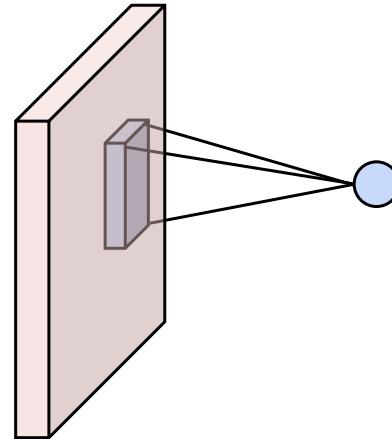


# CS163: Deep Learning for Computer Vision

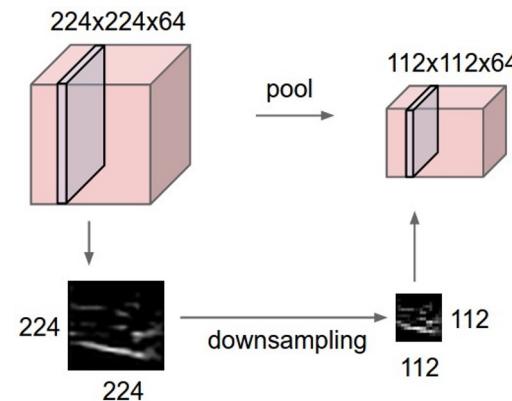
## Lecture 8: Modern CNN Architectures (up to 2018)

# Last Time: Components of Convolutional Networks

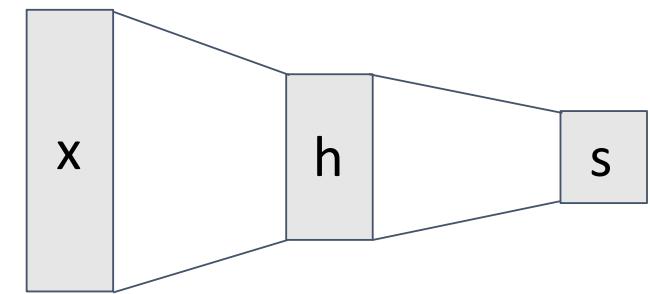
Convolution Layers



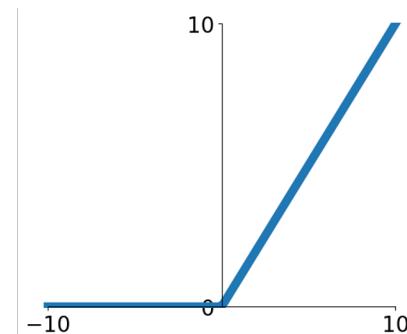
Pooling Layers



Fully-Connected Layers



Activation Function



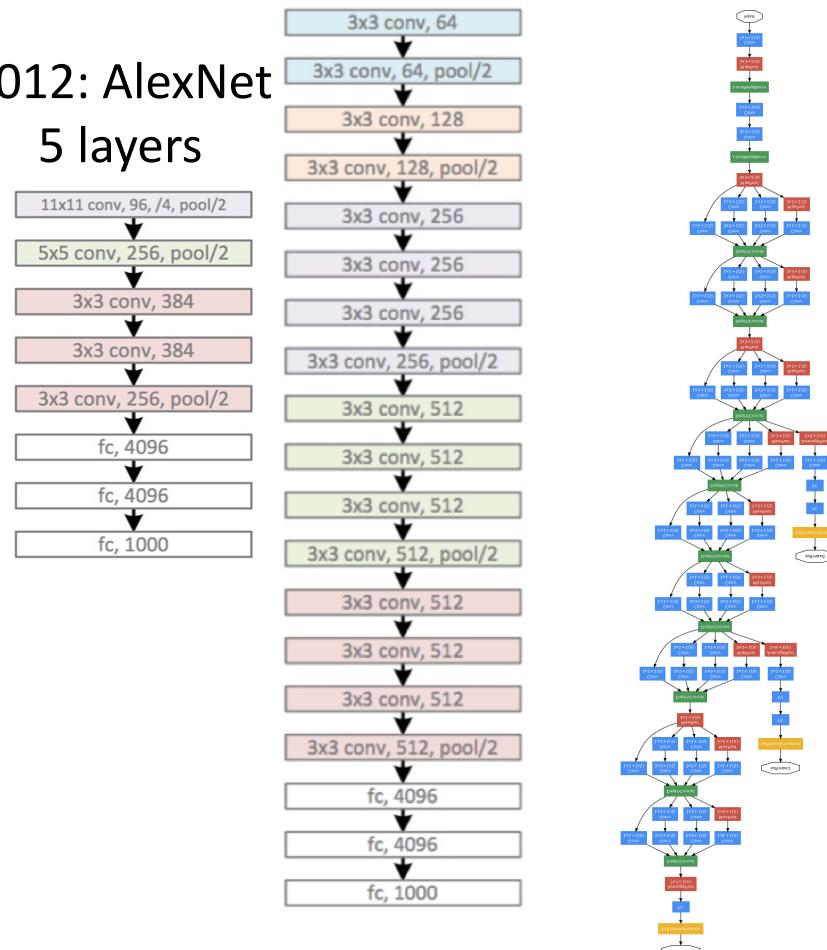
Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

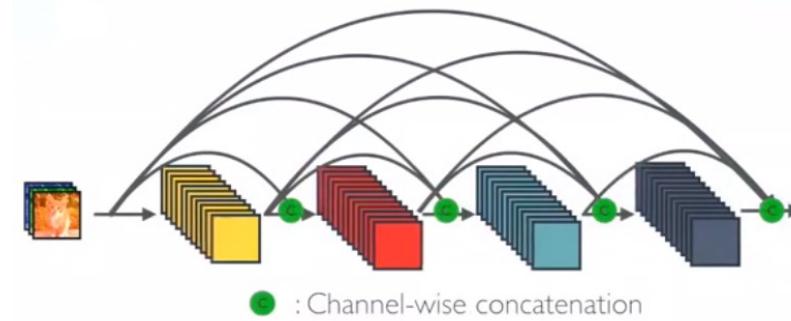
2016: ResNet  
>100 layers

2014: VGG 2015: GoogLeNet  
16 layers 22 layers

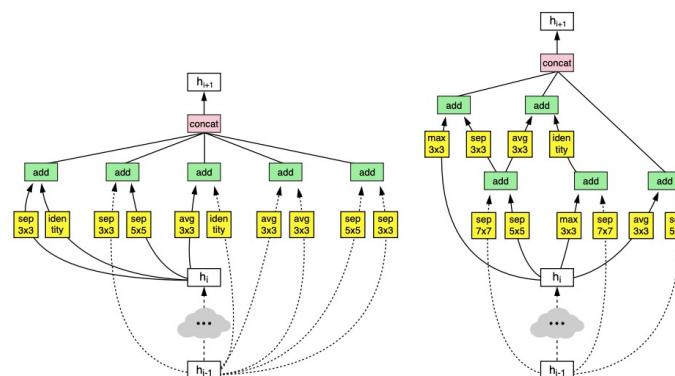
2012: AlexNet  
5 layers



# This Lecture: A Zoo of CNN Architectures

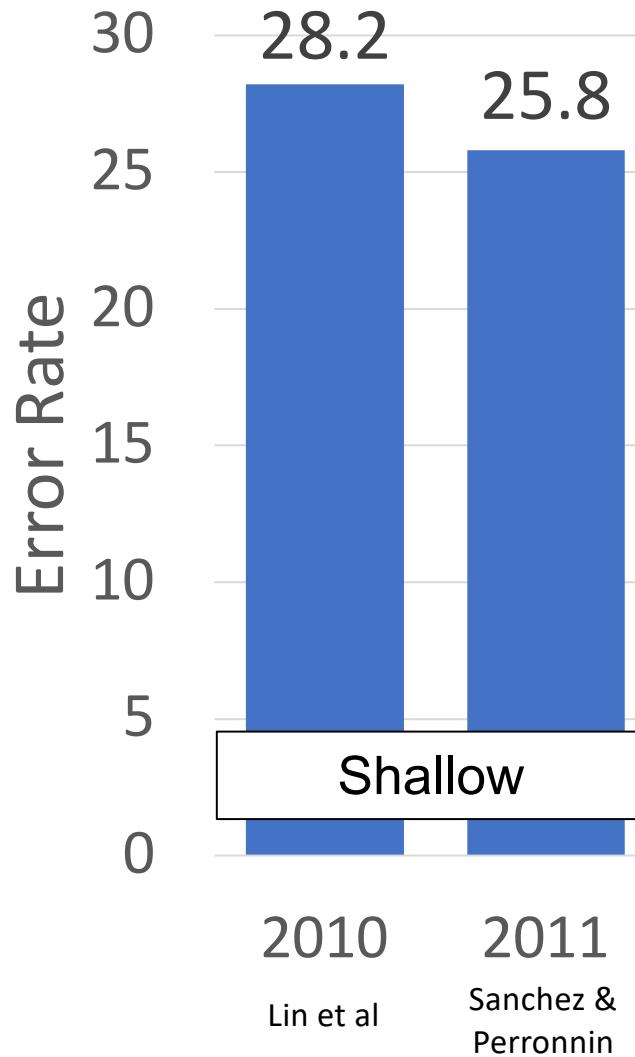


2017: DenseNet  
>100 layers

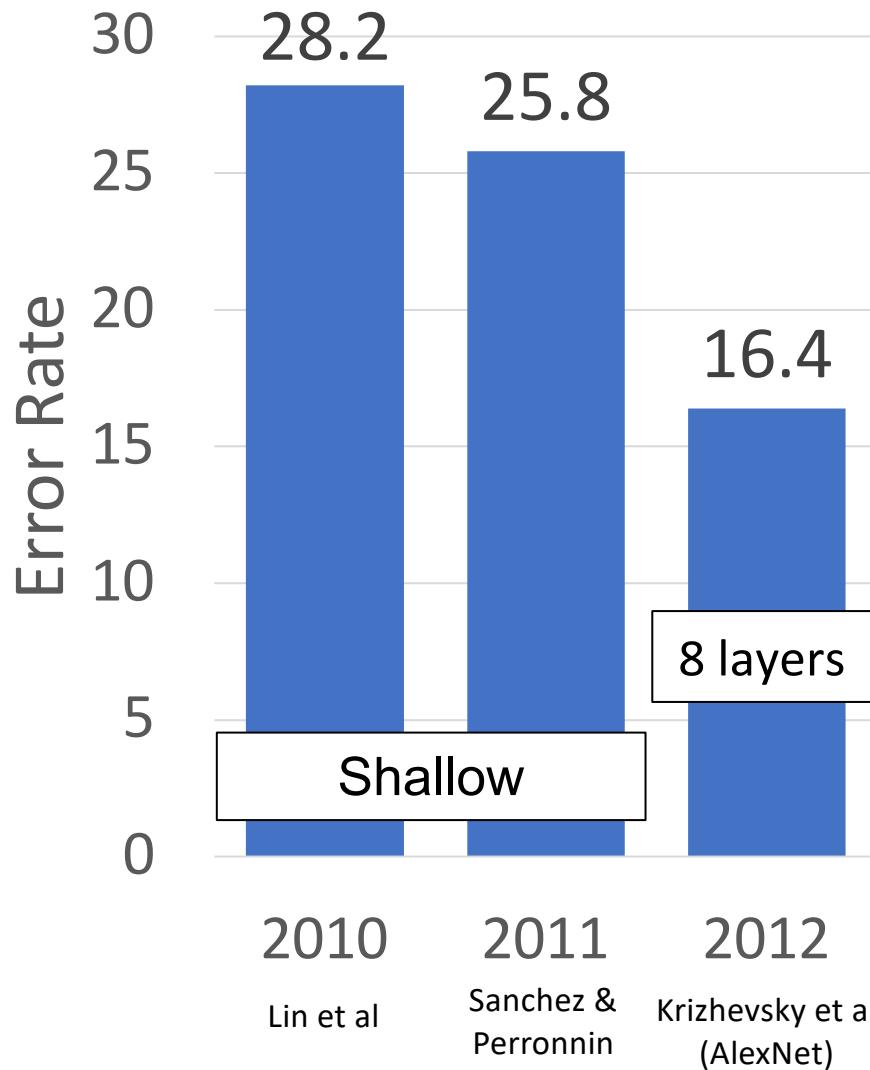


2017: NASNet

# ImageNet Classification Challenge



# ImageNet Classification Challenge



# AlexNet

227 x 227 inputs  
5 Convolutional layers  
Max pooling  
3 fully-connected layers  
ReLU nonlinearities

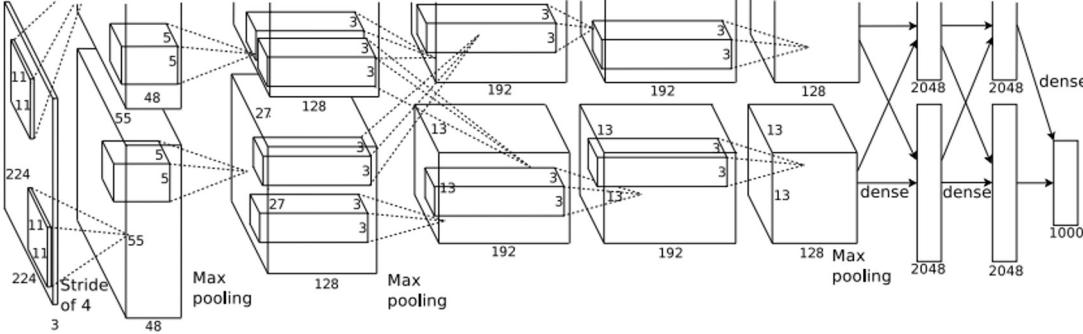
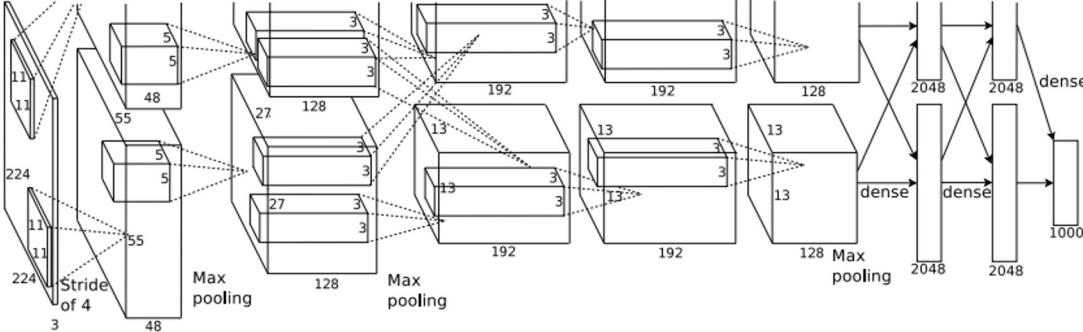


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# AlexNet

227 x 227 inputs  
5 Convolutional layers  
Max pooling  
3 fully-connected layers  
ReLU nonlinearities



Used “Local response normalization”;  
Not used anymore

Trained on two GTX 580 GPUs – only  
3GB of memory each! Model split  
over two GPUs

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# AlexNet

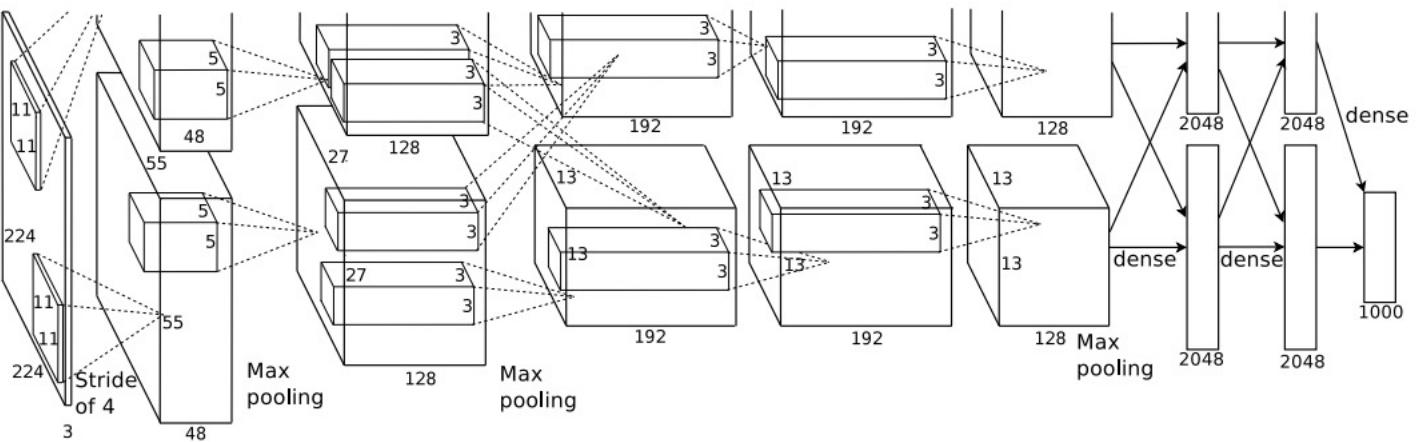
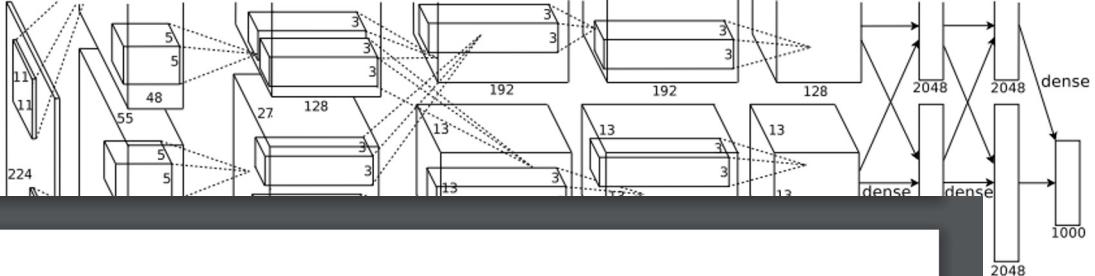
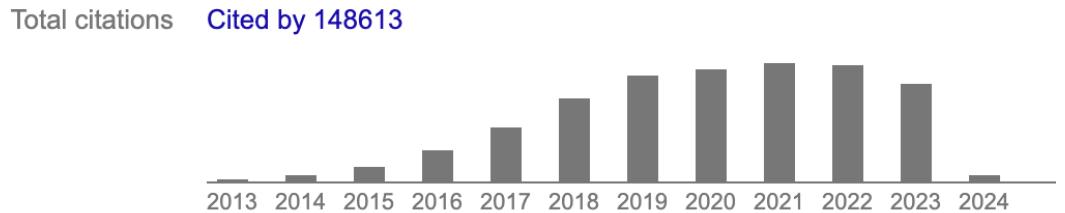
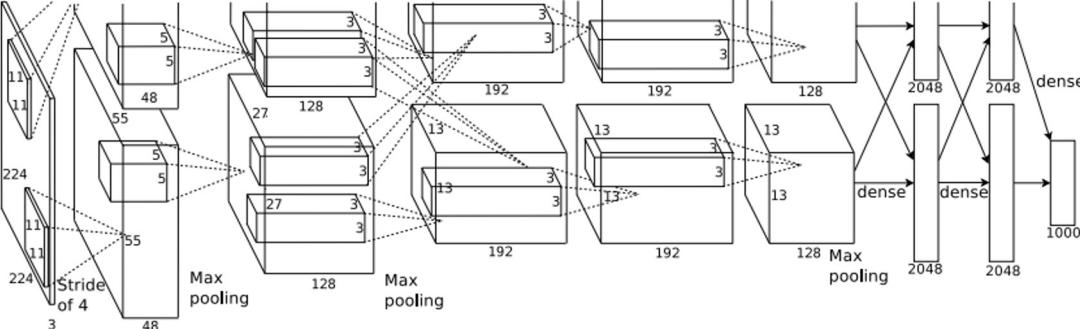


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

# AlexNet



Krizhevsky et al. ImageNet classification with deep convolutional neural networks. NeurIPS'12



## Citation Counts

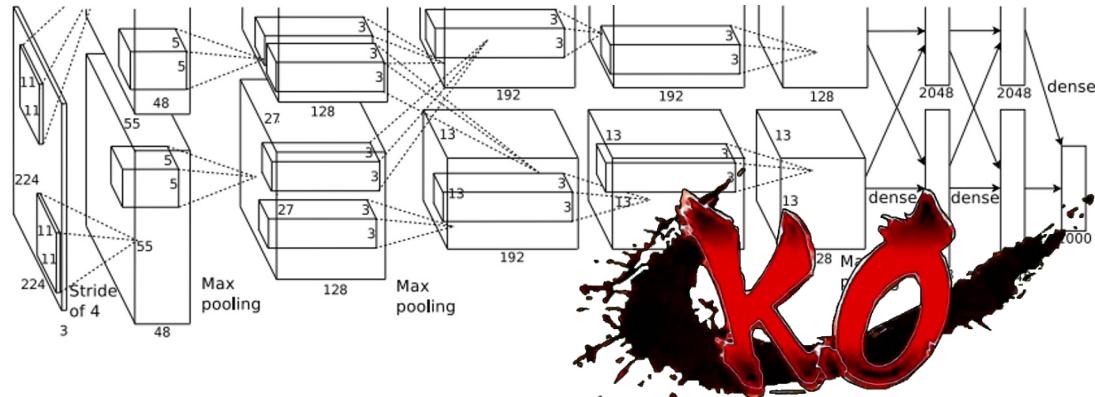
Darwin, “On the origin of species”, 1859: **54,348**

Shannon, “A mathematical theory of communication”, 1948: **131,593**

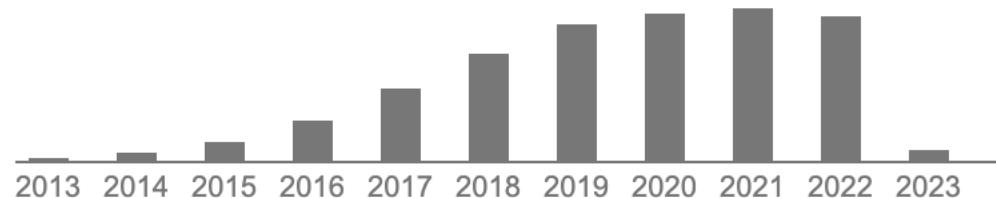
Watson and Crick, “Molecular Structure of Nucleic Acids”, 1953: **16,248**

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

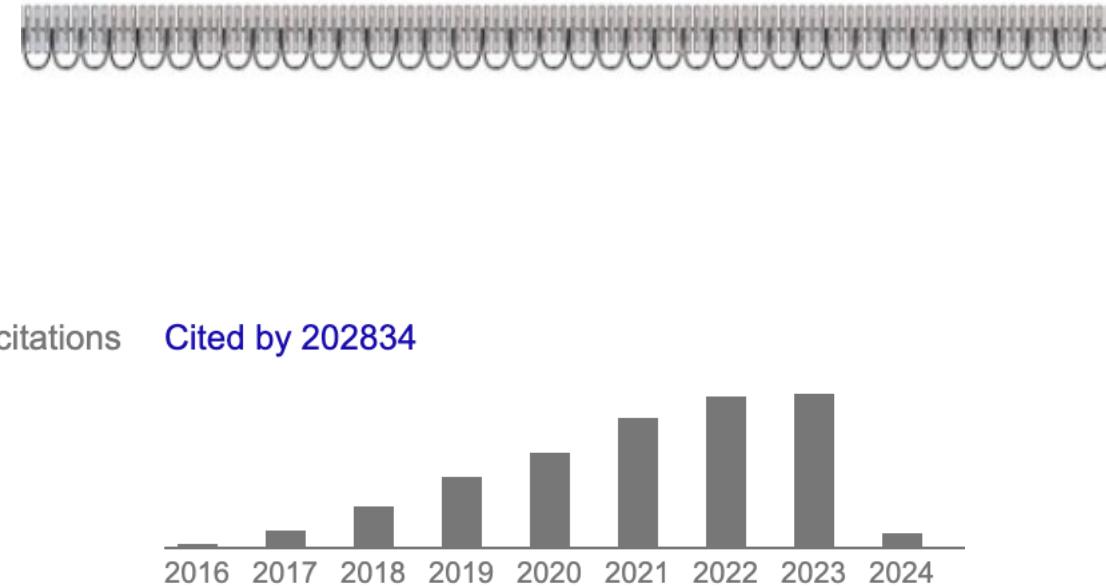
# AlexNet



Cited by 126922



Krizhevsky et al. ImageNet classification with deep convolutional neural networks. NeurIPS'12

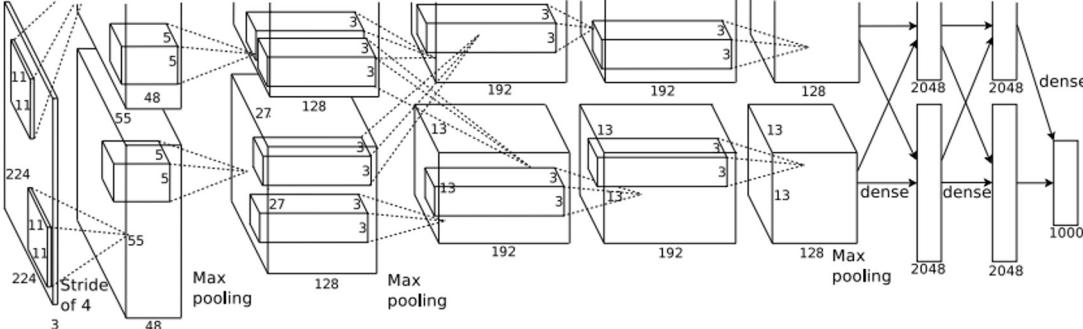


He et al. Deep Residual Learning for Image Recognition. CVPR'16

Dated Feb.1, 2024

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

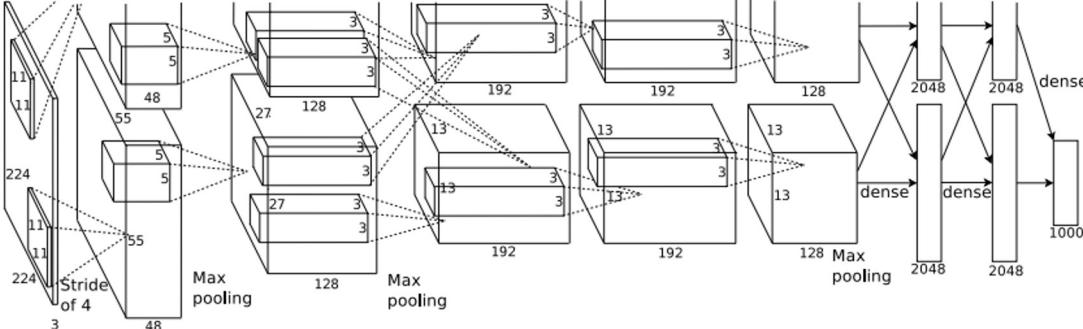
# AlexNet



	Input size		Layer					Output size	
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	
conv1	3	227	64	11	4	2	?		

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# AlexNet

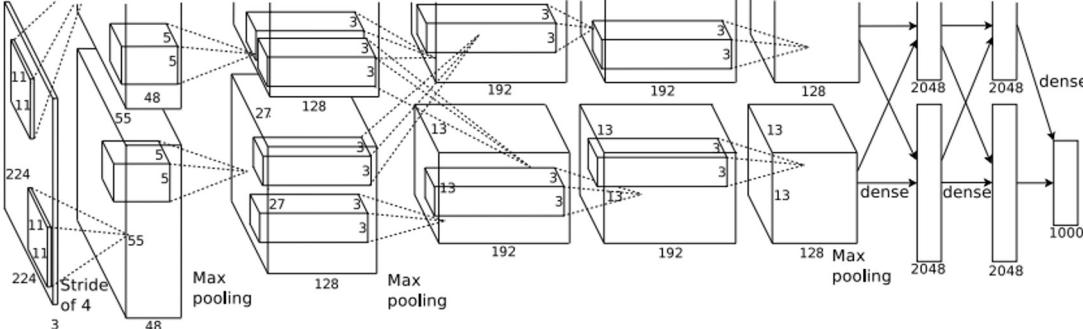


	Input size		Layer					Output size	
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	
conv1	3	227	64	11	4	2	64	?	

