CNN Architectures

Table of Contents

[AlexNet 3](#_Toc138881874)

[ZFNet 5](#_Toc138881875)

[VGG 6](#_Toc138881876)

[GoogLeNet 8](#_Toc138881877)

[Stem Network 8](#_Toc138881878)

[Inception Module 9](#_Toc138881879)

[Global Average Pooling 10](#_Toc138881880)

[Auxiliary Classifiers 11](#_Toc138881881)

[Residual Networks 11](#_Toc138881882)

[Residual Blocks 12](#_Toc138881883)

[Network Architecture 13](#_Toc138881884)

[Bottleneck Blocks 14](#_Toc138881885)

[Pre-Activation Blocks 15](#_Toc138881886)

[Comparing Complexity 16](#_Toc138881887)

[ResNeXt 16](#_Toc138881888)

[Squeeze and Excitation 17](#_Toc138881889)

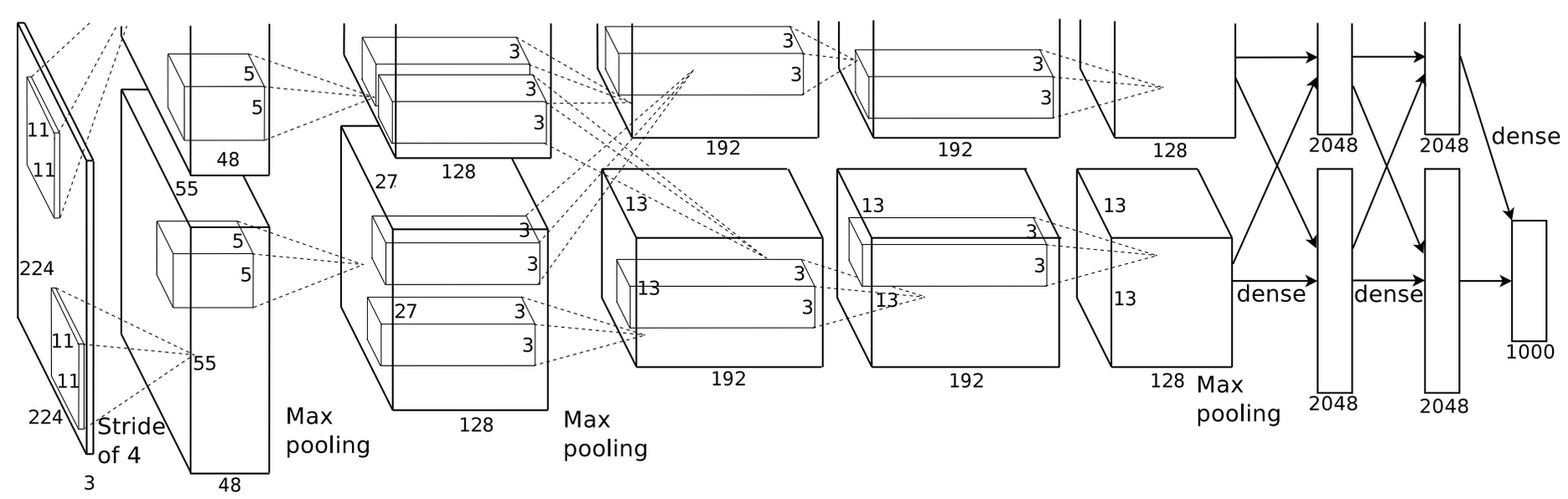
[Densely Connected Neural Networks 18](#_Toc138881890)

[MobileNets 19](#_Toc138881891)

[Neural Architecture Search 19](#_Toc138881892)

The components of CNN that we have previously studied have been used in several different ways to create successful architectures. We will be looking into a few of these architectures now.

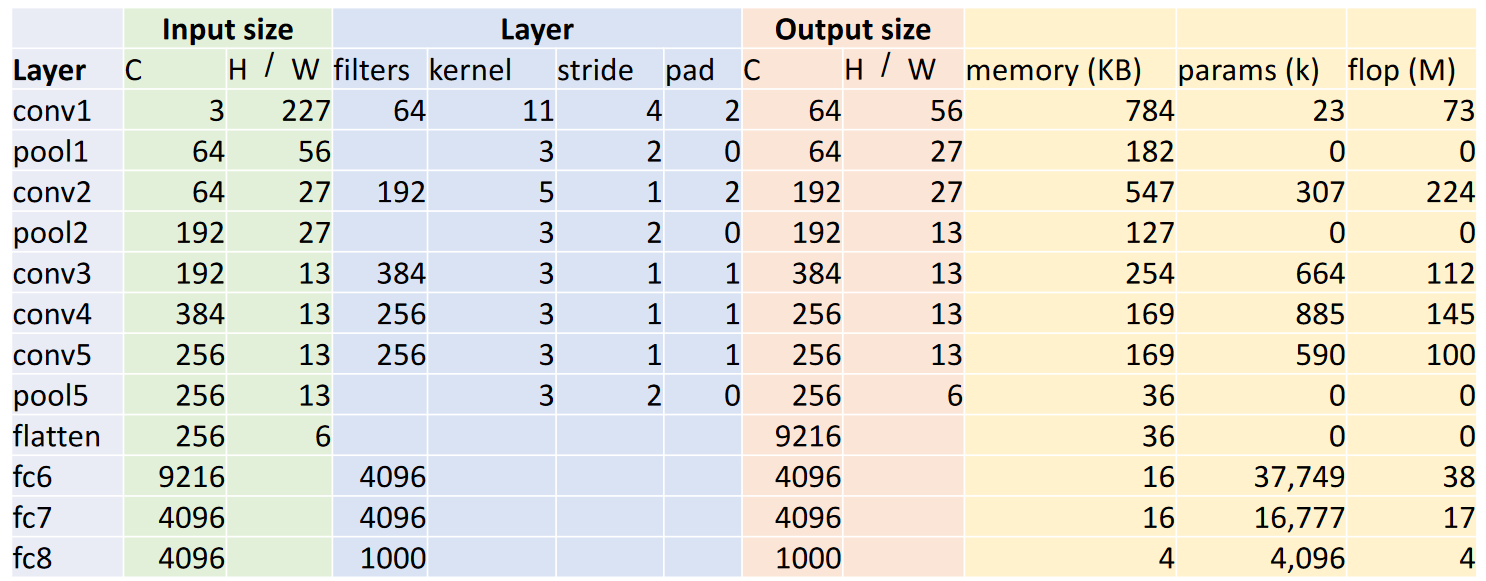
## AlexNet



The **AlexNet** architecture consisted of 5 convolution layers, max-pooling layers, 3 fully-connected layers and ReLU activation functions. We only count the convolution and fully-connected layers, so the AlexNet architecture has a total of 8 layers.

The architecture used something called **local normalization response**, which is comparable to batch normalization. However, batch normalization was not invented at the time.

Additionally, it was trained on two GTX580 GPUs, which only had 3GB of memory each. The model had to be split over the two GPUs to train it.



