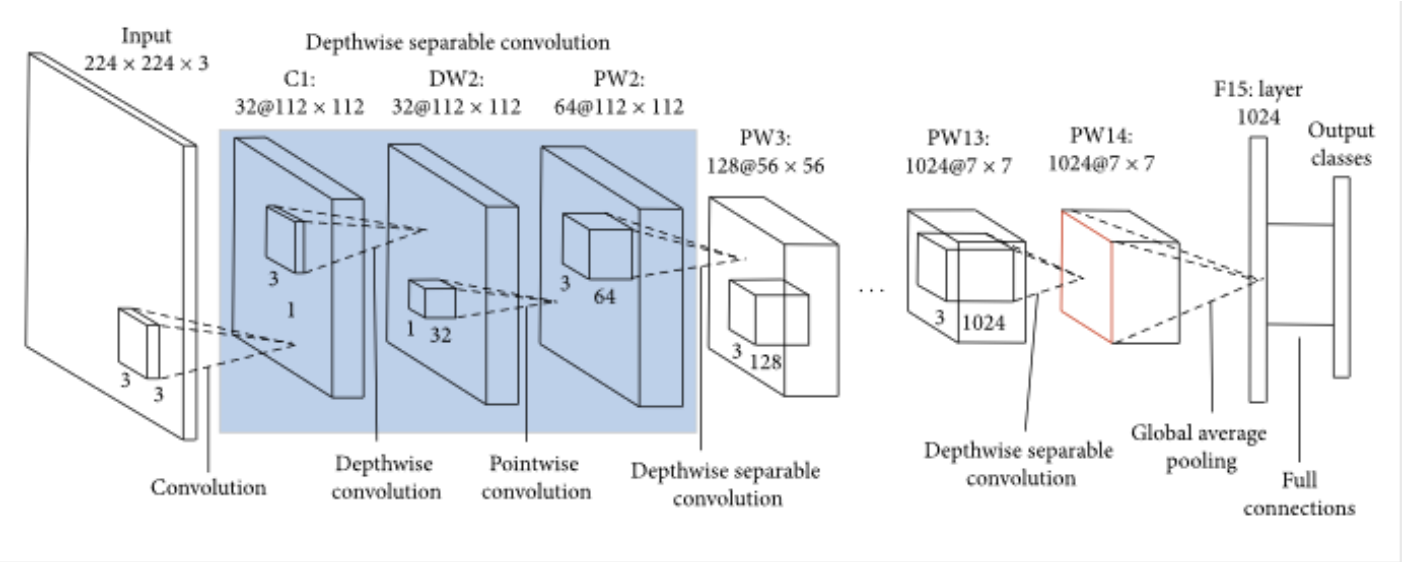
검색어를 입력하세요.

Deep Learning Bible - 2. Classification - English
(/book/7972)

09.04 Understanding of MobileNet - EN



This article covers five parts:

1. What is MobileNet?
2. The Architecture of MobileNet.
3. Depthwise Separable Convolution.
4. Difference between Standard Convolution and Depthwise separable convolution.

1. What is MobileNet?

As the name applied, the MobileNet model is designed to be used in mobile applications, and it is TensorFlow's first mobile computer vision model.

MobileNet is a simple but efficient and not very computationally intensive convolutional neural networks for mobile vision applications. MobileNet is widely used in many real-world applications which includes object detection, fine-grained classifications, face attributes, and localization. In this lecture, I will explain you the

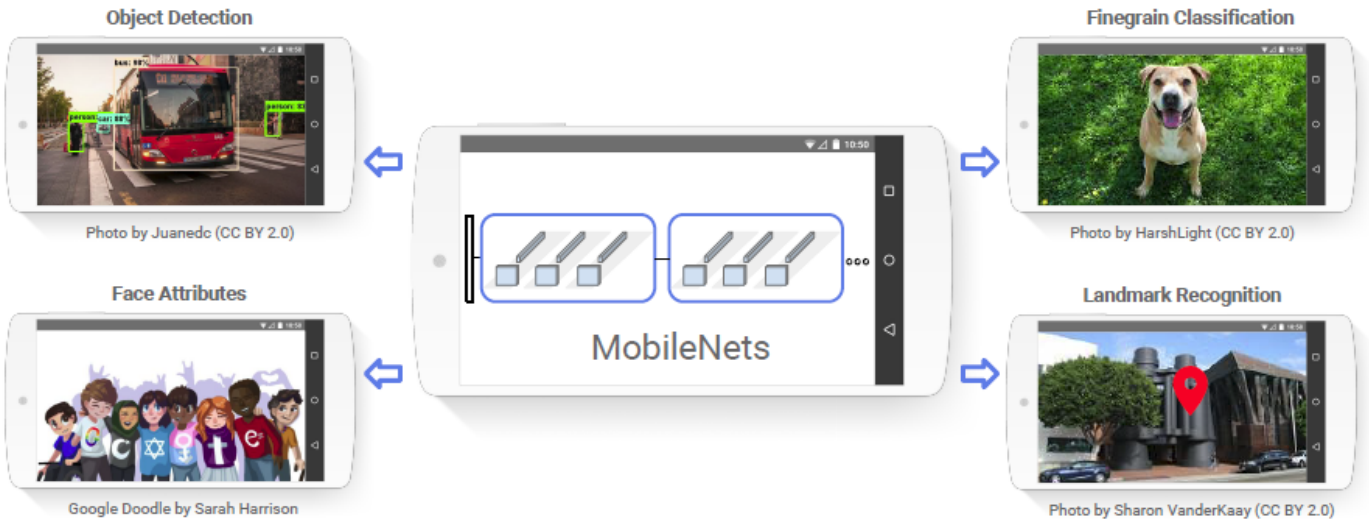
overview of MobileNet and how exactly it becomes the most efficient and lightweight neural network.

MobileNet uses depthwise separable convolutions. It significantly reduces the number of parameters when compared to the network with regular convolutions with the same depth in the nets. This results in lightweight deep neural networks.

A depthwise separable convolution is made from two operations.

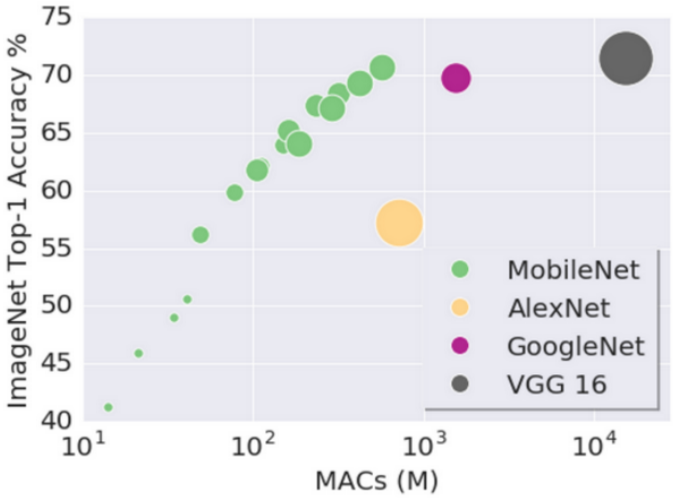
- Depthwise convolution.
- Pointwise convolution.

MobileNet is a class of CNN that was open-sourced by Google, and therefore, this gives us an excellent starting point for training our classifiers that are insanely small and insanely fast.



When MobileNets Applied to Real Life

The speed and power consumption of the network is proportional to the number of MACs (Multiply-Accumulates) which is a measure of the number of fused Multiplication and Addition operations.



2. The Architecture of MobileNet.

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Mobilenet Layers from paper

The full MobileNets network has 30 layers. The design of the network is quite straightforward:

1. convolutional layer with stride 2
2. depthwise layer
3. pointwise layer that doubles the number of channels
4. depthwise layer with stride 2
5. pointwise layer that doubles the number of channels
6. depthwise layer
7. pointwise layer
8. depthwise layer with stride 2
9. pointwise layer that doubles the number of channels

and so on…

After the very first layer (a regular convolution), the depthwise and pointwise layers take turns. Sometimes the depthwise layer has a stride of 2, to reduce the width and height of the data as it flows through the network. Sometimes the pointwise layer doubles the number of channels in the data. All the convolutional layers are followed by a ReLU activation function.

This goes on for a while until the original 224×224 image is shrunk down to 7×7 pixels but now has 1024 channels. After this there's an average-pooling layer that works on the entire image so that we end up with a $1 \times 1 \times 1024$ image, which is really just a vector of 1024 elements.

If we're using MobileNets as a classifier, for example on ImageNet which has 1000 possible categories, then the final layer is a fully-connected layer with a softmax and 1000 outputs. If you wanted to use MobileNets on a different dataset, or as a feature extractor instead of classifier, you'd use some other final layer instead.

Note: The paper actually states that between each convolution and the ReLU there is a batch normalization layer. For our purposes we can ignore those layers as they're only used during training. Since the convolution operation is a linear transform and so is batch normalization, we can multiply the weights learned by batch normalization with the weights of the convolution layer before it. This saves on multiply operations at inference time.