Recall: Output channels = number of filters

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# AlexNet

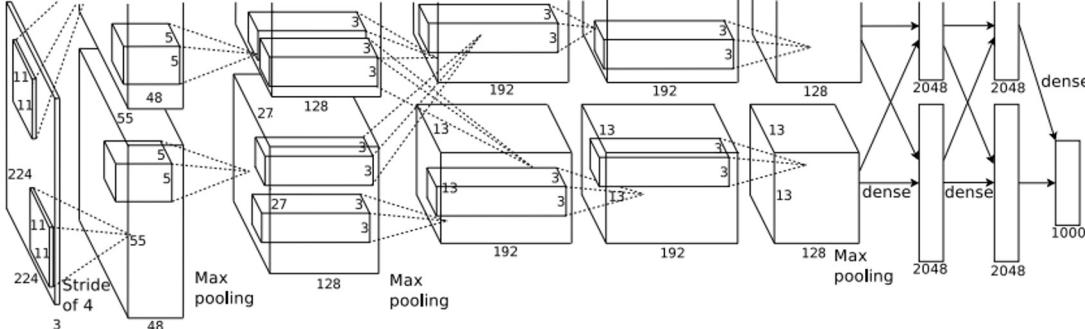


	Input size		Layer					Output size	
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	
conv1	3	227	64	11	4	2	64	56	

$$\begin{aligned}
 \text{Recall: } W' &= (W - K + 2P) / S + 1 \\
 &= (227 - 11 + 2*2) / 4 + 1 \\
 &= 220/4 + 1 = 56
 \end{aligned}$$

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

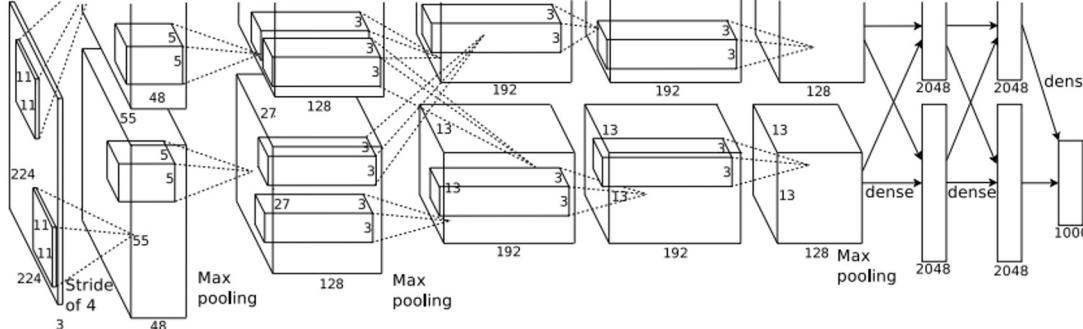
# AlexNet



	Input size		Layer					Output size		
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	
conv1	3	227	64	11	4	2	64	56	?	

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# AlexNet



	Input size		Layer					Output size		
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	
conv1	3	227	64	11	4	2	64	56	784	

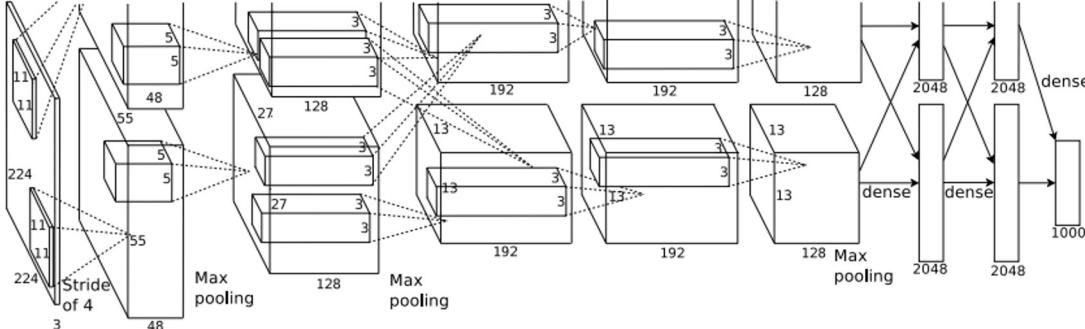
$$\begin{aligned}
 \text{Number of output elements} &= C * H' * W' \\
 &= 64 * 56 * 56 = 200,704
 \end{aligned}$$

Bytes per element = 4 (for 32-bit floating point)

$$\begin{aligned}
 \text{KB} &= (\text{number of elements}) * (\text{bytes per elem}) / 1024 \\
 &= 200704 * 4 / 1024 \\
 &= \mathbf{784}
 \end{aligned}$$

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

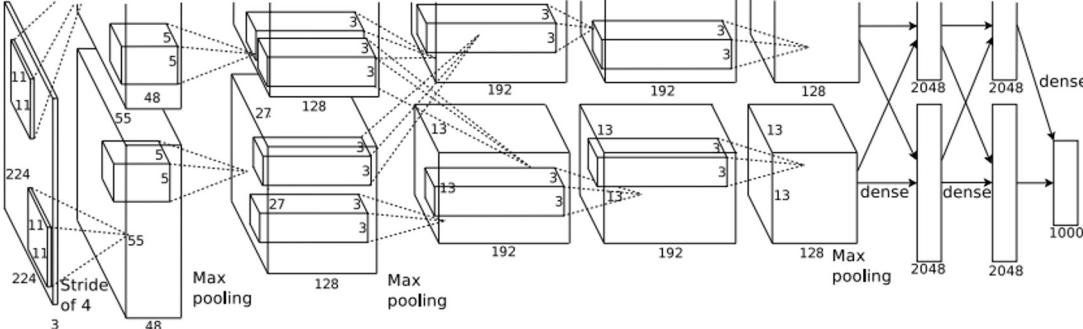
# AlexNet



	Input size		Layer					Output size			
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	
conv1	3	227	64	11	4	2	64	56	784	?	

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# AlexNet



	Input size		Layer					Output size			
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	
conv1	3	227	64	11	4	2	64	56	784	23	

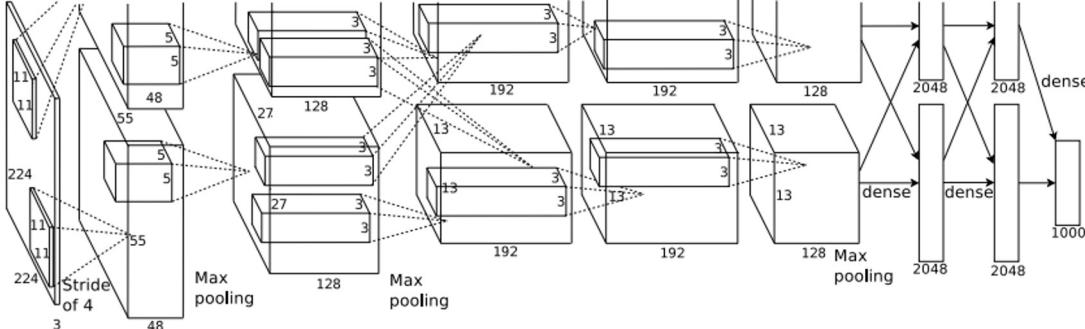
$$\begin{aligned}\text{Weight shape} &= C_{\text{out}} \times C_{\text{in}} \times K \times K \\ &= 64 \times 3 \times 11 \times 11\end{aligned}$$

$$\text{Bias shape} = C_{\text{out}} = 64$$

$$\begin{aligned}\text{Number of weights} &= 64 * 3 * 11 * 11 + 64 \\ &= \mathbf{23,296}\end{aligned}$$

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

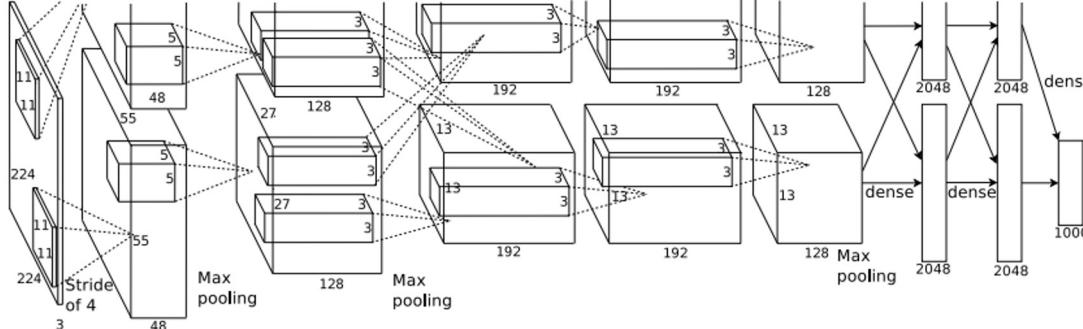
# AlexNet



	Input size		Layer					Output size			
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	?

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# AlexNet

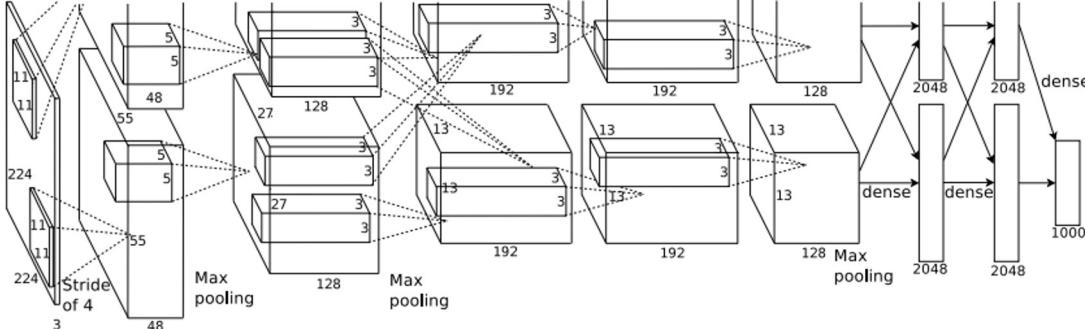


	Input size		Layer					Output size			
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73

Number of floating point operations (multiply+add)  
 $= (\text{number of output elements}) * (\text{ops per output elem})$   
 $= (C_{\text{out}} \times H' \times W') * (C_{\text{in}} \times K \times K)$   
 $= (64 * 56 * 56) * (3 * 11 * 11)$   
 $= 200,704 * 363$   
 $= \mathbf{72,855,552}$

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

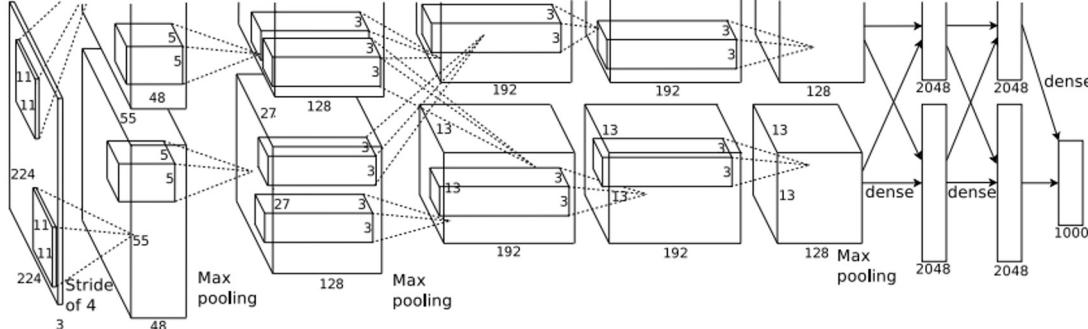
# AlexNet



	Input size		Layer					Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)	
conv1	3	227	64	11	4	2	64	56	784	23	73	
pool1	64	56		3	2	0	?					

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# AlexNet



	Input size		Layer					Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)	
conv1	3	227	64	11	4	2	64	56	784	23	73	
pool1	64	56		3	2	0	64	27				

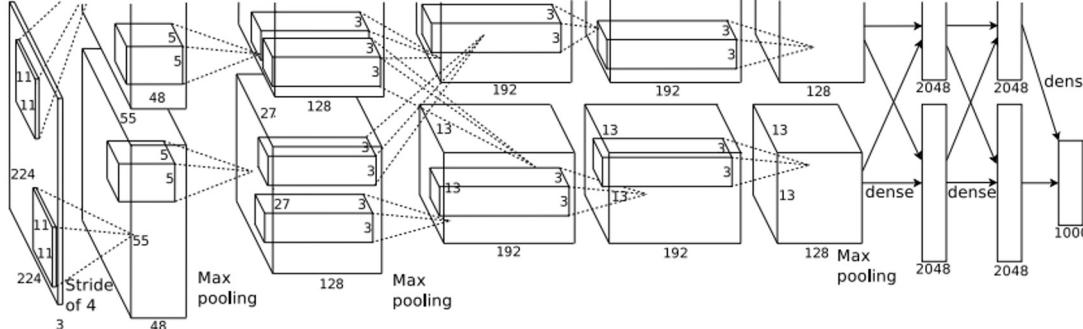
For pooling layer:

$$\text{#output channels} = \text{#input channels} = 64$$

$$\begin{aligned} W' &= \text{floor}((W - K) / S + 1) \\ &= \text{floor}(53 / 2 + 1) = \text{floor}(27.5) = 27 \end{aligned}$$

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# AlexNet



	Input size		Layer					Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)	
conv1	3	227	64	11	4	2	64	56	784	23	73	
pool1	64	56		3	2	0	64	27	182	?		

#output elems =  $C_{out} \times H' \times W'$

Bytes per elem = 4

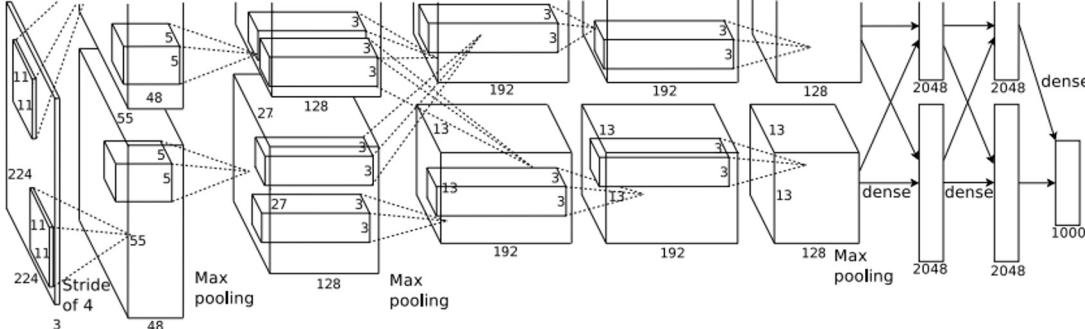
$$KB = C_{out} * H' * W' * 4 / 1024$$

$$= 64 * 27 * 27 * 4 / 1024$$

$$= 182.25$$

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# AlexNet

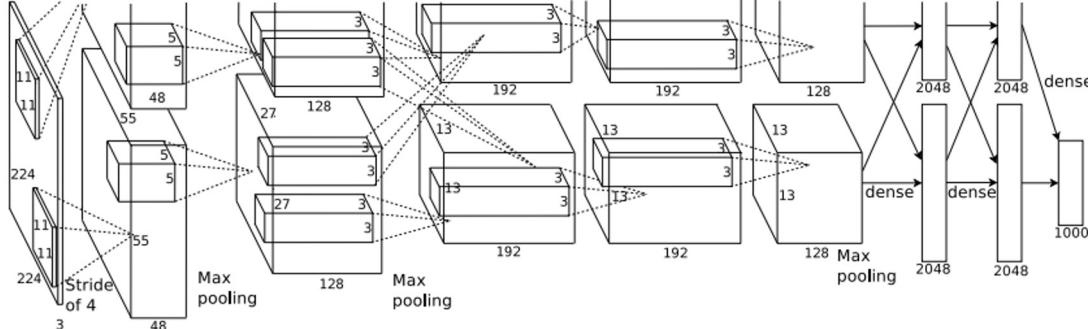


	Input size		Layer					Output size					
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)		
conv1	3	227	64	11	4	2	64	56	784	23	73		
pool1	64	56		3	2	0	64	27	182	0	?		

Pooling layers have no learnable parameters!

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# AlexNet



	Input size		Layer					Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)	
conv1	3	227	64	11	4	2	64	56	784	23	73	
pool1	64	56		3	2	0	64	27	182	0	0	

Floating-point ops for pooling layer

$$= (\text{number of output positions}) * (\text{flops per output position})$$

$$= (C_{\text{out}} * H' * W') * (K * K)$$

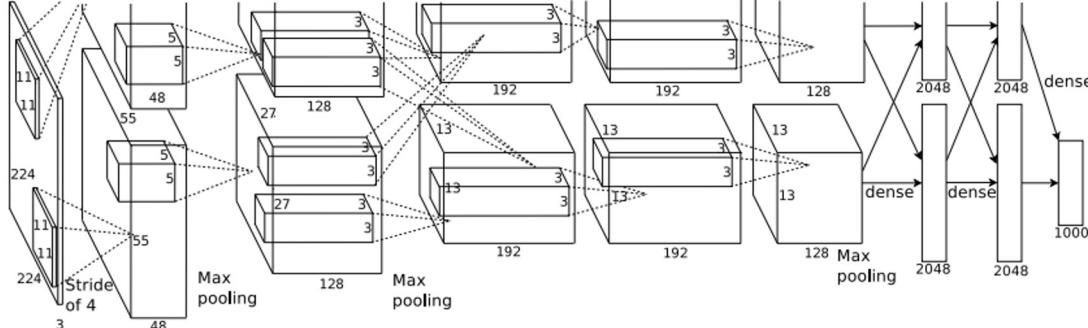
$$= (64 * 27 * 27) * (3 * 3)$$

$$= 419,904$$

$$= \mathbf{0.4 \text{ MFLOP}}$$

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# AlexNet

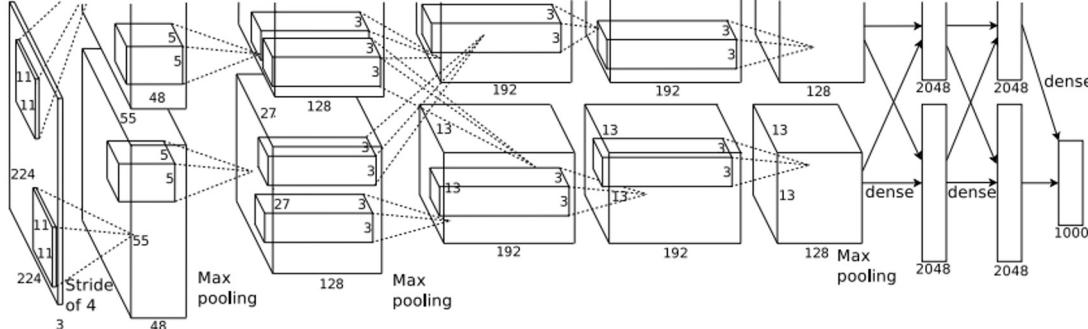


	Input size		Layer					Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)	
conv1	3	227	64	11	4	2	64	56	784	23	73	
pool1	64	56		3	2	0	64	27	182	0	0	
conv2	64	27	192	5	1	2	192	27	547	307	224	
pool2	192	27		3	2	0	192	13	127	0	0	
conv3	192	13	384	3	1	1	384	13	254	664	112	
conv4	384	13	256	3	1	1	256	13	169	885	145	
conv5	256	13	256	3	1	1	256	13	169	590	100	
pool5	256	13		3	2	0	256	6	36	0	0	
flatten	256	6					9216		36	0	0	

$$\begin{aligned}
 \text{Flatten output size} &= C_{\text{in}} \times H \times W \\
 &= 256 * 6 * 6 \\
 &= \mathbf{9216}
 \end{aligned}$$

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# AlexNet

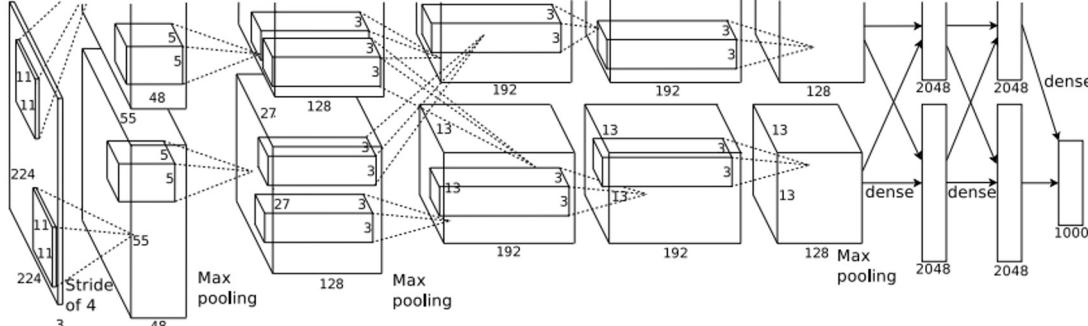


	Input size		Layer					Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)	
conv1	3	227	64	11	4	2	64	56	784	23	73	
pool1	64	56		3	2	0	64	27	182	0	0	
conv2	64	27	192	5	1	2	192	27	547	307	224	
pool2	192	27		3	2	0	192	13	127	0	0	
conv3	192	13	384	3	1	1	384	13	254	664	112	
conv4	384	13	256	3	1	1	256	13	169	885	145	
conv5	256	13	256	3	1	1	256	13	169	590	100	
pool5	256	13		3	2	0	256	6	36	0	0	
flatten	256	6					9216		36	0	0	
fc6	9216		4096				4096		16	37,749	38	

$$\begin{aligned}
 \text{FC params} &= C_{\text{in}} * C_{\text{out}} + C_{\text{out}} \\
 &= 9216 * 4096 + 4096 \\
 &= 37,725,832
 \end{aligned}$$

$$\begin{aligned}
 \text{FC flops} &= C_{\text{in}} * C_{\text{out}} \\
 &= 9216 * 4096 \\
 &= 37,748,736
 \end{aligned}$$

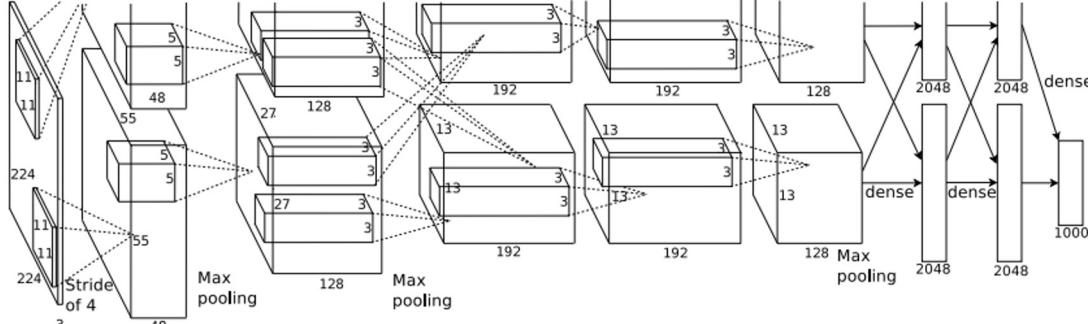
# AlexNet



	Input size		Layer					Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)	
conv1	3	227	64	11	4	2	64	56	784	23	73	
pool1	64	56		3	2	0	64	27	182	0	0	
conv2	64	27	192	5	1	2	192	27	547	307	224	
pool2	192	27		3	2	0	192	13	127	0	0	
conv3	192	13	384	3	1	1	384	13	254	664	112	
conv4	384	13	256	3	1	1	256	13	169	885	145	
conv5	256	13	256	3	1	1	256	13	169	590	100	
pool5	256	13		3	2	0	256	6	36	0	0	
flatten	256	6					9216		36	0	0	
fc6	9216		4096				4096		16	37,749	38	
fc7	4096		4096				4096		16	16,777	17	
fc8	4096		1000				1000		4	4,096	4	

# AlexNet

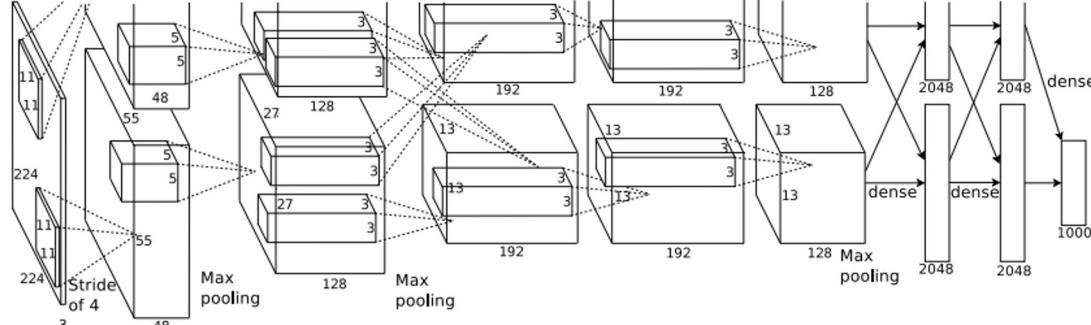
How to choose this?  
Trial and error =(



	Input size		Layer					Output size					
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)		
conv1	3	227	64	11	4	2	64	56	784	23	73		
pool1	64	56		3	2	0	64	27	182	0	0		
conv2	64	27	192	5	1	2	192	27	547	307	224		
pool2	192	27		3	2	0	192	13	127	0	0		
conv3	192	13	384	3	1	1	384	13	254	664	112		
conv4	384	13	256	3	1	1	256	13	169	885	145		
conv5	256	13	256	3	1	1	256	13	169	590	100		
pool5	256	13		3	2	0	256	6	36	0	0		
flatten	256	6					9216		36	0	0		
fc6	9216		4096				4096		16	37,749	38		
fc7	4096		4096				4096		16	16,777	17		
fc8	4096		1000				1000		4	4,096	4		

# AlexNet

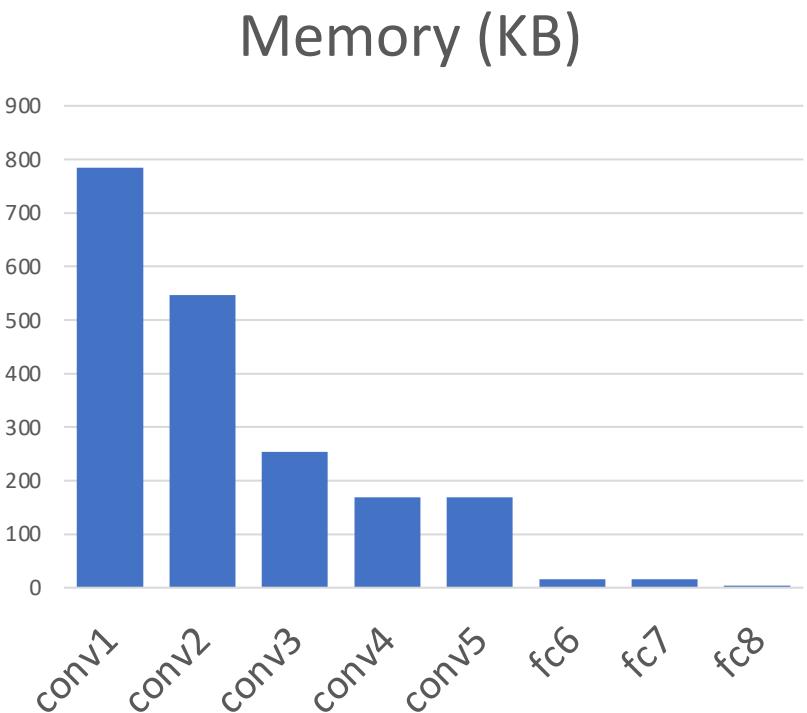
Interesting trends here!



