Object Detection Mobile App for Visually Impaired

Amulya Aankul | Nand Govind Modani
{aankul.a,modani.n}@husky.neu.edu
CSYE 7245, Fall 2017, Northeastern University

Abstract

Based on the fact sheet by WHO, there were approx. 285 million people to be visually impaired worldwide in 2012: 39 million are blind and 246 million have low vision [1]. It is about 4% of the total population. Another fact is that we have now over 2 billion smartphone users worldwide [2], with such a kind of rise in smartphone industry, if combined with artificial intelligence & computer vision could serve as an efficient object detection tool in real time that can help such people to independently access unfamiliar environments & avoid dangers.

The basis of our project is to build a mobile app that can detect and speak out objects in real time using phone camera. We are planning to implement following things:
• Detect object in real time through mobile phone,
• Make the phone speak out the object in front of camera

Introduction

In this project, we aim at deploying a real-time object detection system that operates at a sufficiently high accuracy, low power and high FPS on a low memory device such as mobile phones or Raspberry pi. There has been a lot of work being done to increase the accuracy of computer vision problems using Deep Neural Network but since most of these works are evaluated on high-end GPUs and CPUs, it is very difficult to apply such models in a real-world application such as Surveillance system, camera or mobile phones.

Also, in most of the cases we cannot use cloud because of issues like network delay, power budget and user privacy. In this blog, we will try and reduce the number of parameters in a Convolution Neural Network algorithm that runs on low power and low latency.

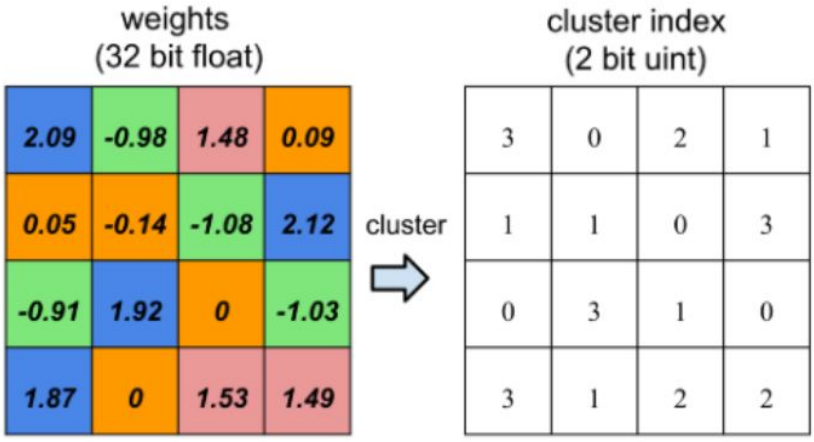
Related works

In this section, we will describe some of the approaches that are used to reduce the size and complexity of an existing deep neural network model. We will keep our discussion limited to those approaches that are used for Convolutional Neural Networks.

1) Quantization

A very popular way to achieve the goal of reducing storage of Deep neural network is to cut the computation of multiplication in convolution layers. Provided that the multiplication of float is much more time consuming than integer, an instinct is to transform float numbers to 8 bit integers. This could be implemented using shrinks by storing the min and max value of weights for each layer, and then compressing each float value to an eight-bit integer representing the

closest real number in a linear set of 256 within the range. This method in theory could reduce the model size by 75% [3]. A support package recently released by TensorFlow develop team enables the 8-bit quantization of tensors [4].



However, the reason why we did not include this approach is that although it does reduce the size of the model significantly, the number of parameters remains the same and therefore there is not much reduction in latency compared to the significant reduction in accuracy.

2) **Removing Small weights**

The idea is borrowed from a very popular paper Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding [5]. Dally which shows that the size of VGG16 can be reduced by a factor 49 by pruning unimportant connections and it will still give the same accuracy.

