can be defined as $\hat{w} = \arg\min_w \sum_{i=1}^{M} (\hat{y}_i - w^T \hat{x}_i^*)^2$

4. Iterate to step 1 to prune the next layer.

The performance of ThiNet on ILSVRC-12 benchmark. ThiNet achieves 3.31× FLOPs reduction and 16.63× compression on VGG-, 6, with only 0.52% top-5 accuracy drop.

The main insight is that the author establishes a well-defined optimization problem, which shows that whether a filter can be pruned depends on the outputs of its next layer, not its own layer. Also because this approach prunes the network in filter level, the computational cost is relatively low, compared with weight pruning.

3.3 Sparsifying Neural Network Connections for Face Recognition [?]

This paper focused on creating a new network for face recognition, while also sparsified the whole network after the training has finished. What's meaningful to our work is the principle he used to prune weights.

The author tends to keep connections (and the corresponding weights) where neurons connected have high correlations and drop connections between weakly correlated neurons. For fully and locally-connected layers, where weights are not shared, given a neuron a_i in the current layer and its K connected neurons $b_{i1}, b_{i2}, \ldots, b_{iK}$ in the previous layer, the correlation coefficient between a_i to each of b_{ik} for $k = 1, 2, \ldots, K$ is

$$r_{ik} = \frac{E[a_i - \mu_{a_i}][b_{ik} - \mu_{b_{ik}}]}{\sigma_{a_i} \sigma_{b_{ik}}}$$

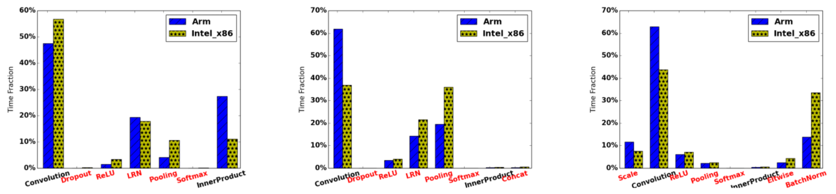
where $\mu_{a_i}, \mu_{b_{ik}}, \sigma_{a_i}, \sigma_{b_{ik}}$denote the mean and standard deviation of a_i and b_{ik}, respectively, which are evaluated on a separated training set. Since both positively and negatively neurons are important, the author used similar but slightly different methods to deal with them. For convolutional layers, the author calculated the mean magnitude, which we don't present here.

The main insight is to measure weight importance based on the previous layer. The author also experimented on directly training the sparse ConvNet from scratch, which failed to find good solutions for face recognition. Although we mainly focus on pruning an existing network, this could be some hint to the training stage.

3.4 DeepRebirth: Accelerating Deep Neural Network Execution on Mobile Devices [?]

This paper finds another solution for speeding up model inference and reducing model size, which is called DeepRebirth. Strictly speaking this method shouldn't be called pruning, but it's useful for deployment in mobile phones and has been tested on some mobile devices, so we list it here.

First the author found that non-tensor layers consume too much time in model execution, which is shown below:



(a) AlexNet (b) GoogLeNet (c) ResNet-50

Figure 2: Time Decomposition for each layer. Non-tensor layers (e.g., dropout, ReLU, LRN, softmax, pooling, etc) shown in red color while tensor layers (e.g., convolution, inner-product) shown in black color.

So the author thought of combining those non-tensor layers with its adjacent tensor layers. This paper presents two ways to combine those layers, called **Streamline Slimming** and **Branch Slimming** respectively.

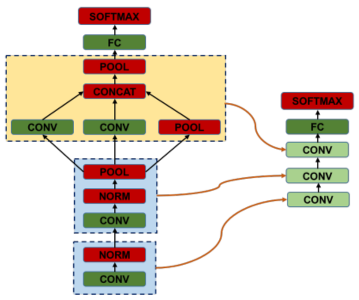


Figure 1: An illustration of proposed DeepRebirth model acceleration pipeline. DeepRebirth optimizes a trained deep learning model (left) to an accelerated "slim" model (right). Such optimization is achieved with two operations: Streamline Slimming which absorbs non-tensor layers (i.e., pooling and normalization) to their bottom convolutional layer (in light blue background) and Branch Slimming which absorbs non-tensor branches and convolutional branches with small convolution filters (e.g., 1x1) to a convolutional branch with large convolution filter (e.g., 5x5) (in light yellow background). We name new generated layers as slim layers.