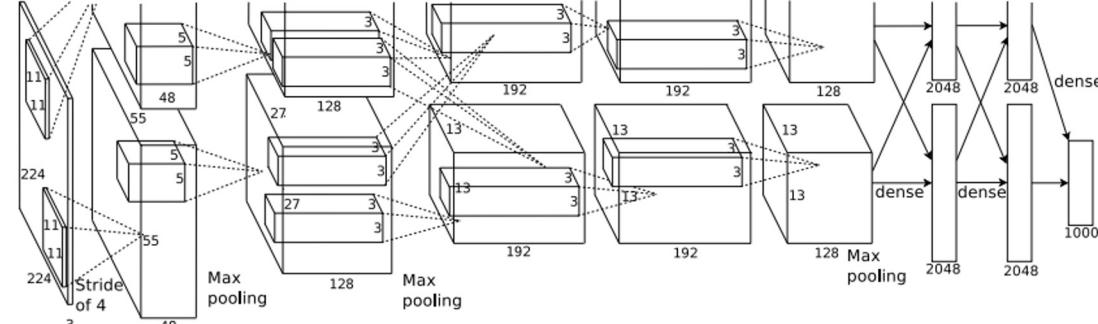
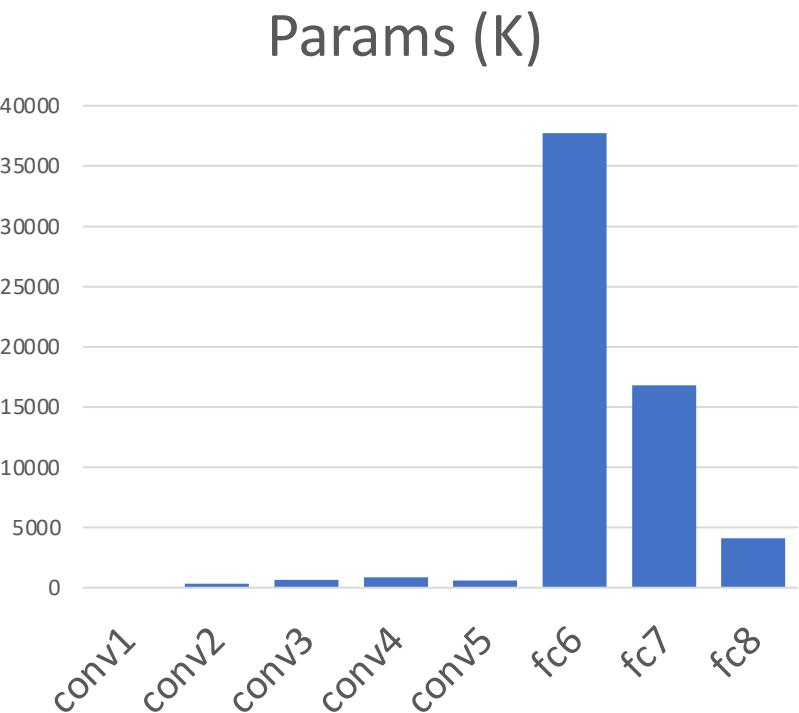
	Input size		Layer					Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)	
conv1	3	227	64	11	4	2	64	56	784	23	73	
pool1	64	56		3	2	0	64	27	182	0	0	
conv2	64	27	192	5	1	2	192	27	547	307	224	
pool2	192	27		3	2	0	192	13	127	0	0	
conv3	192	13	384	3	1	1	384	13	254	664	112	
conv4	384	13	256	3	1	1	256	13	169	885	145	
conv5	256	13	256	3	1	1	256	13	169	590	100	
pool5	256	13		3	2	0	256	6	36	0	0	
flatten	256	6					9216		36	0	0	
fc6	9216		4096				4096		16	37,749	38	
fc7	4096		4096				4096		16	16,777	17	
fc8	4096		1000				1000		4	4,096	4	

# AlexNet

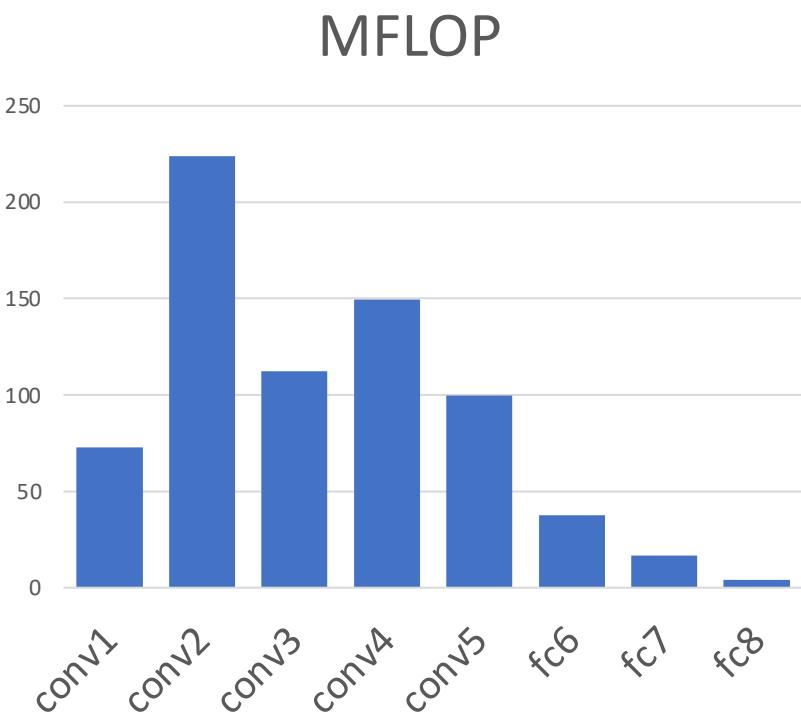
Most of the **memory usage** is in the early convolution layers



Nearly all **parameters** are in the fully-connected layers



Most **floating-point ops** occur in the convolution layers



# Fact about AlexNet's authors

---

## ImageNet Classification with Deep Convolutional Neural Networks

---

**Alex Krizhevsky**  
University of Toronto  
kriz@cs.utoronto.ca

**Ilya Sutskever**  
University of Toronto  
ilya@cs.utoronto.ca

**Geoffrey E. Hinton**  
University of Toronto  
hinton@cs.utoronto.ca



Ilya Sutskever  
Co-Founder of OpenAI  
Chief Scientist

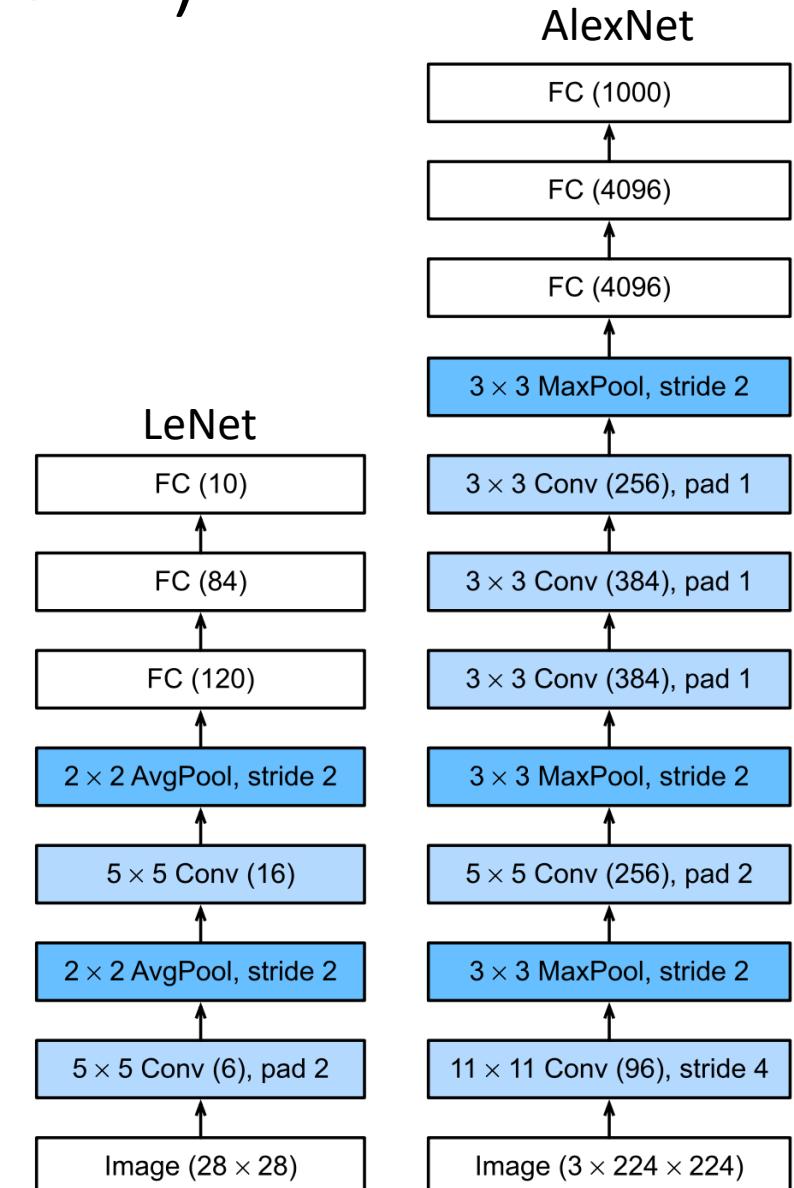




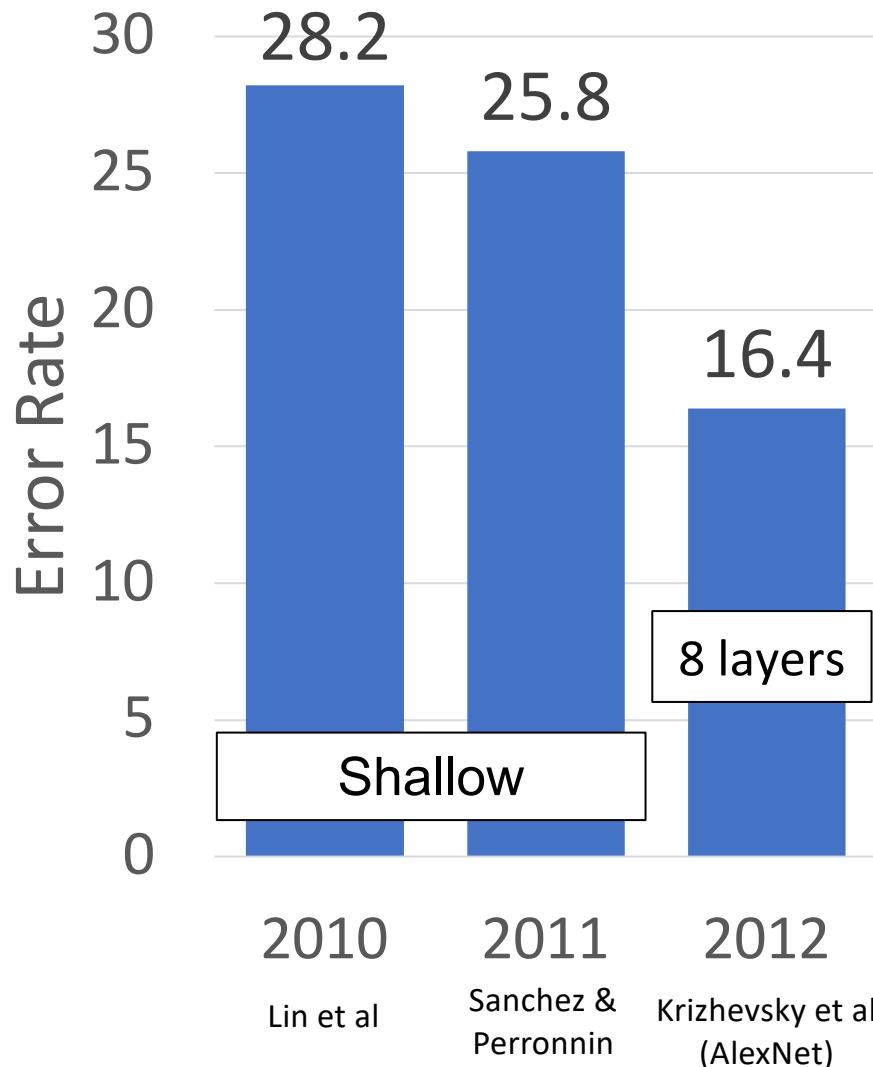
# From LeNet (1989) to AlexNet (2012)

Shared the similar design philosophies, but there are some major differences:

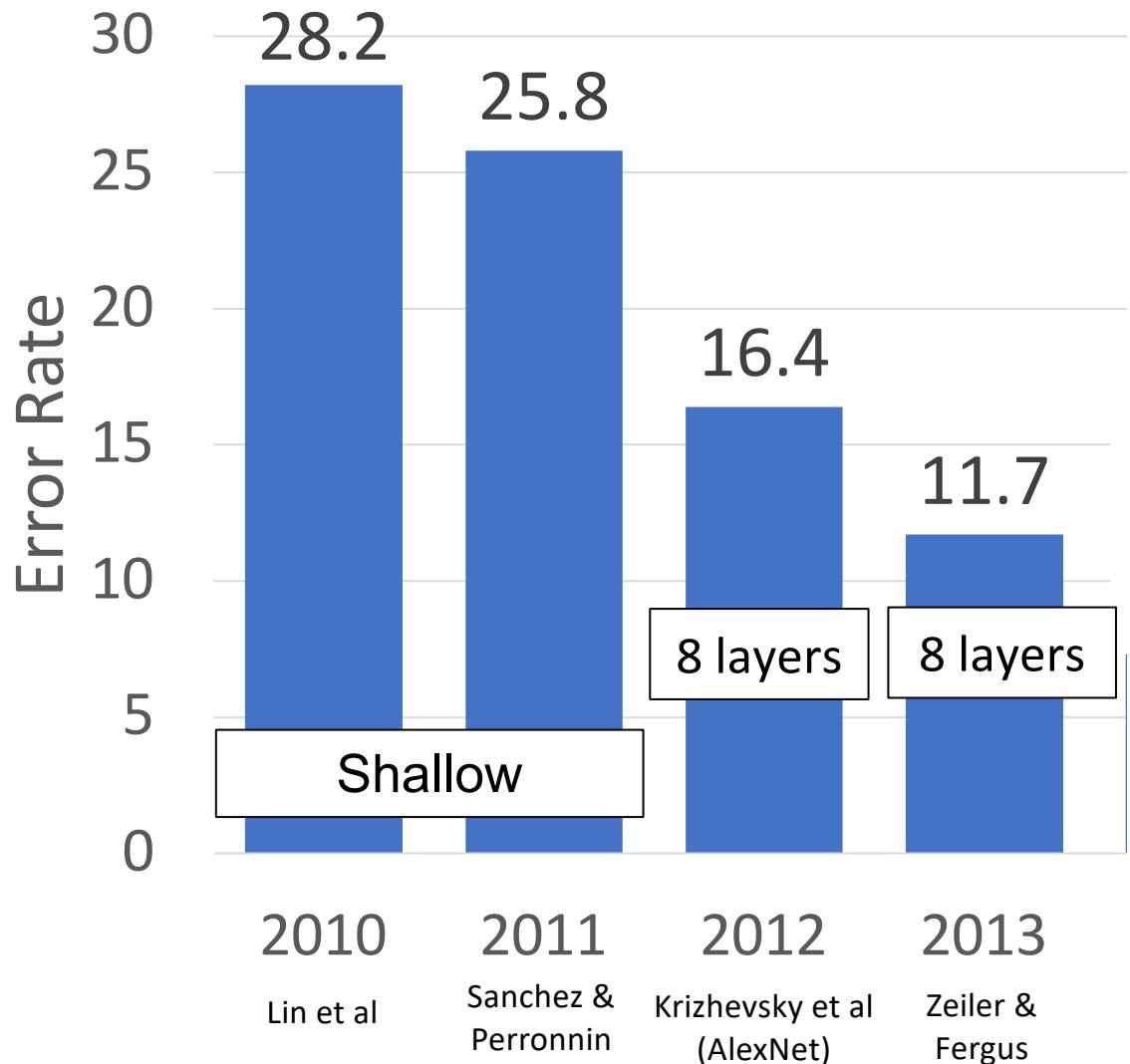
1. Deeper and larger
2. ReLU instead of sigmoid as the activation functions
3. GPUs and large dataset available



# ImageNet Classification Challenge

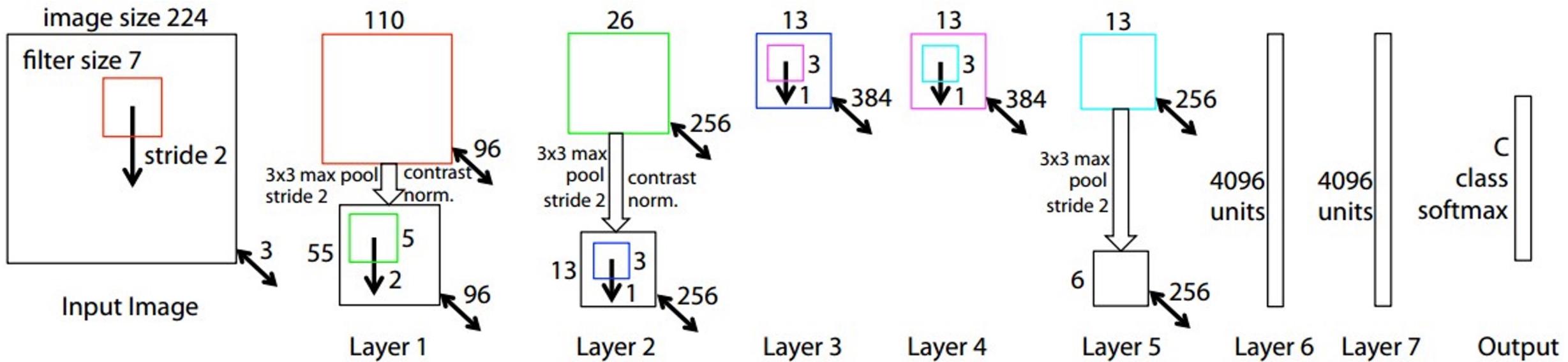


# ImageNet Classification Challenge



# ZFNet: A Bigger AlexNet

ImageNet top 5 error: 16.4% → 11.7%



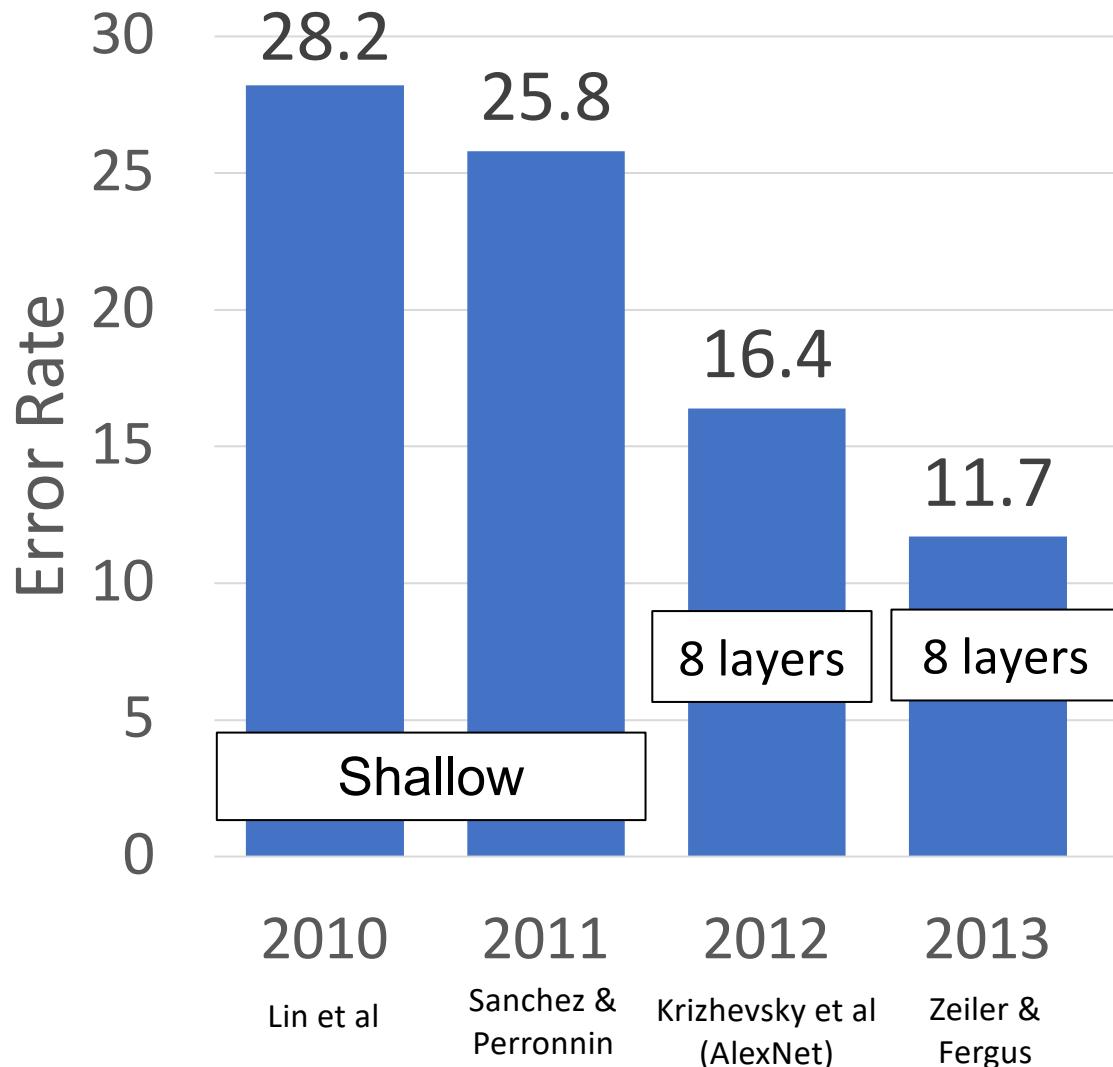
AlexNet but:

CONV1: change from (11x11 stride 4) to (7x7 stride 2)

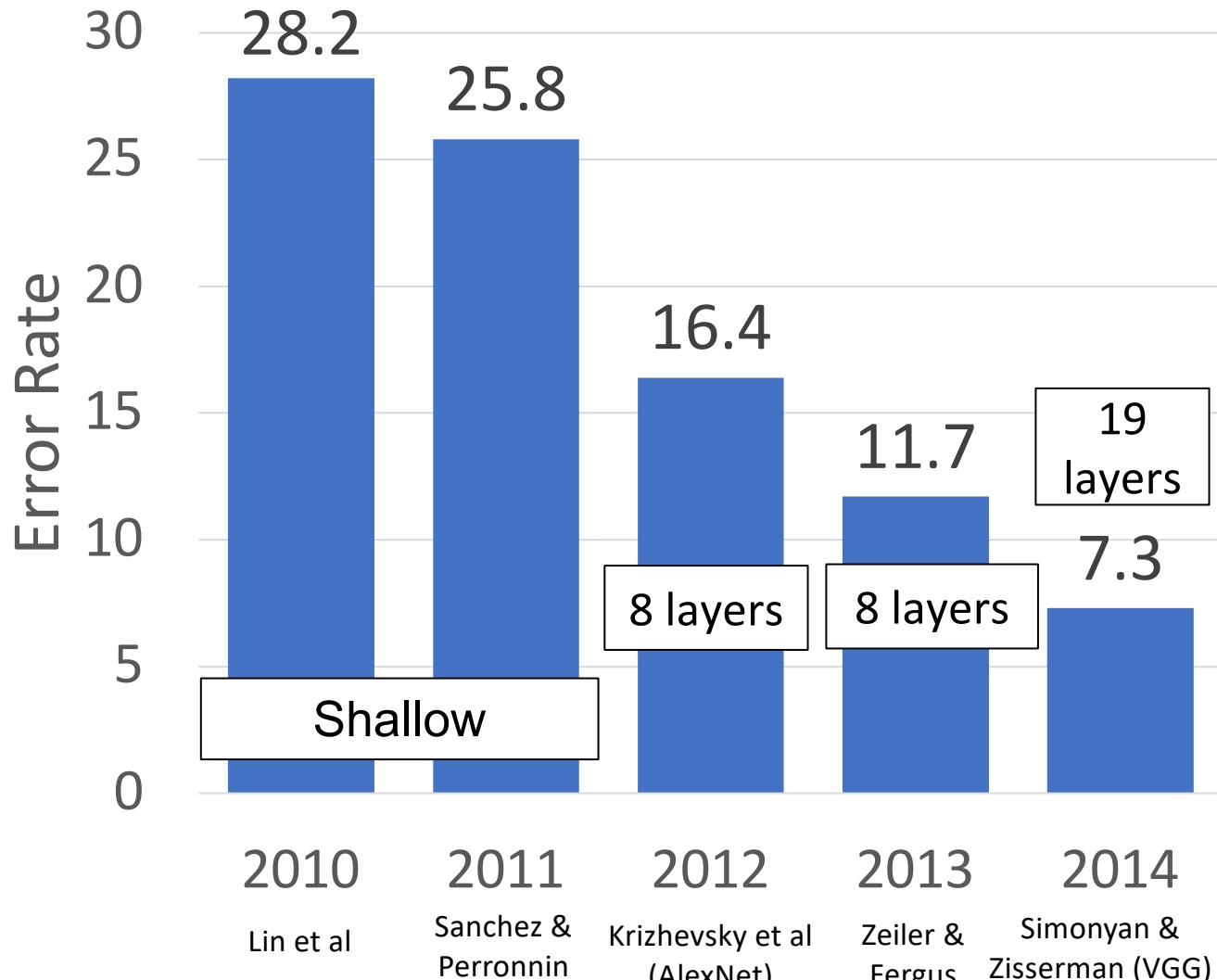
CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512

More trial and error = (

# ImageNet Classification Challenge



# ImageNet Classification Challenge



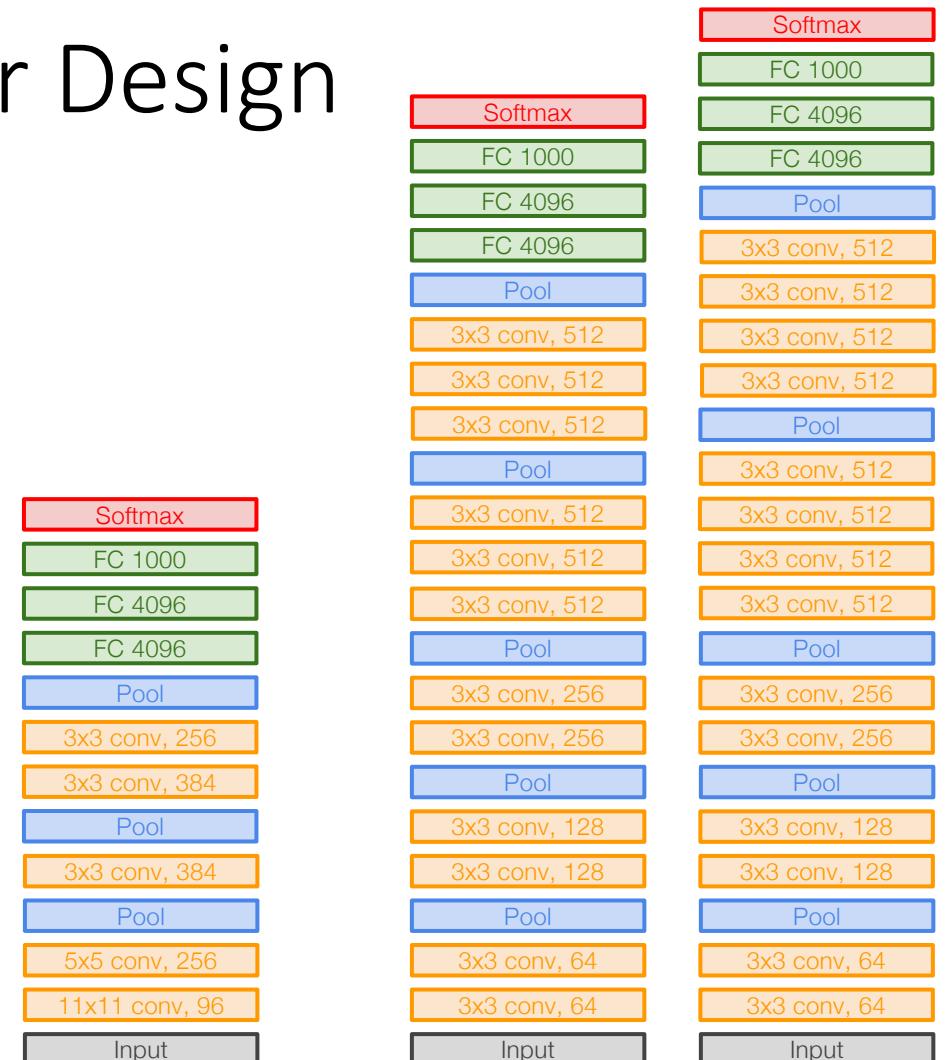
# VGG: Deeper Networks, Regular Design

## VGG Design rules:

All conv are 3x3 stride 1 pad 1

All max pool are 2x2 stride 2

After pool, double #channels



AlexNet

VGG16

VGG19

# VGG: Deeper Networks, Regular Design

## VGG Design rules:

All conv are 3x3 stride 1 pad 1

All max pool are 2x2 stride 2

After pool, double #channels

Network has 5 convolutional **stages**:

Stage 1: conv-conv-pool

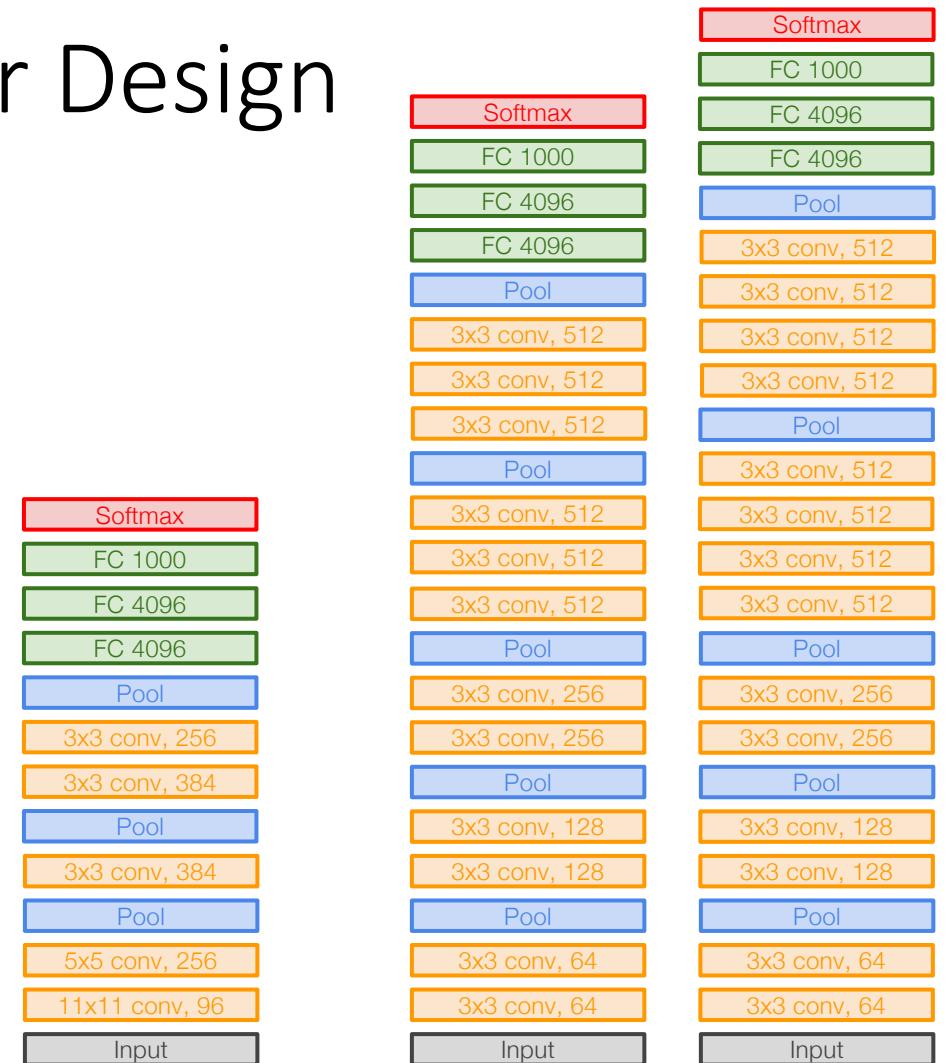
Stage 2: conv-conv-pool

Stage 3: conv-conv-pool

Stage 4: conv-conv-conv-[conv]-pool

Stage 5: conv-conv-conv-[conv]-pool

(VGG-19 has 4 conv in stages 4 and 5)