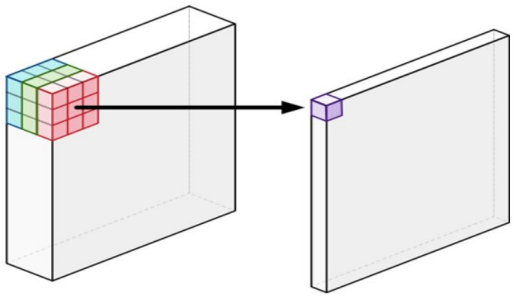
Unfortunately, this results in generation of sparse matrices, which is hard to compute by existing hardware and again, although the size decreases, the inference time increases and this approach fails.

$$\begin{pmatrix} 1.0 & 0 & 5.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3.0 & 0 & 0 & 0 & 0 & 11.0 & 0 \\ 0 & 0 & 0 & 0 & 9.0 & 0 & 0 & 0 \\ 0 & 0 & 6.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7.0 & 0 & 0 & 0 & 0 \\ 2.0 & 0 & 0 & 0 & 0 & 10.0 & 0 & 0 \\ 0 & 0 & 0 & 8.0 & 0 & 0 & 0 & 0 \\ 0 & 4.0 & 0 & 0 & 0 & 0 & 0 & 12.0 \end{pmatrix}$$

Technical Approach

1) Depth-wise Separable Convolutional layer

Depth-wise Convolution was first implemented in the paper Xception: Deep Learning with Depthwise Separable Convolutions [6], where convolutional layers in inception models were modified into Depthwise convolutional layer which resulted in increase in accuracy and a significant decrease in the number of parameters.



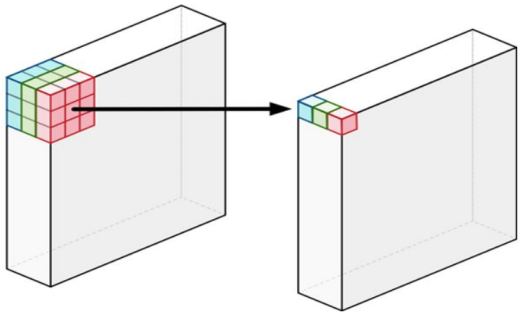
A regular convolutional layer applies a convolution kernel or filters to all of the channels of the input image. It slides this kernel across the image and at each step performs a weighted sum of the input pixels covered by the kernel across all input channels.

The important thing is that the convolution operation combines the values of all the input channels. If the image has 3 input channels, then running a single convolution kernel across this image results in an output image with only 1 channel per pixel. So for each input pixel, no matter how many channels it has, the convolution writes a new output pixel with only a single channel.

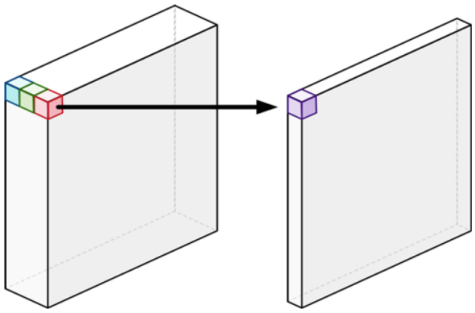
Unlike a regular convolution, Depthwise Separable Convolution does not combine the input channels but it performs convolution on each channel separately. For an image with 3 channels, a depthwise convolution creates an output image that also has 3 channels. Each channel gets its own set of weights.

The depthwise convolution is followed by a pointwise convolution. This really is the same as a regular convolution but with a 1×1 kernel. When we put these two things together — a depthwise convolution followed by a pointwise convolution — the result is called a depthwise separable convolution.

Step 1: Depthwise convolution



Step 2: Pointwise convolution

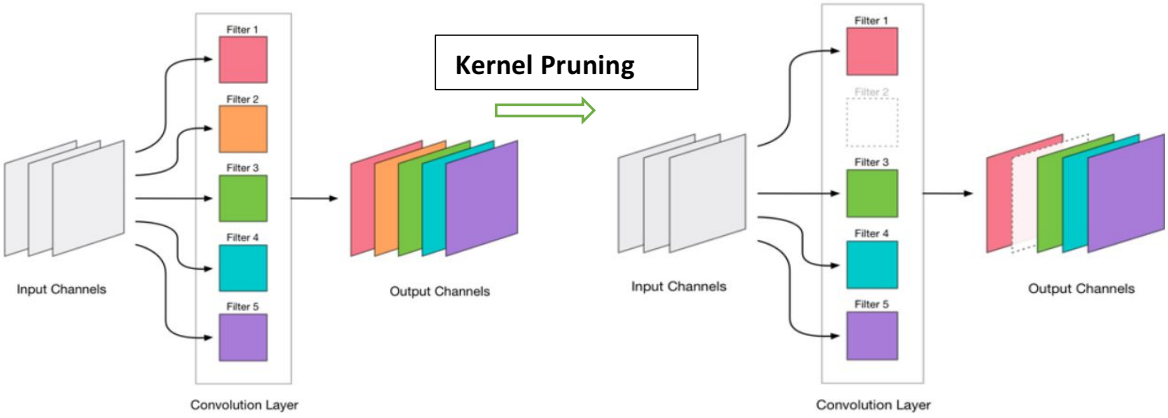


How does this work better? The end results of both approaches are similar — they both filter the data and make new features — but a regular convolution must do much more computational work to get there and needs to learn more weights. So even though it does the same thing, the depthwise separable convolution is going to be much faster. The paper shows the exact formula you can use to compute the speed difference but for 3×3 kernels this new approach is about 9 times as fast and still as effective.

2) Kernel Pruning

Earlier, we saw the problems possessed by Quantization and Sparse weights. To solve these issues, another technique is Kernel Pruning which is well documented in the paper: Pruning Filters for Efficient ConvNets [7].

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.



In Kernel Pruning, we find the kernel or filter which is least important according to the sorted values of l1 norm of its weight which is simply just a sum of absolute values of the kernel's weights. Therefore, we are removing weights for the entire kernel which is not very important thus reducing both the number of parameters as well as the latency.

Dataset

The ImageNet Large Scale Visual Recognition Challenge is a benchmark in object category classification and detection on 1000 object categories and 1.2 million images. The challenge has been run annually from 2010 to present, attracting participation from more than fifty institutions.

First, we downloaded the ILSVRC 2012 dataset from the ImageNet website (http://www.image-net.org/challenges/LSVRC/2012/) by following the instructions. However, the download can be very slow. Therefore, to move our data to cloud, we downloaded the dataset from a legal academic torrent (http://academictorrents.com/browse.php?search=imagenet) which is a much faster way to download the ImageNet ILSVRC 2012 data.

Model Selection & Evaluation

The data which we'll be using for training is the ImageNet dataset which has around 1.2 million images of 1000 categories. Our objective is to take a model with pre-trained weights and try and reduce the number of parameters to see how much it affects the accuracy and the latency.

These are our evaluation metrics:

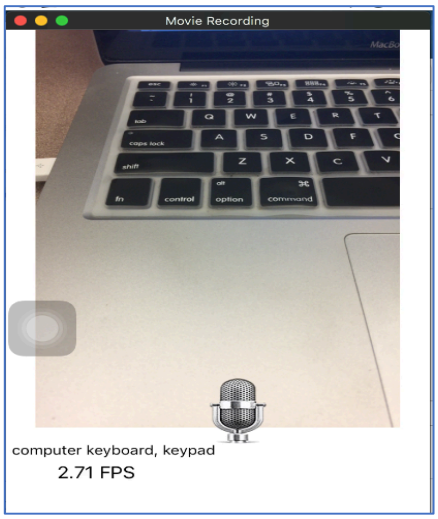
1) Top-1% accuracy for accuracy
2) Frames per second (FPS) for speed
3) Number of parameters for model size & power consumption

Let's look at the accuracy and number of parameters for some of the models:

Year	Model	Top 1% accuracy	# Parameters
2012	Alexnet	57%	62,378,344
2014	GoogleLeNet	68%	102,897,440
2014	VGG-16	71%	138,357,544
2015	ResNet-50	75%	25,636,712
2016	Inception-v3	78%	23,851,784
2017	Mobilenet	69%	4,253,864

After looking at the model features, we chose **Mobilenet** as our base model on which we will be applying the above techniques after going through the paper MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications [8] for the following reasons: -

- It already implements Depthwise Separable Convolution (DSC)
- It has the best accuracy *vs* number of parameters



Mobilenet already looked fast enough so we first evaluated the model on iPhone 6 which has just 1GB of RAM and video recording of 30 FPS.

