

# Reduction of Convolutional Layers in Deep Networks

Vincent Martineau,<sup>1\*</sup>

<sup>1</sup>Department of Computer Science and Software Engineering, Université Laval, Quebec, Canada

\*E-mail: vincent.martineau.1@ulaval.ca.

**There has been research explaining how to reduce convolutional layers in neural networks. However in the proposed articles the models used are usually AlexNet and VGG. When tried on more complex networks such as ResNet, DenseNet and SqueezeNet the proposed solutions do not work. The goal here is to explore the limits of this approach and to propose a solution in order to generalize to more complex and deep networks than those in Molchanov's original article. Experiments will be done on the model transfer between ImageNet to Cifar10 and we will see the gains that can be.**

## Introduction

CNN networks have been with us for a very long time (1). These networks are widely used in image processing in a wide variety of fields. They are very used in the classification or location of objects. Whether with surveillance systems, autonomous cars or smart devices, this type of network is now part of our daily work (2).

These networks have shown that they can achieve results beyond human capabilities (3). In many cases, these models are made available to the general public and can be adapted to simpler problems to get very good results (4).

## 0.1 Description of the problems

In order to use these safe networks of small devices (5) techniques have been evaluated in the past. More recently, an article proposes to make the reduction of convolution filters (6). However the article focuses mainly on AlexNet and VGG networks and does not work on the more complex networks present in PyTorch (7).

The purpose of this article will be to evaluate the work needed to push this logic to more complex networks such as ResNet, DenseNet and SqueezeNet.

## Method

The method used is a variant of the method proposed in the article Pruning Convolutional Neural networks for Resource efficient inference (6). The purpose of the changes is to extend the capabilities of our model to much more complex networks such as ResNet, DenseNet and SqueezeNet.

Here is a list of the steps to follow to get an interesting result:

1. Apply fine tuning on a pre-trained network.
2. Extract the execution graph from our model.
3. From our execution graph, exclude convolutional layers that we do not want to affect.
4. Reduce the filters according to the reduction criterion.
5. Apply fine tuning to restore the network.
6. Repeat the reduction until the network is the size you want.

In this procedure the selection criterion is very important to have good results. For this experiment, we used the Taylor Expansions which proved their effectiveness in the article and the average activations.

In order to be able to run on more complex models, we chose not to explore other selection criteria, but to focus on the algorithm needed to support more complex models.

## 0.1 The layers to ignore

In the proposed solution, it is important not to apply filter reduction just before a flow addition as seen in several complex models. Figure 1 represents this situation. The problem in this case is that it is necessary to apply an equivalent reduction on all incoming branches. It is possible that there is a selection criterion to include these layers in the future.

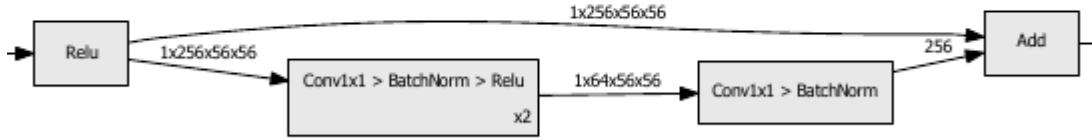


Figure 1: In the following figure it is impossible to reduce the second convolution layer for the bottom branch since it would also be necessary to modify the layer preceding this ResNet block so that the residual connection has the right size.

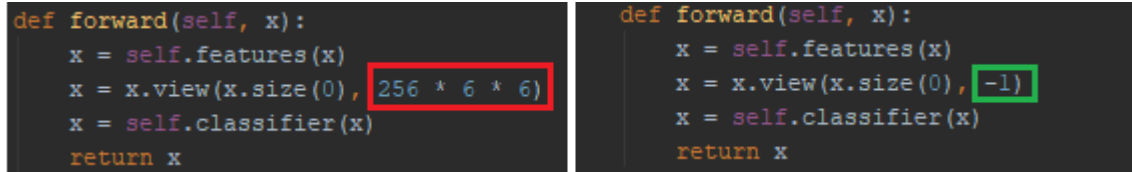
Another potential problem occurs when a layer is reused. In this case it is necessary that the input of our layer is compatible with the exit of the layer which we treat. However, we must assign the previous layers so that the dimensions are also consistent.

Note that it is possible that there is a reliable algorithm that can handle these cases, but this article does not explore these options.

Finally, it is crucial not to affect a layer if it directly influences the number of network outputs. Such a practice would directly affect the performance and training of the model. There could be a lag between classes during classification and the loss function could become invalid.