AlexNet

VGG16

VGG19

# VGG: Deeper Networks, Regular Design

## VGG Design rules:

**All conv are 3x3 stride 1 pad 1**

All max pool are 2x2 stride 2

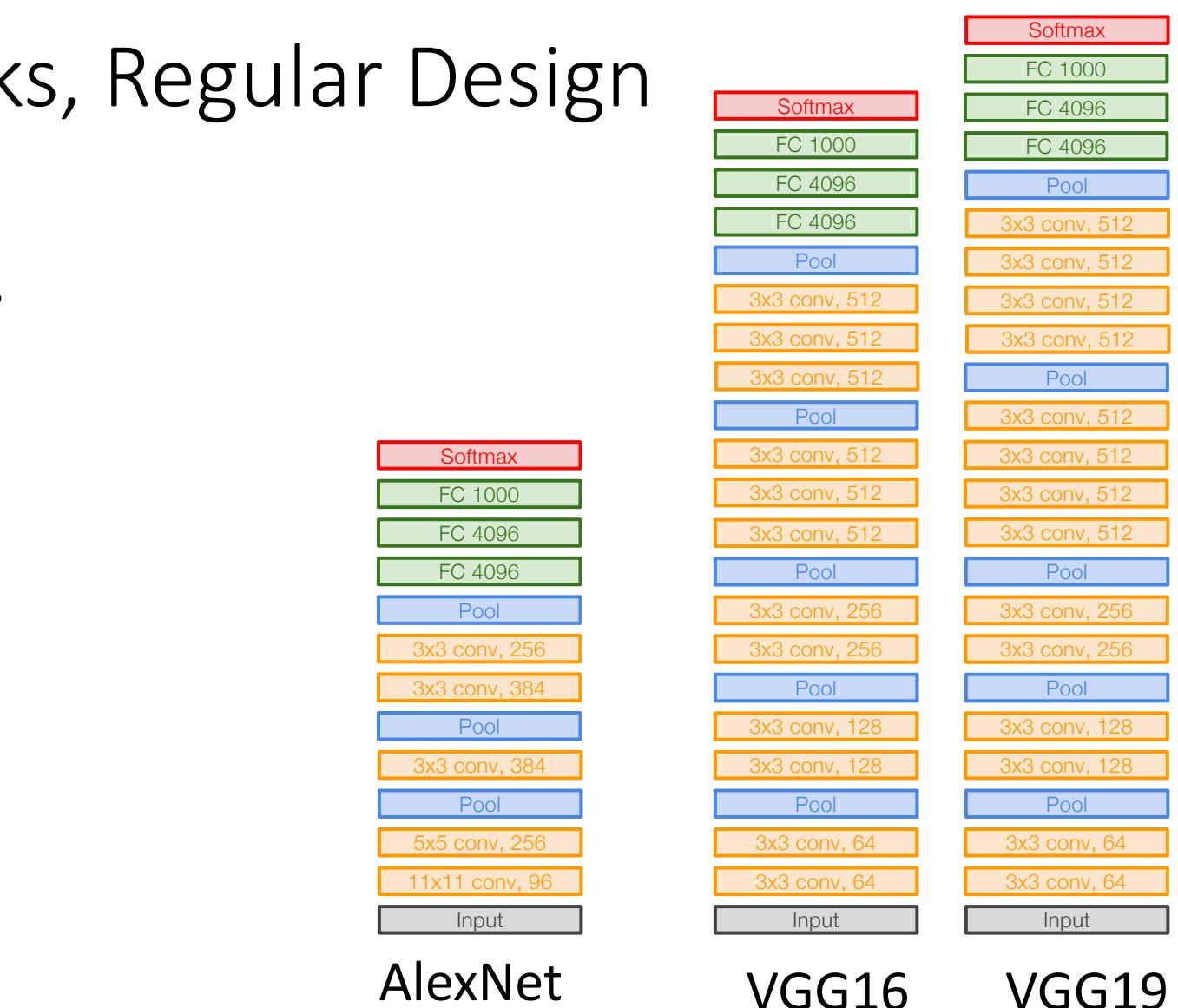
After pool, double #channels

## Option 1:

Conv(5x5, C -> C)

Params:  $25C^2$

FLOPs:  $25C^2HW$



# VGG: Deeper Networks, Regular Design

## VGG Design rules:

**All conv are 3x3 stride 1 pad 1**

All max pool are 2x2 stride 2

After pool, double #channels

### Option 1:

Conv(5x5, C -> C)

Params:  $25C^2$

FLOPs:  $25C^2HW$

### Option 2:

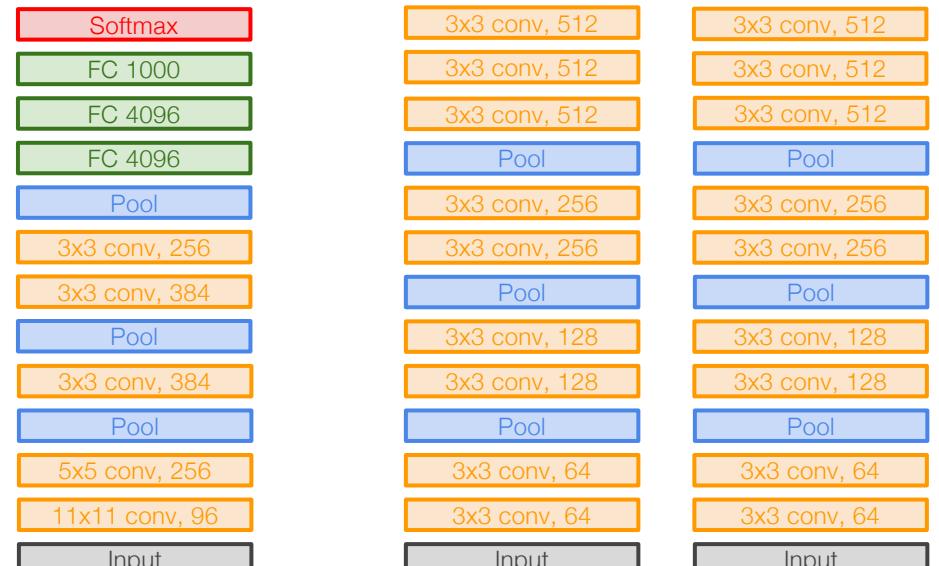
Conv(3x3, C -> C)

Conv(3x3, C -> C)

Params:  $18C^2$

Params:  $18C^2$

FLOPs:  $18C^2HW$



AlexNet

VGG16

VGG19

# VGG: Deeper Networks, Regular Design

## VGG Design rules:

**All conv are 3x3 stride 1 pad 1**

All max pool are 2x2 stride 2

After pool, double #channels

Two 3x3 conv has same receptive field as a single 5x5 conv, but has fewer parameters and takes less computation!

## Option 1:

Conv(5x5, C -> C)

Params:  $25C^2$

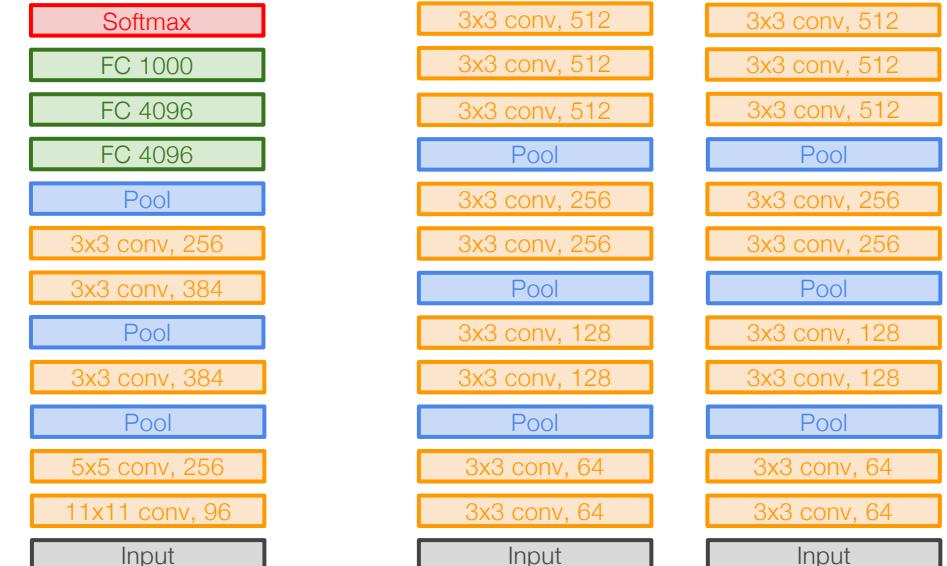
FLOPs:  $25C^2HW$

## Option 2:

Conv(3x3, C -> C)  
Conv(3x3, C -> C)

Params:  $18C^2$

FLOPs:  $18C^2HW$



AlexNet

VGG16

VGG19

# VGG: Deeper Networks, Regular Design

## VGG Design rules:

All conv are 3x3 stride 1 pad 1

All max pool are 2x2 stride 2

After pool, double #channels

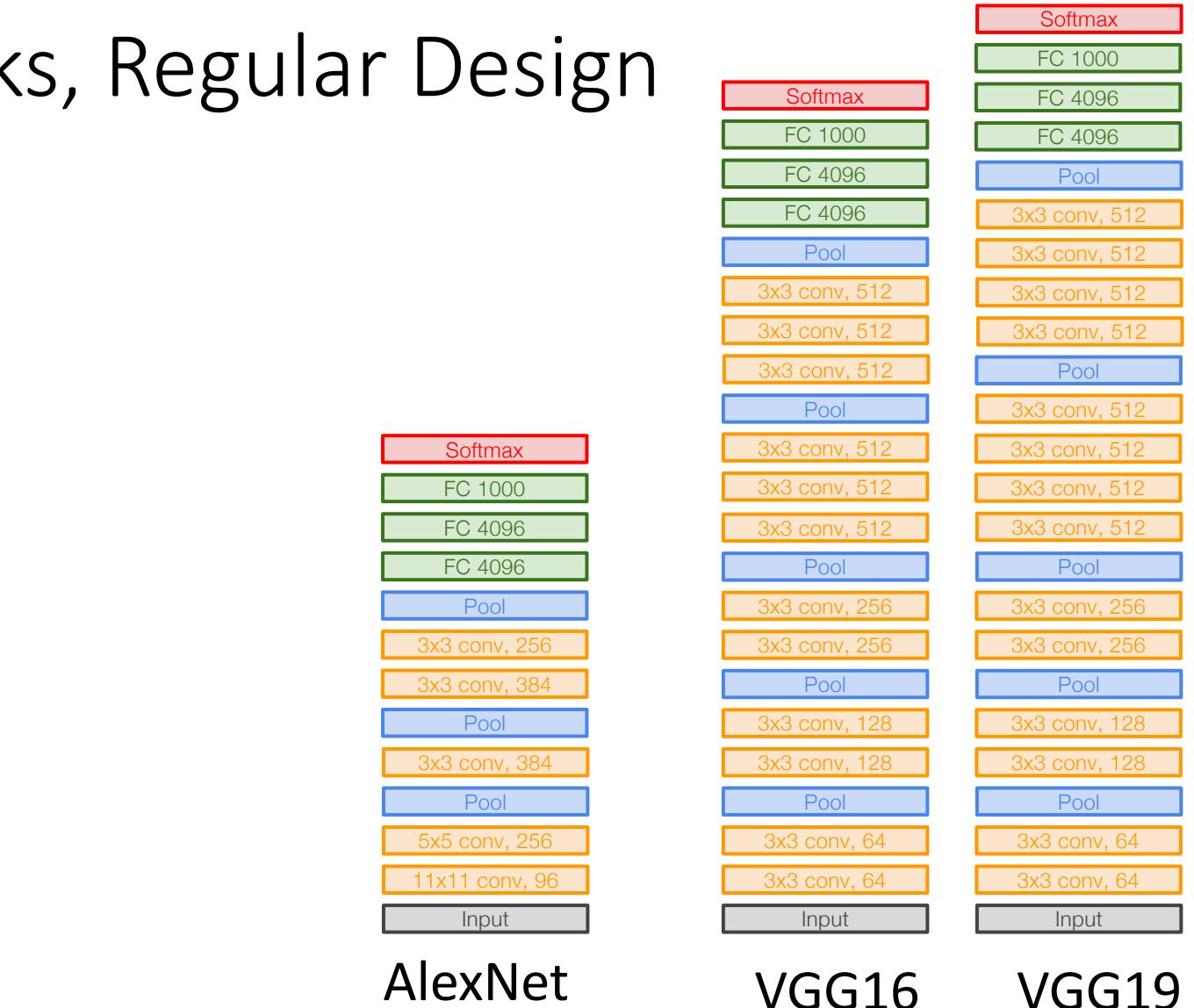
Input:  $C \times 2H \times 2W$

Layer: Conv(3x3,  $C \rightarrow C$ )

Memory: 4HWC

Params:  $9C^2$

FLOPs:  $36HWC^2$



Simonyan and Zissermann, "Very Deep Convolutional Networks for Large-Scale Image Recognition", ICLR 2015

# VGG: Deeper Networks, Regular Design

## VGG Design rules:

All conv are 3x3 stride 1 pad 1

All max pool are 2x2 stride 2

After pool, double #channels

Input:  $C \times 2H \times 2W$

Layer: Conv(3x3,  $C \rightarrow C$ )

Memory: 4HWC

Params:  $9C^2$

FLOPs:  $36HWC^2$

Input:  $2C \times H \times W$

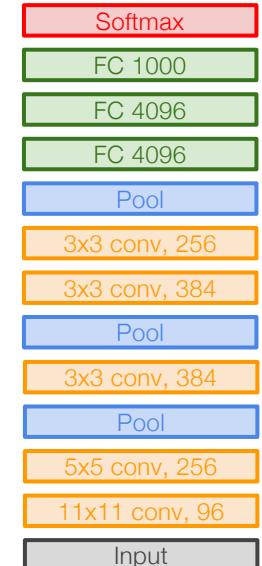
Conv(3x3,  $2C \rightarrow 2C$ )

Memory: 2HWC

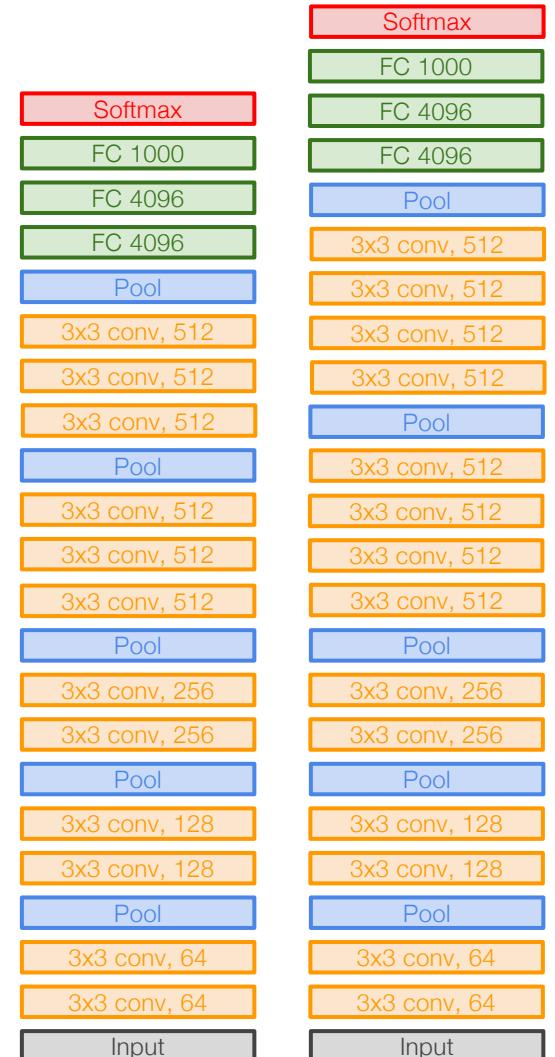
Params:  $36C^2$

FLOPs:  $36HWC^2$

Lecture 8 - 45



AlexNet



VGG16

VGG19

# VGG: Deeper Networks, Regular Design

## VGG Design rules:

All conv are 3x3 stride 1 pad 1

All max pool are 2x2 stride 2

After pool, double #channels

Conv layers at each spatial resolution take the same amount of computation!

Input:  $C \times 2H \times 2W$

Layer: Conv(3x3,  $C \rightarrow C$ )

Memory: 4HWC

Params:  $9C^2$

FLOPs:  $36HWC^2$

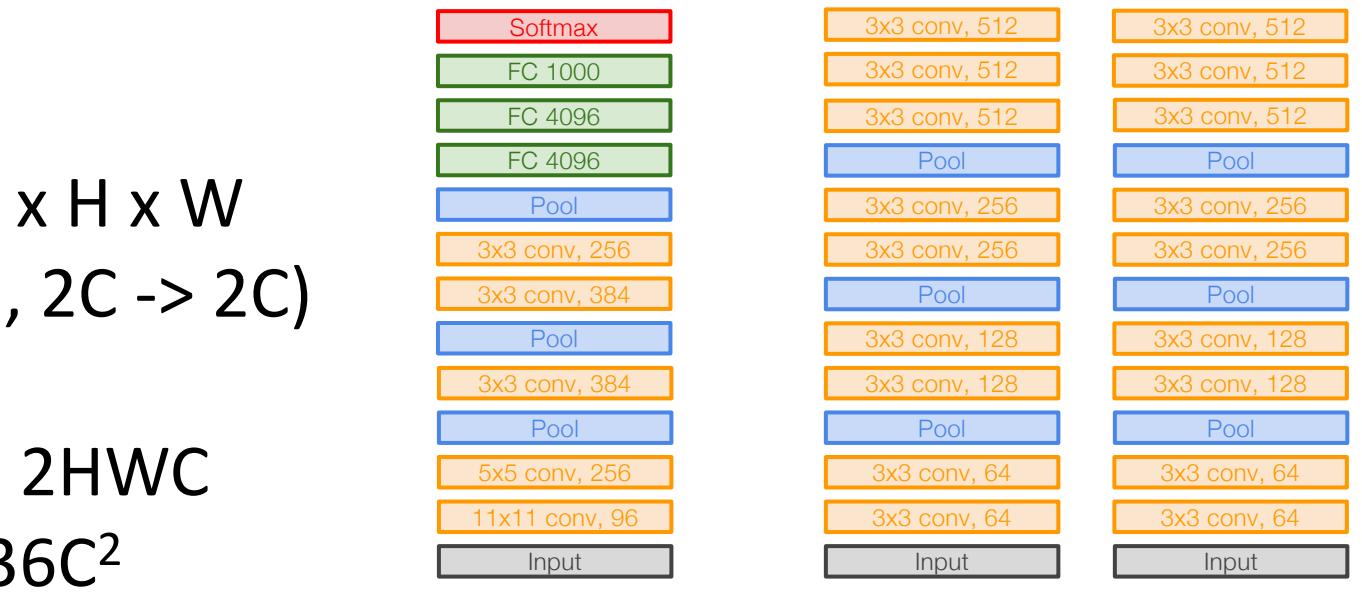
Input:  $2C \times H \times W$

Conv(3x3,  $2C \rightarrow 2C$ )

Memory: 2HWC

Params:  $36C^2$

FLOPs:  $36HWC^2$



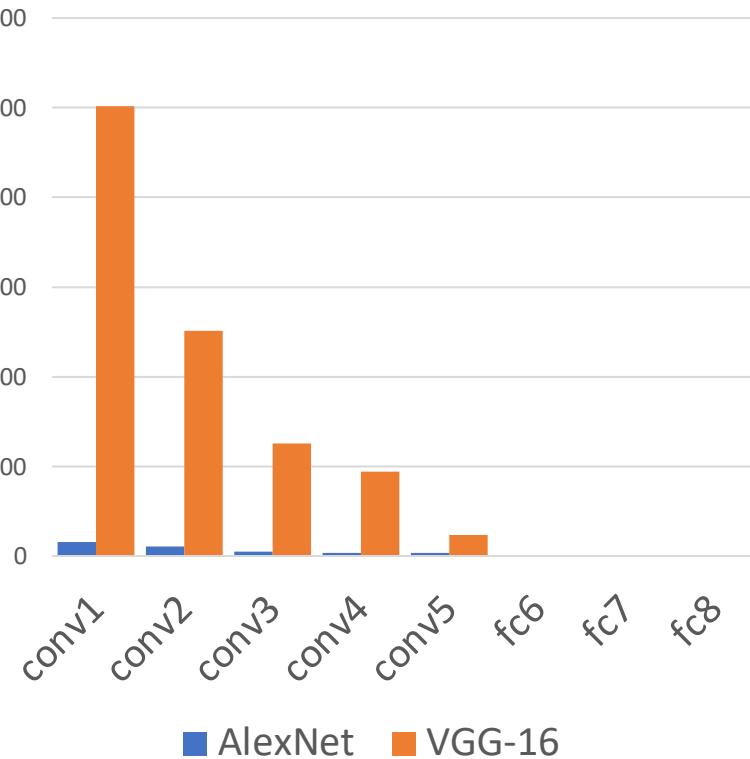
AlexNet

VGG16

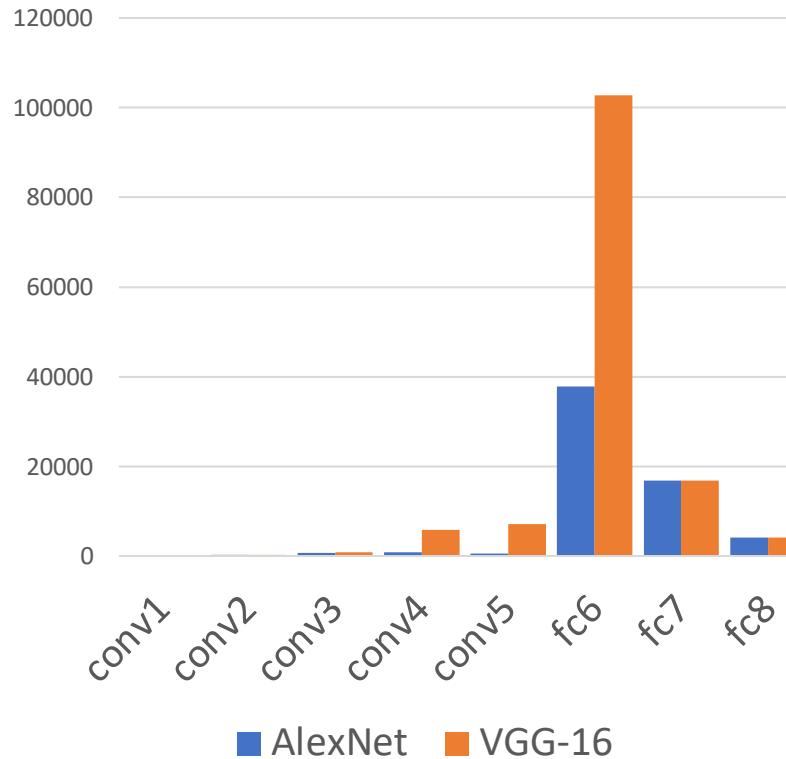
VGG19

# AlexNet vs VGG-16: Much bigger network!

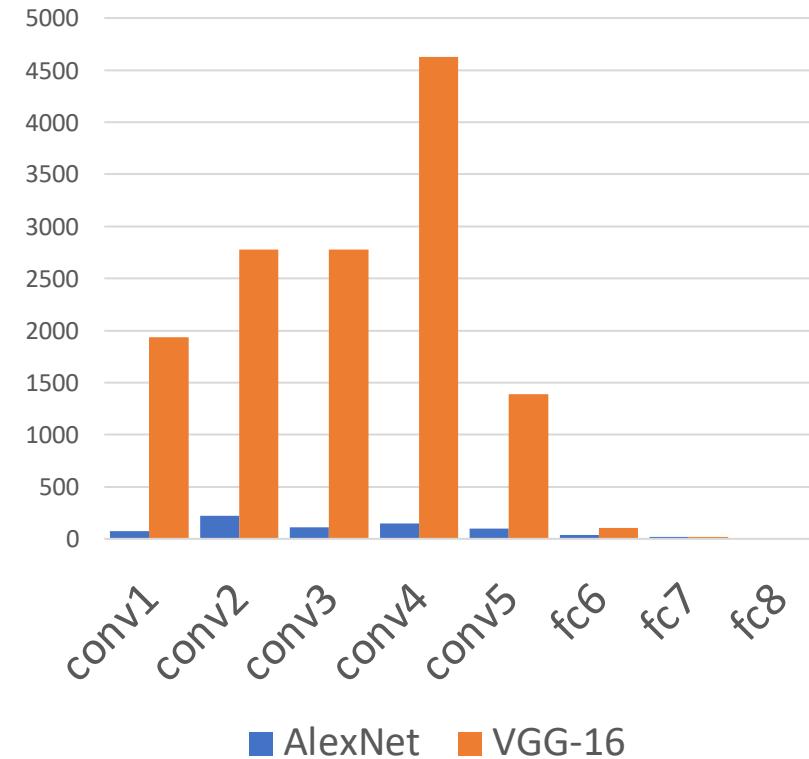
AlexNet vs VGG-16  
(Memory, KB)



AlexNet vs VGG-16  
(Params, M)



AlexNet vs VGG-16  
(MFLOPs)



AlexNet total: 1.9 MB

VGG-16 total: 48.6 MB (25x)

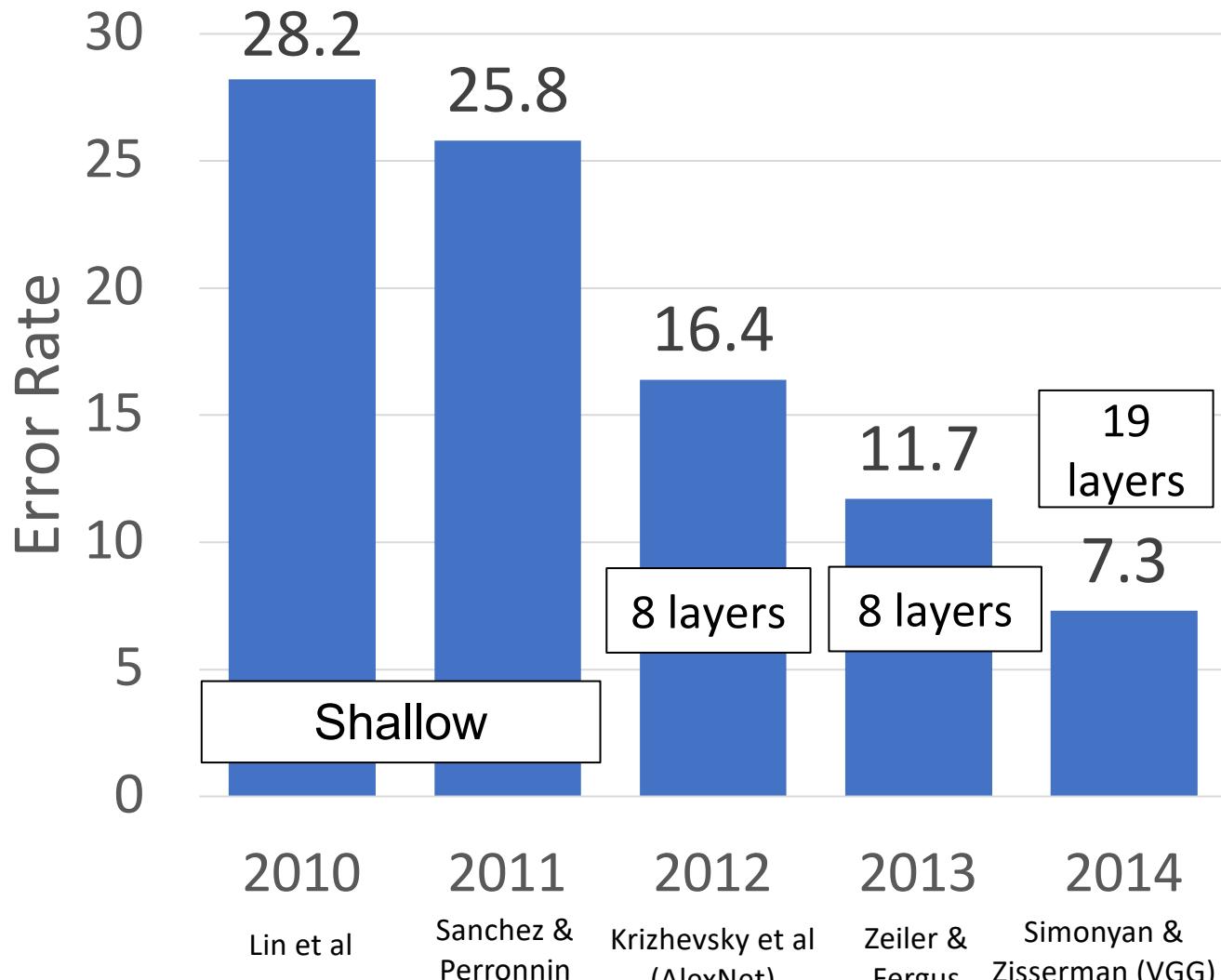
AlexNet total: 61M

VGG-16 total: 138M (2.3x)

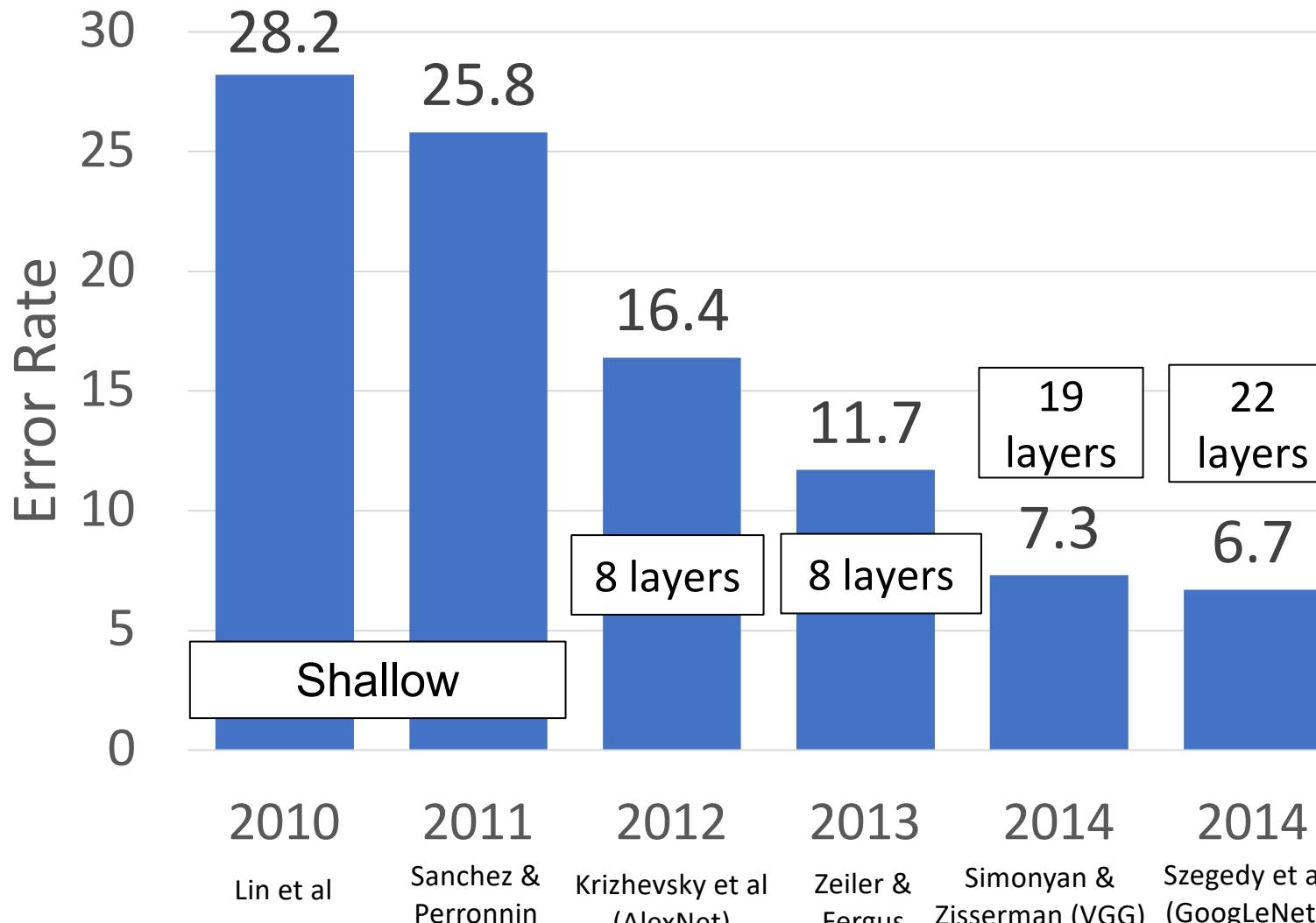
AlexNet total: 0.7 GFLOP

VGG-16 total: 13.6 GFLOP (19.4x)

# ImageNet Classification Challenge

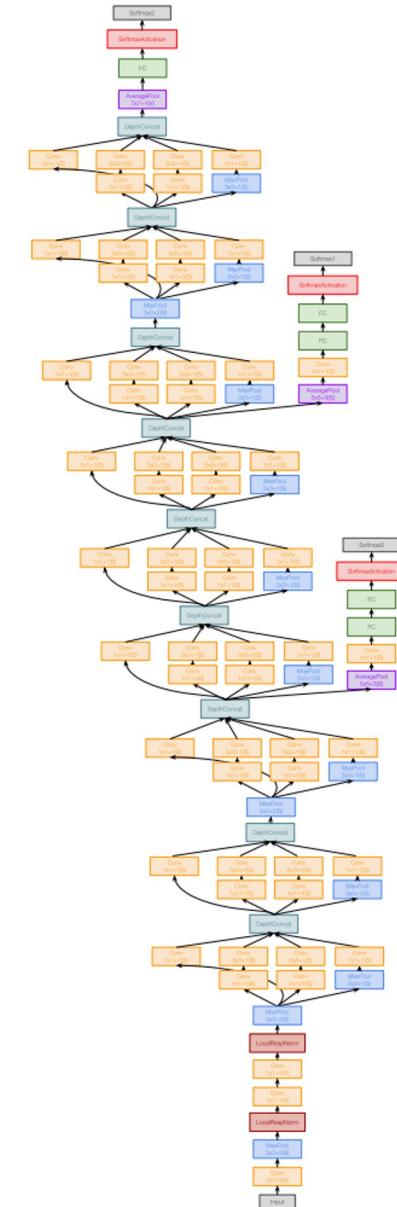


# ImageNet Classification Challenge



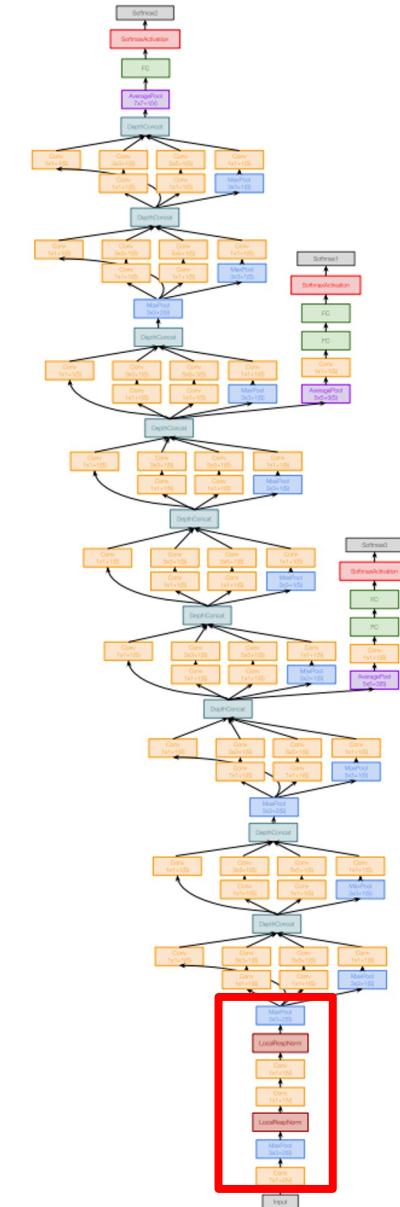
# GoogLeNet: Focus on Efficiency

Many innovations for efficiency: reduce parameter count, memory usage, and computation



# GoogLeNet: Aggressive Stem

**Stem network** at the start aggressively downsamples input  
(Recall in VGG-16: Most of the compute was at the start)



Szegedy et al, "Going deeper with convolutions", CVPR 2015

# GoogLeNet: Aggressive Stem

**Stem network** at the start aggressively downsamples input  
(Recall in VGG-16: Most of the compute was at the start)

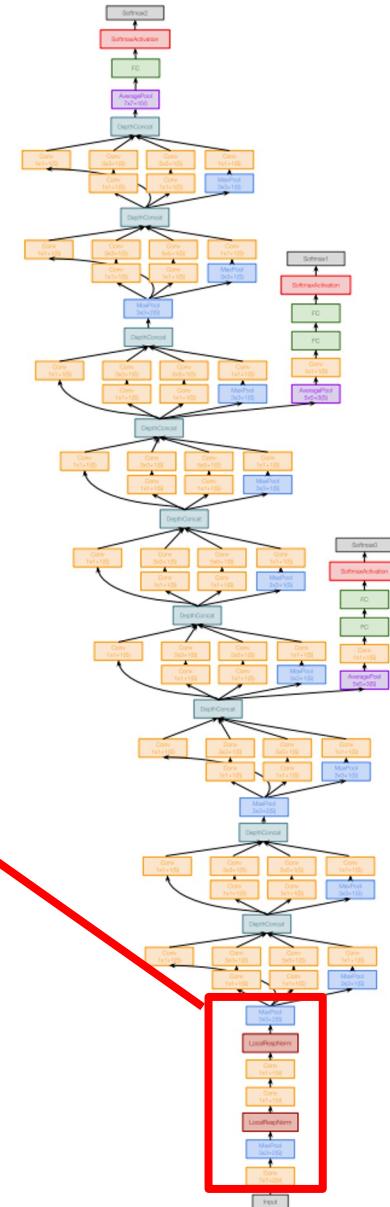
Layer	Input size			Layer			Output size			memory (KB)	params (K)	flop (M)
	C	H / W	filters	kernel	stride	pad	C	H/W				
conv	3	224	64	7	2	3	64	112	3136	9	118	
max-pool	64	112		3	2	1	64	56	784	0	2	
conv	64	56	64	1	1	0	64	56	784	4	13	
conv	64	56	192	3	1	1	192	56	2352	111	347	
max-pool	192	56		3	2	1	192	28	588	0	1	

Total from 224 to 28 spatial resolution:

Memory: 7.5 MB

Params: 124K

MFLOP: 418



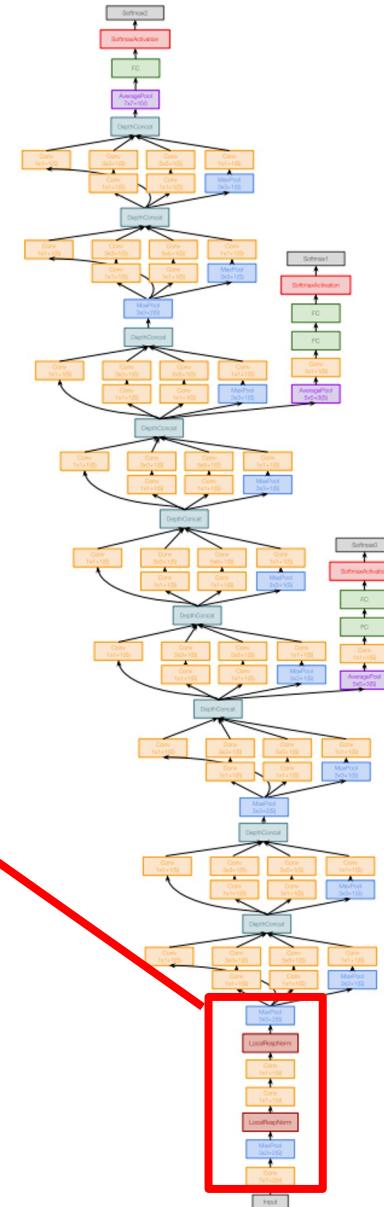
# GoogLeNet: Aggressive Stem

**Stem network** at the start aggressively downsamples input  
(Recall in VGG-16: Most of the compute was at the start)

Layer	Input size		Layer				Output size		memory (KB)	params (K)	flop (M)
	C	H / W	filters	kernel	stride	pad	C	H/W			
conv	3	224	64	7	2	3	64	112	3136	9	118
max-pool	64	112		3	2	1	64	56	784	0	2
conv	64	56	64	1	1	0	64	56	784	4	13
conv	64	56	192	3	1	1	192	56	2352	111	347
max-pool	192	56		3	2	1	192	28	588	0	1

Total from 224 to 28 spatial resolution:  
Memory: 7.5 MB  
Params: 124K  
MFLOP: 418

Compare VGG-16:  
Memory: 42.9 MB (5.7x)  
Params: 1.1M (8.9x)  
MFLOP: 7485 (17.8x)

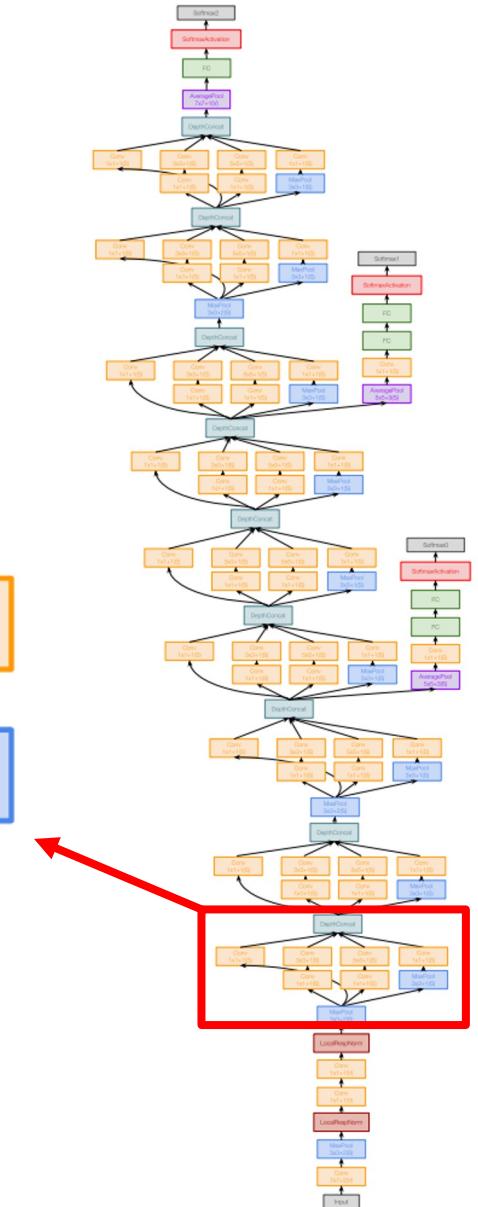
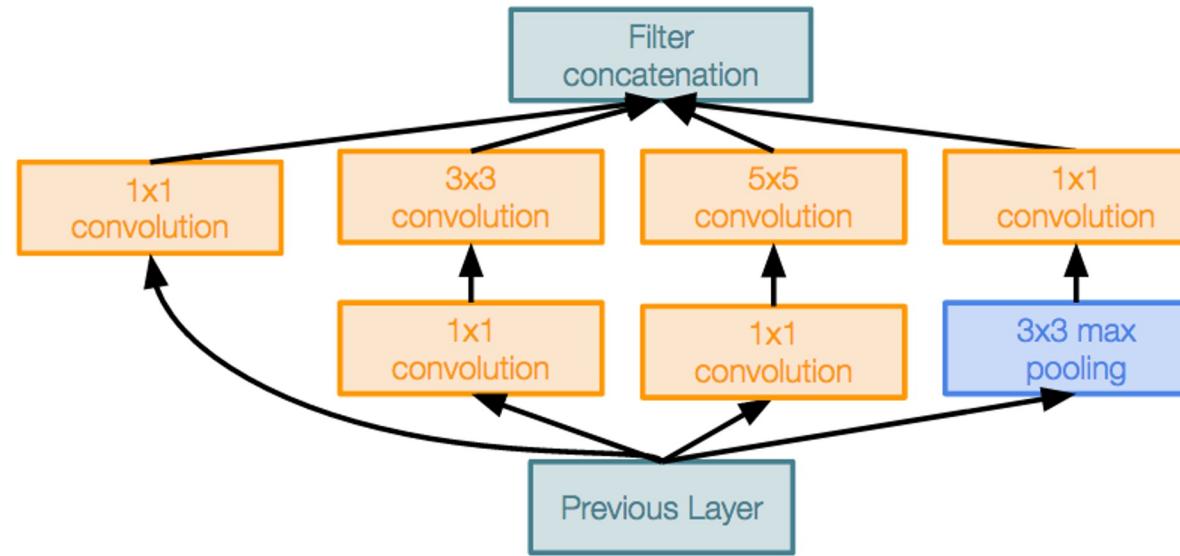


# GoogLeNet: Inception Module

## Inception module

Local unit with parallel branches

Local structure repeated many times throughout the network



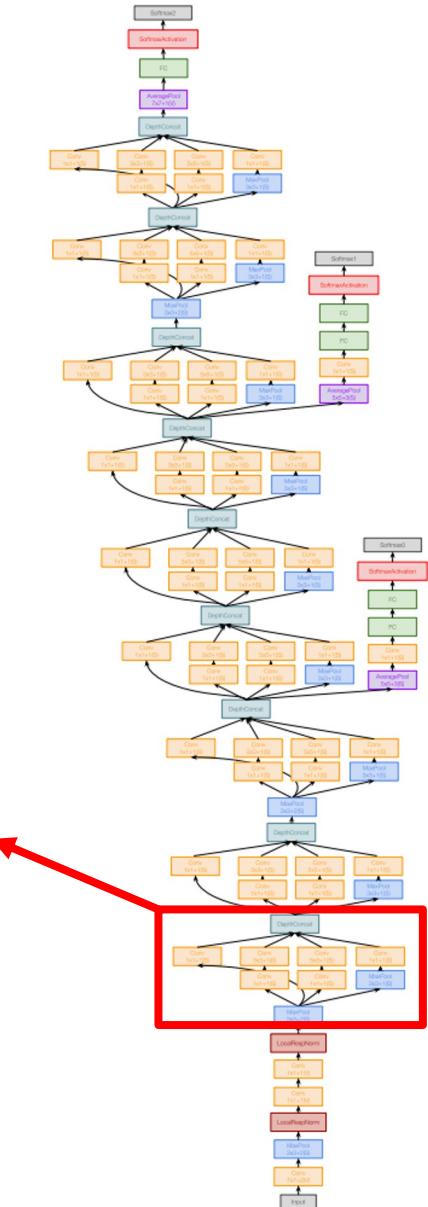
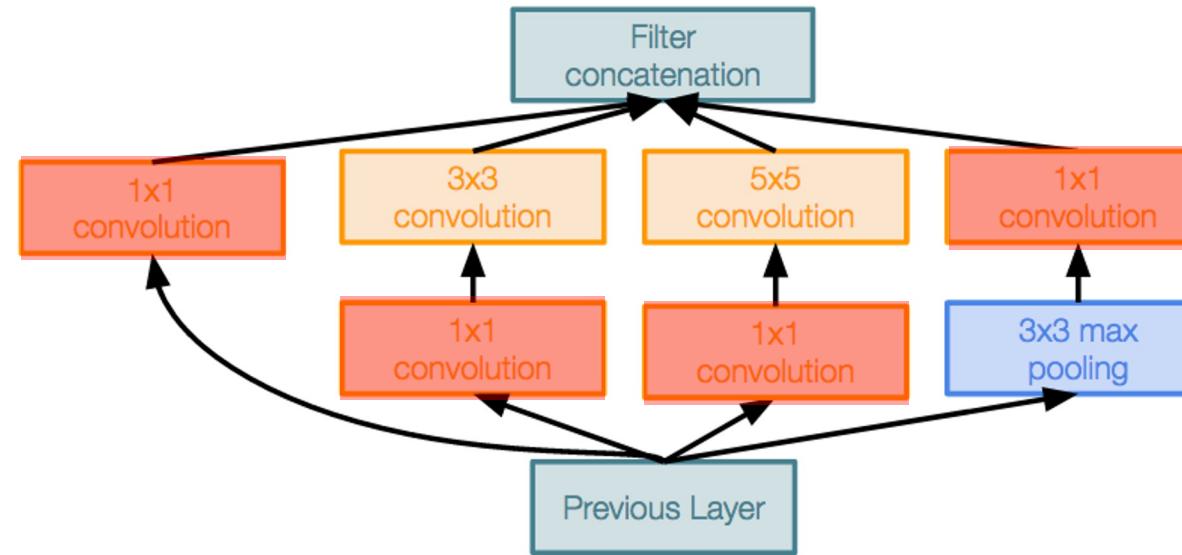
# GoogLeNet: Inception Module

## Inception module

Local unit with parallel branches

Local structure repeated many times throughout the network

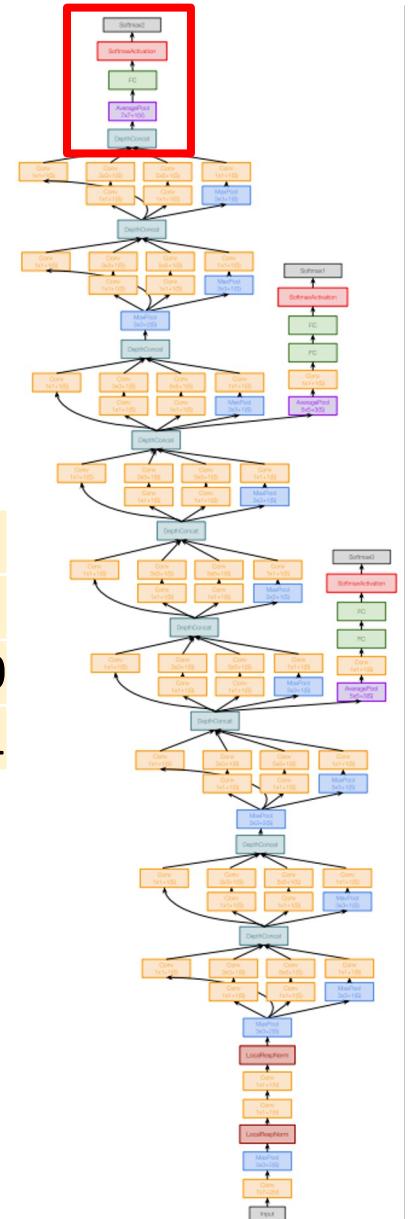
Uses 1x1 “Bottleneck” layers to reduce channel dimension (dimensionality reduction) before expensive conv (we will revisit this with ResNet!)



# GoogLeNet: Global Average Pooling

No large FC layers at the end! Instead uses **global average pooling** to collapse spatial dimensions, and one linear layer to produce class scores  
(Recall VGG-16: Most parameters were in the FC layers!)

	Input size		Layer				Output size				
Layer	C	H/W	filters	kernel	stride	pad	C	H/W	memory (KB)	params (k)	flop (M)
avg-pool	1024	7		7	1	0	1024	1	4	0	0
fc	1024		1000				1000		0	1025	1



# GoogLeNet: Global Average Pooling

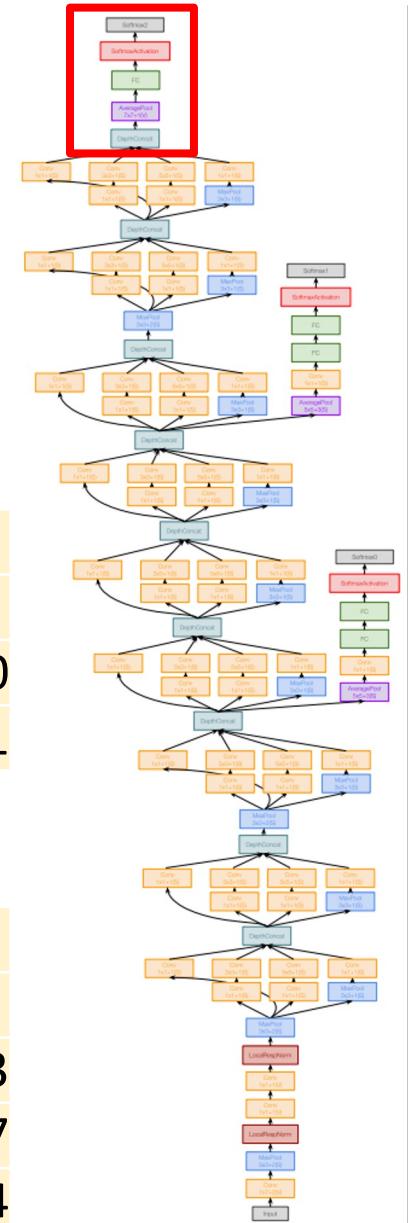
No large FC layers at the end! Instead uses **global average pooling** to collapse spatial dimensions, and one linear layer to produce class scores  
 (Recall VGG-16: Most parameters were in the FC layers!)

e.g. GAP:  $1024 \times 7 \times 7 \rightarrow 1024 \times 1$

	Input size		Layer				Output size				
Layer	C	H/W	filters	kernel	stride	pad	C	H/W	memory (KB)	params (k)	flop (M)
avg-pool	1024	7		7	1	0	1024	1	4	0	0
fc	1024		1000				1000		0	1025	1

Compare with VGG-16:

Layer	C	H/W	filters	kernel	stride	pad	C	H/W	memory (KB)	params (K)	flop (M)
flatten	512	7					25088		98		
fc6	25088		4096				4096		16	102760	103
fc7	4096		4096				4096		16	16777	17
fc8	4096		1000				1000		4	4096	4

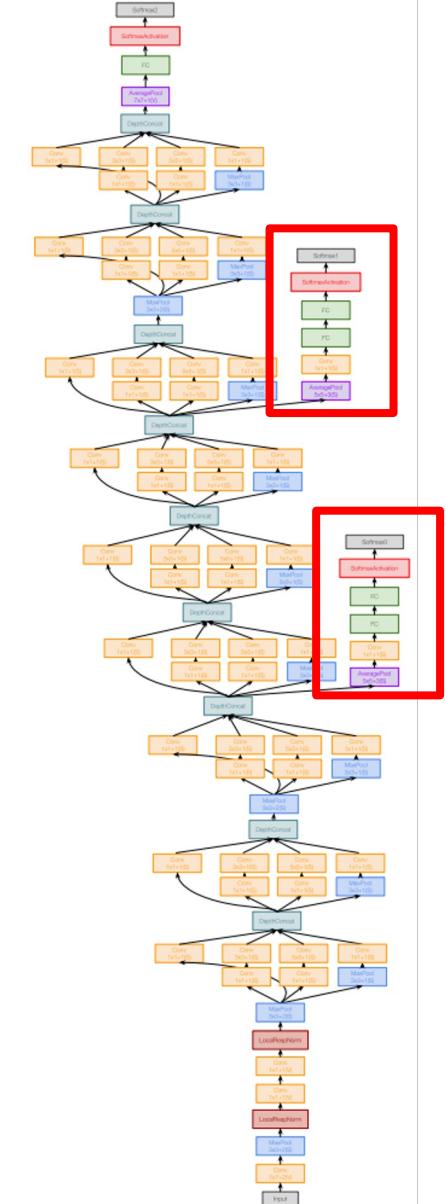


# GoogLeNet: Auxiliary Classifiers

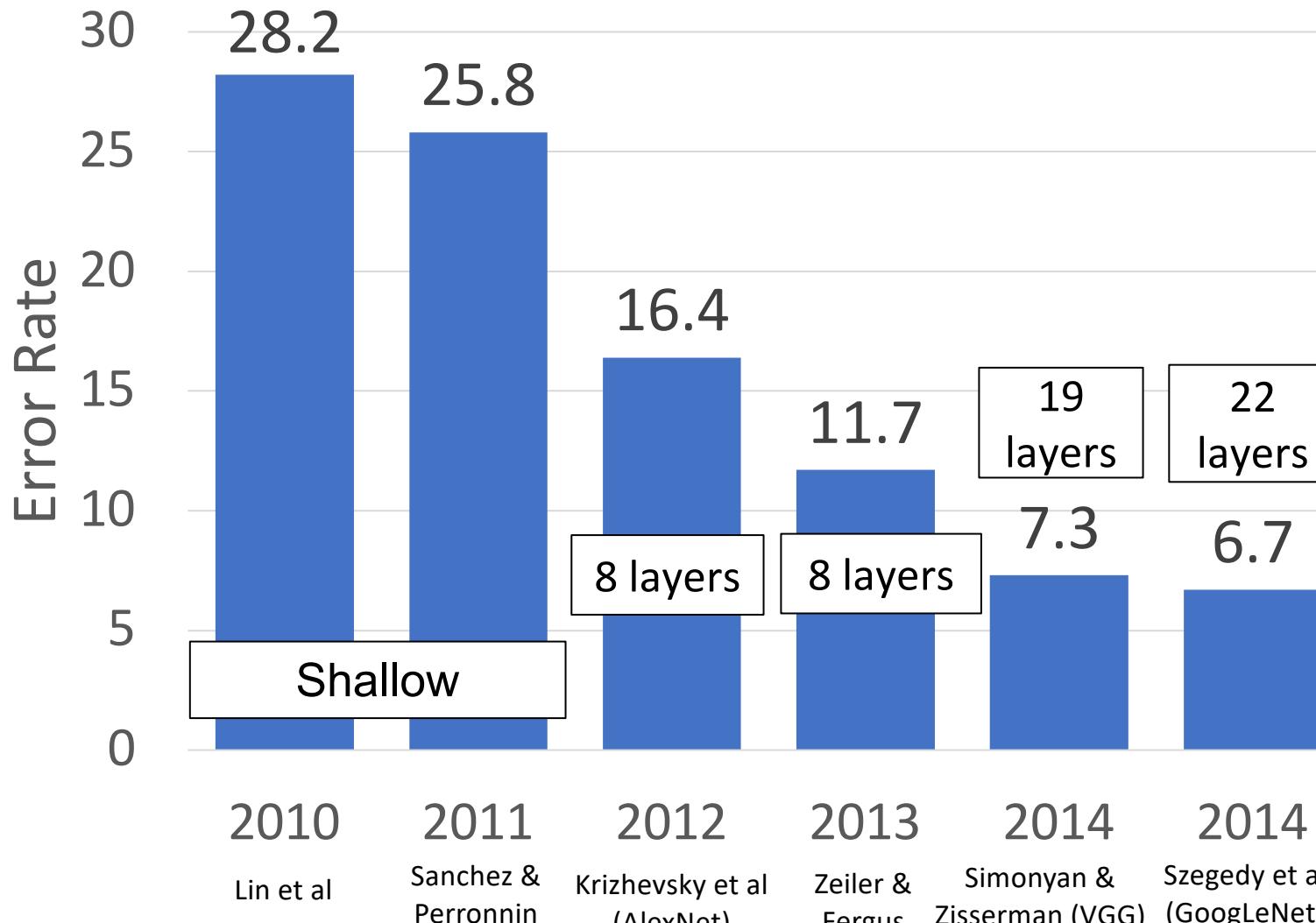
Training using loss at the end of the network didn't work well:  
Network is too deep, gradients don't propagate cleanly