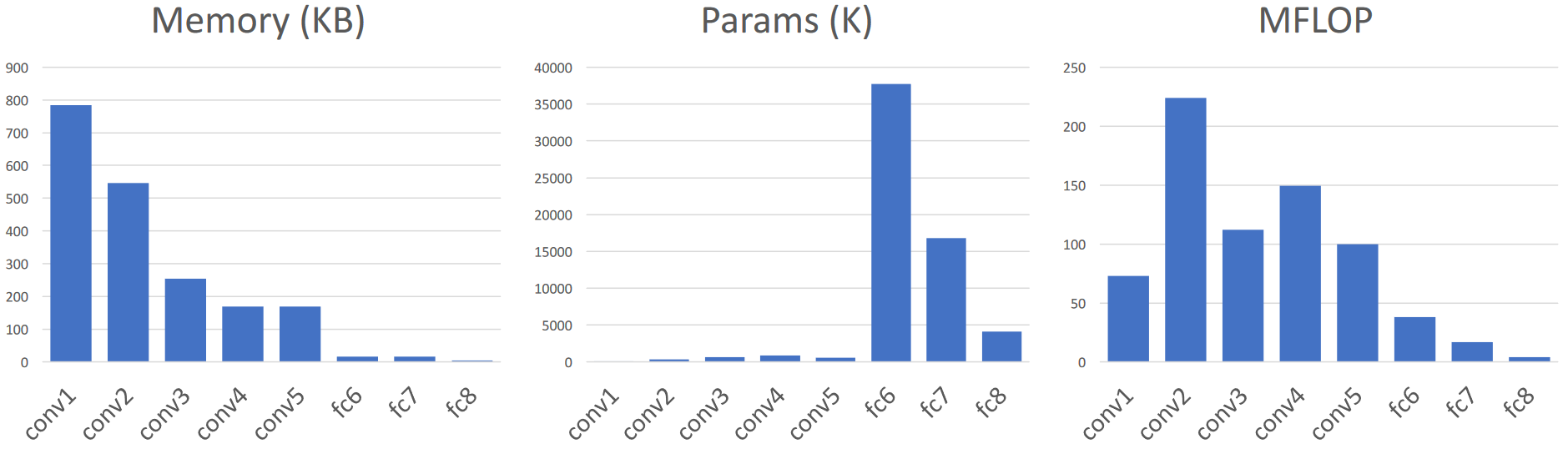
The table above shows some information about each of the layers in AlexNet. The calculations for the output sizes have been discussed previously, but the memory, parameter and flop calculations are being discussed again.

Memory depends on the number of output elements. For the first layer, which is a convolution layer, there are elements. Each element takes bytes of memory giving us kilobytes.

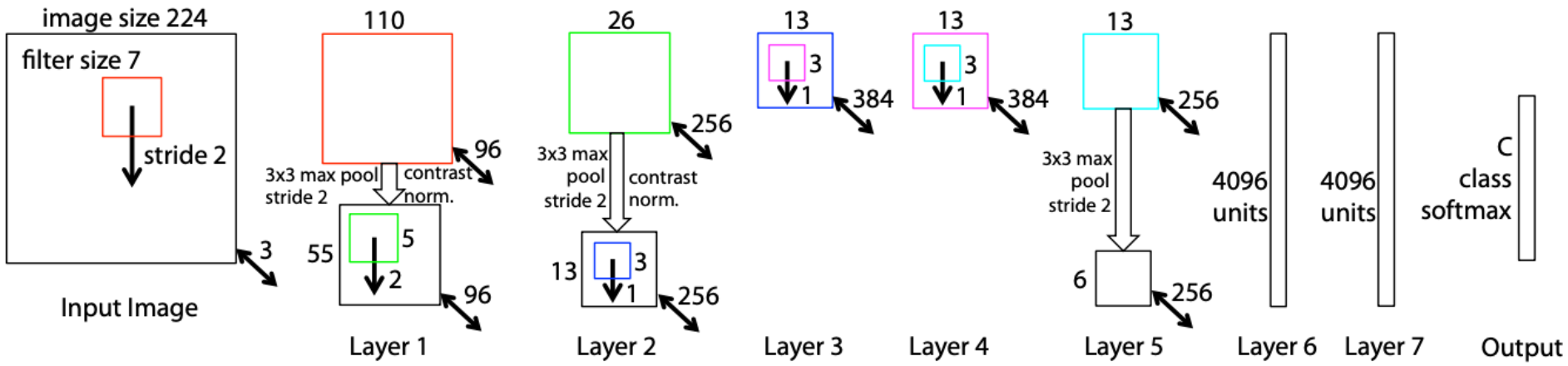
The number of parameters depends on the filter size as well as the number of input and output channels. For the first convolution layer, this is given by . For the first fully connected layer, this is given by .

The number of floating point operations depends on how many operations each output element requires and how many output elements there are. For the first convolution layer, there are outputs, and each output requires operations. This gives flops. For the first pooling layer, there are outputs with each output requiring operations, giving us flops. For the first fully connected layer, there are outputs, each undergoing an operation with one of the inputs. This gives flops.

Unfortunately, all of this information does not teach us anything about the structure of the layers in AlexNet. The structure was found by trial and error. There is however, a trend regarding the memory usage, the number of parameters and the flops. Most of the memory usage is in the early convolution layers. Most of the parameters are in the fully connected layers and most of the floating point operations occur in the convolution layers.



## ZFNet



The **ZFNet** is an extended version of the AlexNet, so it also has 8 layers. The first convolution was changed from 11x11 with a stride of 4 to 7x7 with a stride of 2 and the third, fourth and fifth convolutions used 512, 1024 and 512 filters instead of 384, 384 and 256 filters respectively. Again, this architecture was found by trial and error.

Increasing the convolution size results in a larger receptive field at the cost of a higher computational cost. The entire architecture is also, simply put, bigger. The fact that it achieved a better accuracy suggests that bigger networks work better.

## VGG

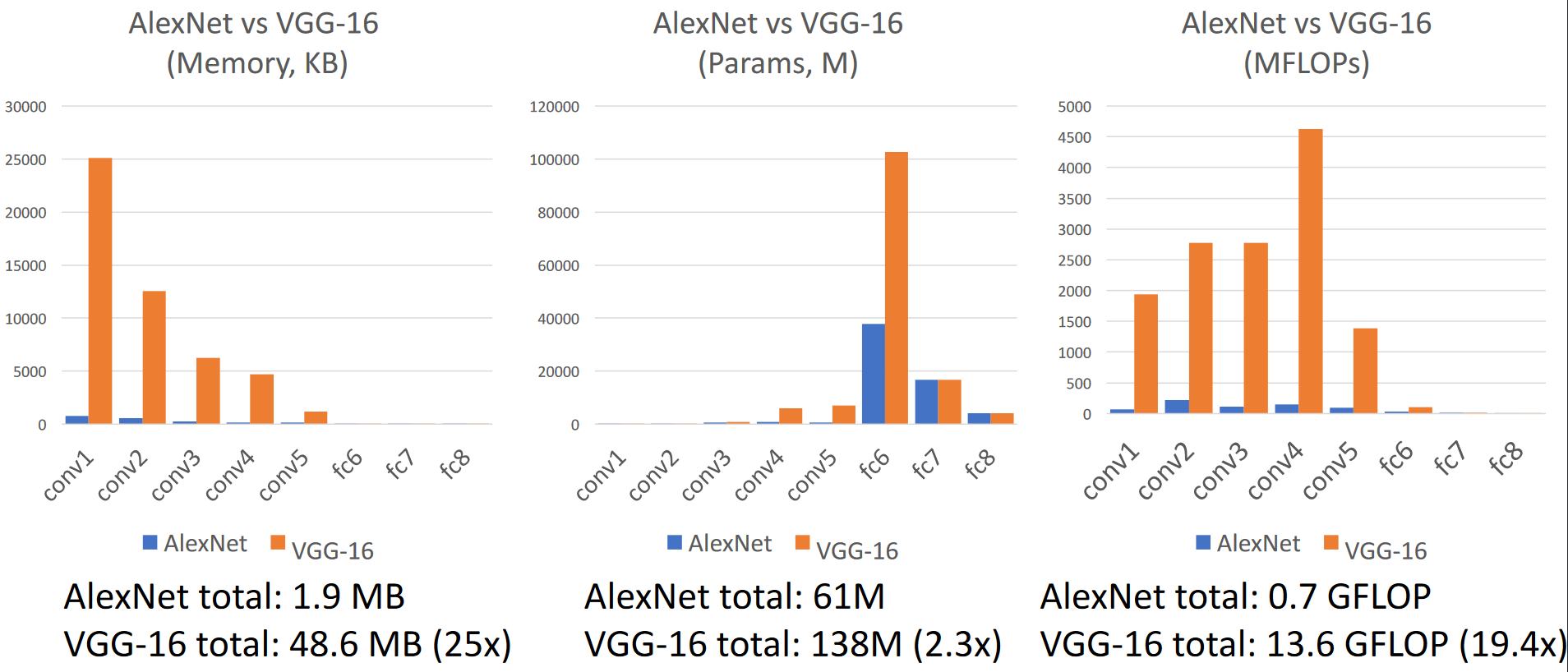


The **VGG Networks** were created with the intention of making a deeper network with a regular design. As such, all of the convolutions are 3x3 with a stride of 1 and a padding of 1, all the max pools are 2x2 with a stride of 2 and after each pooling operation, the number of channels is doubled.

There are five stages to the VGG networks. The first three stages have two convolution layers followed by a pooling layer. The fourth and fifth stages have three convolution layers followed by a pooling layer for VGG-16 and four convolution layers followed by a pooling layer for VGG-19.

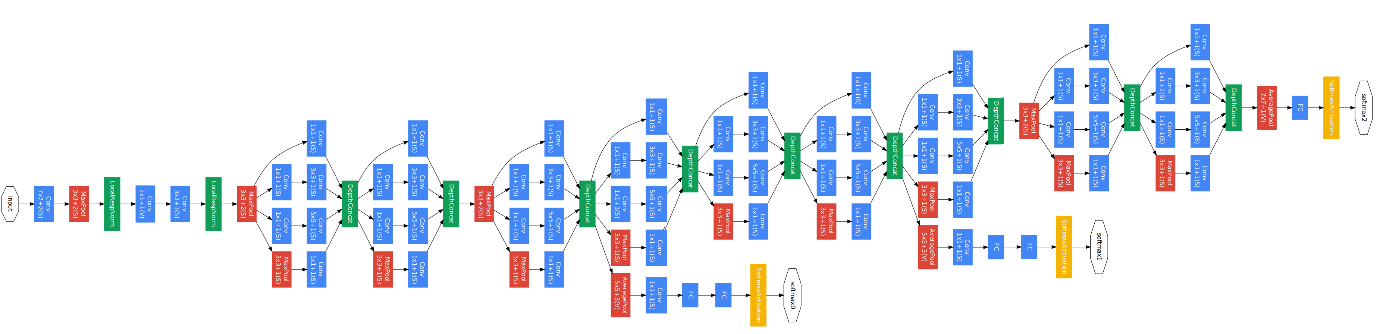
Unlike AlexNet, there are design decisions we can learn from the VGG network. Instead of using 5x5 convolutions, 3x3 convolutions have been used. Using 2 3x3 convolutions gives us the **same receptive field** as a single 5x5 convolution, but has fewer parameters ( compared to , where is the number of channels in the input as well as the output) and fewer flops ( compared to where and are the height and width of the filters respectively).

Next, consider the decisions to use only 2x2 pooling layers with a stride of 2 and doubling the number of channels after each pooling layer. This means that if the stage before the pooling layer has channels and a spatial resolution of , the stage after the pooling layer will have channels and a spatial resolution of . Due to this change, the number of floating point operations remains intact (), the number of parameters increases four times () and the memory consumption halves (). This ensures that each convolutional layer has the same computational cost (due to the same number of flops).



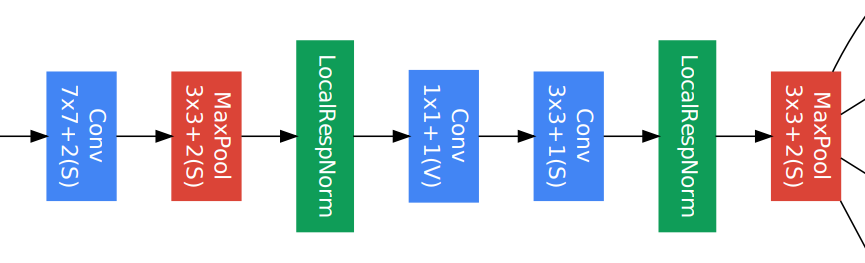
The graphs above make one thing obvious. The VGG networks are absurdly large compared to AlexNet. Again, we find that **deeper networks** give us better results.

## GoogLeNet

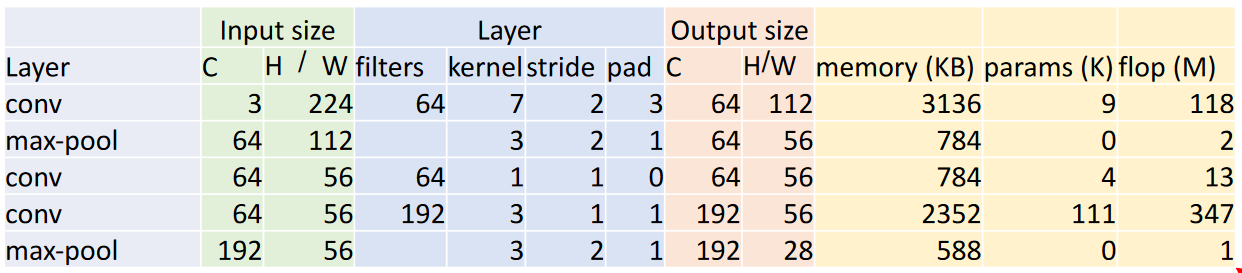


The **GoogLeNet** architecture had a focus on **efficiency** without losing the ability to perform well. As can be seen, the network is huge and has several parts, each of which we will discuss in turn.

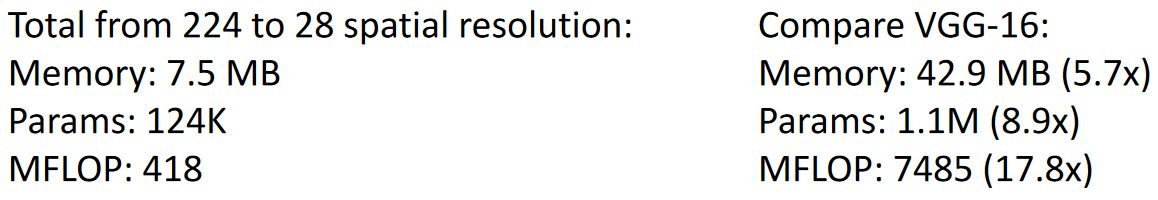
### Stem Network



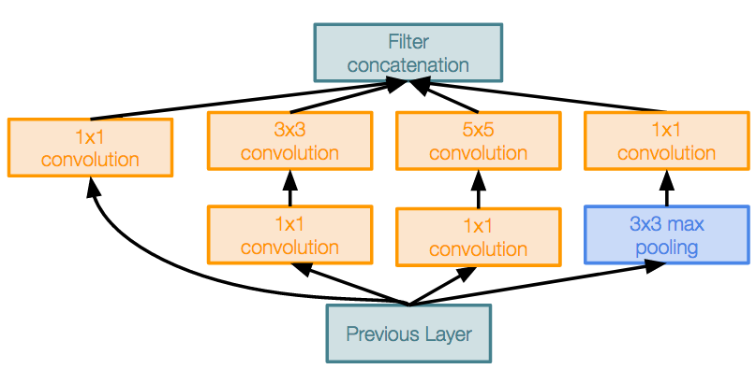
The architecture starts with a **stem network**. Most of the computation in VGG-16 was towards the start of the network, so the job of the stem network is to very aggressively **downsample** the input so as the reduce the computation at this stage.



The reduction form 224x224 images to 28x28 images requires 7.5 MB of memory, has 124K parameters and has 418 MFLOPs. This is almost nothing compared to VGG-16.



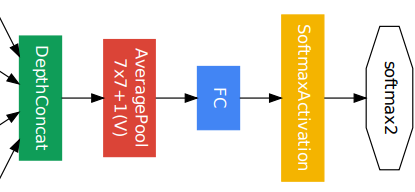
### Inception Module



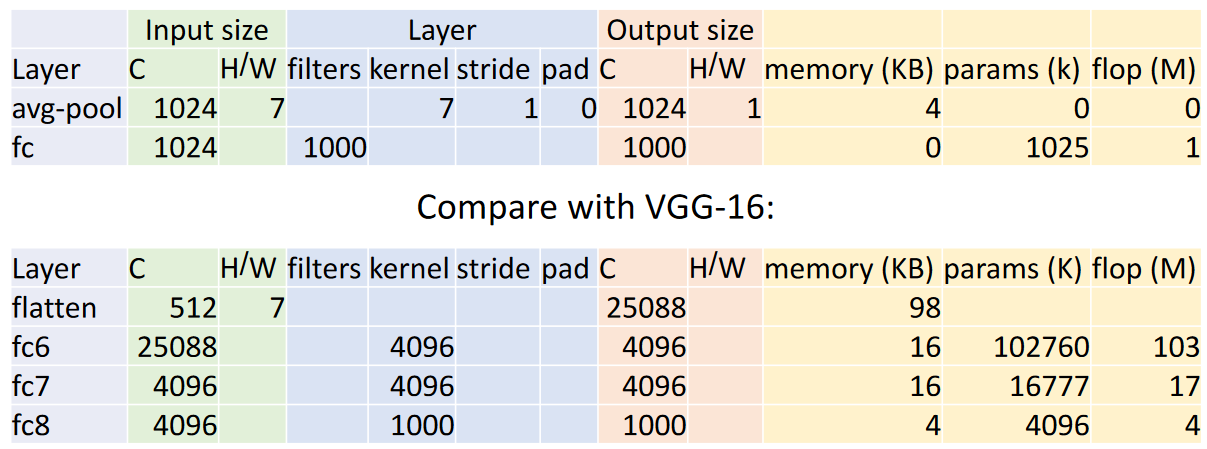
The next innovation of the architecture is the **inception module**. This is a local structure that is repeated throughout the network. This module was also the first to use **parallel branches** of computation. Instead of picking a kernel size, they basically used all of the kernel sizes that are commonly used in parallel.

They also used 1x1 convolutions frequently. These convolutions act as **bottlenecks** to reduce the channel dimension before expensive convolutions. The same idea is also used later in residual networks.

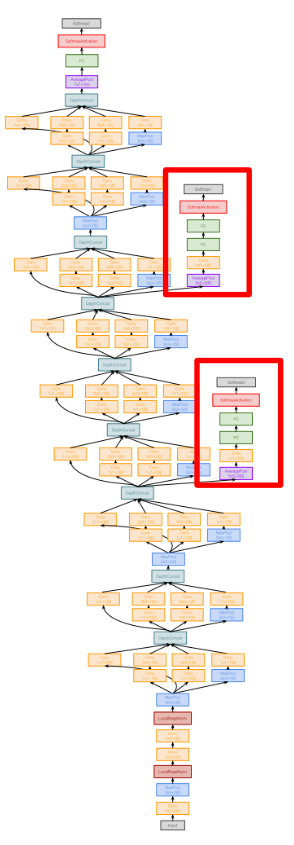
### Global Average Pooling



At the end of the network, the architecture uses something called **global average pooling**. Instead of flattening the multi-dimensional data like previous networks, the use an average pooling operation to do this. In VGG-16, we saw that the majority of the parameters of the model came from the fully-connected layers towards the end. By using global average pooling, these parameters are avoided.



### Auxiliary Classifiers

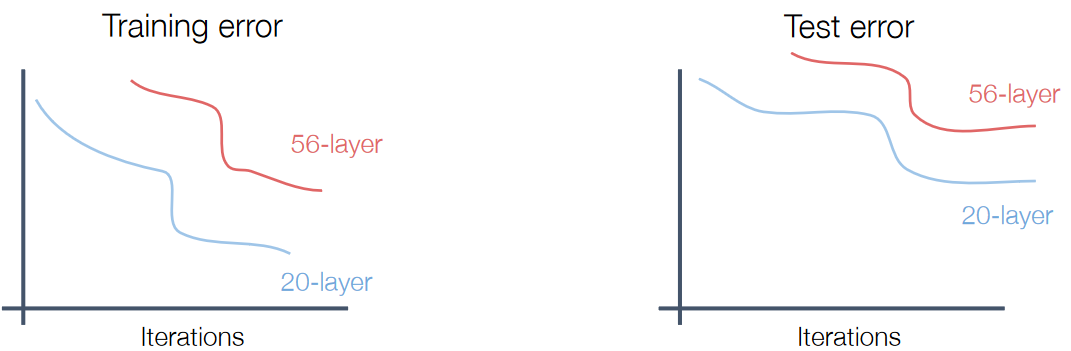


Due to the depth of the network, using the **loss** from the end of the network to update gradients did not work very well. Instead, there are **auxiliary classifiers** at intermediate points which also try to classify the image. The loss calculated from these are also used to update the gradients. The invention of **batch normalization** would later remove the need for these.

## Residual Networks

The invention of **batch normalization** led to very deep networks becoming easy to train. However, this led to an interesting discover. It was seen that very deep models, which we have so far expected to perform better than shallower ones, were actually performing worse.

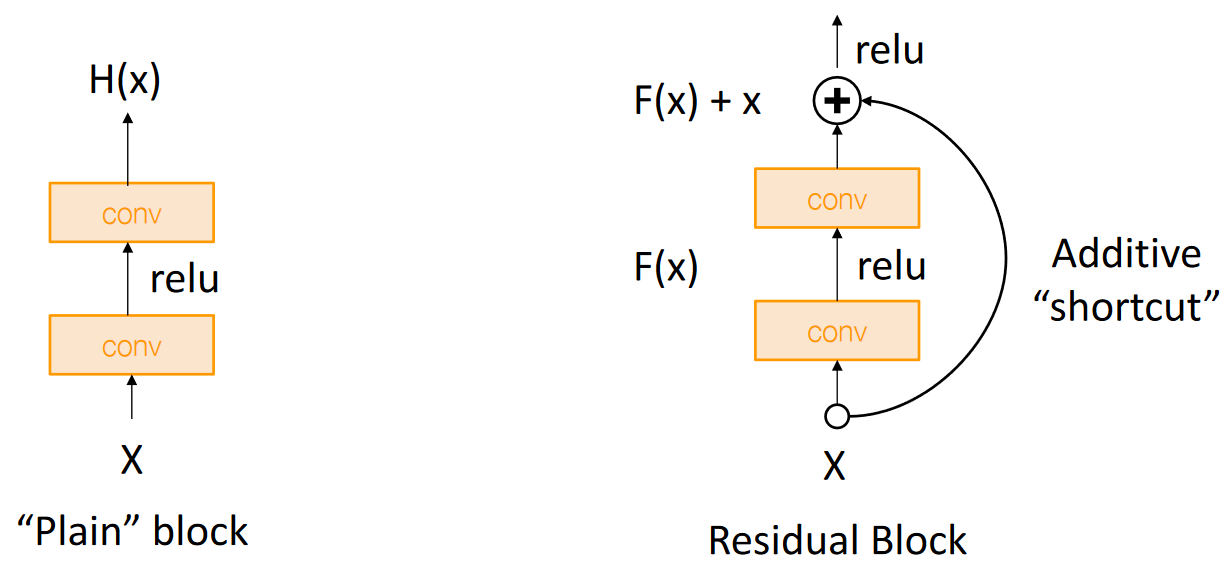
The initial guess as to why this was happening was that the models were simply too large for the amount of data at hand and were thus overfitting. This would explain the high test error rate. However, it was seen that the training error was also high, which indicated that the model was actually **underfitting**. This was absurd.



Ideally, a larger network should be able to **emulate** a shallower one. If it is actually best for the model to stop at the 20th layer for example, the rest of the layers can just be set to **identity** so as to ensure they have no effect. Unfortunately, it seemed that the deeper networks were unable to properly optimize themselves and do this.

### Residual Blocks

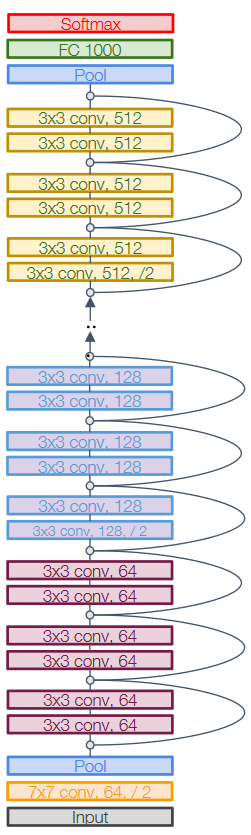
To make it easier to learn identity functions on unused layers, the concept of **residual blocks** was introduced. This essentially added a **shortcut** from one layer to another so that the model could learn to **skip** the layers in between if it needed to.