## 0.2 Practices to avoid

When a model is created it is important to avoid hardcoded value in the inference function. For example, in the AlexNet model, there is the code on the left side of Figure 2 that prevents us from reducing the last convolution layer before the fully connected layer. By performing the modification shown in Figure 2, it is now possible to reduce the convolution layer before the fully connected layer.



<pre>def forward(self, x):     x = self.features(x)     x = x.view(x.size(0), 256 * 6 * 6)     x = self.classifier(x)     return x</pre>	<pre>def forward(self, x):     x = self.features(x)     x = x.view(x.size(0), -1)     x = self.classifier(x)     return x</pre>
--	---

Figure 2: Note in the left-hand section that the original code assumes a fixed size for the last convolution layer. In addition to limiting the model to a certain input image size, the last convolution layer can not be reduced. The proposed solution requires the modification in the right section of the image that allows a variable size of the last convolution layer.

## 0.3 Propagation of the effect

When we remove a convolution filter from a layer, several precautions must be taken to ensure the stability of the system. For example, we must adjust the weights, the number of skewings in the layer and adjust the gradients when they are contained in the layer itself. Which is the case of PyTorch.

Subsequently, the following layers must be adjusted according to their type.

For the following **convolutions** layers. It is necessary to adjust the number of layers in inputs to match the convolutional layer we are dealing with. We must also adjust the weight of this layer.

For **Batchnorm** layers, we must adjust the bias when used and we reset the variables used at runtime. In PyTorch, we talk about running means and running vars.

Finally, we need to adjust **fully connected** layers. In the case of these layers, we adjust the weights and apply the *weight inference rule* (8) to simulate weight removal by *dropout*.

The other layers seem to support the reduction of convolutional layers. It should be noted that we have only tried the models proposed in PyTorch and that other work might be needed for more complex layers.

## Résultats

First, we evaluate the effect of reducing convolution filters on the AlexNet model. Figure 3 shows the effects of the reduction. The first result we can notice is that a succession of too much pronounced reduction significantly reduces the pre-workout effect.

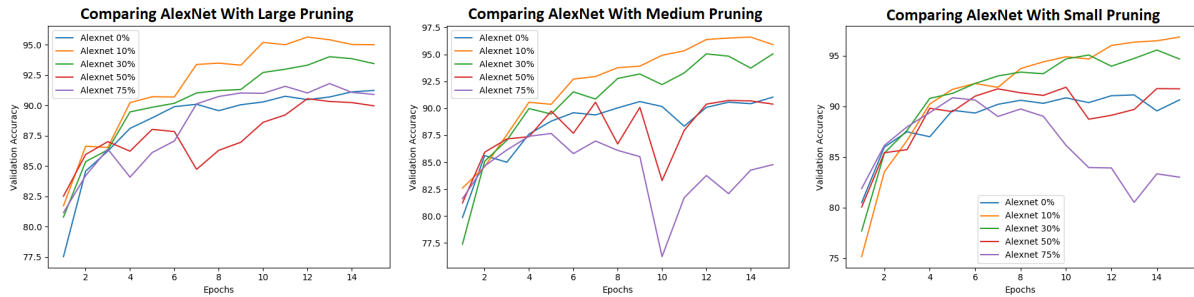


Figure 3: In the figure on the left, there is a significant reduction of the weights at each iteration over two iterations (epoch 4 and 7). The figure of the center represents a reduction on 5 iterations with 2 steps of leveling. In the figure on the right, we see a more gradual reduction made over 10 iterations.

Table 1: Comparison of results after model reduction

	0%	10%	30%	50%	75%
<b>Fast Pruning</b>	91.02	88.94	89.94	88.01	88.79
<b>Average Pruning</b>					
<b>Slow Pruning</b>	90.74	90.14	89.43	87.84	81.02

Figure 3 also shows an important aspect. It shows that the way to reduce weights is important. Reducing the weight too fast will cause more noise in the training. However, the number of training epochs performed after the last reduction seems essential to obtain good results. In contrast, we notice that a gradual reduction over a longer period brings more stability to the model for a certain time, but that the number of iterations of reduction eventually reduces the final performance of the model. An approach between the two gets good stability in the model and the final results are more consistent.

Then we compare the difference in efficiency between the models. This difference is visible in Figure 4 which compares the shallow models in the left section and the deep Pytorch models in the right graph. In any case, we note that even with a reduction of 30% that can be cut, it is possible to significantly reduce the network and maintain a level of performance.