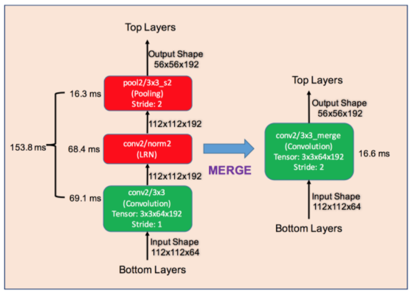


Figure 3: Streamline Slimming: The GoogLeNet example and the running time is measured using bvlc_googlenet model in Caffe on a Samsung Galaxy S5. Left panel: convolution (in green), LRN (in red), pooling (in red). Right Panel: single convolution layer. The three layers in the left panel are merged and regenerated as a convolution layer (i.e., slim layer) in the right panel.

The weights and biases are retrained to attain similar performance with original network, which is given by

$$(\tilde{W}^*, \tilde{B}^*) = \arg\min_{W,B} \sum_i ||Y^i_{CNN} - \tilde{f}(W, B; X^i)||^2_F$$

where W and B represent weight and bias matrix respectively.

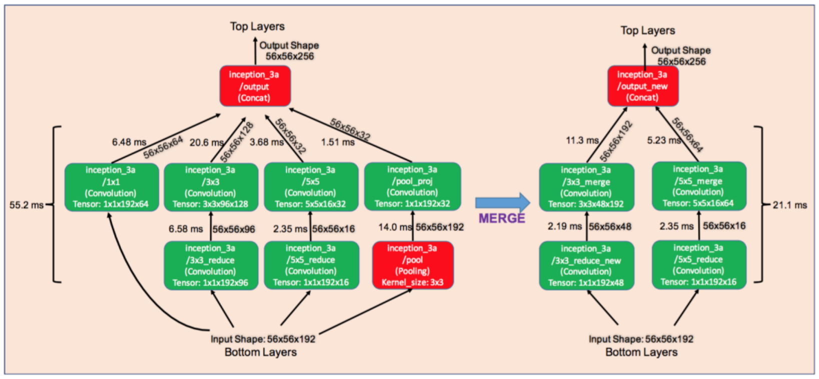

Top Layers
Output Shape 56x56x256
inception_3a /output (Concat)
6.48 ms
56x56x64
20.6 ms
56x56x128
3.68 ms
56x56x32
1.51 ms
56x56x32
55.2 ms
inception_3a /1x1 (Convolution) Tensor: 1x1x192x64
inception_3a /3x3 (Convolution) Tensor: 3x3x96x128
inception_3a /5x5 (Convolution) Tensor: 5x5x16x32
inception_3a /pool_proj (Convolution) Tensor: 1x1x192x32
6.58 ms
56x56x96
2.35 ms
56x56x16
14.0 ms
56x56x192
inception_3a /3x3_reduce (Convolution) Tensor: 1x1x192x96
inception_3a /5x5_reduce (Convolution) Tensor: 1x1x192x16
inception_3a /pool (Pooling) Kernel_size: 3x3
Input Shape: 56x56x192
Bottom Layers
MERGE
Top Layers
Output Shape 56x56x256
inception_3a /output_new (Concat)
11.3 ms
56x56x192
5.23 ms
56x56x64
inception_3a /3x3_merge (Convolution) Tensor: 3x3x48x192
inception_3a /5x5_merge (Convolution) Tensor: 5x5x16x64
21.1 ms
2.19 ms
56x56x48
2.35 ms
56x56x16
inception_3a /3x3_reduce_new (Convolution) Tensor: 1x1x192x48
inception_3a /5x5_reduce (Convolution) Tensor: 1x1x192x16
Input Shape: 56x56x192
Bottom Layers

Figure 4: Branch Slimming: The GoogLeNet example and the running time is measured using bvlc_googlenet model in Caffe on a Samsung Galaxy S5. Left panel: four branches in parallel, convolution layer, convolution + convolution, convolution + convolution, convolution + pooling. Right panel: two branches in parallel, convolution + convolution, convolution + convolution. Two branches are reduced.

The main insight in this article is to evaluate execution time of non-tensor layers and solve this problem by merging certain layers, and it has achieved great success in speeding up the whole network. However, this is a completely engineering practice. It could be added to any network after the training has finished.

4 Summary

These articles all present some ways to accelerate or compress neural networks, or both. In general, we may derive the process for pruning an existing network as follows:

1. Get a trained network.
2. Prune some components based on its importance. It's better to evaluate that based on its relation with output rather than on one layer exclusively.
3. Retrain the whole network to minimize error induced by pruning. After that iterate to step 2.

*4 Reconstruct a new network for less model latency.

In order for step 4 to be combined, there needs some extra efforts, or we could only focus on model speed or model size. Anyway, that's the main framework for network pruning. Also we could combine other methods like quantization and entropy coding to reduce the model size further.