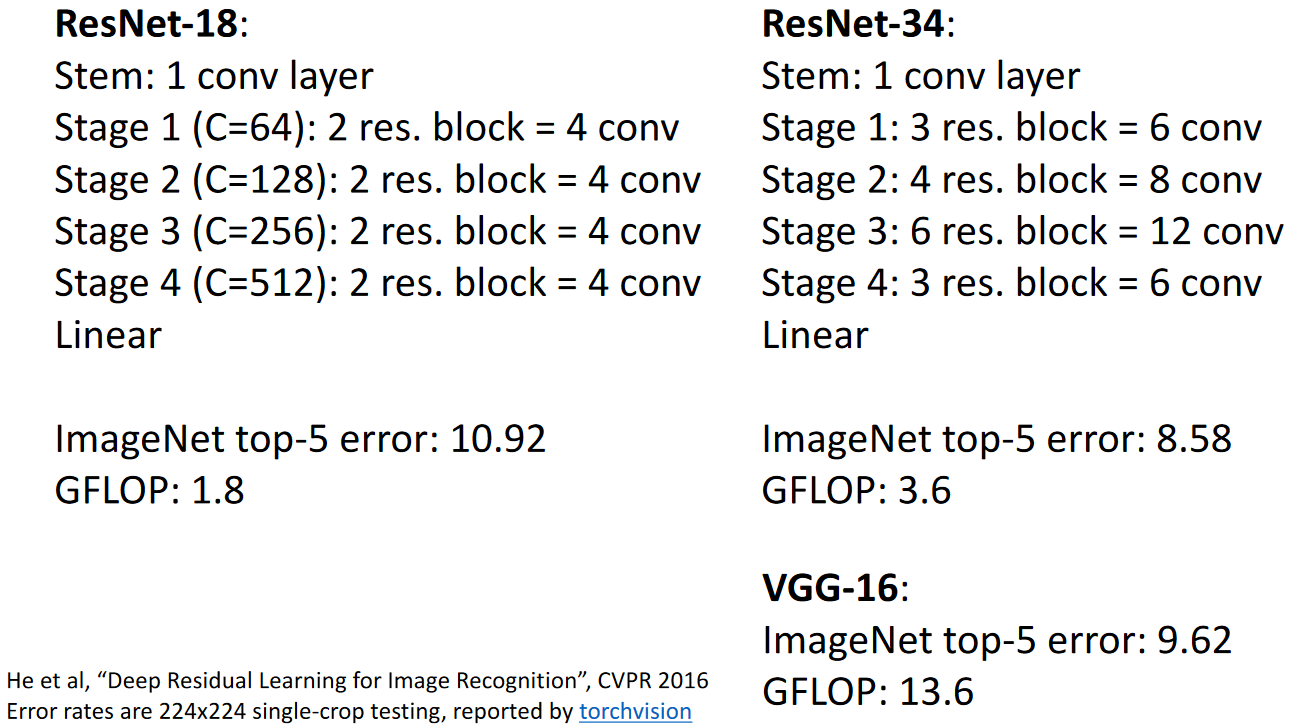
This works well due to how backpropagation behaves for **add gates**. At an add gate which combines the output from two inputs, the gradient is **copied** to both inputs. This means that the earlier layer is able to **directly receive** the gradient from the layer layers without having to go through the layers in between.

### Network Architecture

The entire residual network is just a stack of residual blocks. The original architecture uses ideas from both VGG and GoogLetNet, dividing the network into stages which each halve the spatial resolution and double the number of channels, using 3x3 convolutions exclusively, aggressively downsampling in the stem and using global average pooling at the end.

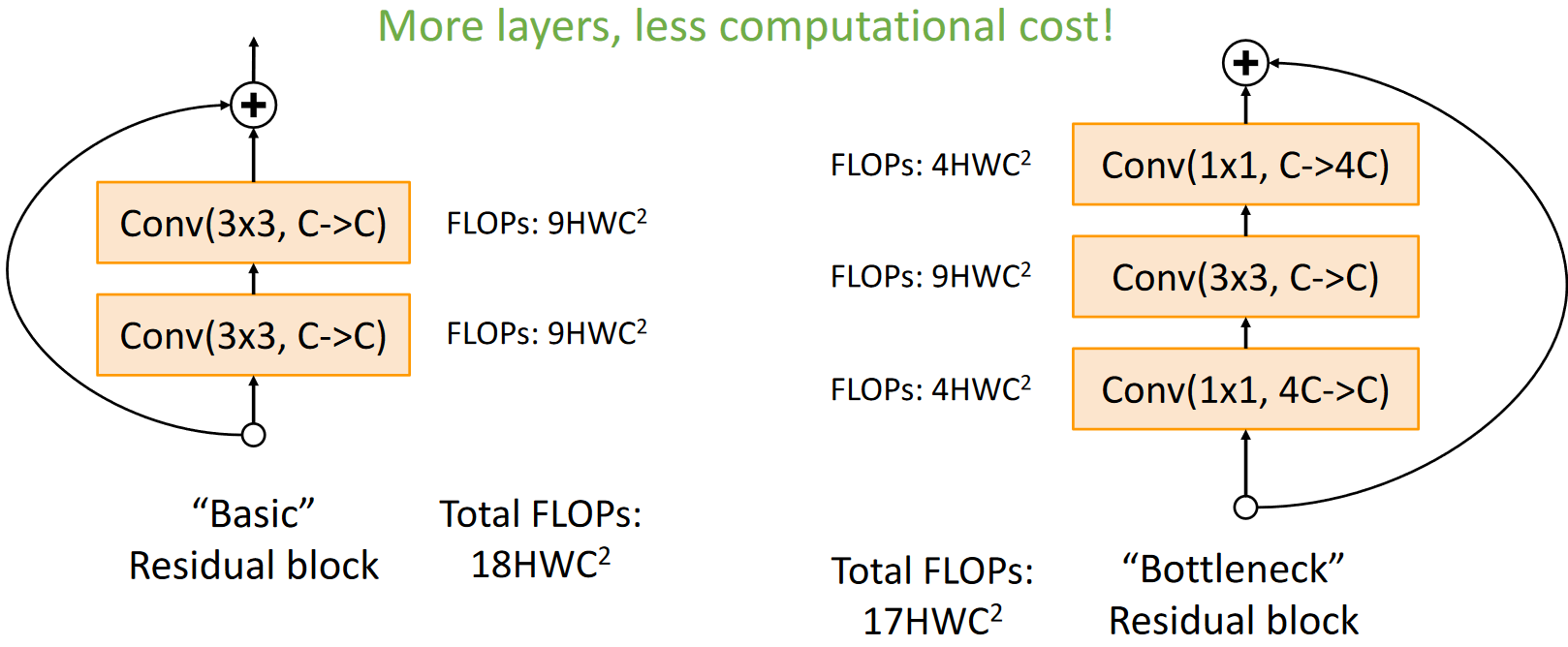


There are several variants of the residual network, each of which used a different number of residual blocks and convolutions per stage. Overall, these architectures achieved similar accuracy to VGG with only a fraction of the computation cost.



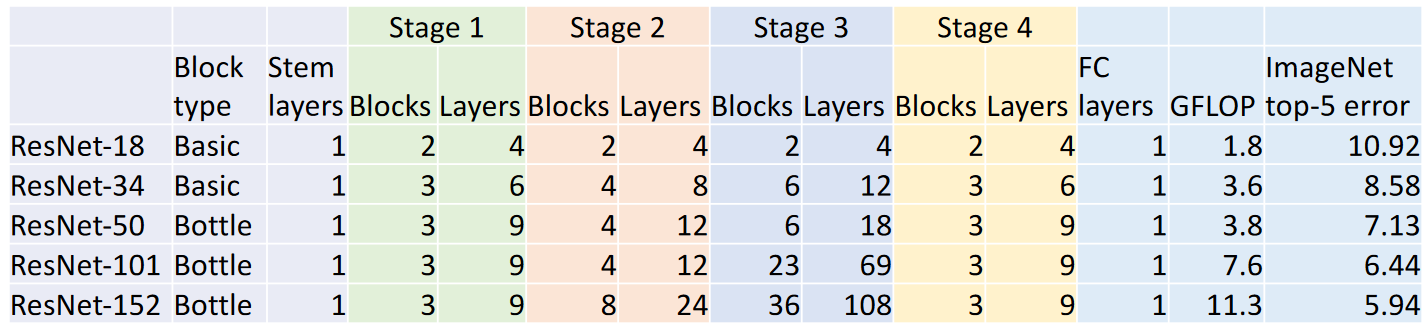
### Bottleneck Blocks

Deeper residual networks used a modified version of the block design called a **bottleneck block** which has a lower computational cost.

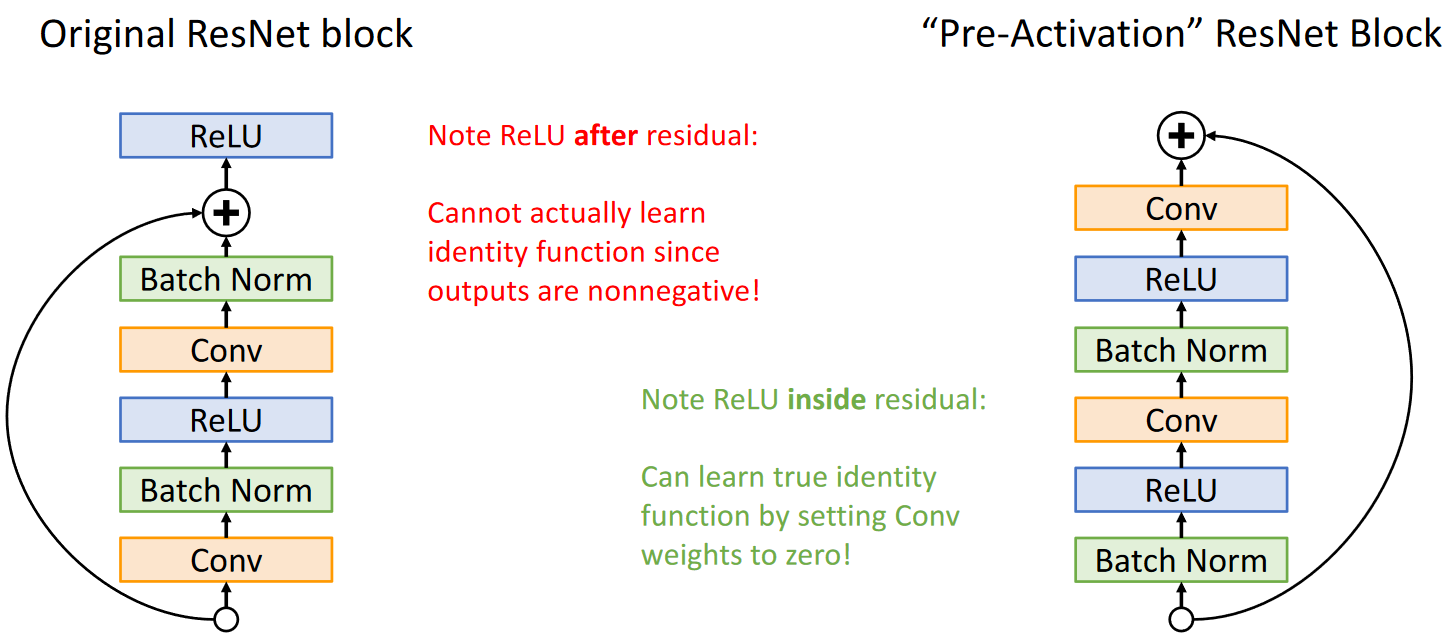


The bottleneck blocks also introduce **non-linearity**, with the intuition being that this will help the model learn more complex decisions boundaries.

These blocks have been used in deeper versions of Res-Net to achieve better error rates without a huge increase in computation cost.