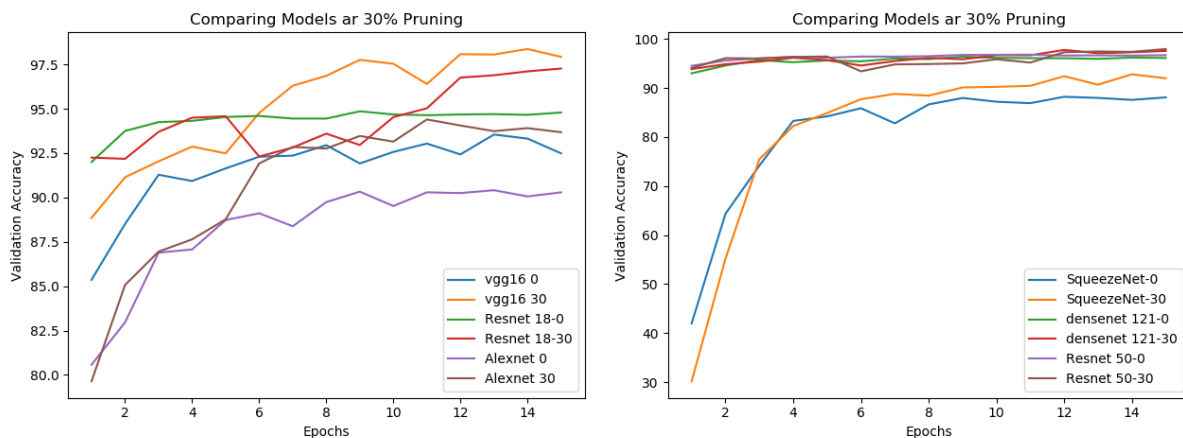


Figure 4: In the image on the left we can see the effect of the reduction on shallow models. In the image on the right we see the effect of the reduction on the deep networks of PyTorch.

The strategy used for the reduction is to make a gradual reduction. That is, cut 5 % filters that can be reduced and repeat for 6 iterations. It can be seen that this strategy gives better results on all models at the validation level. However, we note that the final scores are slightly lower.

Table 2: Comparison of the reduction according to the model

	<b>VGG</b>			<b>AlexNet</b>			<b>ResNet18</b>		
	0%	30%	Diff	0%	30%	Diff	0%	30%	Diff
<b>FLOPs (G)</b>	17.31	10.55	-39.1%	0.815	0.500	-38.7%	1.83	1.40	-23.5%
<b>Params (M)</b>	138.3	80.5	-42.8%	57.0	47.9	-16.0%	11.18	7.53	-32.6%
<b>Score (%)</b>	91.92	92.03	0.11	90.98	88.57	-2.41	94.8	93.11	-1.69

	<b>SqueezeNet</b>			<b>DenseNet121</b>			<b>ResNet50</b>		
	0%	30%	Diff	0%	30%	Diff	0%	30%	Diff
<b>FLOPs (G)</b>	0.516	0.511	-0.97%	2.91	2.14	-26.4%	4.14	2.86	-30.9%
<b>Params (M)</b>	1.24	1.23	-0.81%	7.98	6.12	-23.3%	23.53	11.18	-52.5%
<b>Score (%)</b>	88.22	88.87	0.65	95.99	95.31	-0.68	96.32	94.76	-1.56

By examining the effect of the reduction on the parameters and the number of FLOPs in the table1. We note that a 30% reduction in available filters varies greatly from model to model. For example, in SqueezeNet, only the first convolution layer can be reduced. In contrast, ResNet and VGG networks offer an excellent reduction.

We also notice a loss of up to 2% for most models. In some cases we even notice a gain on the set of tests. However, the loss does not seem directly related to the percentage of parameters removed from the network.

## Conclusion

In conclusion we notice: 1- the technique of reduction of the convolution filters can be applied on more complex models by creating a module analyzing the execution of the model. 2 - Reduced gains can reduce material requirements for many architectures. 3 - The reduction of convolution filters offers a form of regularization that can be used to train a model when applied moderately. 4 - How to apply the reduction is important in order to obtain good results.



## References and Notes

1. Y. LeCun et al. (1989). Backpropagation Applied to Handwritten Zip Code Recognition.
2. 30 amazing applications of deep learning: <http://www.yaronhadad.com/deep-learning-most-amazing-applications/> .
3. K. He, X. Zhang, S. Ren, and J. Sun (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification..
4. Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, Chunfang Liu (2018). A Survey on Deep Transfer Learning.
5. Song Han, William J. Dally (2018). Bandwidth-Efficient Deep Learning.
6. Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, Jan Kautz (2017). Pruning Convolutional Neural Networks for Resource Efficient Inference.
7. <https://pytorch.org/docs/stable/torchvision/models.html> .
8. Srivastava et al. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting..