Pruning Deep Neural Nets

Neelam Babel | babel.n@husky.neu.edu

Abstract

Deep Neural Nets have a wide scope, be it image recognition or it's application in game theory. But the question is - can we make them fast and small without making them worse ? Can they run seamlessly on our mobile devices ? As part of this research project, I focus on pruning of neural nets to make them more efficient. I have used L-1 Norm to prune the Convolutional Layers/filters while trying to maintain the accuracy level. The model accuracy was 67% initially (before pruning) and it dropped to 56% after pruning the network to 16%. I re-trained the model to 1 epoch and got the accuracy back up to 64.9%.

Introduction

We are going to take MobileNet-224 and make it 16% smaller. In other words, we're going to reduce it from 4 million parameters to approximately 3.5 million — without losing a lot on accuracy.

The question is - does MobileNet have any connections it doesn't really need? Even though this model is already quite small, can we make it even smaller - without making it worse?
When we prune a neural network, the tradeoff is **network size vs. accuracy**. In general, the smaller the network, the faster it runs (and the less battery power it uses) but the worse its predictions are. MobileNet scores better than SqueezeNet, for example, but is also 3.4 times larger.
Ideally, we want to find the smallest possible neural net that can most accurately represent the thing we want it to learn. This is an open problem in machine learning, and until there is a good theory of how to do this, we're going to have to start with a larger network and lobotomize it.

Dataset

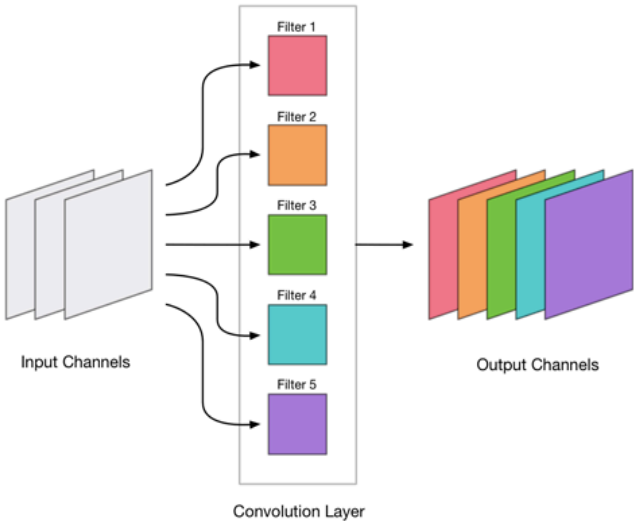
We have used ImageNet ILSVRC 2012 validation set data to evaluate our pruned MobileNet Model.
The data can be downloaded from the below link -
http://www.image-net.org/challenges/LSVRC/2012/

Model Architecture Design

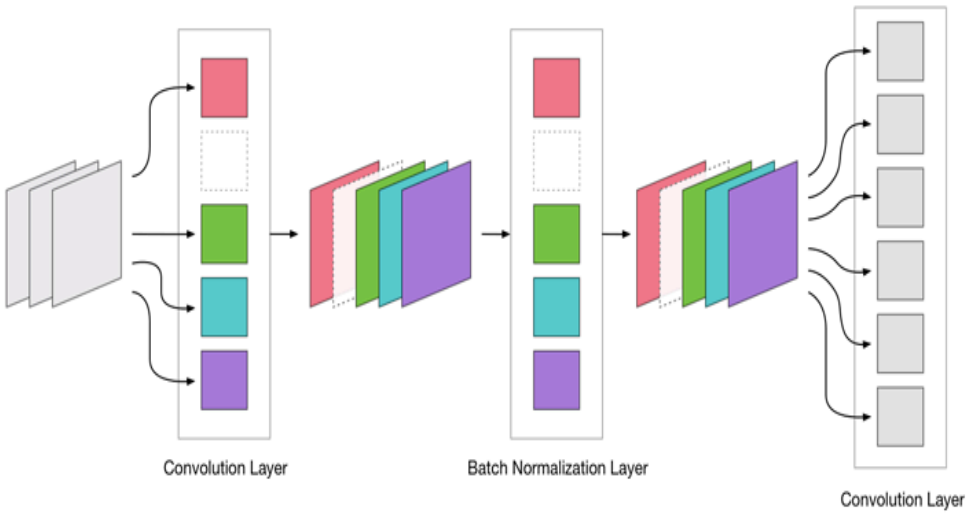
Pruning Neural Nets

Instead of pruning away individual connections we'll remove complete convolution filters. This keeps our connections dense and the GPU happy.

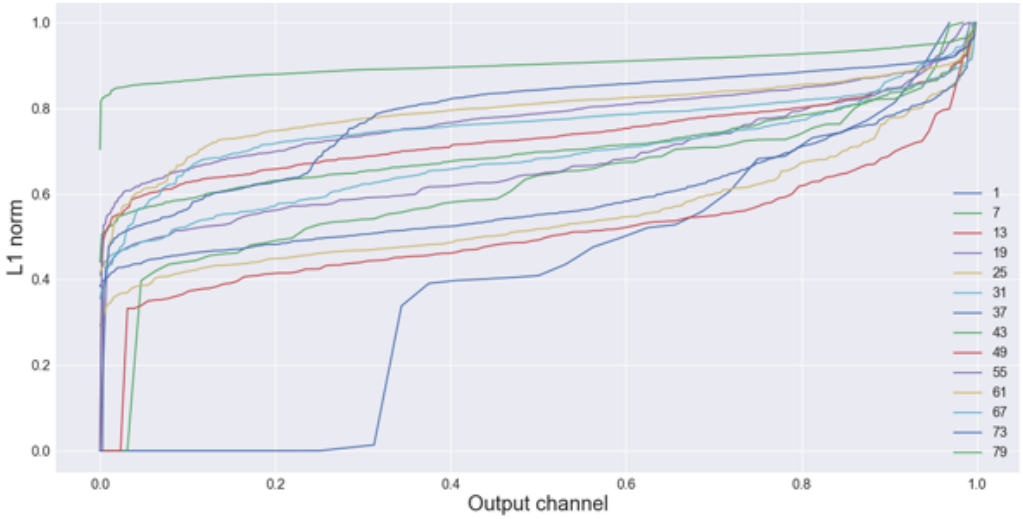
A convolution layer produces an image with a certain number of output channels. Each of these output channels contains the result of a single convolution filter. Such a filter takes the weighted sum over all the input channels and writes this sum to a single output channel. We're going to find the convolution filters that are the least important and remove their output channels from the layer.



There are different metrics we can use to estimate a filter's relevance, but we'll be using a very simple one - the **L1-norm** of the filter's weights, which just means we take the absolute values of the filter's weights and add them all up.
Removing filters from a layer means the number of output channels for that layer becomes smaller. Naturally, this has an effect on the next layer in the network too because that layer now receives fewer input channels.
As a result, we also have to remove the corresponding input channels from that layer. And when the convolution is followed by batch normalization, we also have to remove these channels from the batch norm parameters.



Below is a plot of the L1-norms of all the convolutional layers in MobileNet. You can see that many layers have filters that do not appear to contribute much to the network (low L1-norm).



MobileNet actually has three kinds of convolutional layers:
-one regular 3×3 convolution (the very first layer)
-depth-wise convolutions
-1×1 convolutions (also known as pointwise convolutions)

We can only remove filters from the 3×3 and 1×1 convolutions, but not from the depth-wise ones. A depth-wise convolution must always have the same number of output channels as it has input channels. There isn't much to gain by compressing, and depth-wise convolutions are pretty fast anyway (since they do way less work than a regular convolution). So we'll primarily focus on the layers with 3×3 and 1×1 convolutions.

Retraining Pruned Neural Nets

Removing filters from a layer makes the accuracy of the network worse - after all we are throwing away things that the neural net has learned, even if they may not be very important - we need to retrain the network a little, so that it can learn to compensate for the parts we just cut out.
Retraining just means we call model.fit() again. A little trial-and-error led me to a learning rate of 0.00003 - quite small, but anything larger might make the training spin out of control. The reason the learning rate must be so small is that at this point the network is mostly trained already and we only want to make small changes to tweak the results.

The process then is:
-remove filters (i.e. output channels) from a layer, in multiples of 4
-re-train the network for a few epochs
-evaluate on the validation set to see if the network has regained its previous accuracy
-move to the next layer and repeat these steps

This process is quite labor-intensive, since we only compress one layer at a time and we need to retrain the network after every change. Figuring out how many filters to drop in each layer is not obvious either.

Code with Documentation

GitHub Link: The complete code with documentation can be found on the below link https://github.com/babeln/Data_Science_Notebooks/blob/master/Research%20Study.zip

Results

On the original model the validation set accuracy is:
Top-1 accuracy over 50000 images = 67.1%
Top-5 accuracy over 50000 images = 86.4%

The final pruned model has accuracy:
Before: 4231976 parameters
After: 3583656 parameters
Saved: 648320 parameters
Compressed to 84.68% of original

Top-1 accuracy over 50000 images = 56.9
Top-5 accuracy over 50000 images = 77.8

After re-training the model for 1 epoch the accuracy becomes:
Top-1 accuracy over 50000 images = 64.9
Top-5 accuracy over 50000 images = 84.3

Note that this isn't necessarily the best we can do. This approach of compressing the network layer-by-layer is very much trial & error. We can keep pruning the convolutional layers in the same fashion and re-train the model each time and finally evaluate the performance.

Acknowledgment

I would like to show my sincere gratitude to professor Nik Bear Brown for guiding me during this project.

Discussion

The result is a little short of my goal: the network did become ~16% smaller but the accuracy is a little worse- but definitely not 16% worse!
The workflow could use some improvement, as I could only re-train the model on 1 Epoch (Since I am running it on a CPU, every Epoch takes about an hour to run). I also wasn't very scientific about how I chose what filters to remove, or how I picked the order in which to compress the layers. But for this project, I just wanted to get an idea of what was possible.

Obviously, I did not arrive at an optimal pruning for this network. Using L1-norms may not be the best way to determine how important a filter is, and maybe it's better to only remove a few filters at a time than to chop off 1/4th of the layer's output channels in one go. But I am happy that using samples worked so well for retraining. Not having to wait hours for retraining means I could quickly run new experiments.

Is this sort of thing worth doing? I'd say so. Suppose you have a neural network that runs at 25 FPS on your iPhone, which means every frame takes 0.04 seconds to process. If the neural net is 25% smaller — and let's assume this means it will also be 25% faster — then each frame will only take 0.03 seconds, which translates to over 30 FPS with some time to spare. This could be the difference between an app that runs choppy and an app that runs butter smooth.

References

[1] Matthijs Hollemans, Compressing Deep Neural Nets**,**
http://machinethink.net/blog/compressing-deep-neural-nets/
[2] Jacob Gildenblat, Pruning Deep Neural Networks to make them fast and small,
https://jacobgil.github.io/deeplearning/pruning-deep-learning