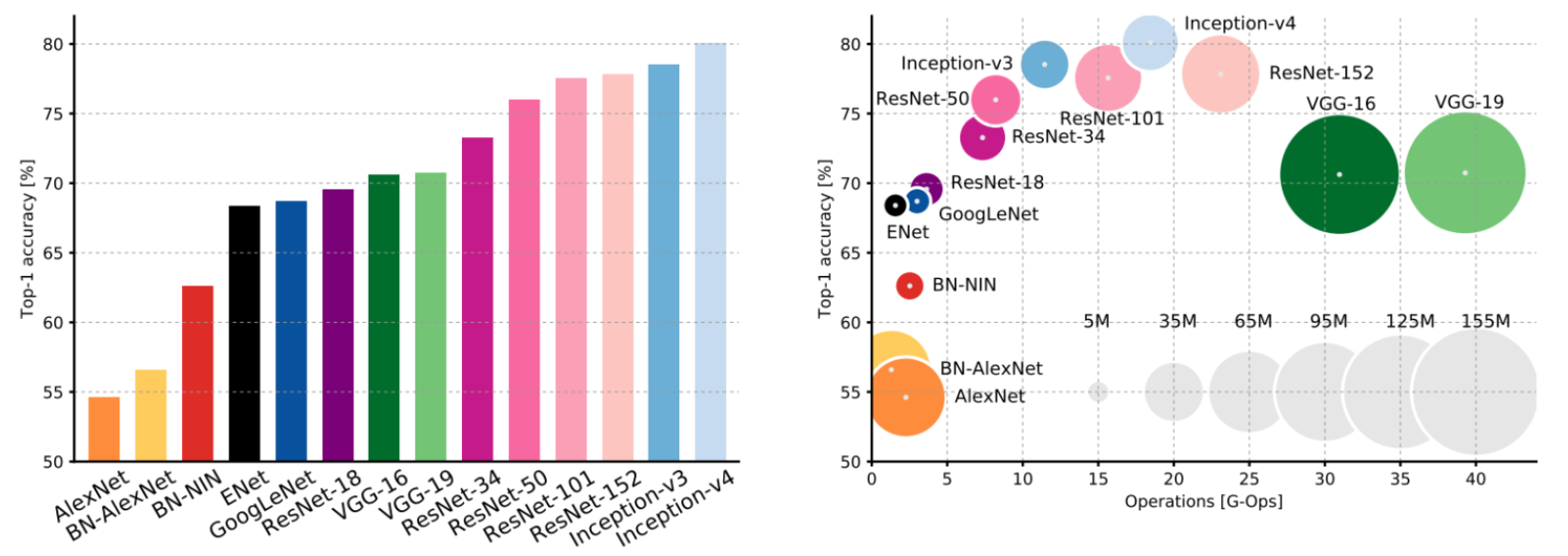


### Pre-Activation Blocks

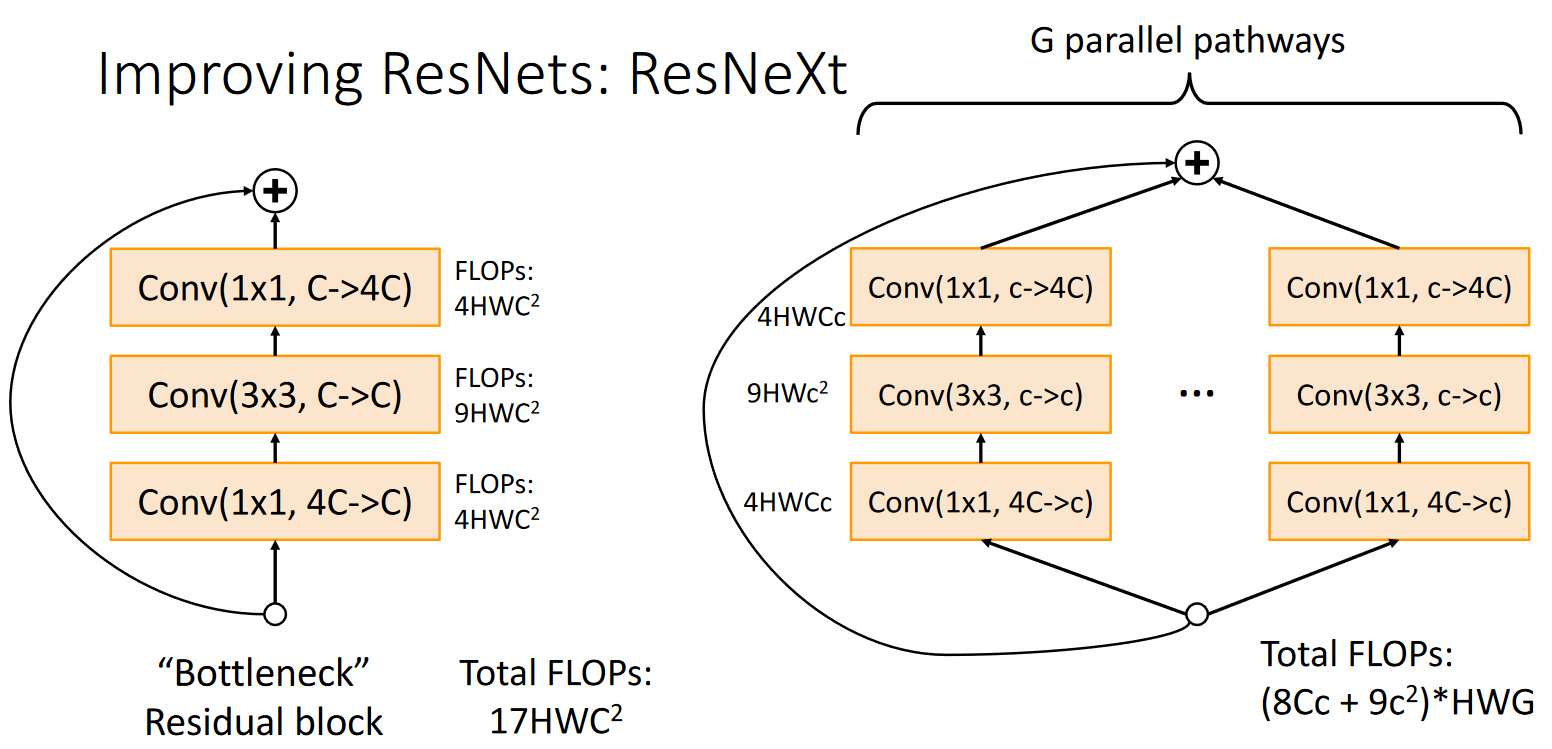


The original block design was modified further by changing the point at which the addition operation takes place. This led to a slight improvement in the error rates but is not actually used.

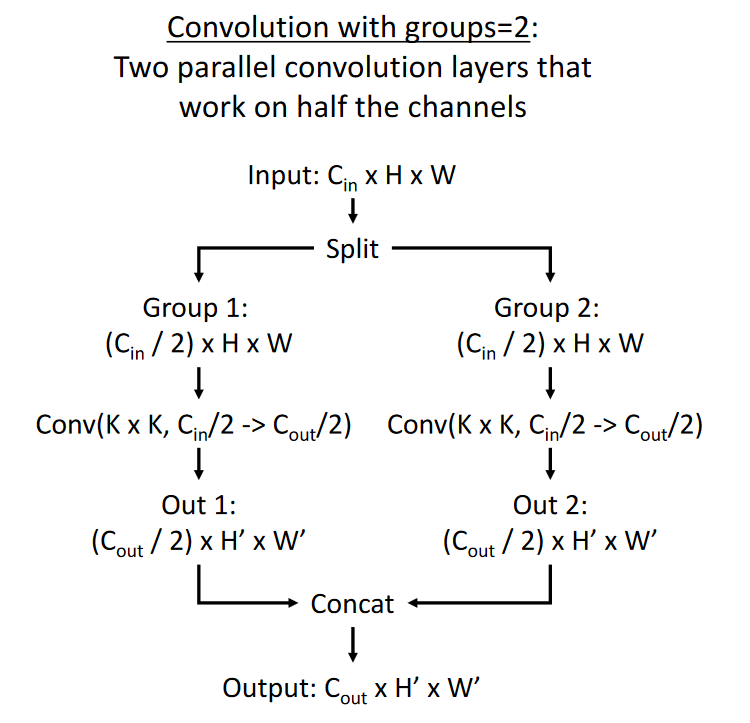
## Comparing Complexity



## ResNeXt



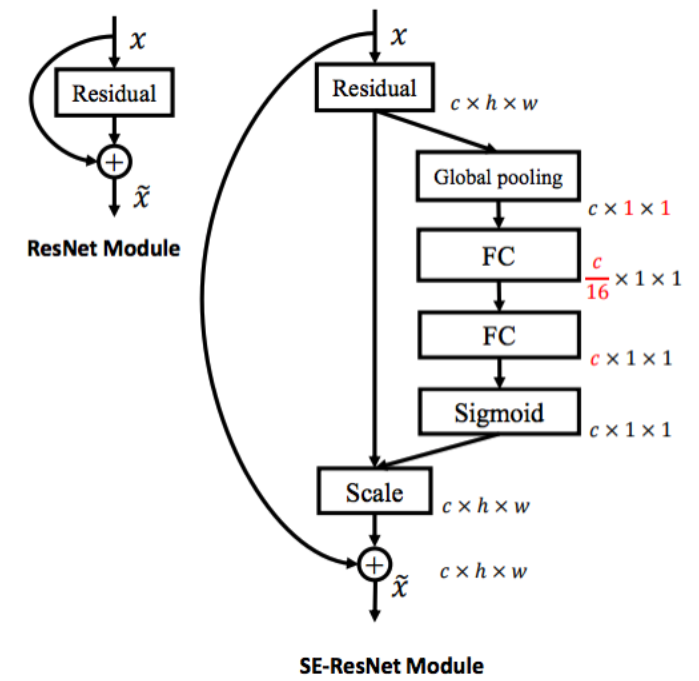
The idea behind **ResNeXt** networks is to have multiple bottleneck residual blocks in **parallel**. Each of the blocks where changed to use instead of channels, where , giving a total of . Interestingly enough, if , the computational cost does not change. The idea of parallel pathways can be implemented using a concept called **grouped convolution**.



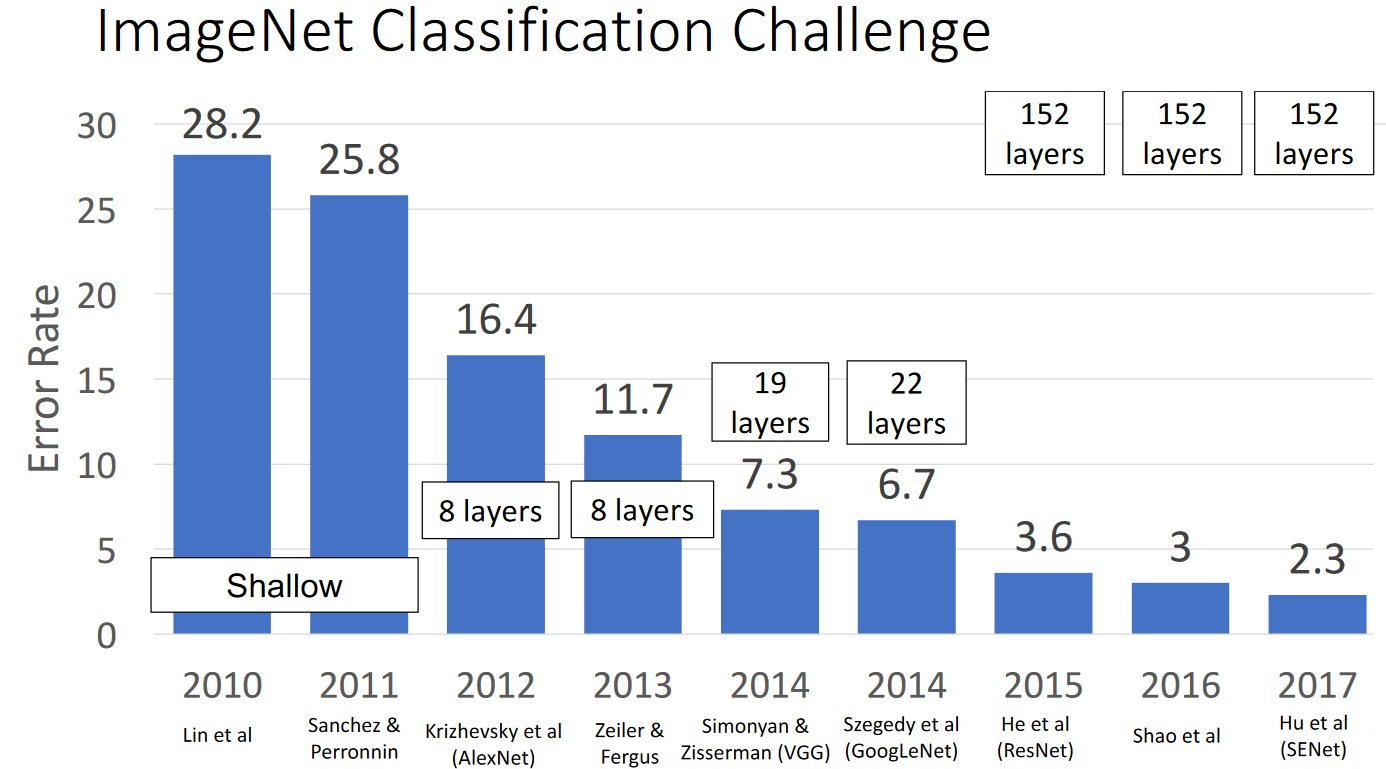
Increasing the number of parallel pathways without increasing the computational cost results in increased performance.