The Mobilenet Prediction gives a speed of 3-6 FPS, which is slow for a real-time application. Hence, it would be slower on lower hardware devices such as Raspberry Pi or a surveillance camera.

Now the problem to solve is, can we make this algorithm faster?

Technology Stack

Hardware:

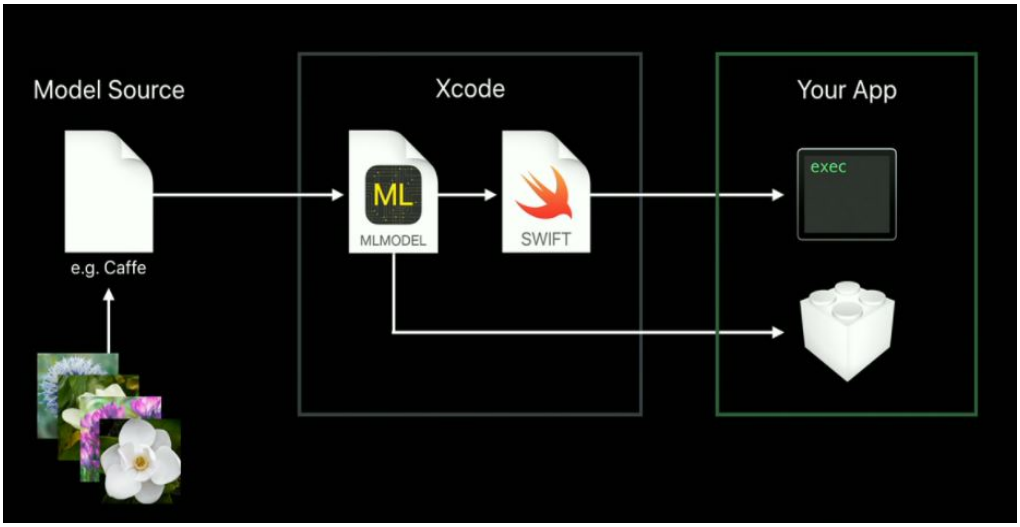
- Google Cloud Platform
- 8 Core CPU, 30 GB Memory
- 1 NVIDIA Tesla K80 GPU

```
ngmodani2094@csye-project-instance:~$ nvidia-smi
Wed Dec  6 11:20:04 2017
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 384.90                 Driver Version: 384.90                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   38C    P0   134W / 149W |  10964MiB / 11439MiB |    100%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      1976      G   /usr/lib/xorg/Xorg                            15MiB |
|    0      2646      C   /home/ngmodani2094/anaconda3/bin/python    10935MiB |
+-----------------------------------------------------------------------------+
```

Software:

- Python (Numpy, pandas, matplotlib) for pre-processing
- Keras with Tensorflow backend for deep learning
- CoreML - Apple's machine learning framework & Xcode for iOS app development



Methodology

1) **Load pre-trained model**
   Keras provides several pre-trained models on ImageNet which can be directly loaded into the model. We can also look at the summary of the model which provides us details about the all the layers, size and number of parameters in the model.

```
original_model = MobileNet(weights="imagenet")
original_model.compile(loss="categorical_crossentropy",optimizer="adam",
              metrics=["categorical_accuracy", "top_k_categorical_accuracy"])
original_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 224, 224, 3)       0
_________________________________________________________________
conv1 (Conv2D)               (None, 112, 112, 32)      864
_________________________________________________________________
conv1_bn (BatchNormalization (None, 112, 112, 32)      128
_________________________________________________________________
conv1_relu (Activation)      (None, 112, 112, 32)      0
_________________________________________________________________
conv_dw_1 (DepthwiseConv2D)  (None, 112, 112, 32)      288
_________________________________________________________________
conv_dw_1_bn (BatchNormaliza (None, 112, 112, 32)      128
_________________________________________________________________
conv_dw_1_relu (Activation)  (None, 112, 112, 32)      0
_________________________________________________________________
conv_pw_1 (Conv2D)           (None, 112, 112, 64)      2048
_________________________________________________________________
conv_pw_1_bn (BatchNormaliza (None, 112, 112, 64)      256
```