As a hack, attach “auxiliary classifiers” at several intermediate points in the network that also try to classify the image and receive loss

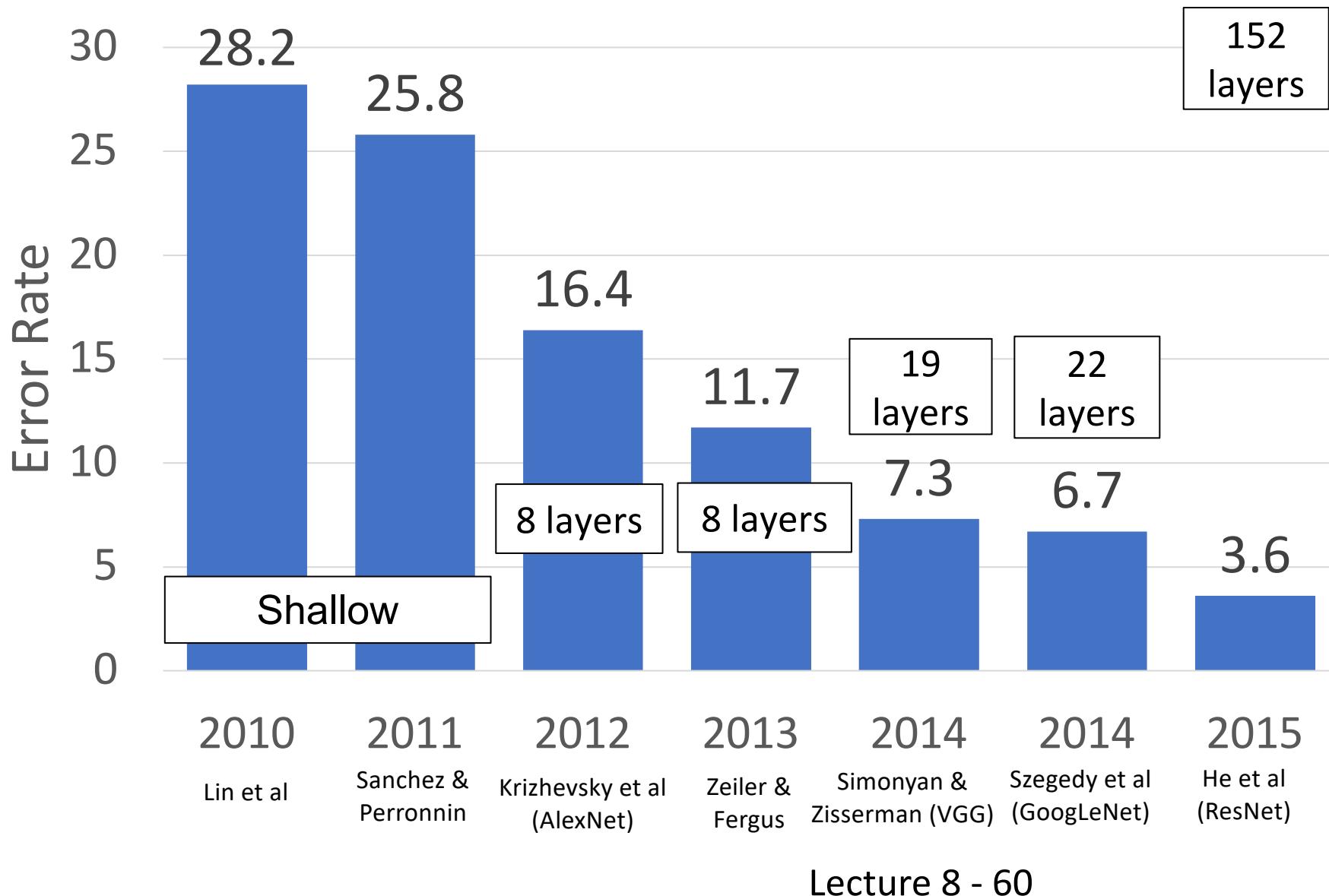
GoogLeNet was before batch normalization! With BatchNorm no longer need to use this trick



# ImageNet Classification Challenge



# ImageNet Classification Challenge



Kaiming He



Previously at Microsoft Research Asia, then at Facebook AI Research (Meta), then Associate Prof at MIT!

Won the most number of best paper awards in computer vision history

- PAMI Young Researcher Award in 2018
- Best Paper Award in CVPR 2009, CVPR 2016, ICCV 2017,
- Best Student Paper Award in ICCV 2017
- Best Paper Honorable Mention in ECCV 2018, CVPR 2021,
- Everingham Prize in ICCV 2021.
- His paper on Deep Residual Networks (ResNets) is the most cited paper in all areas in Google Scholar Metrics 2020.



Kaiming He

Research Scientist, Facebook AI Research (FAIR)

Verified email at fb.com - [Homepage](#)

Computer Vision Machine Learning

FOLLOW

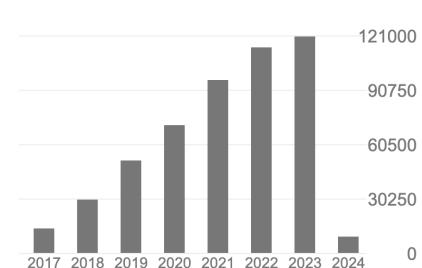
Cited by [VIEW ALL](#)

All Since 2019

Citations 526963 465333

h-index 68 64

i10-index 72 71



TITLE	CITED BY	YEAR
Deep Residual Learning for Image Recognition K He, X Zhang, S Ren, J Sun Computer Vision and Pattern Recognition (CVPR), 2016	202653	2016
Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks S Ren, K He, R Girshick, J Sun Neural Information Processing Systems (NIPS), 2015	70319	2015
Mask R-CNN K He, G Gkioxari, P Dollár, R Girshick International Conference on Computer Vision (ICCV), 2017	33379	2017
Focal Loss for Dense Object Detection TY Lin, P Goyal, R Girshick, K He, P Dollár International Conference on Computer Vision (ICCV), 2017	27448	2017
Feature Pyramid Networks for Object Detection TY Lin, P Dollár, R Girshick, K He, B Hariharan, S Belongie	23991	2017

After joining MIT, he became the most cited professor at MIT

# Residual Networks

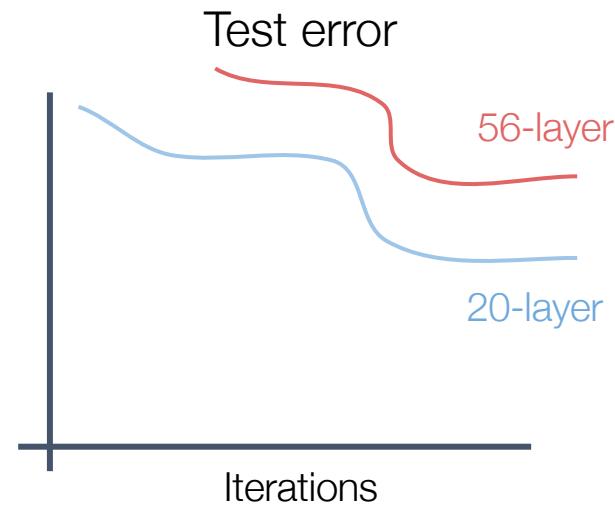
Once we have Batch Normalization, we can train networks with 10+ layers.  
What happens as we go deeper?

# Residual Networks

Once we have Batch Normalization, we can train networks with 10+ layers.  
What happens as we go deeper like 100 layers?

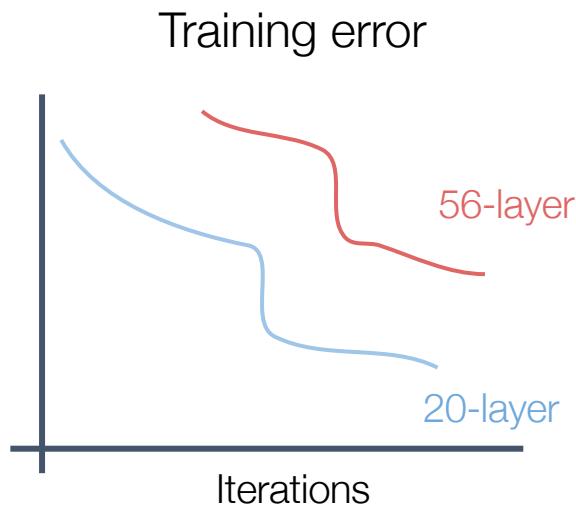
Deeper model does worse than shallow model!

Initial guess: Deep model is **overfitting** since it is much bigger than the other model



# Residual Networks

Once we have Batch Normalization, we can train networks with 10+ layers.  
What happens as we go deeper like 100 layers?



In fact the deep model seems to be **underfitting** since it also performs worse than the shallow model on the training set! It is actually **underfitting**

# Residual Networks

A deeper model can emulate a shallower model: copy layers from shallower model, set extra layers to identity

Thus deeper models should do at least as good as shallow models

**Hypothesis:** This is an optimization problem. Deeper models are harder to optimize, and in particular don't learn identity functions to emulate shallow models

# Residual Networks

A deeper model can emulate a shallower model: copy layers from shallower model, set extra layers to identity

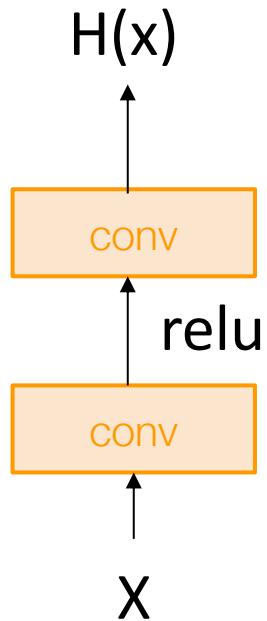
Thus deeper models should do at least as good as shallow models

**Hypothesis:** This is an optimization problem. Deeper models are harder to optimize, and in particular don't learn identity functions to emulate shallow models

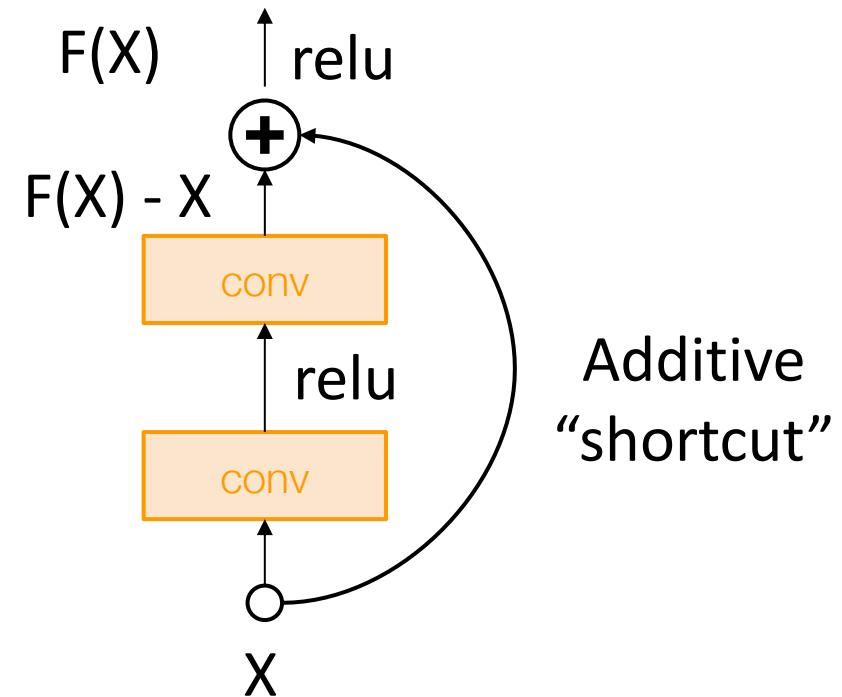
**Solution:** Change the network so learning identity functions with extra layers is easy!

# Residual Networks

**Solution:** Change the network so learning identity functions with extra layers is easy!



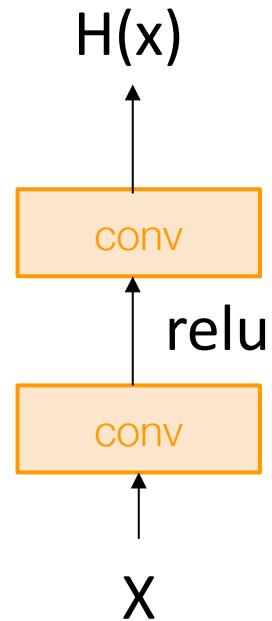
“Plain” block



Residual Block

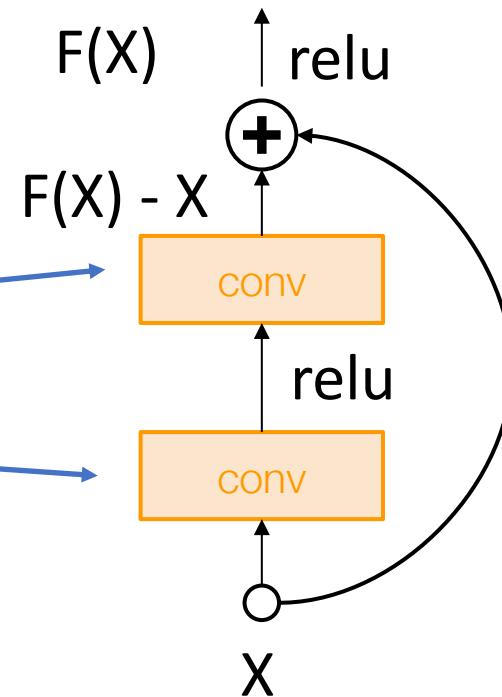
# Residual Networks

**Solution:** Change the network so learning identity functions with extra layers is easy!



“Plain” block

If you set these to  
0, the whole block  
will compute the  
identity function!



Residual Block

# Residual Block Implementation

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l

class Residual(nn.Module): #@save
    """The Residual block of ResNet."""
    def __init__(self, input_channels, num_channels, use_1x1conv=False,
                 strides=1):
        super().__init__()
        self.conv1 = nn.Conv2d(input_channels, num_channels, kernel_size=3,
                             padding=1, stride=strides)
        self.conv2 = nn.Conv2d(num_channels, num_channels, kernel_size=3,
                             padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2d(input_channels, num_channels,
                                 kernel_size=1, stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm2d(num_channels)
        self.bn2 = nn.BatchNorm2d(num_channels)

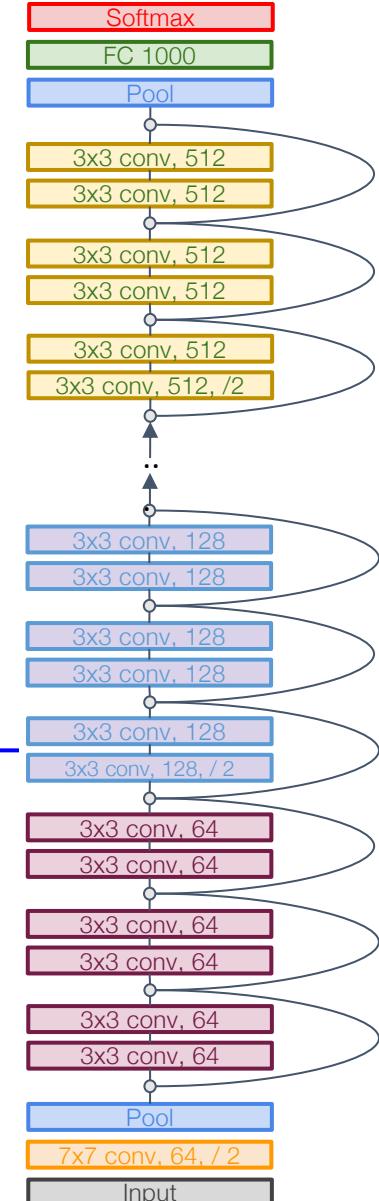
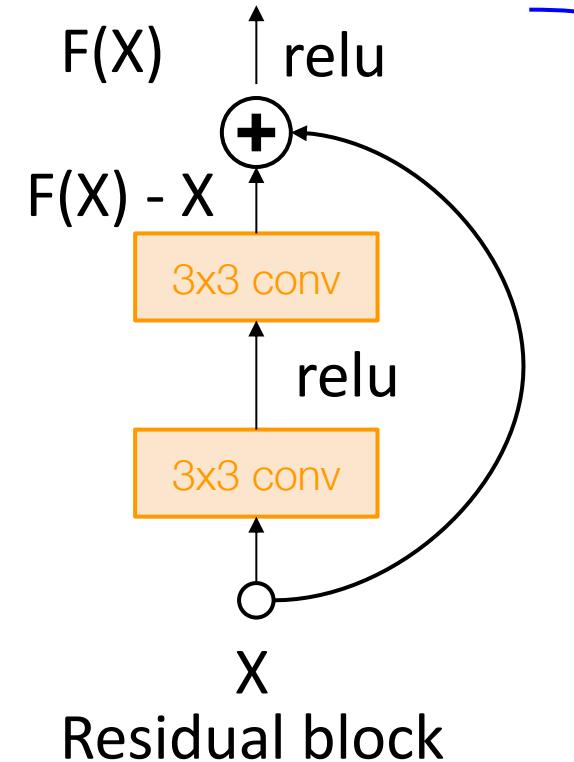
    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)
```

# Residual Networks

A residual network is a stack of many residual blocks

Regular design, like VGG: each residual block has two 3x3 conv

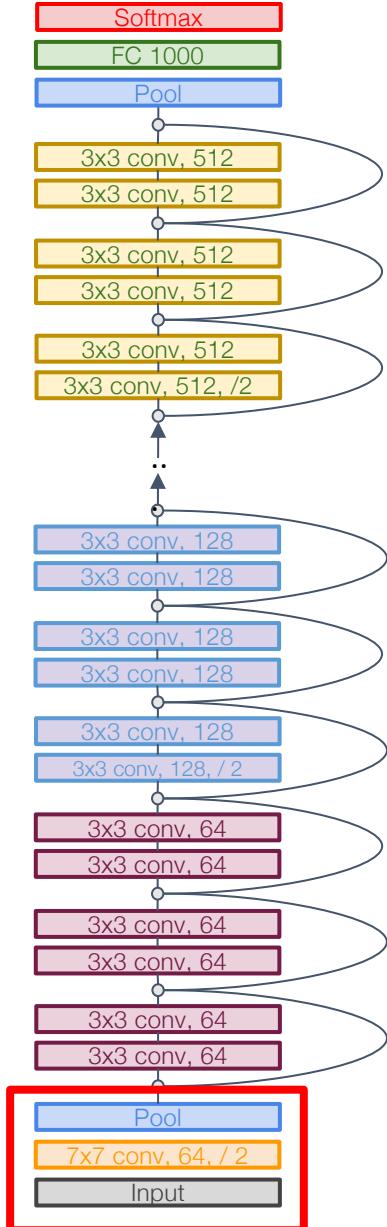
Network is divided into **stages**: the first block of each stage halves the resolution (with stride-2 conv) and doubles the number of channels



# Residual Networks

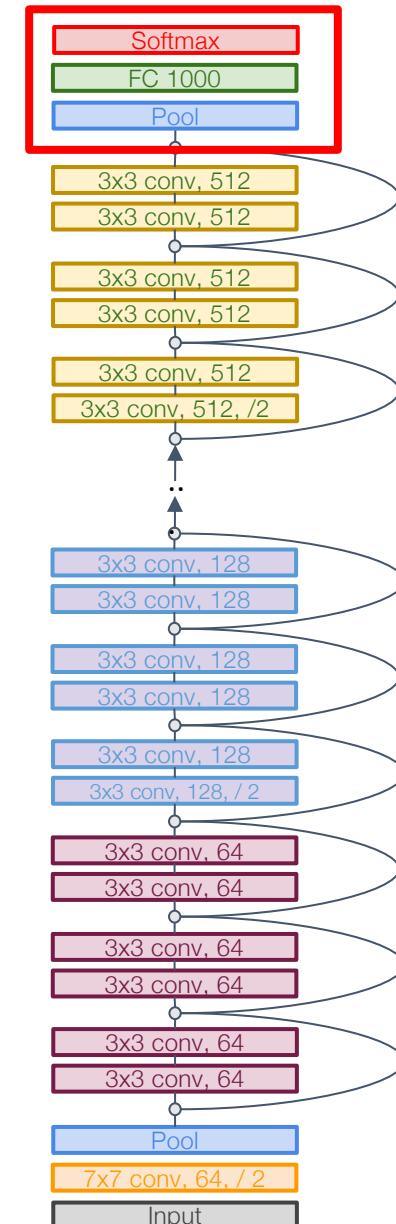
Uses the same aggressive **stem** as GoogleNet to downsample the input 4x before applying residual blocks:

Layer	Input size		Layer				Output size		params (k)	flop (M)
	C	H/W	filters	kernel	stride	pad	C	H/W		
conv	3	224	64	7	2	3	64	112	3136	9 118
max-pool	64	112		3	2	1	64	56	784	0 2



# Residual Networks

Like GoogLeNet, no big fully-connected-layers: instead, use **global average pooling** and a single linear layer at the end



# Residual Networks

## ResNet-18:

Stem: 1 conv layer

Stage 1 (C=64): 2 res. block = 4 conv

Stage 2 (C=128): 2 res. block = 4 conv

Stage 3 (C=256): 2 res. block = 4 conv

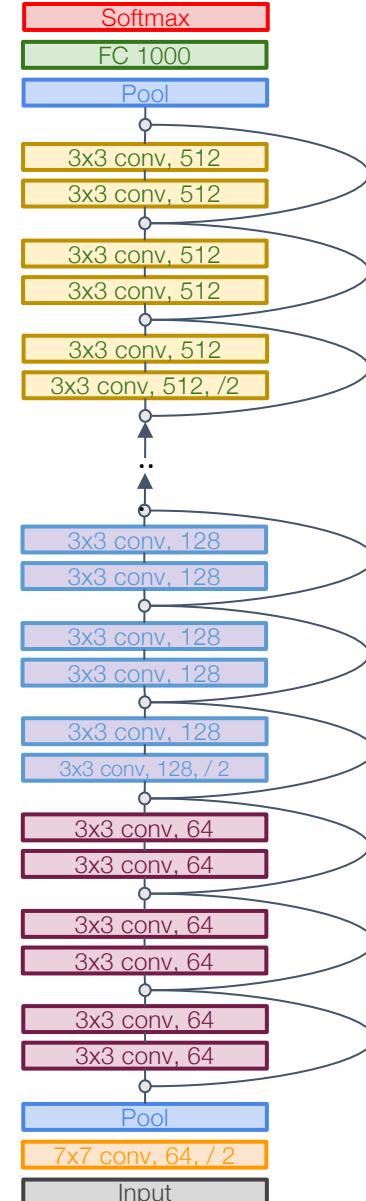
Stage 4 (C=512): 2 res. block = 4 conv

Linear

ImageNet top-5 error: 10.92

GFLOP: 1.8

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016  
Error rates are 224x224 single-crop testing, reported by [torchvision](#)



# Residual Networks

## ResNet-18:

Stem: 1 conv layer

Stage 1 (C=64): 2 res. block = 4 conv

Stage 2 (C=128): 2 res. block = 4 conv

Stage 3 (C=256): 2 res. block = 4 conv

Stage 4 (C=512): 2 res. block = 4 conv

Linear

ImageNet top-5 error: 10.92

GFLOP: 1.8

## ResNet-34:

Stem: 1 conv layer

Stage 1: 3 res. block = 6 conv

Stage 2: 4 res. block = 8 conv

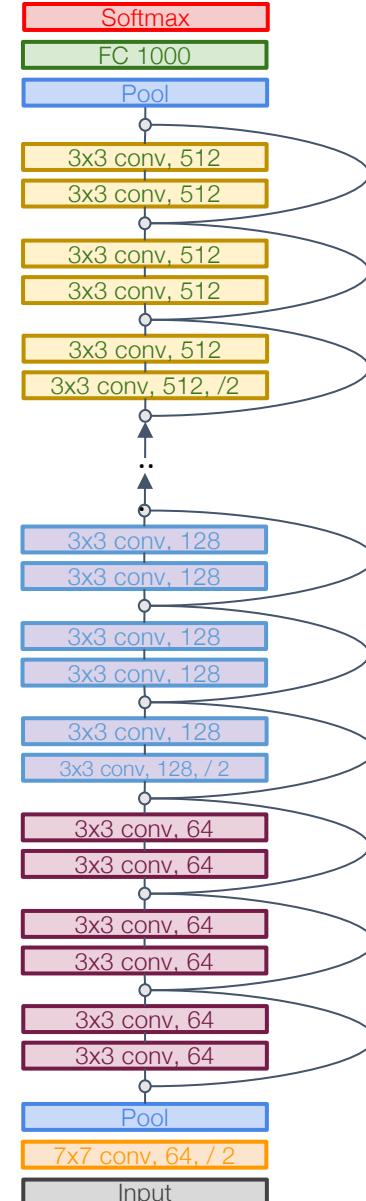
Stage 3: 6 res. block = 12 conv

Stage 4: 3 res. block = 6 conv

Linear

ImageNet top-5 error: 8.58

GFLOP: 3.6



He et al, "Deep Residual Learning for Image Recognition", CVPR 2016  
Error rates are 224x224 single-crop testing, reported by [torchvision](#)

# Residual Networks

## ResNet-18:

Stem: 1 conv layer

Stage 1 (C=64): 2 res. block = 4 conv

Stage 2 (C=128): 2 res. block = 4 conv

Stage 3 (C=256): 2 res. block = 4 conv

Stage 4 (C=512): 2 res. block = 4 conv

Linear

ImageNet top-5 error: 10.92

GFLOP: 1.8

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016  
Error rates are 224x224 single-crop testing, reported by [torchvision](#)

## ResNet-34:

Stem: 1 conv layer

Stage 1: 3 res. block = 6 conv

Stage 2: 4 res. block = 8 conv

Stage 3: 6 res. block = 12 conv

Stage 4: 3 res. block = 6 conv

Linear

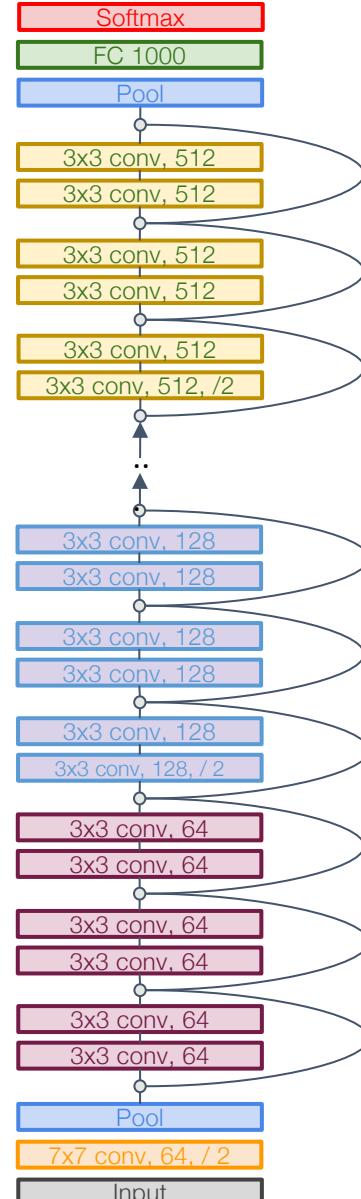
ImageNet top-5 error: 8.58

GFLOP: 3.6

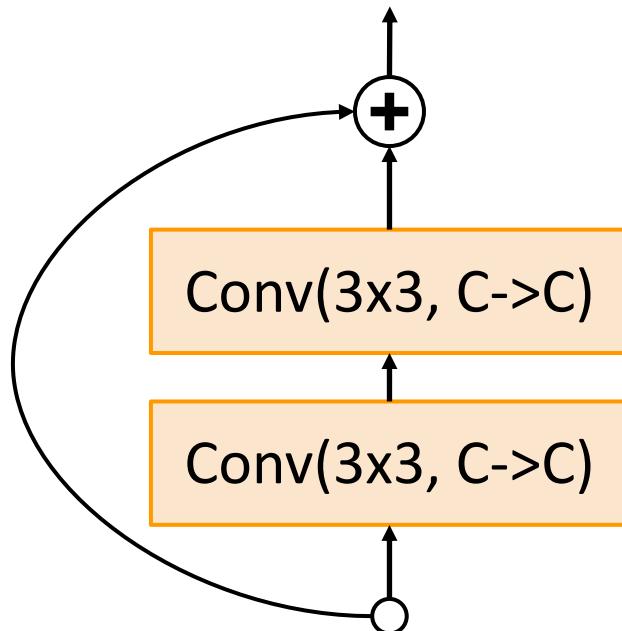
## VGG-16:

ImageNet top-5 error: 9.62

GFLOP: 13.6

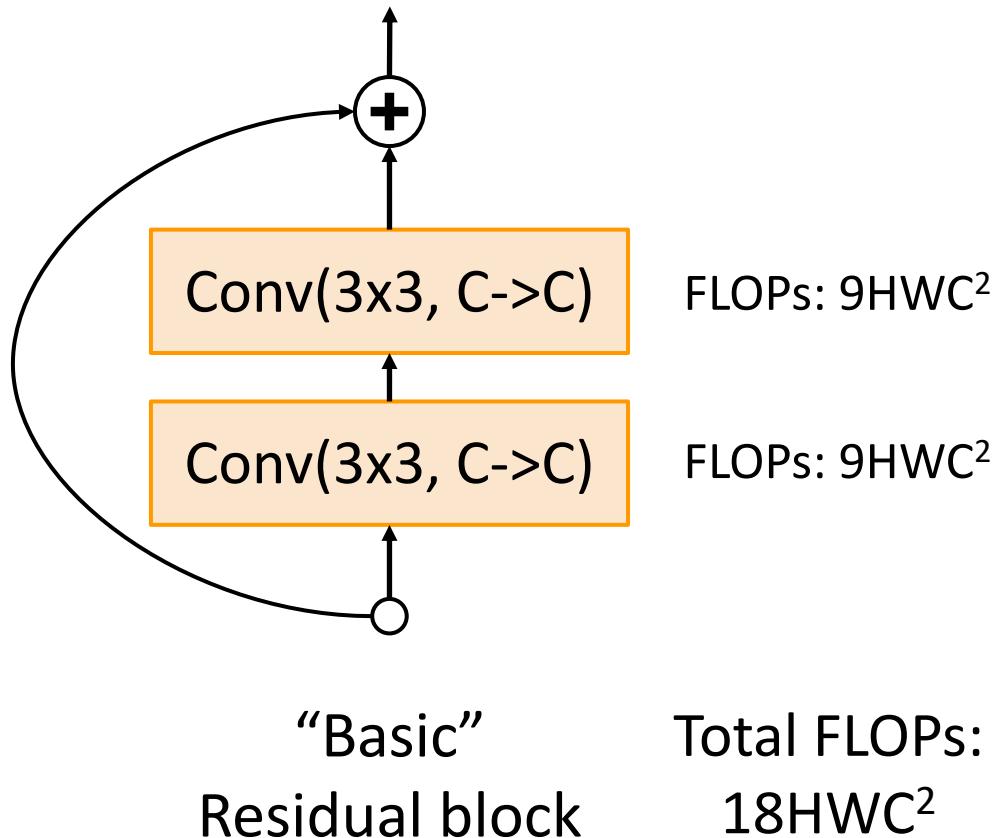


# Residual Networks: Basic Block

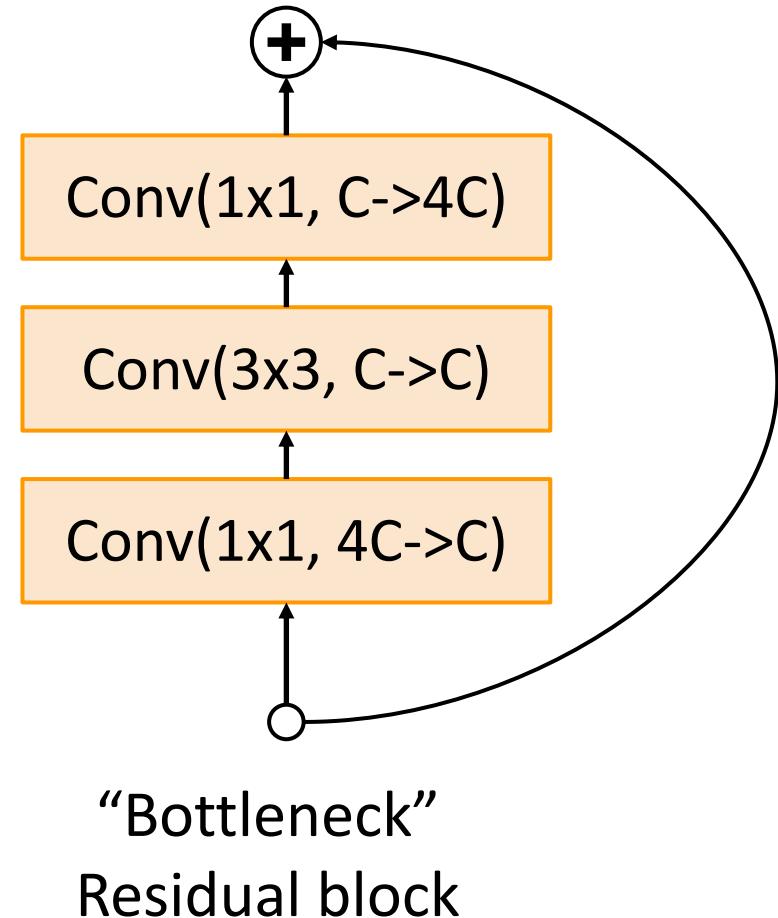
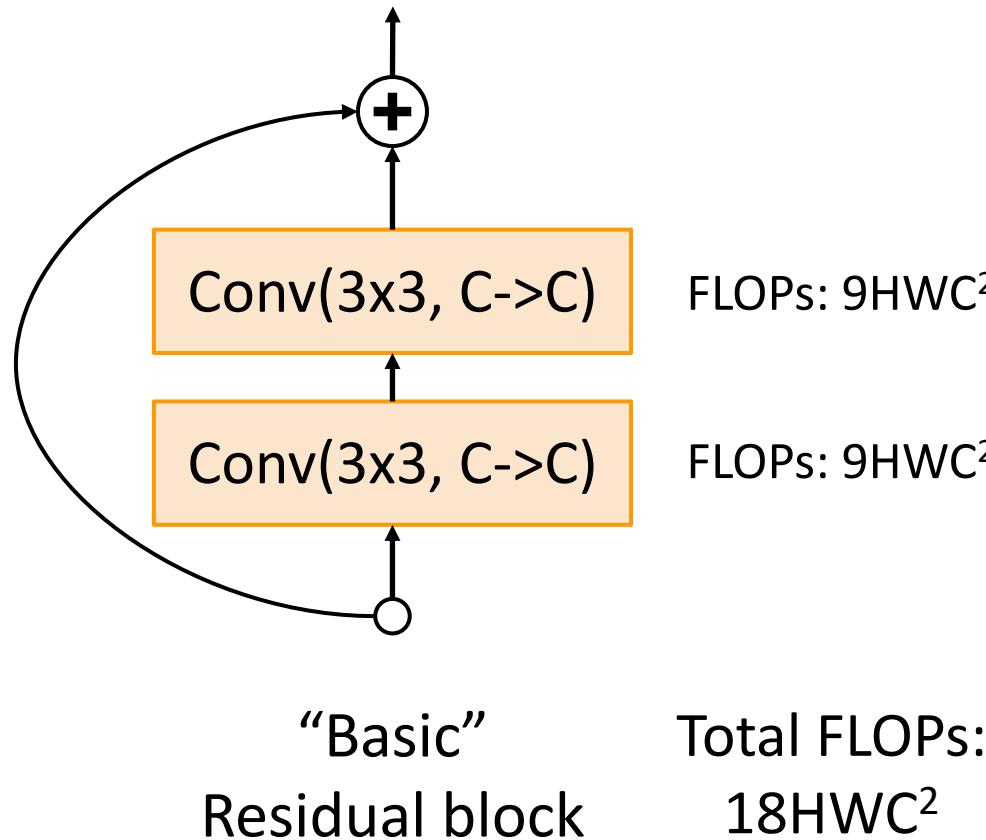


“Basic”  
Residual block

# Residual Networks: Basic Block

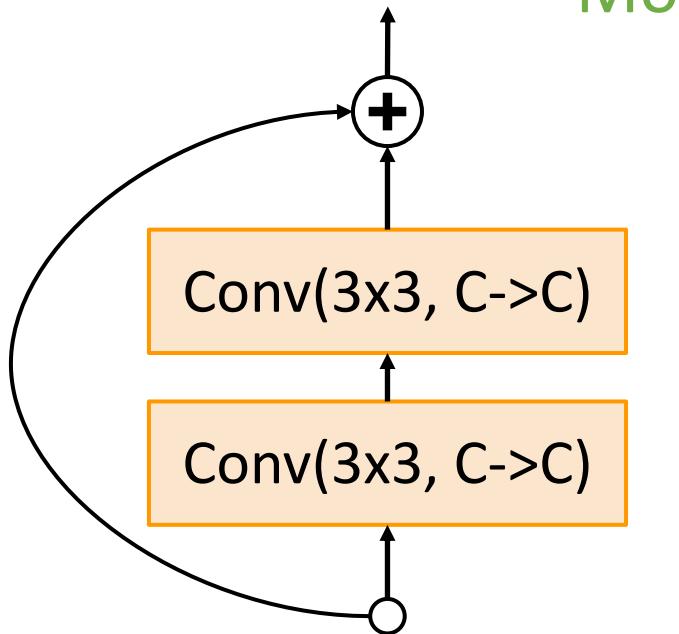


# Residual Networks: Bottleneck Block



# Residual Networks: Bottleneck Block

More layers, less computational cost!



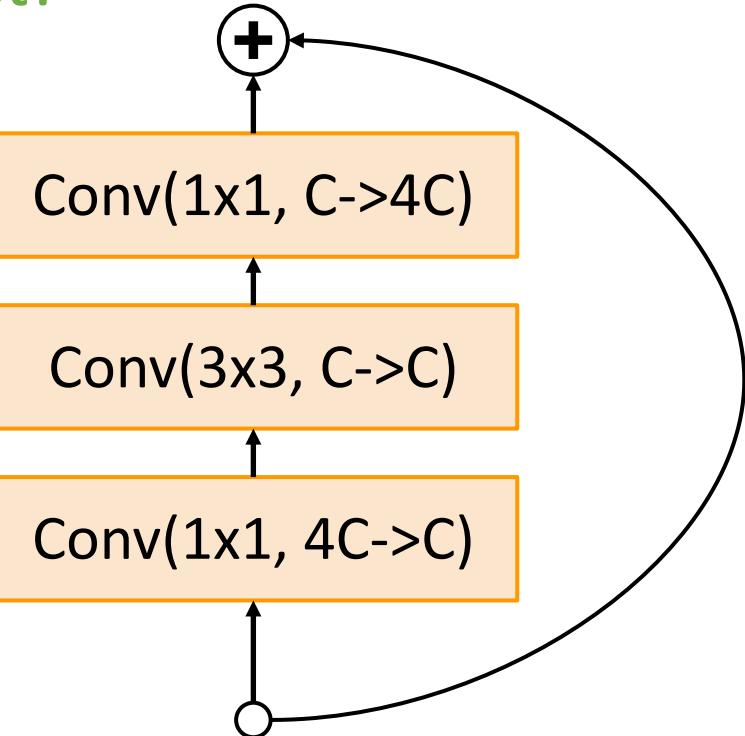
"Basic"  
Residual block

Total FLOPs:  
 $18HWC^2$

FLOPs:  $9HWC^2$

FLOPs:  $9HWC^2$

FLOPs:  $4HWC^2$



"Bottleneck"  
Residual block

Total FLOPs:  
 $17HWC^2$

# Residual Networks



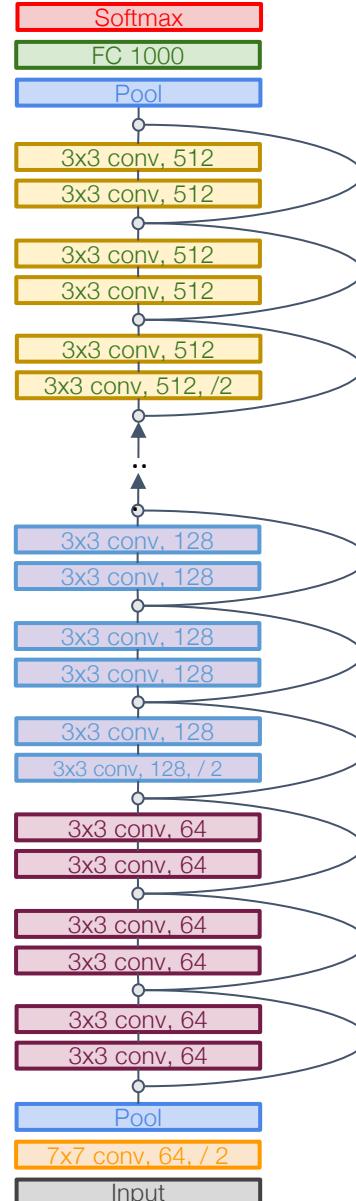
			Stage 1		Stage 2		Stage 3		Stage 4		FC	ImageNet	
	Block type	Stem layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	GFLOP	top-5 error	
ResNet-18	Basic	1	2	4	2	4	2	4	2	4	1	1.8	10.92
ResNet-34	Basic	1	3	6	4	8	6	12	3	6	1	3.6	8.58

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016  
 Error rates are 224x224 single-crop testing, reported by [torchvision](#)

# Residual Networks

ResNet-50 is the same as ResNet-34, but replaces Basic blocks with Bottleneck Blocks.  
This is a great baseline architecture for many tasks even today!

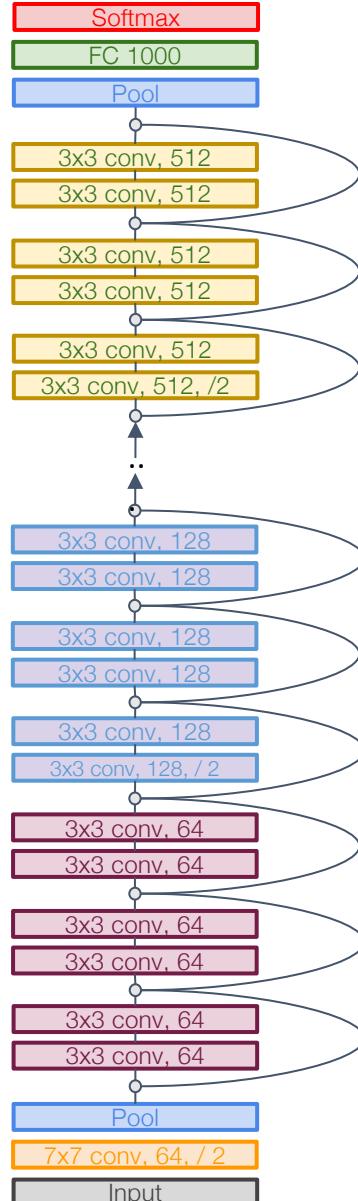
			Stage 1		Stage 2		Stage 3		Stage 4		FC	ImageNet
	Block type	Stem layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	layers	GFLOP top-5 error
ResNet-18	Basic	1	2	4	2	4	2	4	2	4	1	1.8 10.92
ResNet-34	Basic	1	3	6	4	8	6	12	3	6	1	3.6 8.58
ResNet-50	Bottle	1	3	9	4	12	6	18	3	9	1	3.8 7.13



He et al, "Deep Residual Learning for Image Recognition", CVPR 2016  