## Squeeze and Excitation

The final modification that won the ImageNet classification challenge before it was closed is called the **Squeeze and Excitation Network**.

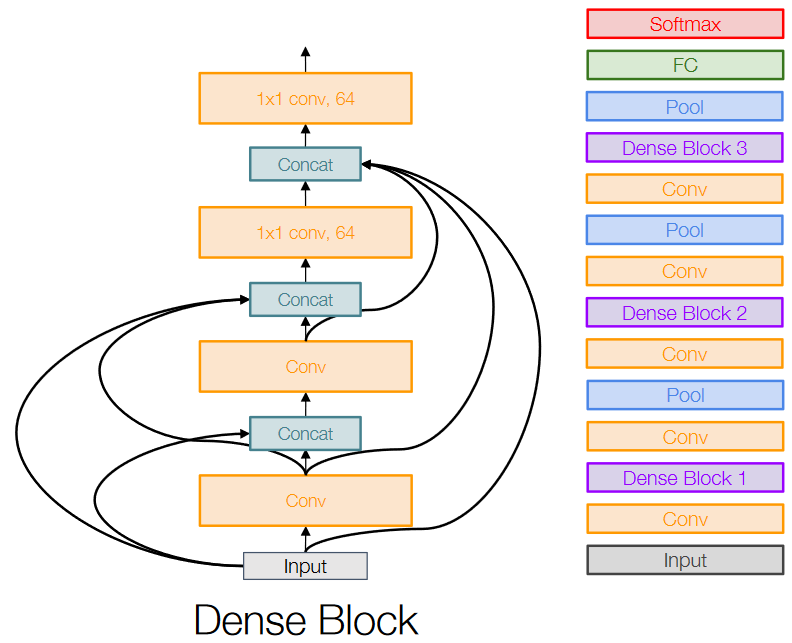


After this, the ImageNet challenge was officially closed.



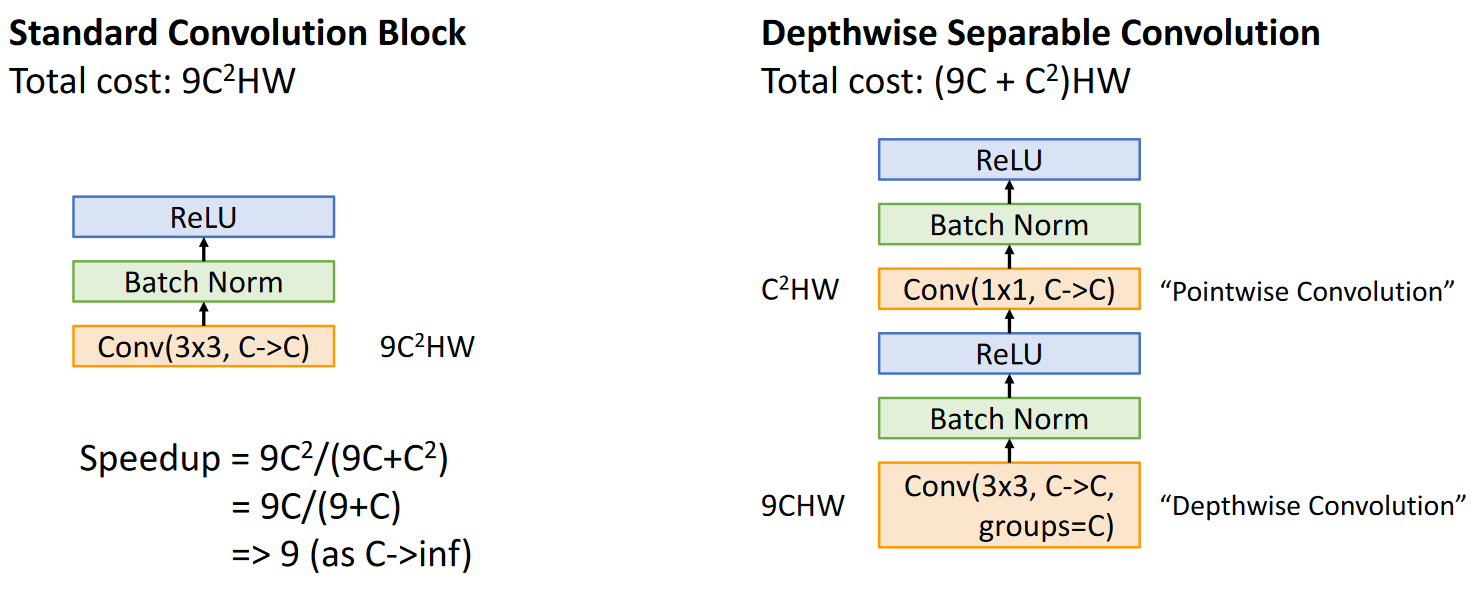
## Densely Connected Neural Networks

Despite the end of the ImageNet Challenge, several architecture continued to try to improve the performance on the dataset. One such architecture is **Densely Connected Neural Networks**. This takes the idea of the residual connections in ResNets and instead of adding features together, **concatenates** them.



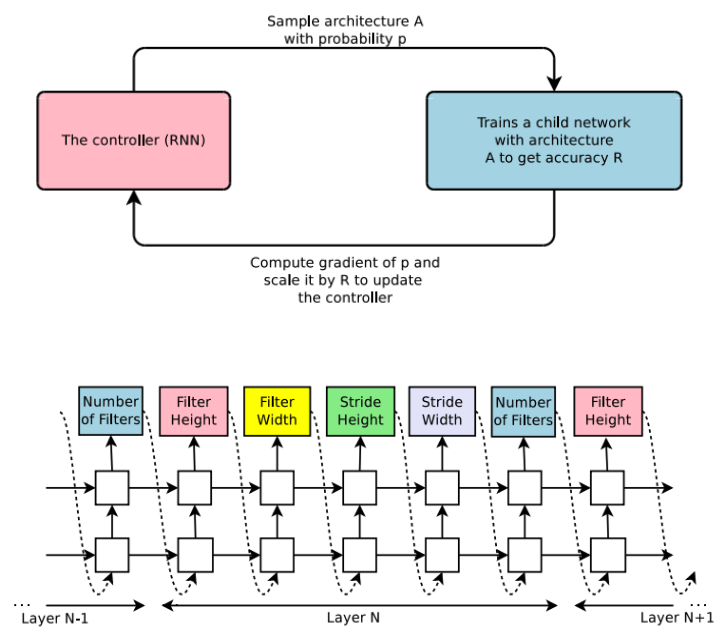
## MobileNets

Another trend was concentrating on extremely efficient models while maintaining the performance. One of the most famous models from this category is the **MobileNet** architecture.



## Neural Architecture Search

An extreme idea is to automate the entire process of designing neural networks. Some architectures have taken this approach. They create a **controller network** which can generate network architectures. Once a batch of child networks have been trained, the controller network is makes a gradient step using a **policy gradient**. Over time, the controller learns to generate children that have good performance.



This is an extremely expensive process. The original paper implemented this using 800 GPUs over 28 days, although later works concentrated on decreasing this expense. However, the process does work and can be used to find efficient CNN architectures.