2) Evaluate the pre-trained model

Now we will validate the pre-trained model on the ImageNet 2012 validation sample which consists of 10,000 images and its labels.

```
val_sample = create_new_val_sample()
%time eval_on_sample(original_model, val_sample)
```

```
Found 1280 images belonging to 1000 classes.
CPU times: user 15 s, sys: 896 ms, total: 15.9 s
Wall time: 17.1 s

[1.2982454121112823, 0.69062500000000004, 0.88046875000000002]
```

Here, the returned list is: - ['*loss*', *'categorical_accuracy'*, '*top_k_categorical_accuracy*'] Therefore, we have a ***top 1% accuracy of 69.06%***

3) Model Architecture

If we look at the architecture, we find that there are three types of convolutional layers:

a) Regular 3x3 convolution
b) Depthwise convolution
c) Pointwise convolution

We can only remove filters/kernels from the regular and pointwise convolution layers, but not from the depthwise. A depthwise convolution has a constraint that it must always have the same number of output channels as it has input channels. On the brighter side, depthwise convolutions are pretty fast already and has few number of parameters, so there was not much gain there anyway. For this purpose, we will primary focus only on the 13 Regular 3x3 convolution and the pointwise layers.

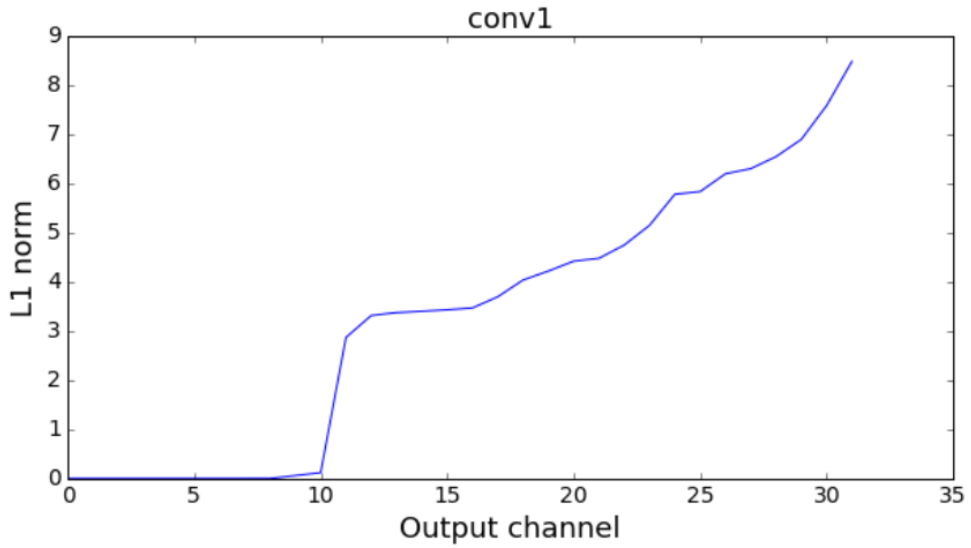
4) Kernel pruning

If we look at the number of filters in each convolution layer, we will notice that it increases with increasing depth. The first convolution layer has 32 filters/864 parameters, and the last convolution has 1024 filters/1048576 parameters. Therefore, we decided to start with the last layer and then move up to the upper layers.

Which filters channels do we remove? We only want to get rid of output channels that do not influence the outcome too much. There are different metrics you 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 as mentioned in the paper [7].

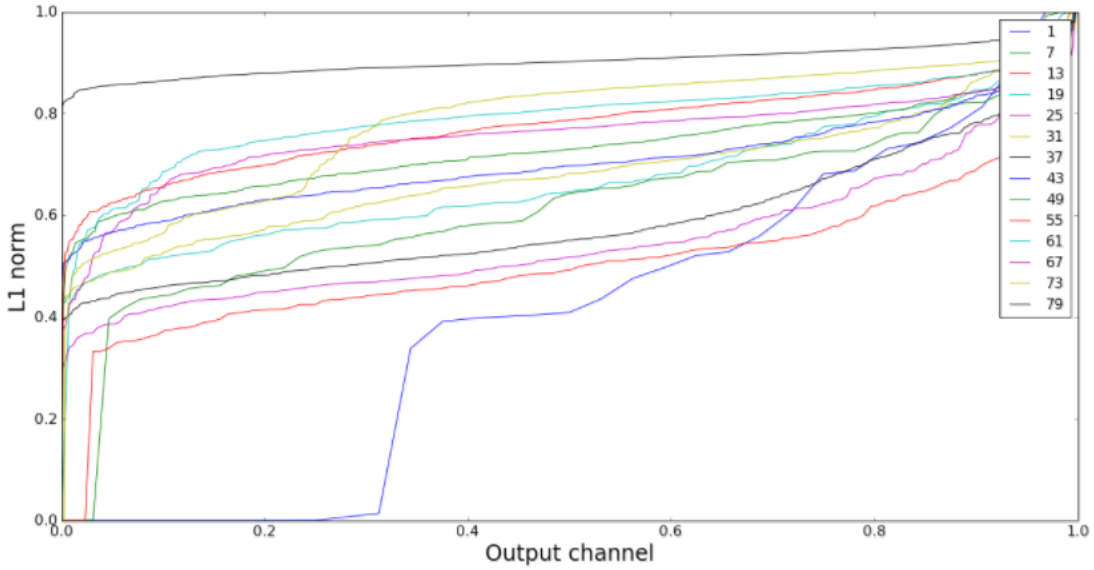
```
: # last Conv Layer is "conv1" at index = 1
  fig = plt.figure(figsize=(10, 5))
  plt.plot(list(map(lambda x: x[1], get_l1_norms(original_model, 1))))
  plt.xlabel("Output channel", fontsize=18)
  plt.ylabel("L1 norm", fontsize=18)
  plt.title(original_model.layers[1].name, fontsize=18)

: <matplotlib.text.Text at 0x7fd6241052b0>
```

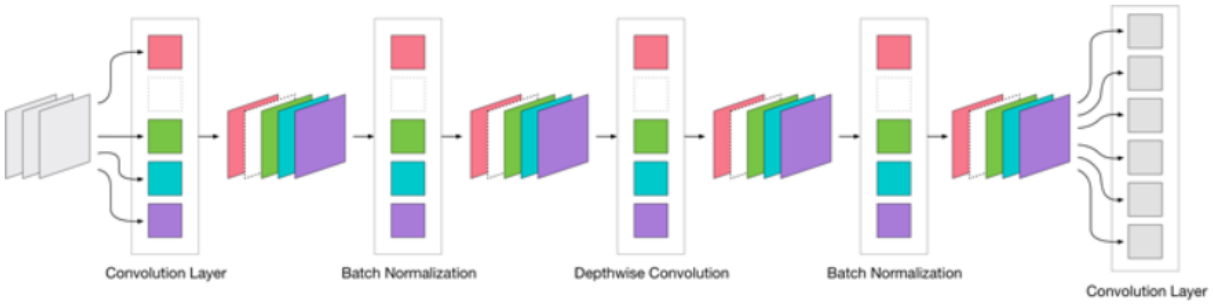


The above plot shows that out of 32 filters, there are 10 filters in the first layer with an L1 norm that is very small. So, it makes sense to remove these layers from the network because they hardly contribute much to the decision or the accuracy.

As discussed before, the last convolution layer has 1024 filters compared to the first convolution layer which is equivalent to 1048576 parameters. Therefore, we can make big gains here. The plot below shows the 13 convolution layers and its l1 norms. Since all these layers have different number of filters, we have normalized the x and y values between 0 and 1 so that it is easier to compare. The legend indicates the depth of the layer.



Another thing we must note is that since we are removing filters from the output channels, the layer becomes smaller. Therefore, it also affects the subsequent layer in the network as well because the next layer receives fewer inputs than it expects. As a result, we also must remove the corresponding input channels from that layer. And when the convolution is followed by batch normalization, we also must remove these channels from the batch norm parameters.



5) Retraining the model

When we removed the filters, and evaluated the model we noticed that the accuracy of the network drops. For example, after pruning the last convolution layer, our accuracy dropped from 69% to 62%. Therefore, we re-trained the model to compensate for the parts which we dropped.

We used trial and error to get to a learning rate of 0.00001 which might seem very small but makes sense because the network is already trained and we only performed small changes in the network.

```
new_model.fit_generator(train_gen,steps_per_epoch=156,epochs=epochs,validation_data=val_gen,validation_steps=20)
```

```
Epoch 1/5
156/156 [==============================] - 148s 947ms/step - loss: 1.1153 - categorical_accuracy: 0.7177 - top_k_categorical_ac
curacy: 0.9189 - val_loss: 1.6122 - val_categorical_accuracy: 0.6250 - val_top_k_categorical_accuracy: 0.8367
Epoch 2/5
156/156 [==============================] - 140s 895ms/step - loss: 0.6926 - categorical_accuracy: 0.8538 - top_k_categorical_ac
curacy: 0.9684 - val_loss: 1.5195 - val_categorical_accuracy: 0.6391 - val_top_k_categorical_accuracy: 0.8523
Epoch 3/5
156/156 [==============================] - 139s 893ms/step - loss: 0.4906 - categorical_accuracy: 0.9228 - top_k_categorical_ac
curacy: 0.9851 - val_loss: 1.4902 - val_categorical_accuracy: 0.6477 - val_top_k_categorical_accuracy: 0.8547
Epoch 4/5
156/156 [==============================] - 139s 893ms/step - loss: 0.3584 - categorical_accuracy: 0.9598 - top_k_categorical_ac
curacy: 0.9947 - val_loss: 1.4754 - val_categorical_accuracy: 0.6531 - val_top_k_categorical_accuracy: 0.8492
Epoch 5/5
156/156 [==============================] - 140s 894ms/step - loss: 0.2668 - categorical_accuracy: 0.9827 - top_k_categorical_ac
curacy: 0.9976 - val_loss: 1.4718 - val_categorical_accuracy: 0.6539 - val_top_k_categorical_accuracy: 0.8539

<keras.callbacks.History at 0x7fd5bf593a58>
```

```
%time eval_on_sample(new_model, val_sample)
```

```
Found 1280 images belonging to 1000 classes.
CPU times: user 13.4 s, sys: 540 ms, total: 14 s
Wall time: 13.3 s

[1.4717752397060395, 0.65390625000000002, 0.85390624999999998]
```

Since, we have 1.2 million images, it would be very time intensive to perform trial and error on so many images. Hence, we sampled 10,000 images from the training data (10 images

per classes) to see if the accuracy increases. If we see good results in the sample data, we performed full training on the 1.2 million images to get back our original accuracy.

```
new_model.fit_generator(train_gen,steps_per_epoch=20018,epochs=epochs,
    validation_data=val_gen,validation_steps= 781)
```

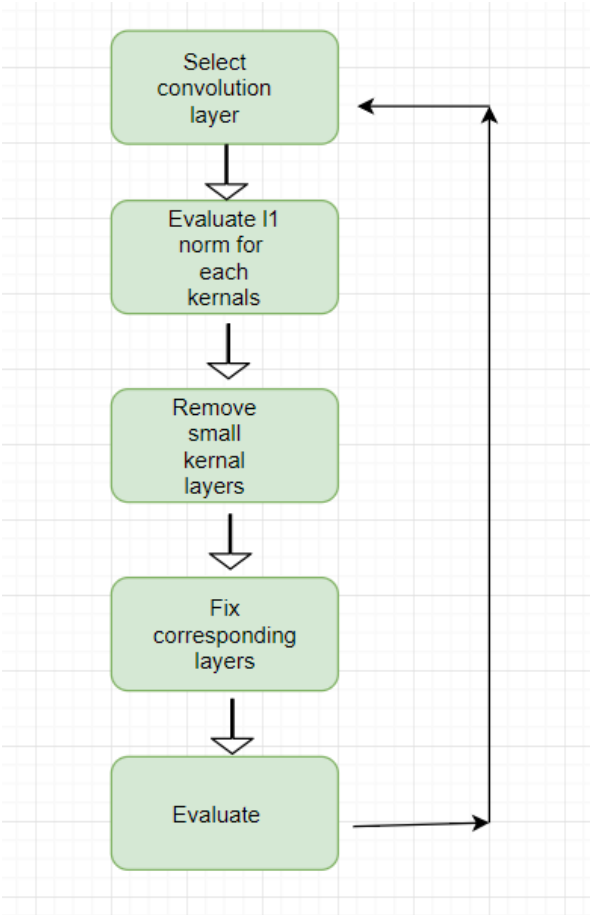
```
Epoch 1/1
20018/20018 [==============================] - 16661s 832ms/step - loss: 0.7493 - categorical_accuracy: 0.8047 - top_k_categori
cal_accuracy: 0.9574 - val_loss: 1.3115 - val_categorical_accuracy: 0.6801 - val_top_k_categorical_accuracy: 0.8828

<keras.callbacks.History at 0x7fd56d808f28>
```

After re-training the model on the full dataset, our accuracy went up from 65% to 68%.

Therefore, the process is:

a) Select the convolution layer
b) Remove the filters from the layer using the l1 norm plot
c) Retrain the layer for a few epochs
d) Evaluate the results
e) Repeat step (a) to (d) for the next convolution layer

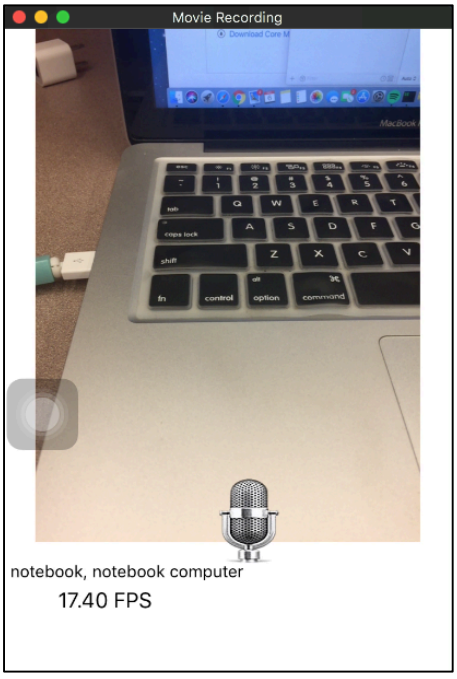


Conclusion

We reduced the number of parameters by 16% with less than 1% drop in accuracy. This was better than what we had expected. This was achieved by just pruning the last layer. We still have 12 more convolution layers to go. However, we decided to stop after pruning the last layer to maintain the accuracy and complexity associated with the configuration of the model. The result is summarized below:

	Original model	Pruned model (only last layer)
Accuracy	69.06 %	68.12 %
No of parameters	4,253,864	3,583,656 (84% of the original)
FPS	0 to 5	10 to 15

After testing on iPhone 6, we also decreased the inference time from 5 FPS to 15 FPS. 5 FPS meant that every frame took 0.2 seconds. This increased to 15 FPS which is about 0.07 seconds which is an improvement of about 65% from the previous model. This could make a huge difference and make the app much smoother. Also, since we have decreased the number of parameters, it has also improved the size of the model and the power consumption as the app will need to make 16% less computation than before.



In future, we expect to perform kernel pruning in all the 13 convolution layers and achieve a reduction in number of parameters by 30%.

Acknowledgment

We would like to show our gratitude to professor Nik Bear Brown for guiding us and encouraging us during the course of this project.

References

[1]. WHO Media Center: *http://www.who.int/mediacentre/factsheets/fs282/en/*
[2]. Statista – The portal for statistics*: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/*
[3]. Chen, Xi, et al. "FxpNet: Training a Deep Convolutional Neural Network in Fixed-Point Representation." 2017 International Joint Conference on Neural Networks (IJCNN), 2017, *doi:10.1109/ijcnn.2017.7966159.*
[4]. Quantize Neural Networks with TensorFlow *(https://www.tensorflow.org/performance/quantization)*
[5]. Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding*. arXiv preprint arXiv:1510.00149, 2015.*
[6]. F. Chollet. Xception: Deep learning with depthwise separable convolutions. *arXiv preprint arXiv:1610.02357, 2016. 1, 2, 3, 4, 5, 6*
[7]. H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets*. arXiv preprint arXiv:1608.08710, 2016.*
[8]. A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang,T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861, 2017.*
[9]. Compressing Deep neural net *(http://machinethink.net/blog/compressing-deep-neural-nets/)*