Even though MobileNets is designed to be pretty fast already, it's possible to use a reduced version of this network architecture. There are three hyperparameters you can set that determine the size of the network:

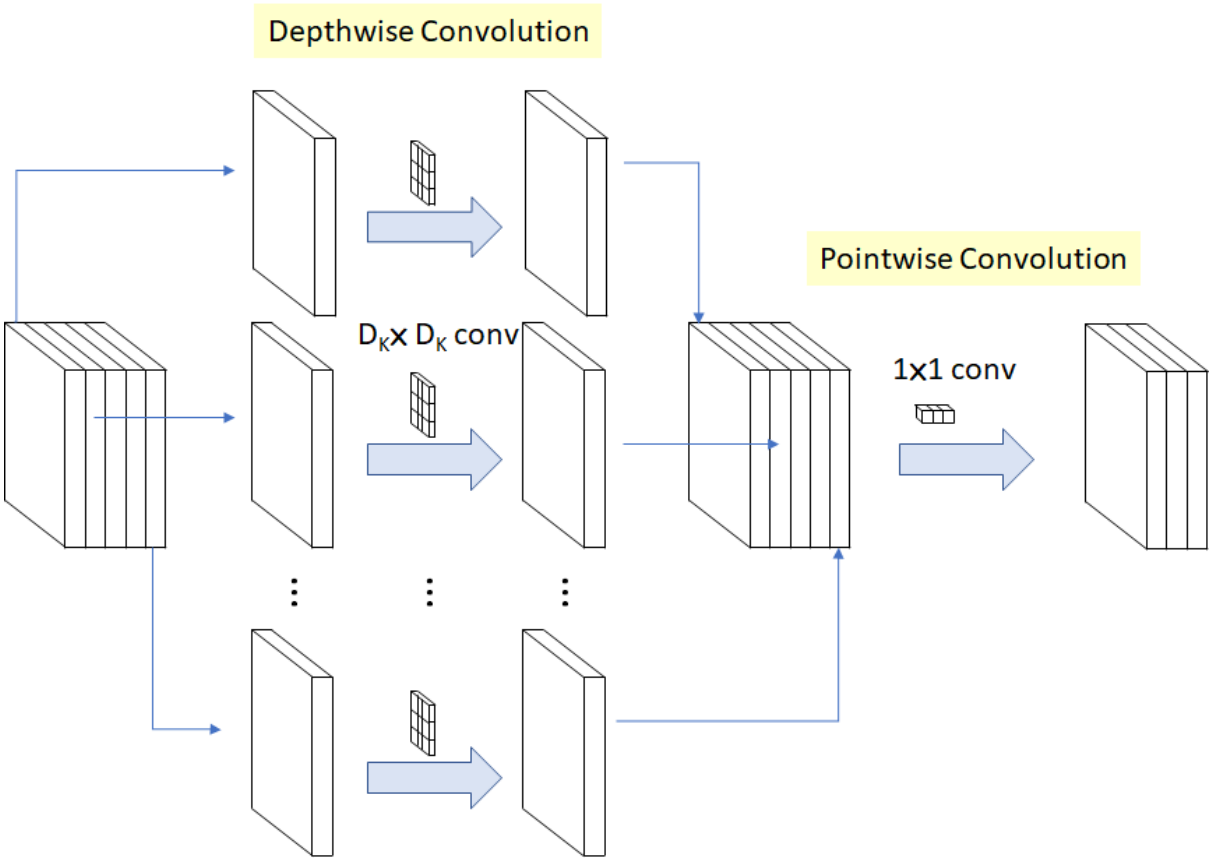
- Width multiplier (the paper calls this "alpha"): this shrinks the number of channels. If the width multiplier is 1, the network starts off with 32 channels and ends up with 1024.
- Resolution multiplier ("rho" in the paper): this shrinks the dimensions of the input image. The default input size is 224×224 pixels.
- Shallow or deep. The full network has a group of 5 layers in the middle that you can leave out.

These settings can be used to make the network smaller — and therefore faster — but at the cost of prediction accuracy. (If you're curious, the paper shows the effects of changing these hyperparameters on the accuracy.)

For the full network the total number of learned parameters is 4,221,032 (after folding the batch normalization layers). That's certainly a lot less than VGGNet, which has over 130 million!

3. Depthwise Separable Convolution.

A depthwise separable convolution is a depthwise convolution followed by a pointwise convolution as follows.



Depthwise Separable Convolution

Depthwise convolution is the channel-wise $D_K \times D_K$ spatial convolution. Suppose in the figure above, and we have five channels; then, we will have 5 $D_K \times D_K$ spatial convolutions.

Pointwise convolution is the 1×1 convolution to change the dimension.

3.1 Sobel operator

This convolution originated from the idea that a filter's depth and spatial dimension can be separated- thus, the name separable. Let us take the example of Sobel filter, used in image processing to detect edges.

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Sobel Filter. Gx for the vertical edge, Gy for horizontal edge detection

You can separate the height and width dimensions of these filters. Gx filter can be viewed as a matrix product of [1 2 1] transpose with [-1 0 1].

We notice that the filter had disguised itself. It shows it had nine parameters, but it has 6. This has been possible because of the separation of its height and width dimensions.

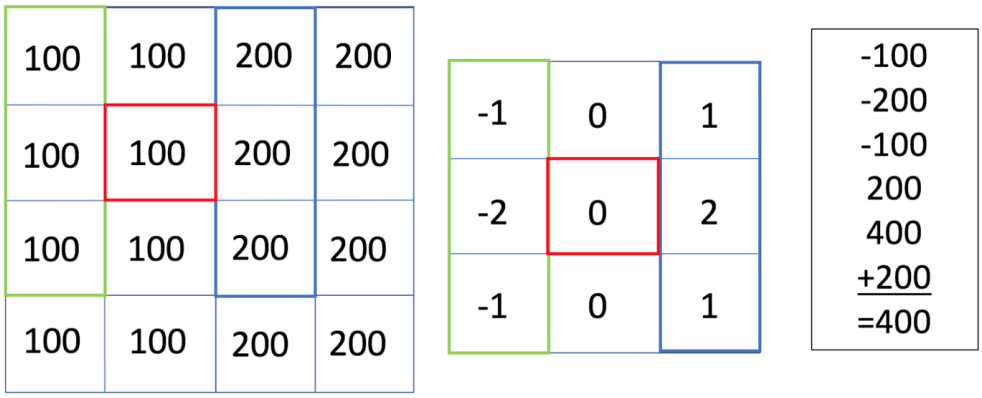
The same idea applied to separate depth dimension from horizontal ($width \times height$) gives us depth-wise separable convolution whare we perform depth-wise convolution. After that, we use a 1×1 filter to cover the depth dimension.

One thing to notice is how much parameters are reduced by this convolution to output the same no. of channels. To produce one channel, we need $3 \times 3 \times 3$ parameters to perform depth-wise convolution and 1×3 parameters to perform further convolution in-depth dimension.

But If we need three output channels, we only need 31×3 depth filter, giving us a total of $36 (= 27 + 9)$ parameters while for the same no. of output channels in regular convolution, we need $33 \times 3 \times 3$ filters giving us a total of 81 parameters.

Depthwise separable convolution is a depthwise convolution followed by a pointwise convolution as follows:

Essentially what we are trying to do here with the Sobel Operator is trying to find out the amount of the difference by placing the gradient matrix over each pixel of our image. We get two images as output, one for X- Direction and other for Y-Direction. By using Kernel Convolution, we can see in the example image below there is an edge between the column of 100 and 200 values.

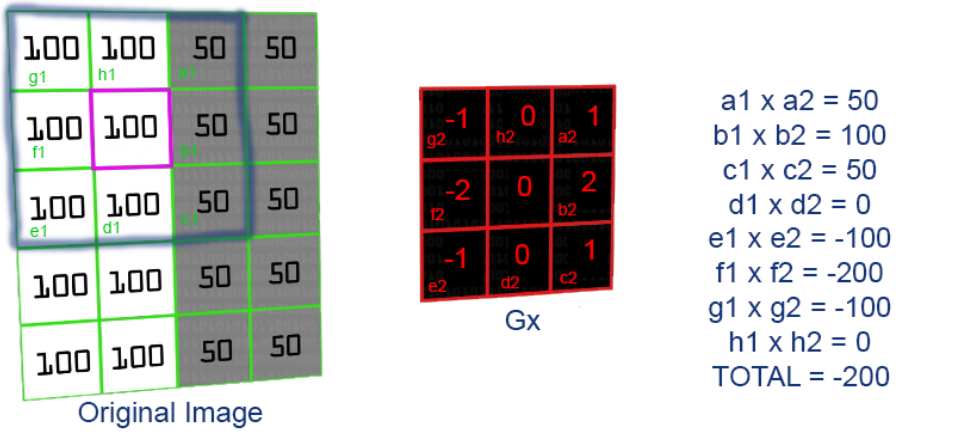


Kernel Convolution: The bigger the value at the end, the more noticeable the edge will be.

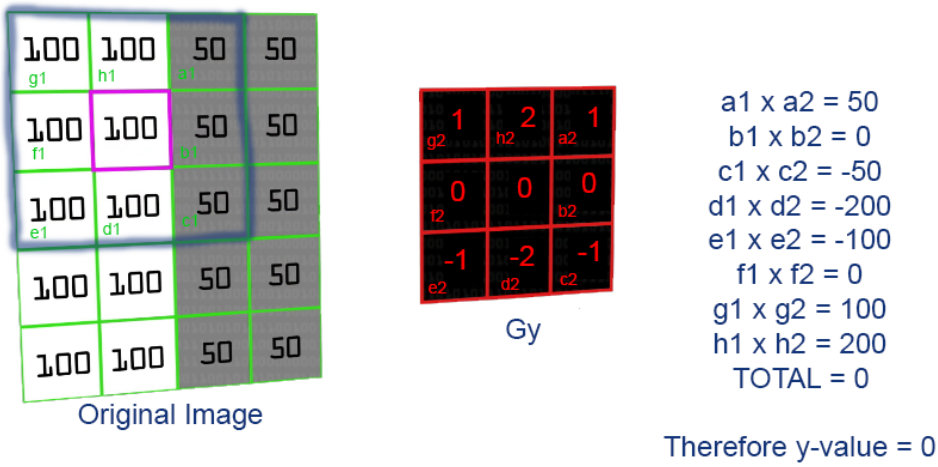
The above example shows the result of doing convolution by placing the Gradient matrix X over a red marked 100 of images. The calculation is shown on the right which sums up to 400, which is non-zero, hence there is an edge. If all the pixels of images were of the same value, then the convolution would

result in a resultant sum of zero. So the gradient matrix will provide a big response when one side is brighter . Another point to note here is that the sign of the output resultant does not matter.

Lets take a look at another example :



Illustrating how the calculation is done for a pixel in the x-direction using Gx.

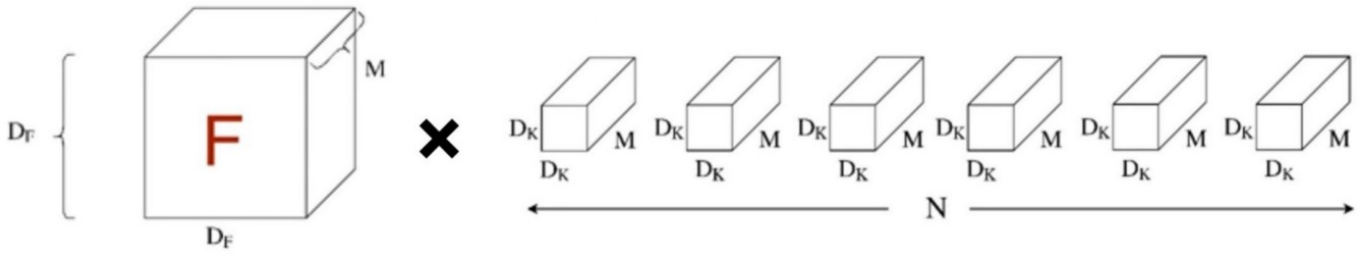


Illustrating how the calculation is done for a pixel in the y-direction using Gy.

There are many other useful deductions that can be derived after calculating the Edge using Gx and Gy.

3.2 Standard Convolution

Let's take a standard convolution,



Standard Convolution

From the above image, the computational cost can be calculated as :

$$D_K \times D_K \times M \times N \times D_F \times D_F$$

Standard Convolution Cost

Where D_F is the special dimensions of the input feature map and D_K is the size of the convolution kernel. Here M and N are the number of input and output channels respectively.

For a standard convolution, the computational cost depends multiplicatively on the number of input and output channels and on the spatial dimensions of the input feature map and convolution kernel.

In this convolution, we apply a 2-d depth filter at each depth level of input tensor. Lets understand this through an example. Suppose our input tensor is $3 \times 8 \times 8$ (input_channels × width × height). Filter is $3 \times 3 \times 3$. In a standard convolution we would directly convolve in depth dimension as well (fig 1).

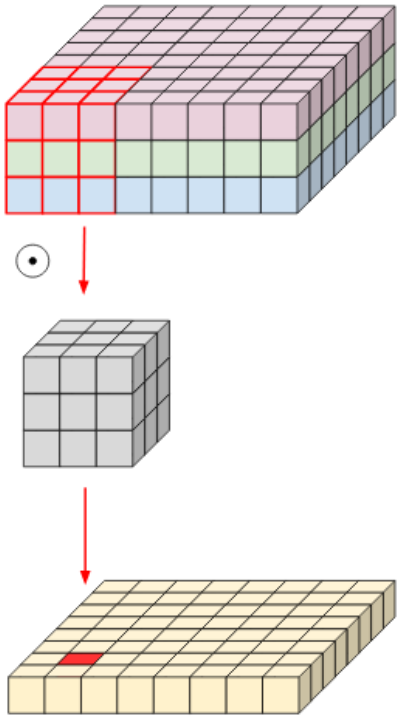
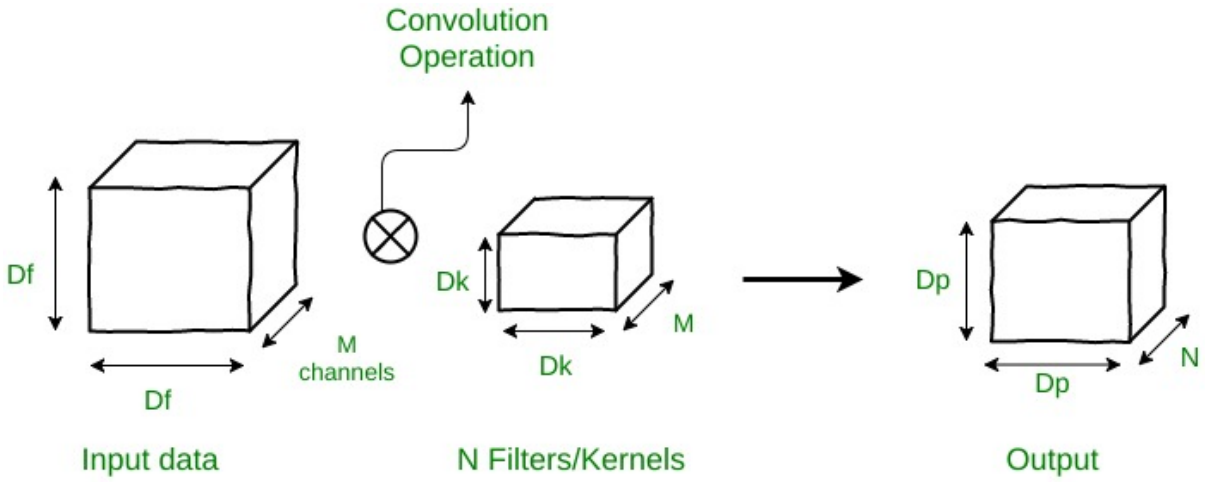


Fig 1. Normal convolution

If the Stride has a size other than 1, it is expressed slightly differently. In the above case, since size of stide is 1, both input and output sizes are given as D_F. In MobileNet, if stide is 2, it is done only in the first layer, and stride is 1 in all other cases. If the stride is 2, the equation is given as follows.

Suppose there is an input data of size Df × Df × M, where Df × Df can be the image size and M is the number of channels (3 for an RGB image). Suppose there are N filters/kernels of size Dk × Dk × M. If a normal convolution operation is done, then, the output size will be Dp × Dp × N.



The number of multiplications in 1 convolution operation = size of filter = Dk × Dk × M

Since there are N filters and each filter slides vertically and horizontally Dp times,

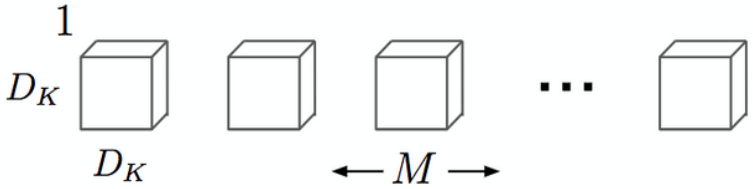
the total number of multiplications become N × Dp × Dp × (Multiplications per convolution)

So for normal convolution operation

Total no of multiplications = $N \times Dp^2 \times Dk^2 \times M$

3.3 Depthwise Convolution

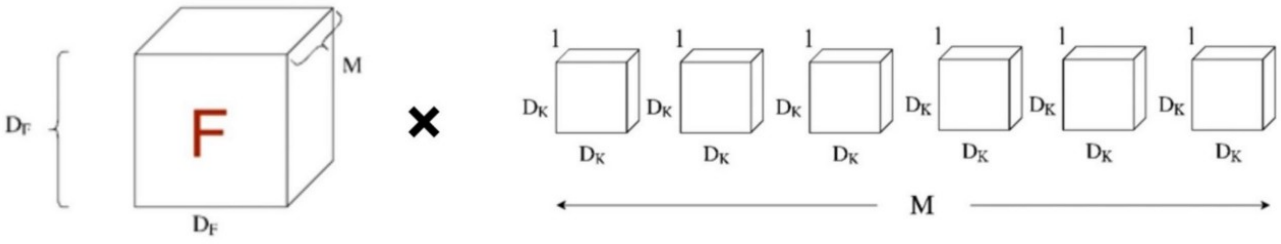
The depth-wise convolutions are used to apply a single filter into each input channel. This is different from a standard convolution in which the filters are applied to all of the input channels.



Depthwise convolution.

It is a map of a single convolution on each input channel separately. Therefore its number of output channels is the same as the number of the input channels. Its computational cost is $D_F \times D_F \times M \times D_K \times D_K$.

Incase of depthwise convolution, as seen in the below image, contains an input feature map of dimension $D_F \times D_F$ and M number of kernels of channel size 1.



Depth-wise Convolution

As per the above image, we can clearly see that the total computational cost can be calculated as:

$$D_K \times D_K \times M \times D_F \times D_F$$

Depth-Wise Convolution cost

In depth-wise convolution, we use each filter channel only at one input channel. In the example, we have 3 channel filter and 3 channel image. What we do is — break the filter and image into three different channels and then convolve the corresponding image with corresponding channel and then stack them back (Fig 2)

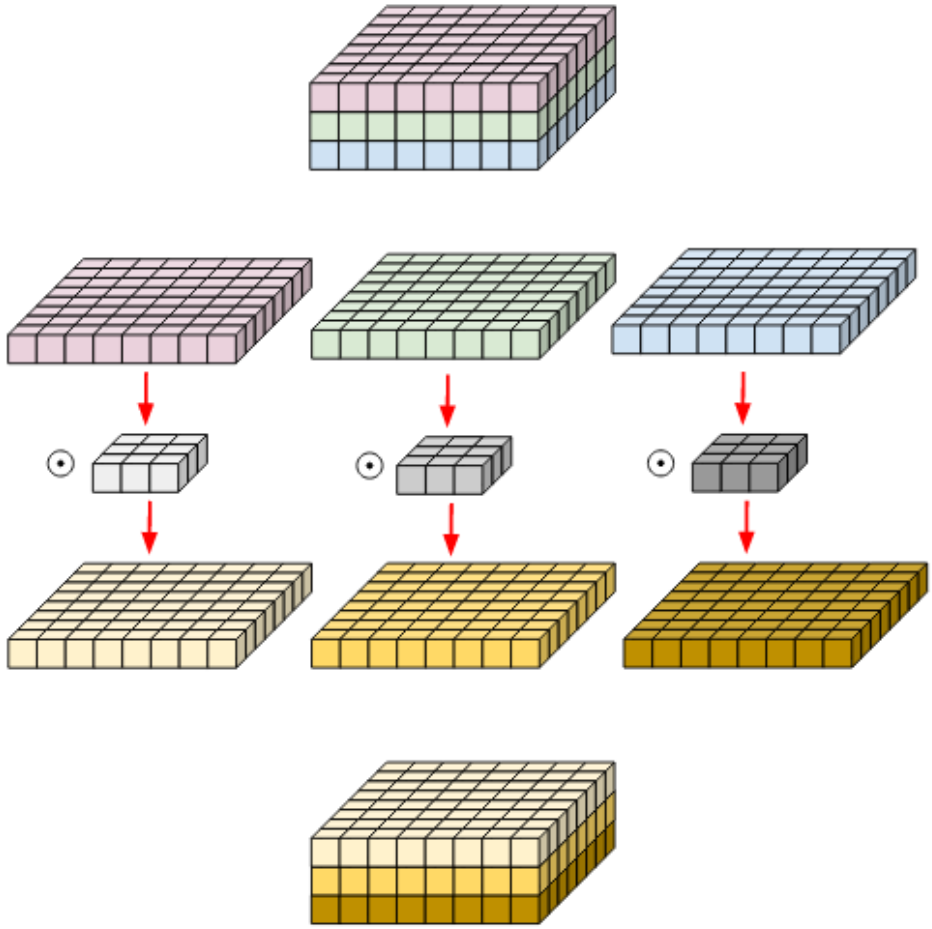
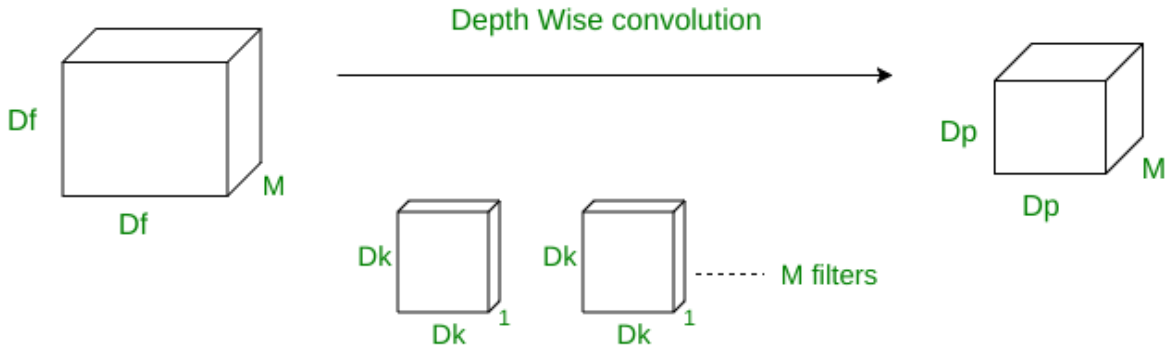


Fig 2. Depth-wise convolution. Filters and image have been broken into three different channels and then convolved separately and stacked thereafter

To produce same effect with normal convolution, what we need to do is- select a channel, make all the elements zero in the filter except that channel and then convolve. We will need three different filters — one for each channel. Although parameters are remaining same, this convolution gives you three output channels with only one 3-channel filter while, you would require three 3-channel filters if you would use normal convolution.

In depth-wise operation, convolution is applied to a single channel at a time unlike standard CNN's in which it is done for all the M channels. So here the filters/kernels will be of size Dk × Dk × 1. Given there are M channels in the input data, then M such filters are required. Output will be of size Dp × Dp × M.



Cost of this operation:

A single convolution operation require Dk × Dk multiplications.

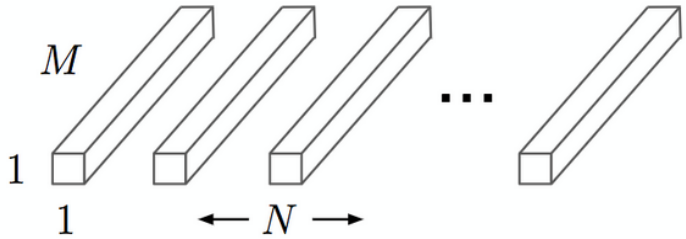
Since the filter are slided by Dp × Dp times across all the M channels,

the total number of multiplications is equal to $M \times Dp \times Dp \times Dk \times Dk$

So for depth wise convolution operation

Total no of multiplications = $M \times Dk^2 \times Dp^2$

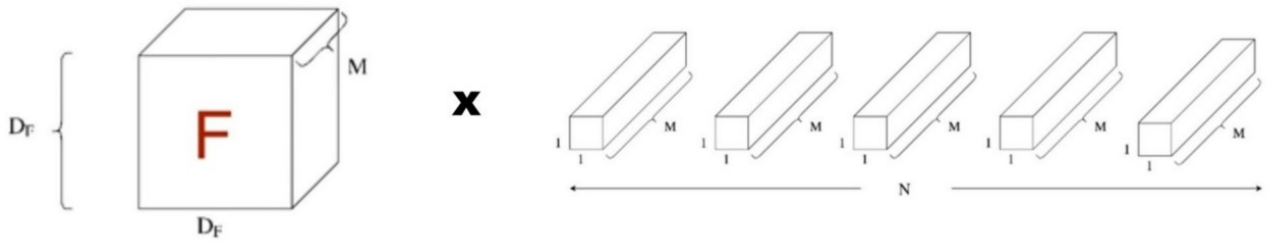
3.4 Pointwise Convolution



pointwise convolution

Convolution with a kernel size of 1×1 that simply combines the features created by the depthwise convolution. Its computational cost is $M \times N \times D_F \times D_F$.

Since the depthwise convolution is only used to filter the input channel, it does not combine them to produce new features. So an additional layer called pointwise convolution layer is made, which computes a linear combination of the output of depthwise convolution using a 1×1 convolution.



Point-wise Convolution

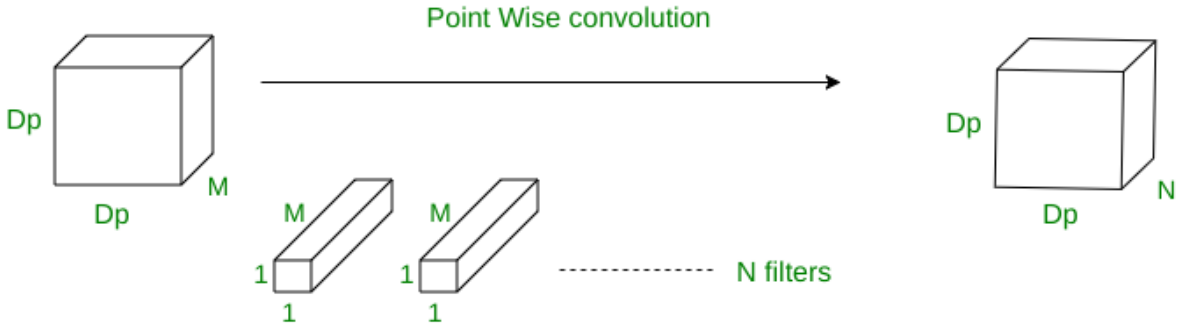
As per the image, let's calculate the computational cost again:

$$M \times N \times D_F \times D_F$$

Pointwise convolution cost

The above case applies when Stide is 1. Point-wise convolution can also be applied when strider is not 1.

In point-wise operation, a 1×1 convolution operation is applied on the M channels. So the filter size for this operation will be $1 \times 1 \times M$. Say we use N such filters, the output size becomes $Dp \times Dp \times N$.



Cost of this operation:

A single convolution operation require 1 × M multiplications.

Since the filter is being slided by Dp × Dp times,

the total number of multiplications is equal to M × Dp × Dp × (no. of filters)

So for point wise convolution operation

Total no of multiplications = $M \times Dp^2 \times N$

3.5 Depth-wise Separable Convolution

This convolution originated from the idea that depth and spatial dimension of a filter can be separated- thus the name separable. Let us take the example of Sobel filter, used in image processing to detect edges. You can separate the height and width dimension of these filters. Gx filter (see fig 3) can be viewed as matrix product of [1 2 1] transpose with [-1 0 1]. We notice

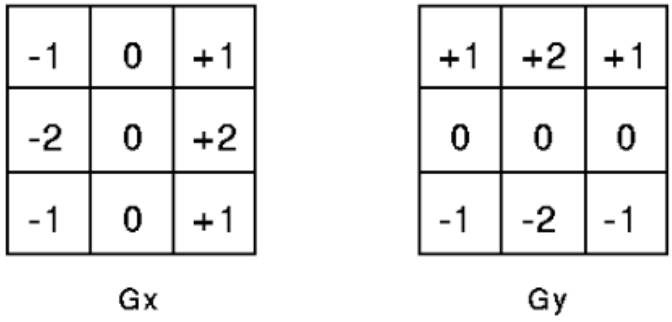


Fig 3. Sobel Filter. Gx for vertical edge, Gy for horzontal edge detection

that the filter had disguised itself. It shows it had 9 parameters but it has actually 6. This has been possible because of separation of its height and width dimensions. The same idea applied to separate depth dimension from horizontal (width × height) gives us depth-wise separable convolution whare we perform depth-wise convolution and after that we use a 1×1 filter to cover the depth dimension (fig 3).

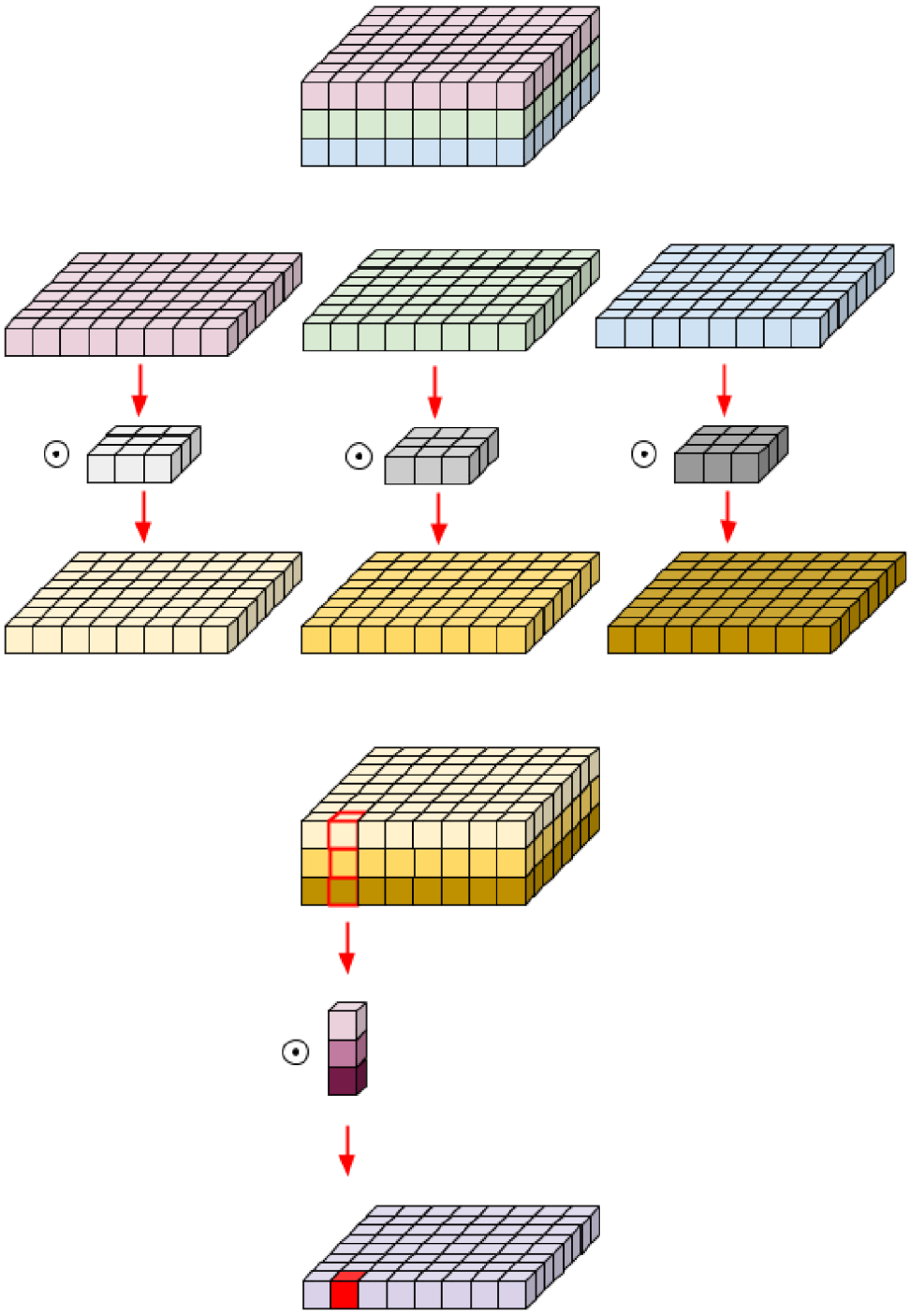


Fig 4. Depth-wise separable convolution

One thing to notice is, how much parameters are reduced by this convolution to output same no. of channels. To produce one channels we need $3\times3\times3$ parameters to perform depth-wise convolution and 1×3 parameters to perform further convolution in depth dimension. But If we need 3 output channels, we only need 3 1×3 depth filter giving us total of $36(=27+9)$ parameters while for same no. of output channels in normal convolution, we need 3 $3\times3\times3$ filters giving us total of 81 parameters. Having too many parameters forces function to memorize lather than learn and thus over-fitting. Depth-wise separable convolution saves us from that.

So the total computational cost of Depthwise separable convolutions can be calculated as:

$$D_K \times D_K \times M \times D_F \times D_F + M \times N \times D_F \times D_F$$

Depthwise separable convolutions cost

Comparing it with the computational cost os standard convolution, we get a reduction in computation, which can be expressed as:

$$\frac{D_K \times D_K \times M \times D_F \times D_F + M \times N \times D_F \times D_F}{D_K \times D_K \times M \times N \times D_F \times D_F} = \frac{1}{N} + \frac{1}{D_K^2}$$

Depthwise separable convolutions cost with standard convolution cost

To put this in a perspective to check the effectiveness of this depthwise separable convolution. Let's take an example.
Let's take $N = 1024$ and $D_k = 3$, plugging the values into the equation.
We get 0.112, or in another word, standard convolution has 9 times more the number of multiplication than that of the Depthwise convolution.

If Stride is not 1, for overall operation:

Total multiplications = Depth wise conv. multiplications + Point wise conv. multiplications

Total multiplications = $M * Dk2 * Dp2 + M * Dp2 * N = M * Dp^2 * (Dk^2 + n)$

So for depth wise separable convolution operation

Total no of multiplications = $M \times Dp^2 \times (Dk^2 + N)$

Comparison between the complexities of these types of convolution operations

$$\begin{aligned} \text{Type of Convolution} &: \text{Complexity} < \\ \text{Standard} &: N \times Dp^2 \times Dk^2 \times M \\ \text{Depth wise separable} &: M \times Dp^2 \times (Dk^2 + N) \end{aligned}$$

Ratio R is define as below:

$$\frac{\text{Complexity of depth wise separable convolutions}}{\text{Complexity of standard convolution}} = Ratio(R)$$

Upon solving:

$$Ratio(R) = \frac{1}{N} + \frac{1}{Dk^2}$$

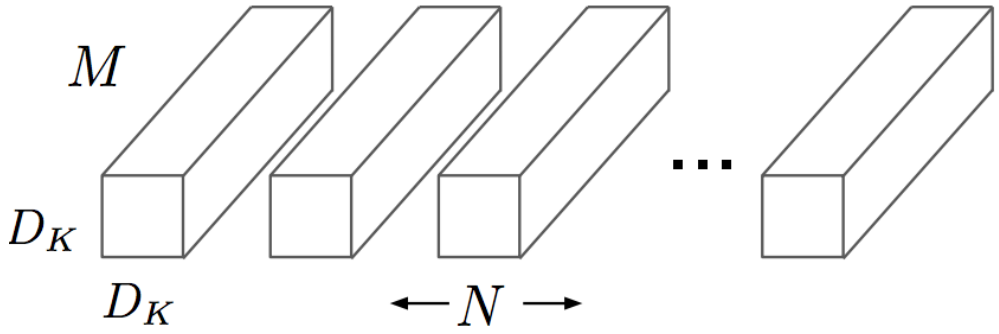
As an example, consider N = 100 and $Dk = 512$. Then the ratio $R = 0.010004$

This means that the depth wise separable convolution network, in this example, performs 100 times lesser multiplications as compared to a standard constitutional neural network.

This implies that we can deploy faster convolution neural network models without losing much of the accuracy.

When Stride is 1 and when it is not 1, $Ratio(R)$ is defined the same. Therefore, based on this theoretical background, it is possible to create a more complex extended version of mobilenet in actual implementation.

4. Difference between Standard Convolution and Depthwise separable convolution.



Standard Convolution.

The main difference between MobileNet architecture and a traditional CNN instead of a single 3×3 convolution layer followed by the batch norm and ReLU. Mobile Nets split the convolution into a 3×3 depth-wise conv and a 1×1 pointwise conv, as shown in the figure.

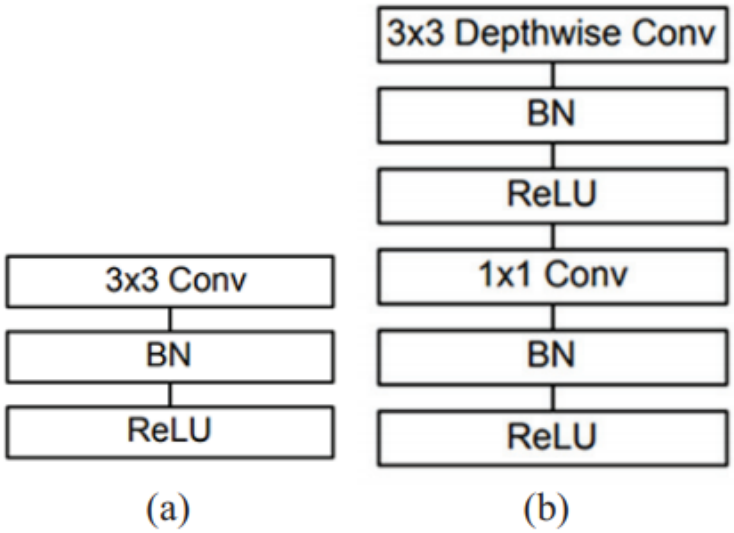


Fig. (a) Standard convolutional layer with batch normalization and ReLU. (b) Depth-wise separable convolution with depth-wise and pointwise layers followed by batch normalization and ReLU.

5. Parameters of MobileNet

Although the base MobileNet architecture is already small and computationally not very intensive, it has two different global hyperparameters to effectively reduce the computational cost further. One is the width multiplayer and another is the resolution wise multiplayer.

5.1 Width Multiplier: Thinner Models

For further reduction of computational cost, they introduced a simple parameter called Width Multiplier also refer as α.

For each layer, the width multiplier α will be multiplied with the input and the output channels(N and M) in order to narrow a network.

So the computational cost with width multiplier would become.

$$D_K \times D_K \times \alpha M \times D_F \times D_F + \alpha M \times \alpha N \times D_F \times D_F$$

Computational Cost: Depthwise separable convolution with width multiplier

Here α will vary from 0 to 1, with typical values of [1, 0.75, 0.5 and 0.25]. When $\alpha = 1$, called as baseline MobileNet and $\alpha < 1$, called as reduced MobileNet. Width Multiplier has the effect of reducing computational cost by α^2.

5.2 Resolution Multiplier: Reduced Representation

The second parameter to reduce the computational cost effectively. Also known as ρ.

For a given layer, the resolution multiplier ρ will be multiplied with the input feature map. Now we can express the computational cost by applying width multiplier and resolution multiplier as:

$$D_K \times D_K \times \alpha M \times \rho D_F \times \rho D_F + \alpha M \times \alpha N \times \rho D_F \times \rho D_F$$

Computational cost by applying width multiplier and resolution multiplier

6. MobileNet version 2

A little less than a year ago I wrote about MobileNets, a neural network architecture that runs very efficiently on mobile devices. Since then I've used MobileNet V1 with great success in a number of client projects, either as a basic image classifier or as a feature extractor that is part of a larger neural network.

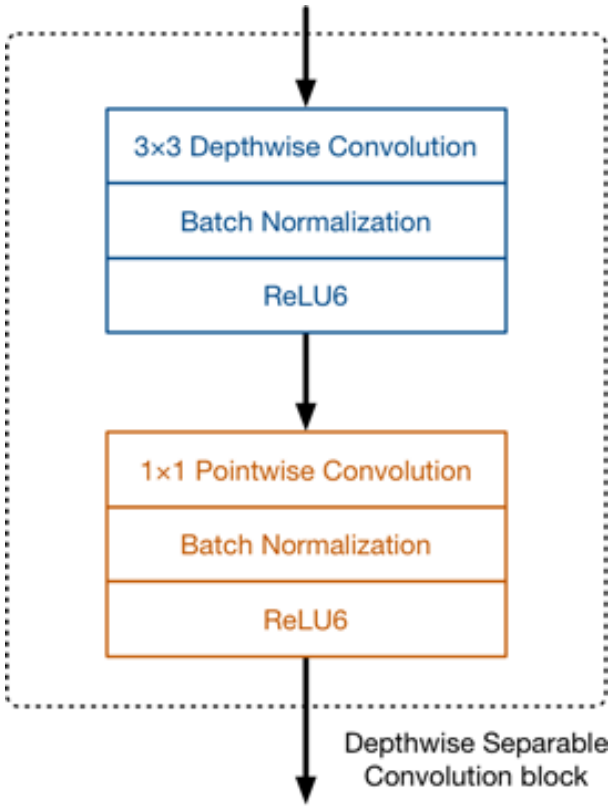
Recently researchers at Google announced MobileNet version 2. This is mostly a refinement of V1 that makes it even more efficient and powerful. Naturally, I made an implementation using Metal Performance Shaders and I can confirm it lives up to the promise.

In this article, I'll explain what's new in MobileNet V2.

6.1 Quick recap of version 1

The big idea behind MobileNet V1 is that convolutional layers, which are essential to computer vision tasks but are quite expensive to compute, can be replaced by so-called depthwise separable convolutions.

The job of the convolution layer is split into two subtasks: first there is a depthwise convolution layer that filters the input, followed by a 1×1 (or pointwise) convolution layer that combines these filtered values to create new features:



Together, the depthwise and pointwise convolutions form a "depthwise separable" convolution block. It does approximately the same thing as traditional convolution but is much faster.

The full architecture of MobileNet V1 consists of a regular 3×3 convolution as the very first layer, followed by 13 times the above building block("depthwise separable" convolution block).

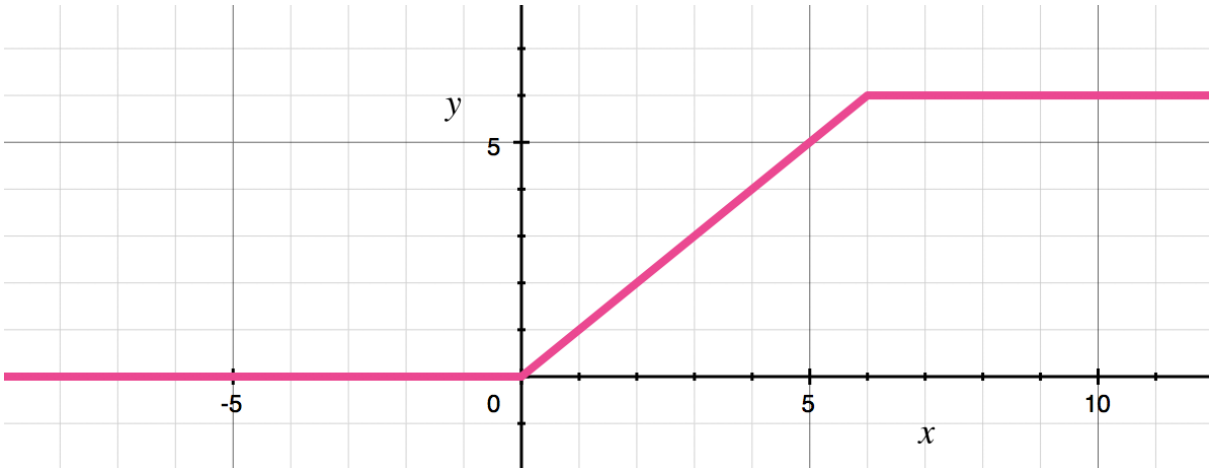
There are no pooling layers in between these depthwise separable blocks. Instead, some of the depthwise layers have a stride of 2 to reduce the spatial dimensions of the data. When that happens, the corresponding pointwise layer also doubles the number of output channels. If the input image is $224 \times 224 \times 3$ then the output of the network is a $7 \times 7 \times 1024$ feature map.

As is common in modern architectures, the convolution layers are followed by batch normalization. The activation function used by MobileNet is ReLU6. This is like the well-known ReLU but it prevents activations from becoming too big:

$$y = min(max(0, x), 6)$$

The authors of the MobileNet paper found that ReLU6 is more robust than regular ReLU when using low-precision computation. (I think "low-precision" here refers to fixed-point arithmetic and not so much the 16-bit floats used with Metal on iOS.)

It also makes the shape of the function look more like a sigmoid:



In a classifier based on MobileNet, there is typically a global average pooling layer at the very end, followed by a fully-connected classification layer or an equivalent 1×1 convolution, and a softmax.

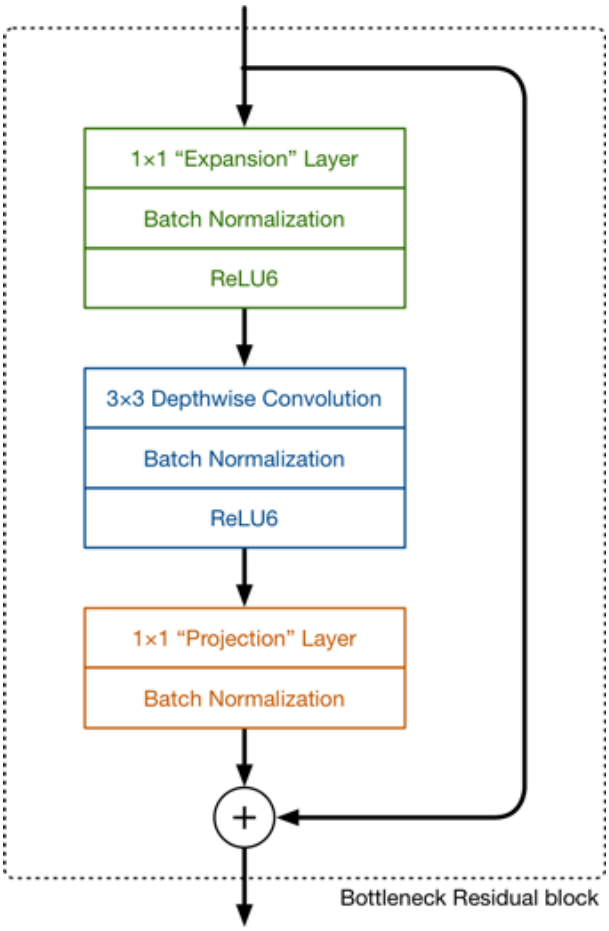
There is actually more than one MobileNet. It was designed to be a family of neural network architectures. There are several hyperparameters that let you play with different architecture trade-offs.

The most important of these hyperparameters is the depth multiplier, confusingly also known as the “width multiplier”. This changes how many channels are in each layer. Using a depth multiplier of 0.5 will halve the number of channels used in each layer, which cuts down the number of computations by a factor of 4 and the number of learnable parameters by a factor 3. It is therefore much faster than the full model but also less accurate.

Thanks to the innovation of depthwise separable convolutions, MobileNet has to do about 9 times less work than comparable neural nets with the same accuracy. This type of layer works so well that I’ve been able to get models with 200+ layers to run in real-time, even on an iPhone 6s.

6.2 The all new version 2

MobileNet V2 still uses depthwise separable convolutions, but its main building block now looks like this:



This time there are three convolutional layers in the block. The last two are the ones we already know: a depthwise convolution that filters the inputs, followed by a 1×1 pointwise convolution layer. However, this 1×1 layer now has a different job.

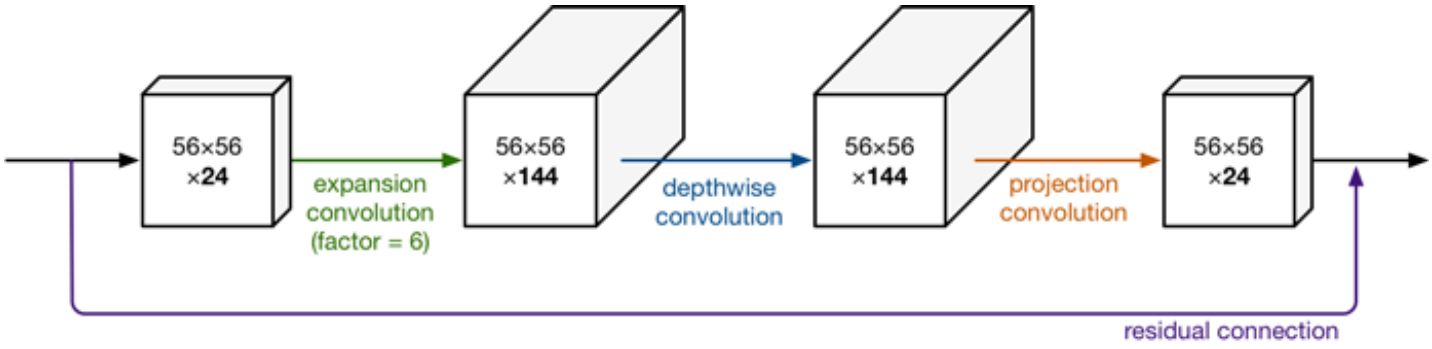
In V1 the pointwise convolution either kept the number of channels the same or doubled them. In V2 it does the opposite: it makes the number of channels smaller. This is why this layer is now known as the projection layer — it projects data with a high number of dimensions (channels) into a tensor with a much lower number of dimensions.

For example, the depthwise layer may work on a tensor with 144 channels, which the projection layer will then shrink down to only 24 channels. This kind of layer is also called a bottleneck layer because it reduces the amount of data that flows through the network. (This is where the "bottleneck residual block" gets its name from: the output of each block is a bottleneck.)

The first layer is the new kid in the block. This is also a 1×1 convolution. Its purpose is to expand the number of channels in the data before it goes into the depthwise convolution. Hence, this expansion layer always has more output channels than input channels — it pretty much does the opposite of the projection layer.

Exactly by how much the data gets expanded is given by the expansion factor. This is one of those hyperparameters for experimenting with different architecture tradeoffs. The default expansion factor is 6.

For example, if there is a tensor with 24 channels going into a block, the expansion layer first converts this into a new tensor with 24 * 6 = 144 channels. Next, the depthwise convolution applies its filters to that 144-channel tensor. And finally, the projection layer projects the 144 filtered channels back to a smaller number, say 24 again.



So the input and the output of the block are low-dimensional tensors, while the filtering step that happens inside block is done on a high-dimensional tensor.

The second new thing in MobileNet V2's building block is the residual connection. This works just like in ResNet and exists to help with the flow of gradients through the network. (The residual connection is only used when the number of channels going into the block is the same as the number of channels coming out of it, which is not always the case as every few blocks the output channels are increased.)

As usual, each layer has batch normalization and the activation function is ReLU6. However, the output of the projection layer does not have an activation function applied to it. Since this layer produces low-dimensional data, the authors of the paper found that using a non-linearity after this layer actually destroyed useful information.

NOTE: The pre-trained models from tensorflow/models only use batch normalization after the depthwise convolution layer, the 1×1 convolutions use bias instead. I'm not sure why that is the case but in practice it doesn't matter — for inference the batch norm operation gets folded into the convolution layer anyway.

The full MobileNet V2 architecture, then, consists of 17 of these building blocks in a row. This is followed by a regular 1×1 convolution, a global average pooling layer, and a classification layer. (Small detail: the very first block is slightly different, it uses a regular 3×3 convolution with 32 channels instead of the expansion layer.)

6.3 Motivation for these changes

Why did the authors of MobileNet V2 make these choices?

The idea behind V1 was to replace expensive convolutions with cheaper ones, even if it meant using more layers. That was a great success. The main changes in the V2 architecture are the residual connections and the expand/projection layers.

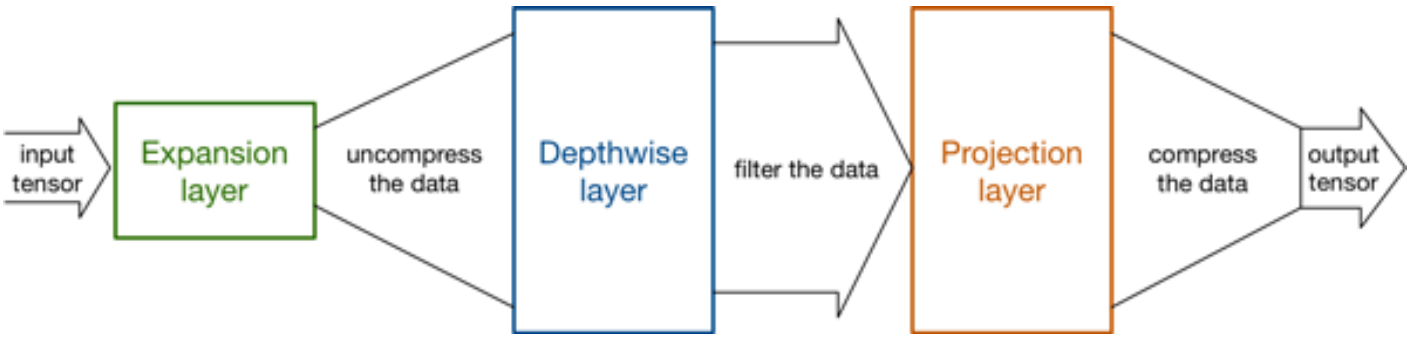
If we look at the data as it flows through the network, notice how the number of channels stays fairly small between the blocks:

As is usual for this kind of model, the number of channels is increased over time (and the spatial dimensions cut in half). But overall, the tensors remain relatively small, thanks to the bottleneck layers that make up the connections between the blocks. Compared to this, V1 lets its tensors become much larger (up to $7 \times 7 \times 1024$).

Using low-dimension tensors is the key to reducing the number of computations. After all, the smaller the tensor, the fewer multiplications the convolutional layers have to do.

However… only using low-dimensional tensors doesn't work very well. Applying a convolutional layer to filter a low-dimensional tensor won't be able to extract a whole lot of information. So to filter the data we ideally want to work with large tensors. MobileNet V2's block design gives us the best of both worlds.

Think of the low-dimensional data that flows between the blocks as being a compressed version of the real data. In order to run filters over this data, we need to uncompress it first. That's what happens inside each block:



The expansion layer acts as an decompressor (like unzip) that first restores the data to its full form, then the depthwise layer performs whatever filtering is important at this stage of the network, and finally the projection layer compresses the data to make it small again.

The trick that makes this all work, of course, is that the expansions and projections are done using convolutional layers with learnable parameters, and so the model is able to learn how to best (de)compress the data at each stage in the network.

6.4 Battle of the versions

Let's compare MobileNet V1 to V2, starting with the sizes of the models in terms of learned parameters and required amount of computation:

Version	MACs (millions)	Parameters (millions)
MobileNet V1	569	4.24
MobileNet V2	300	3.47

These numbers are taken from 1 and 2. They are for the model versions with a 1.0 depth multiplier. In this table, lower numbers are better.

"MACs" are multiply-accumulate operations. This measures how many calculations are needed to perform inference on a single 224×224 RGB image. (The larger the image, the more MACs are needed.)

From the number of MACs alone, V2 should be almost twice as fast as V1. However, it's not just about the number of calculations. On mobile devices, memory access is much slower than computation. But here V2 has the advantage too: it only has 80% of the parameter count that V1 has.

NOTE: I'm not entirely sure how they counted these parameters. My Metal version of the V1 model has 4,254,889 parameters and the V2 model has 3,510,505, so that's slightly more than what is reported above. It's possible they're not counting the batch normalization parameters as these typically get folded into a single set of biases for each layer.

I also measured the actual speed difference between the two models on a few devices, running inference on a sequence of 224×224 images. The following table shows the maximum FPS (frames-per-second) I was able to squeeze from these models:

Version	iPhone 7	iPhone X	iPad Pro 10.5
MobileNet V1	118	162	204
MobileNet V2	145	233	220

For optimal throughput I used a double-buffering approach where the next request is already being prepared (by the CPU) while the current one is still being processed (by the GPU). This way the CPU and GPU are never waiting for one another.

(Fun fact: for V2 it was actually worth doing triple buffering but for V1 that made no difference in speed. This shows that V2 is much more efficient.)

Having a fast model is great… but it's only useful if it actually computes the right thing. So exactly how good are these models?

Version	Top-1 Accuracy	Top-5 Accuracy
MobileNet V1	70.9	89.9
MobileNet V2	71.8	91.0

The reported top-1 and top-5 accuracy are on the ImageNet classification dataset. (The source for these numbers claims they're from the test set but looking at the code it appears to be the 50,000-image validation set.)

It can be a bit misleading to compare accuracy numbers between models, since you need to understand exactly how the model is evaluated. To get the above numbers, the central region of the image was cropped to an area containing 87.5% of the original image, and then that crop was resized to 224×224 pixels. Only a single crop was used per image.

NOTE: Naturally, I did verify that my Metal version of MobileNet V2 comes up with the same answers as the TensorFlow reference model, but I have not tried it on the ImageNet validation set yet. It will be interesting to see if the Metal version gets the same score. :–)

Conclusion: In all of these metrics, V2 scores better than V1. I'm especially pleased that it uses fewer parameters because that's where most of the speed gains come from on mobile devices.

6.5 More than just classification

While the classification score on the ImageNet dataset is useful to know, in practice you'll probably never use the pre-trained ImageNet classifier in your apps.

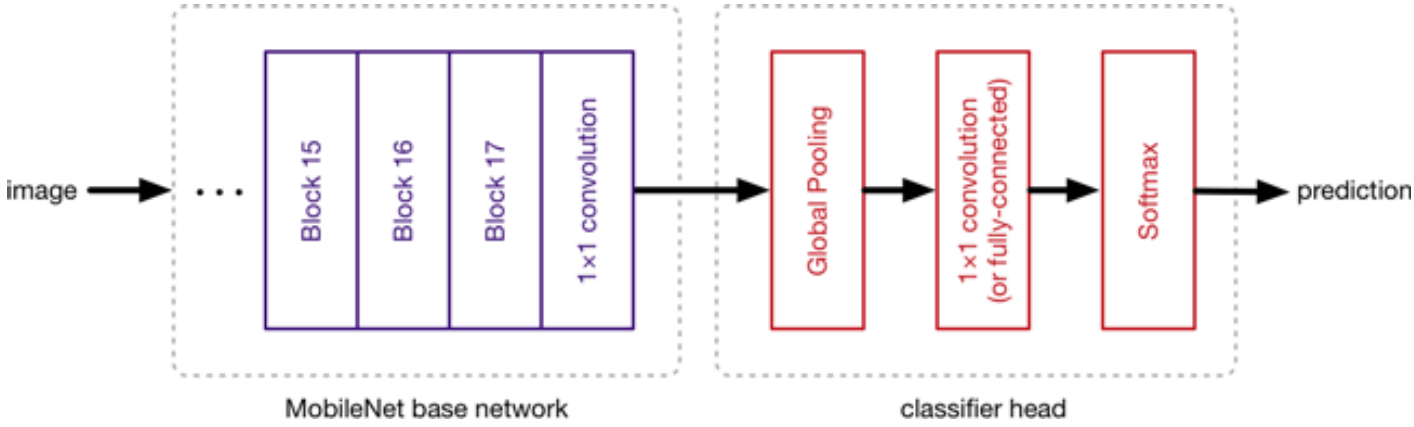
You'll either re-train the classifier on your own dataset, or use the base network as a feature extractor for something like object detection (finding multiple objects in the same image) or image segmentation (making a class prediction for every pixel instead of a single prediction for the whole image) or some other exciting computer vision task.

Re: object detection, I've written about YOLO before. Since then, SSD (Single Shot Detector) has been making a name for itself. It uses many of the same ideas as YOLO but works even better — the main difference is that YOLO makes predictions for only a single feature map while SSD combines predictions across multiple feature maps at different sizes.

The problem with YOLO on mobile is that, while the actual detection portion of the neural network is simple and fast, the feature extractor (Darknet-19) uses regular convolutional layers. Running YOLO on an iPhone only gets you about 10 – 15 FPS. YOLO would be much faster if it was running on top of MobileNet instead of the Darknet feature extractor.

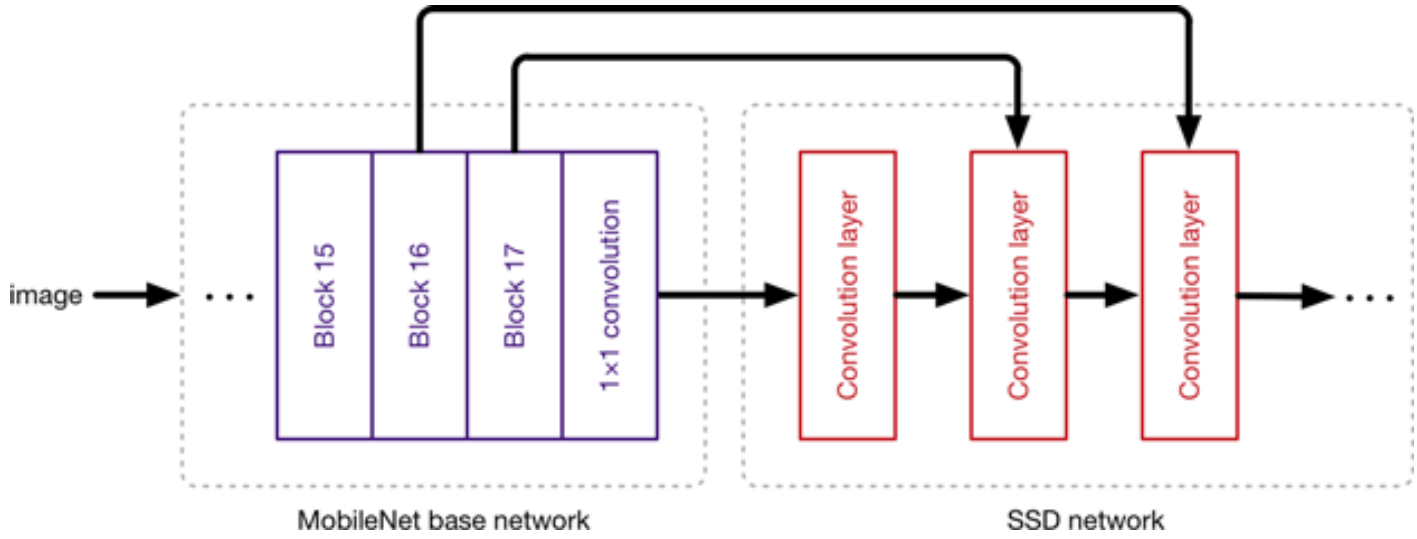
SSD is designed to be independent of the base network, and so it can run on top of pretty much anything, including MobileNet. Even better, MobileNet+SSD uses a variant called SSDLite that uses depthwise separable layers instead of regular convolutions for the object detection portion of the network. With SSDLite on top of MobileNet, you can easily get truly real-time results (i.e. 30 FPS or more).

How does this work? When doing classification, the last layers of the neural network look like this:



The output of the base network is typically a 7×7 pixel image. The classifier first uses a global pooling layer to reduce the size from 7×7 to 1×1 pixel — essentially taking an ensemble of 49 different predictors — followed by a classification layer and a softmax.

To use something like SSDLite with MobileNet, the last layers will look like this instead:



Not only do we take the output of the last base network layer but also the outputs of several previous layers, and we feed these outputs into the SSD layers. The job of the MobileNet layers is to convert the pixels from the input image into features that describe the contents of the image, and pass these along to the other layers. Hence, MobileNet is used here as a feature extractor for a second neural network.

In the case of classification, we're interested in the features that describe high-level concepts, such as "there is a face" and "there is fur", which the classifier layer then can use to draw a conclusion — "this image contains a cat".

In the case of object detection with SSD, we want to know not just these high-level features but also lower-level ones, which is why we also read from the previous layers. Since object detection is more complicated than classification, SSD adds many additional convolutional layers on top of the base network. So it's important to have a feature extractor that is fast — and that's exactly what MobileNet V2 is.

The MobileNet V2 paper also shows that it's possible to run an advanced semantic segmentation model such as DeepLabv3 on top of MobileNet-extracted features.

마지막 편집일시 : 2022년 11월 11일 5:41 오후

- **이전글** : 09.03 Understanding of Inception - EN
- **다음글** : 09.05 Understanding of ResNet - EN

TOP