Error rates are 224x224 single-crop testing, reported by [torchvision](#)

# Residual Networks

Deeper ResNet-101 and ResNet-152 models are more accurate, but also more computationally heavy



			Stage 1		Stage 2		Stage 3		Stage 4		FC	ImageNet	
	Block type	Stem layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	GFLOP	top-5 error	
ResNet-18	Basic	1	2	4	2	4	2	4	2	4	1	1.8	10.92
ResNet-34	Basic	1	3	6	4	8	6	12	3	6	1	3.6	8.58
ResNet-50	Bottle	1	3	9	4	12	6	18	3	9	1	3.8	7.13
ResNet-101	Bottle	1	3	9	4	12	23	69	3	9	1	7.6	6.44
ResNet-152	Bottle	1	3	9	8	24	36	108	3	9	1	11.3	5.94

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016  
 Error rates are 224x224 single-crop testing, reported by [torchvision](#)

# Residual Networks

- Able to train very deep networks
- Deeper networks do better than shallow networks (as expected)
- Swept 1st place in all ILSVRC and COCO 2015 competitions
- Still widely used today!

## MSRA @ ILSVRC & COCO 2015 Competitions

- **1st places** in all five main tracks
  - ImageNet Classification: “Ultra-deep” (quote Yann) **152-layer** nets
  - ImageNet Detection: **16%** better than 2nd
  - ImageNet Localization: **27%** better than 2nd
  - COCO Detection: **11%** better than 2nd
  - COCO Segmentation: **12%** better than 2nd

# Residual Blocks in AlphaGo Zero in 2017

## ARTICLE

doi:10.1038/nature24270

### Mastering the game of Go without human knowledge

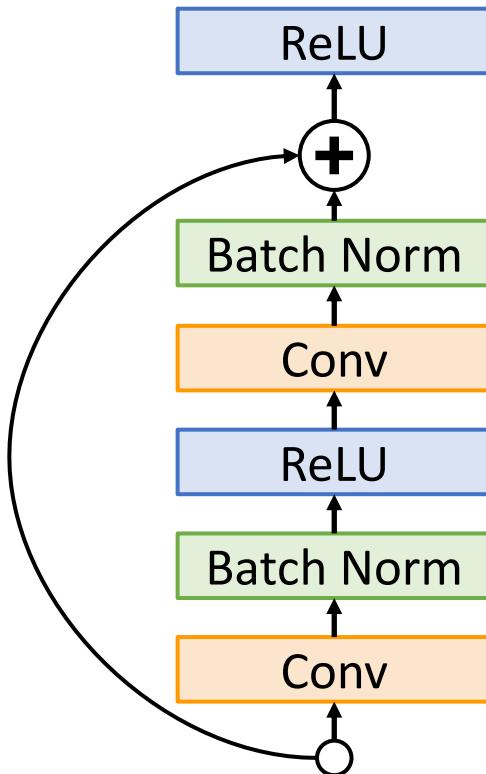
David Silver<sup>1\*</sup>, Julian Schrittwieser<sup>1\*</sup>, Karen Simonyan<sup>1\*</sup>, Ioannis Antonoglou<sup>1</sup>, Aja Huang<sup>1</sup>, Arthur Guez<sup>1</sup>, Thomas Hubert<sup>1</sup>, Lucas Baker<sup>1</sup>, Matthew Lai<sup>1</sup>, Adrian Bolton<sup>1</sup>, Yutian Chen<sup>1</sup>, Timothy Lillicrap<sup>1</sup>, Fan Hui<sup>1</sup>, Laurent Sifre<sup>1</sup>, George van den Driessche<sup>1</sup>, Thore Graepel<sup>1</sup> & Demis Hassabis<sup>1</sup>

#### Reinforcement learning in AlphaGo Zero

Our new method uses a deep neural network  $f_\theta$  with parameters  $\theta$ . This neural network takes as an input the raw board representation  $s$  of the position and its history, and outputs both move probabilities and a value,  $(\mathbf{p}, v) = f_\theta(s)$ . The vector of move probabilities  $\mathbf{p}$  represents the probability of selecting each move  $a$  (including pass),  $p_a = \Pr(a|s)$ . The value  $v$  is a scalar evaluation, estimating the probability of the current player winning from position  $s$ . This neural network combines the roles of both policy network and value network<sup>12</sup> into a single architecture. The neural network consists of many residual blocks<sup>4</sup> of convolutional layers<sup>16,17</sup> with batch normalization<sup>18</sup> and rectifier nonlinearities<sup>19</sup> (see Methods).

# Improving Residual Networks: Block Design

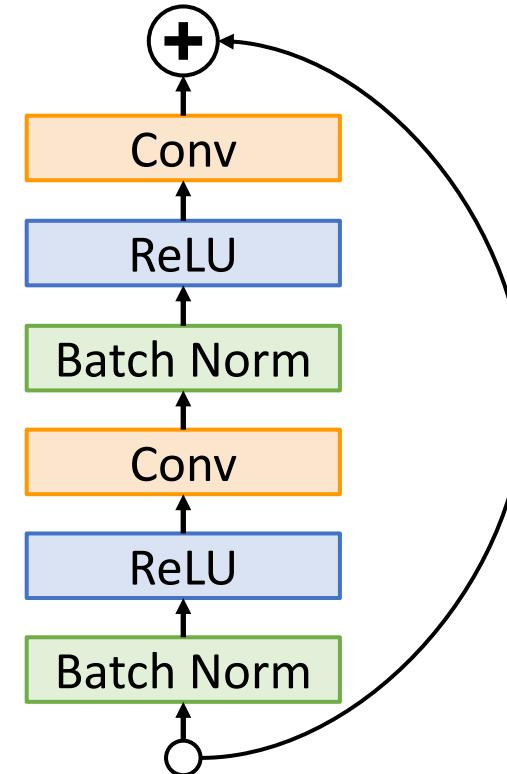
Original ResNet block



Note ReLU **after** residual:

Cannot actually learn  
identity function since  
outputs are nonnegative!

“Pre-Activation” ResNet Block

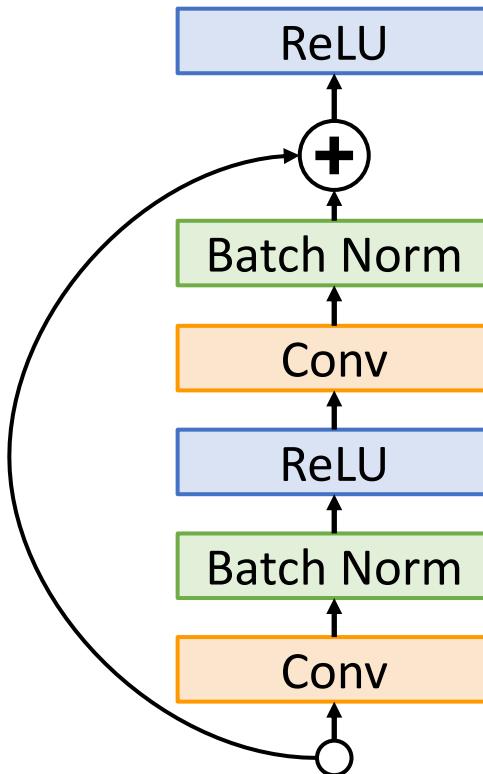


Note ReLU **inside** residual:

Can learn true identity  
function by setting Conv  
weights to zero!

# Improving Residual Networks: Block Design

Original ResNet block



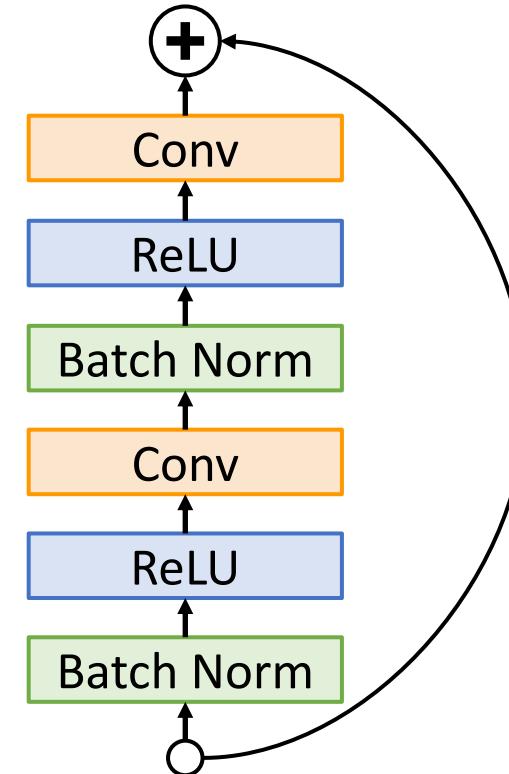
Slight improvement in accuracy  
(ImageNet top-1 error)

ResNet-152: 21.3 vs **21.1**

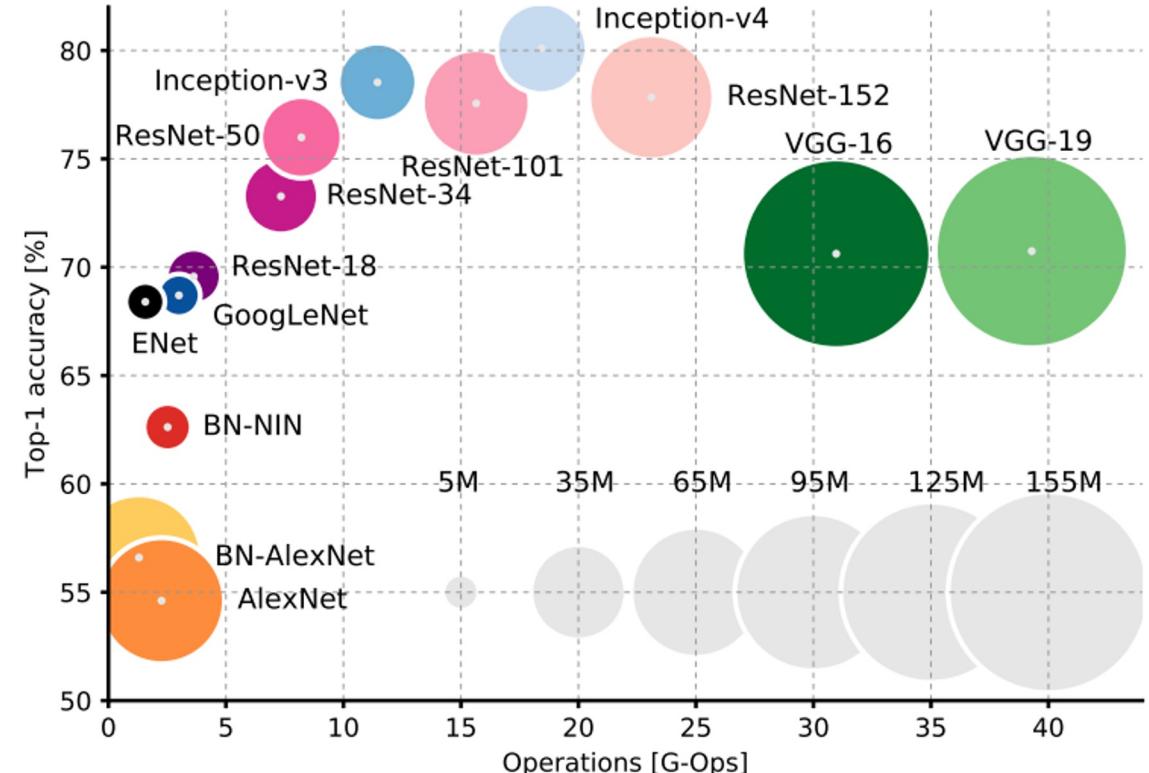
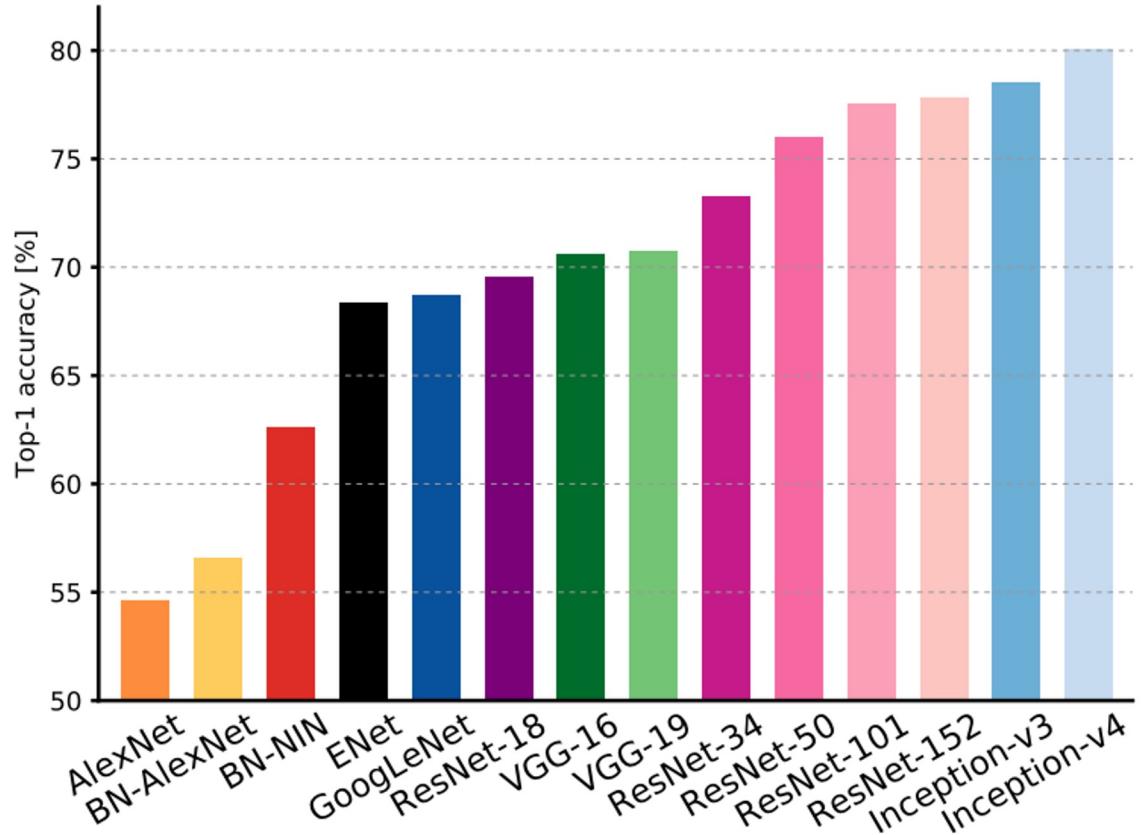
ResNet-200: 21.8 vs **20.7**

Not actually used that much in practice

“Pre-Activation” ResNet Block



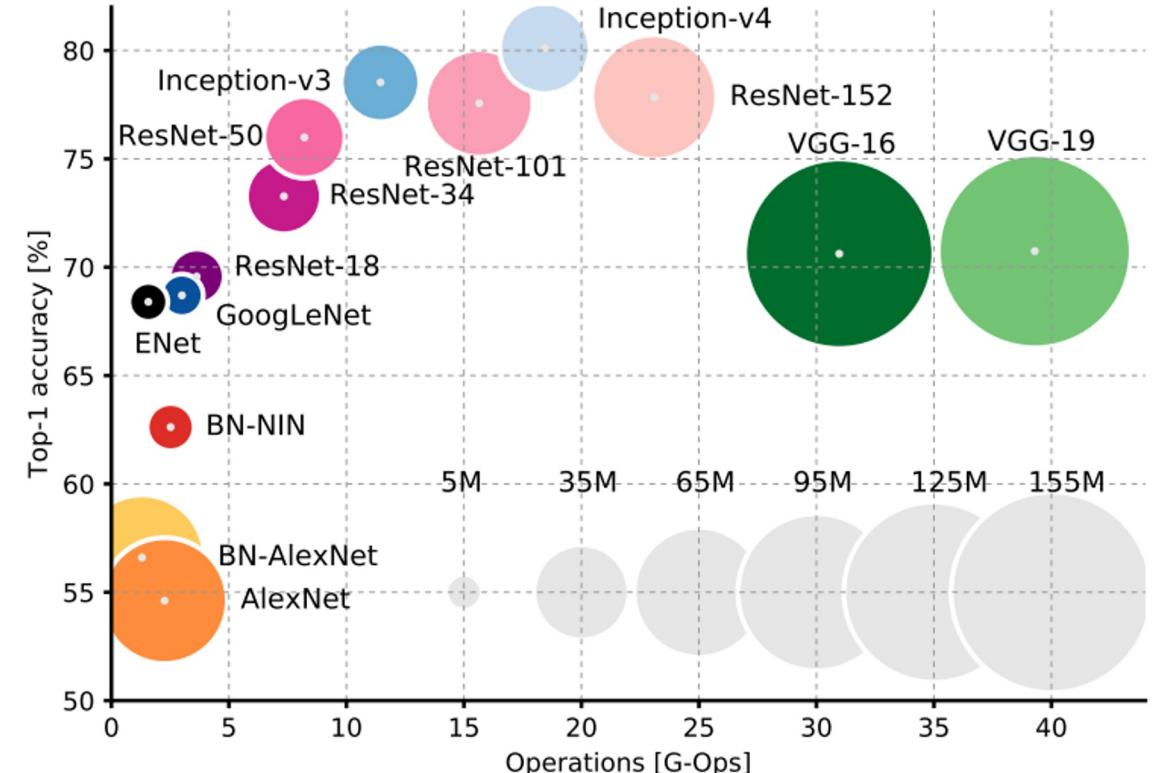
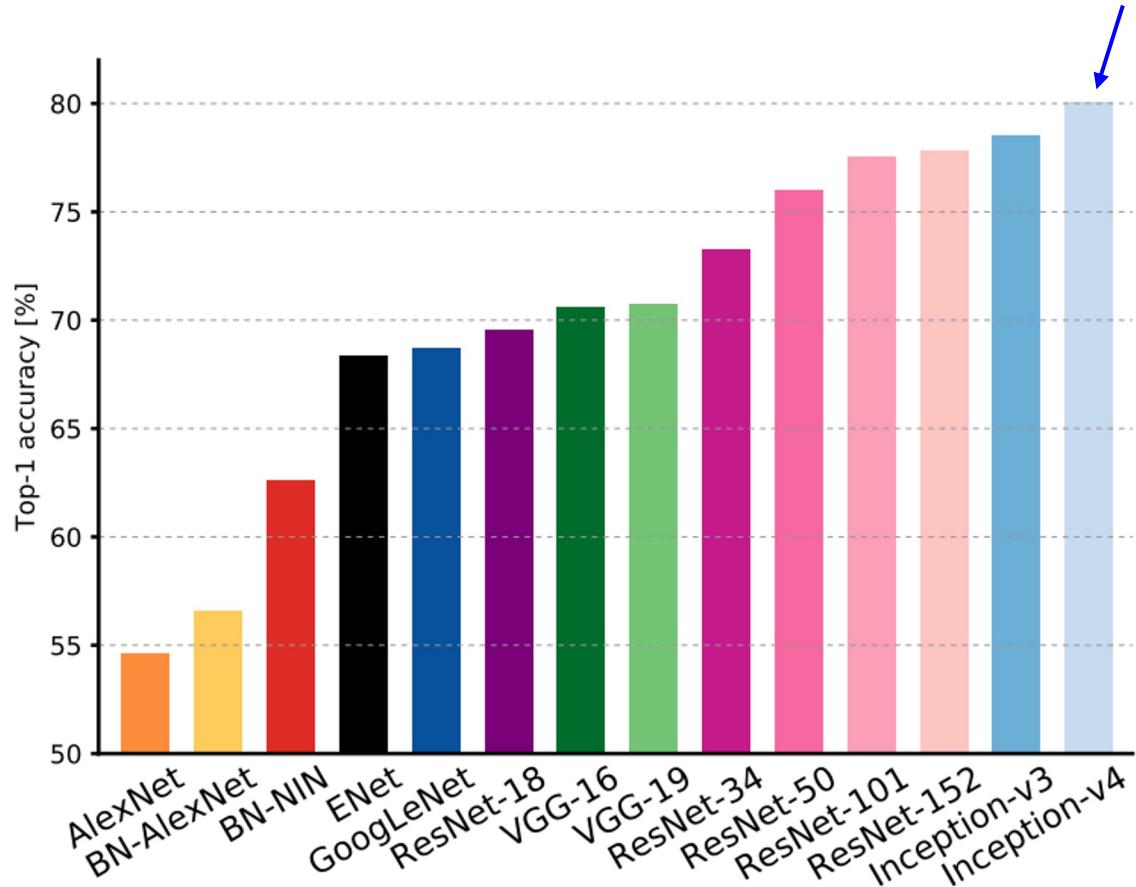
# Comparing Complexity



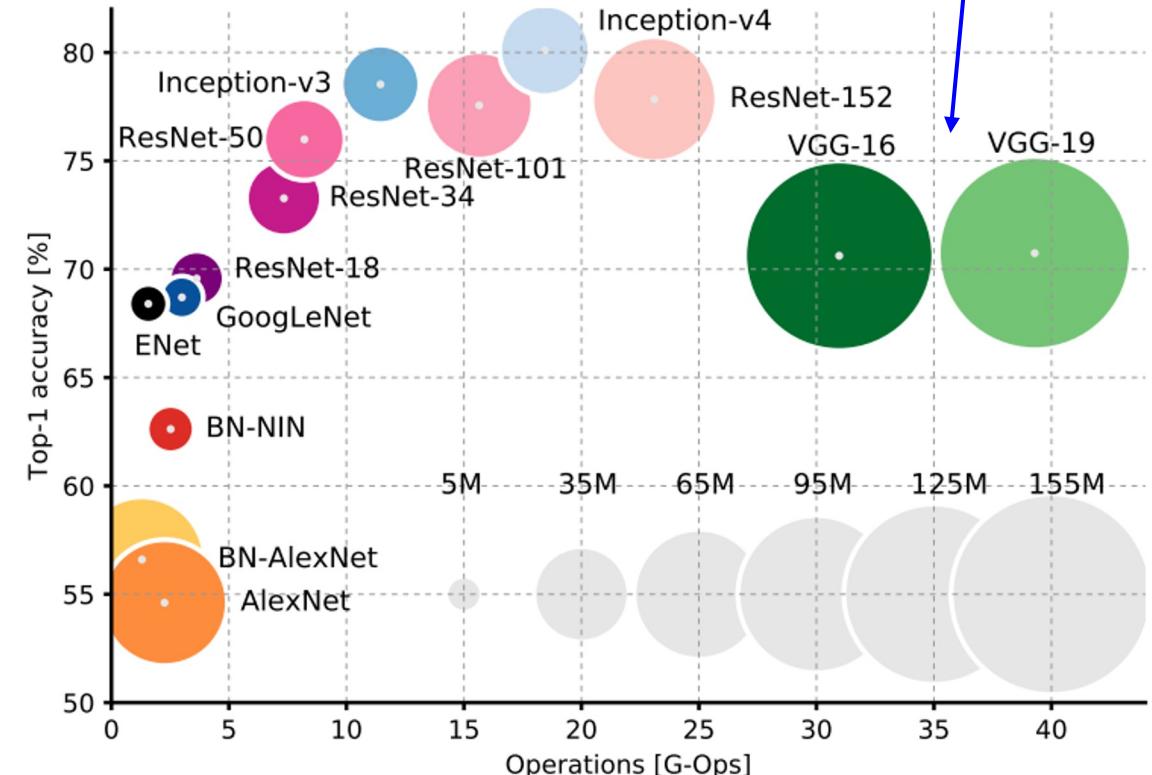
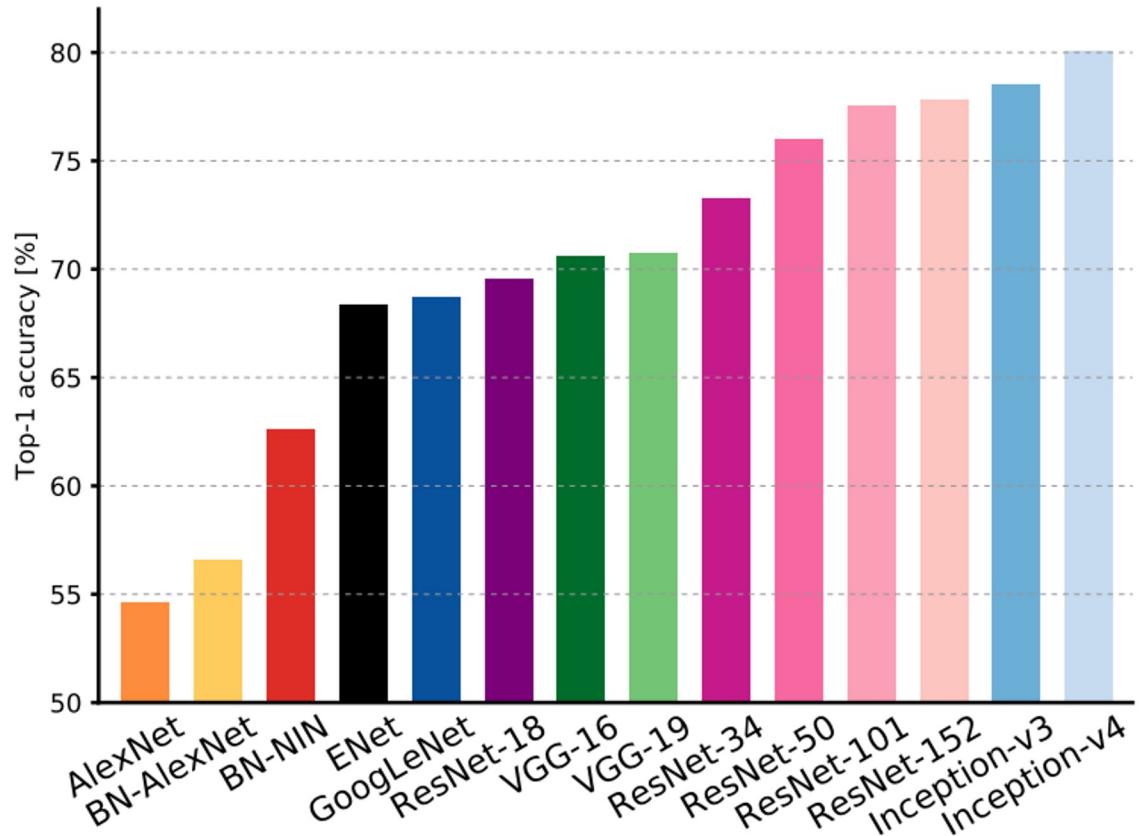
Canziani et al, "An analysis of deep neural network models for practical applications", 2017

# Comparing Complexity

Inception-v4: Resnet + Inception!

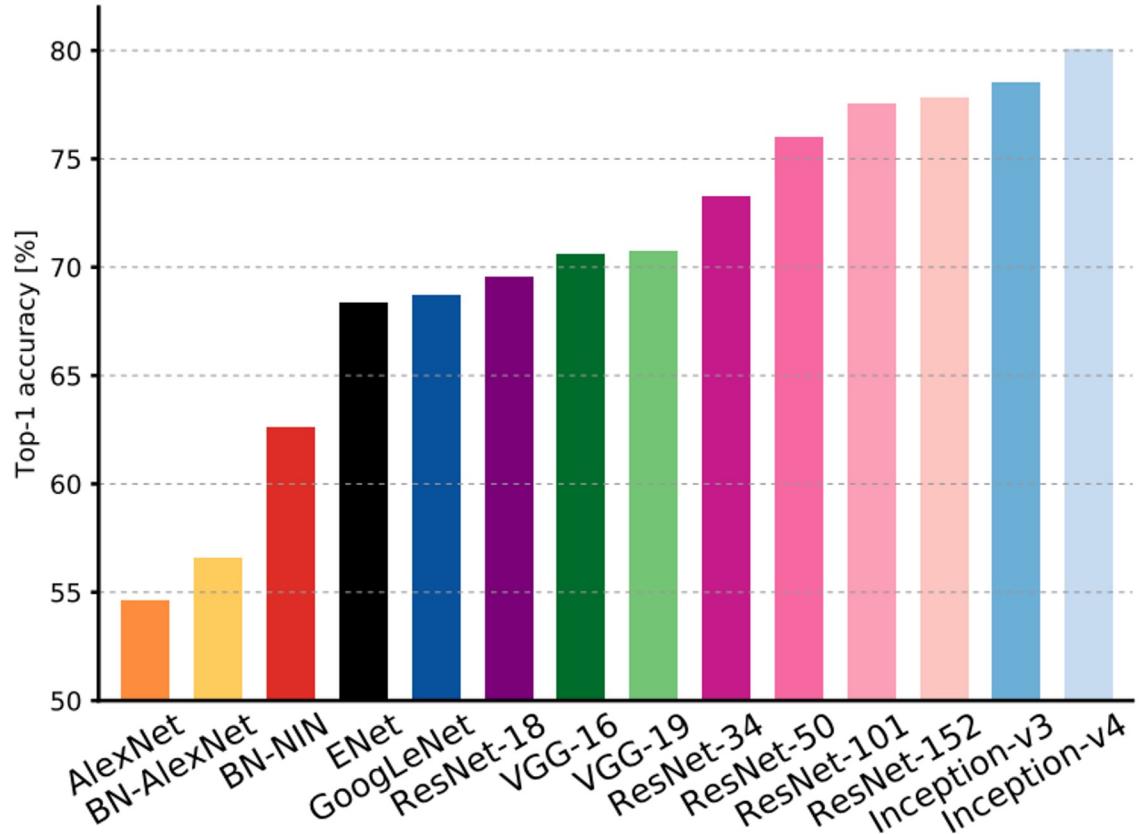


# Comparing Complexity

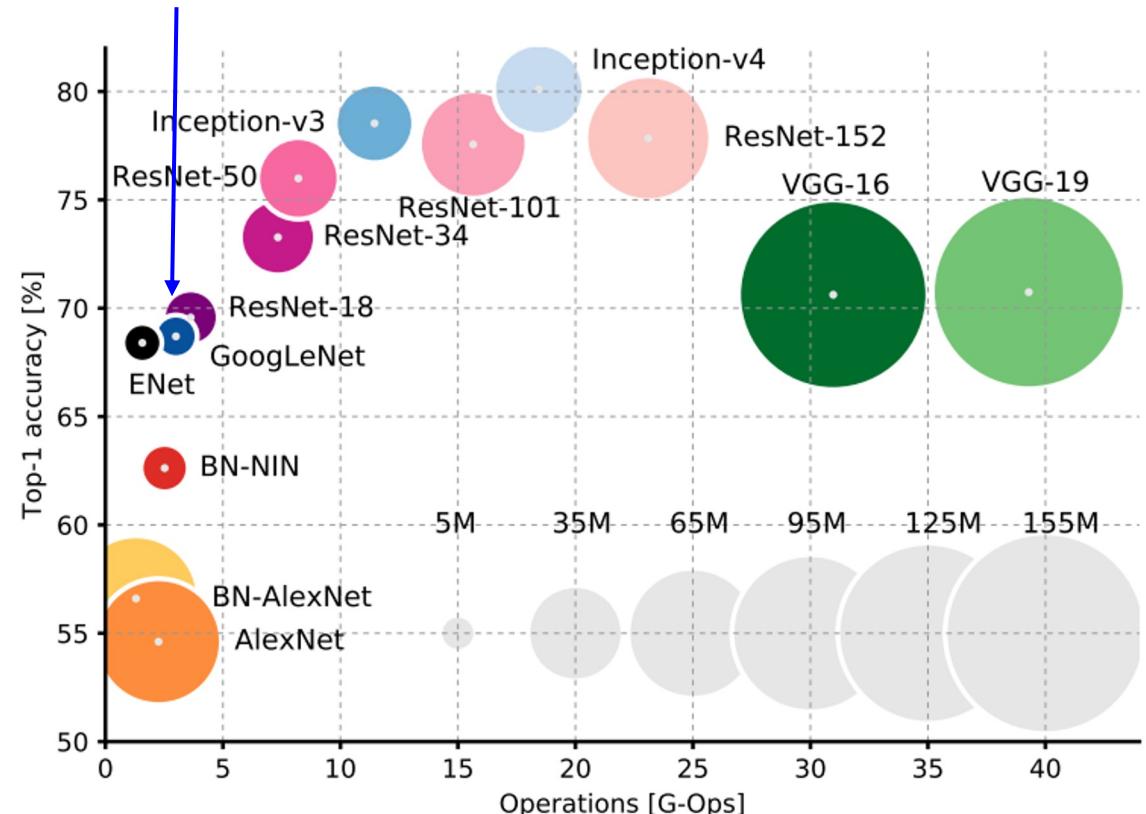


VGG: Highest memory, most operations

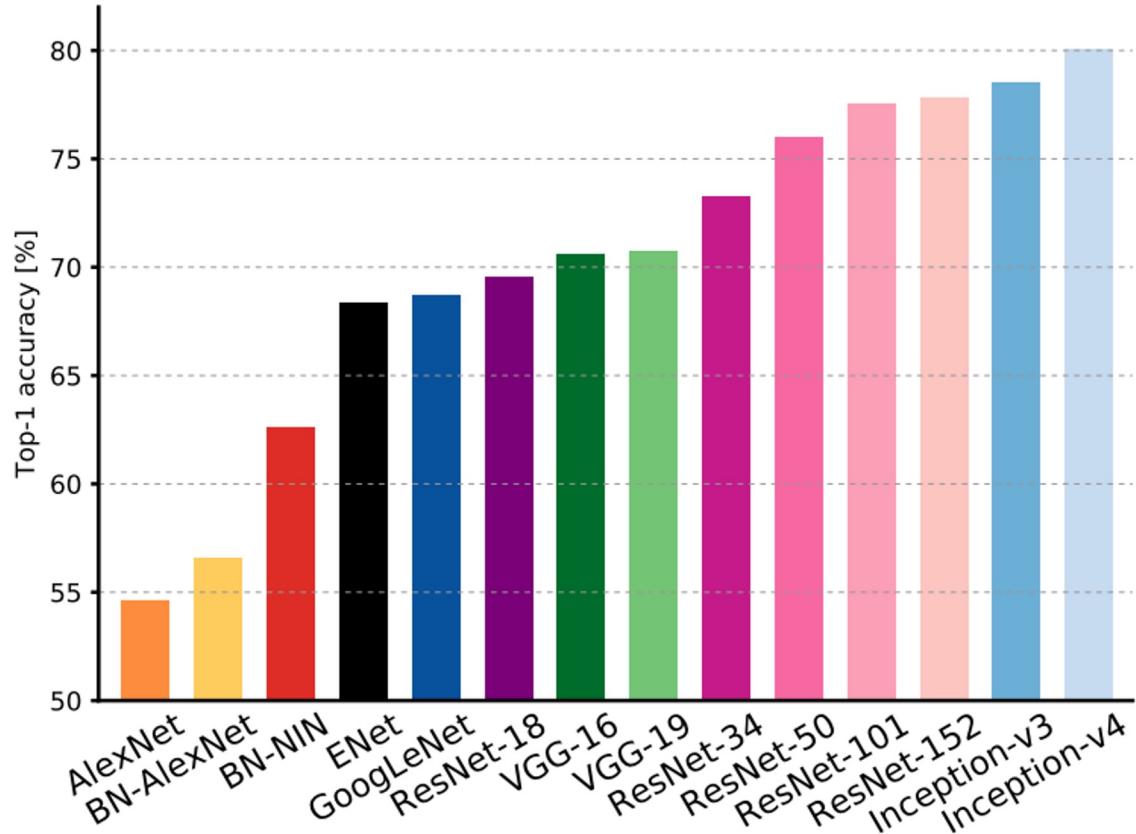
# Comparing Complexity



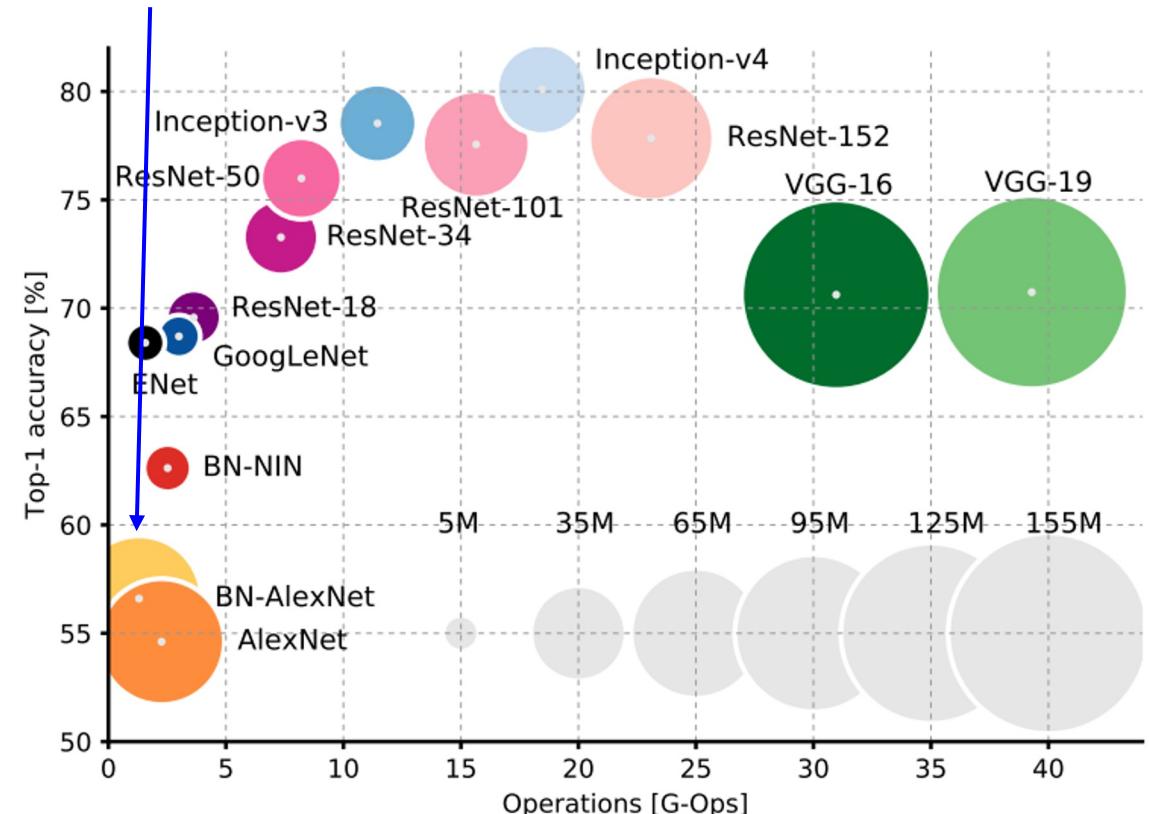
GoogLeNet:  
Very efficient!



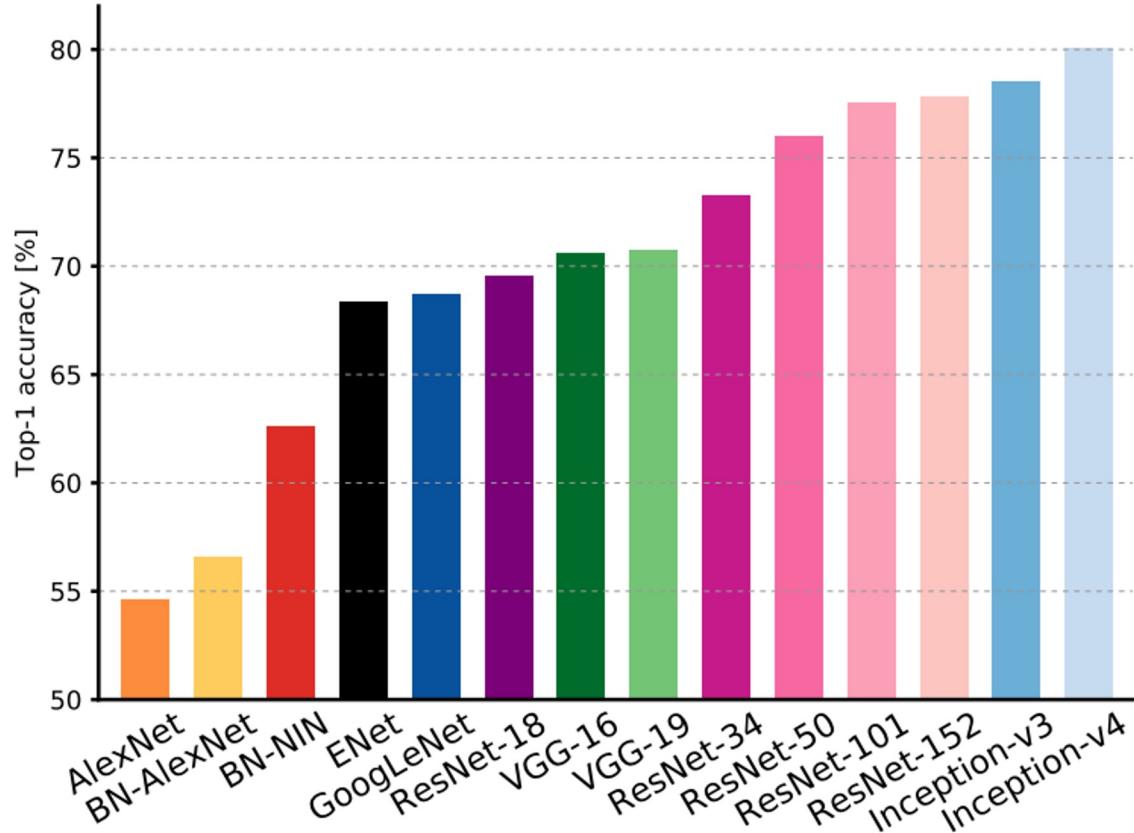
# Comparing Complexity



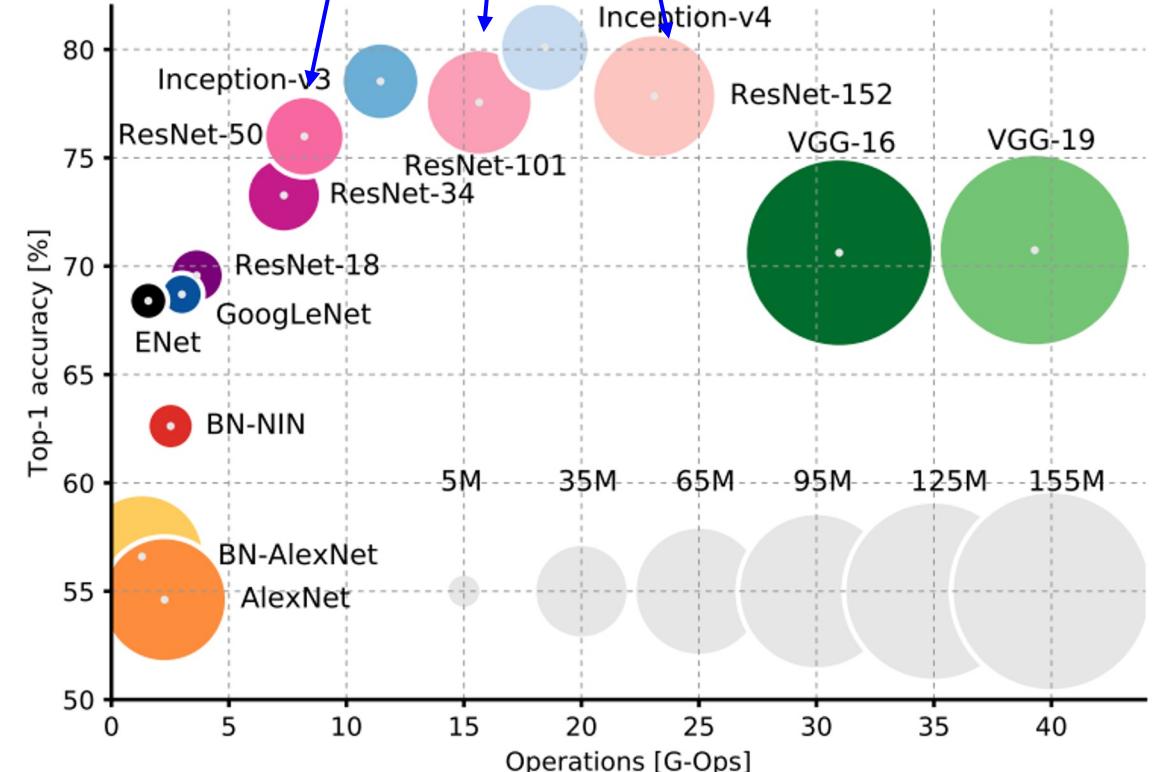
AlexNet: Low compute, lots of parameters



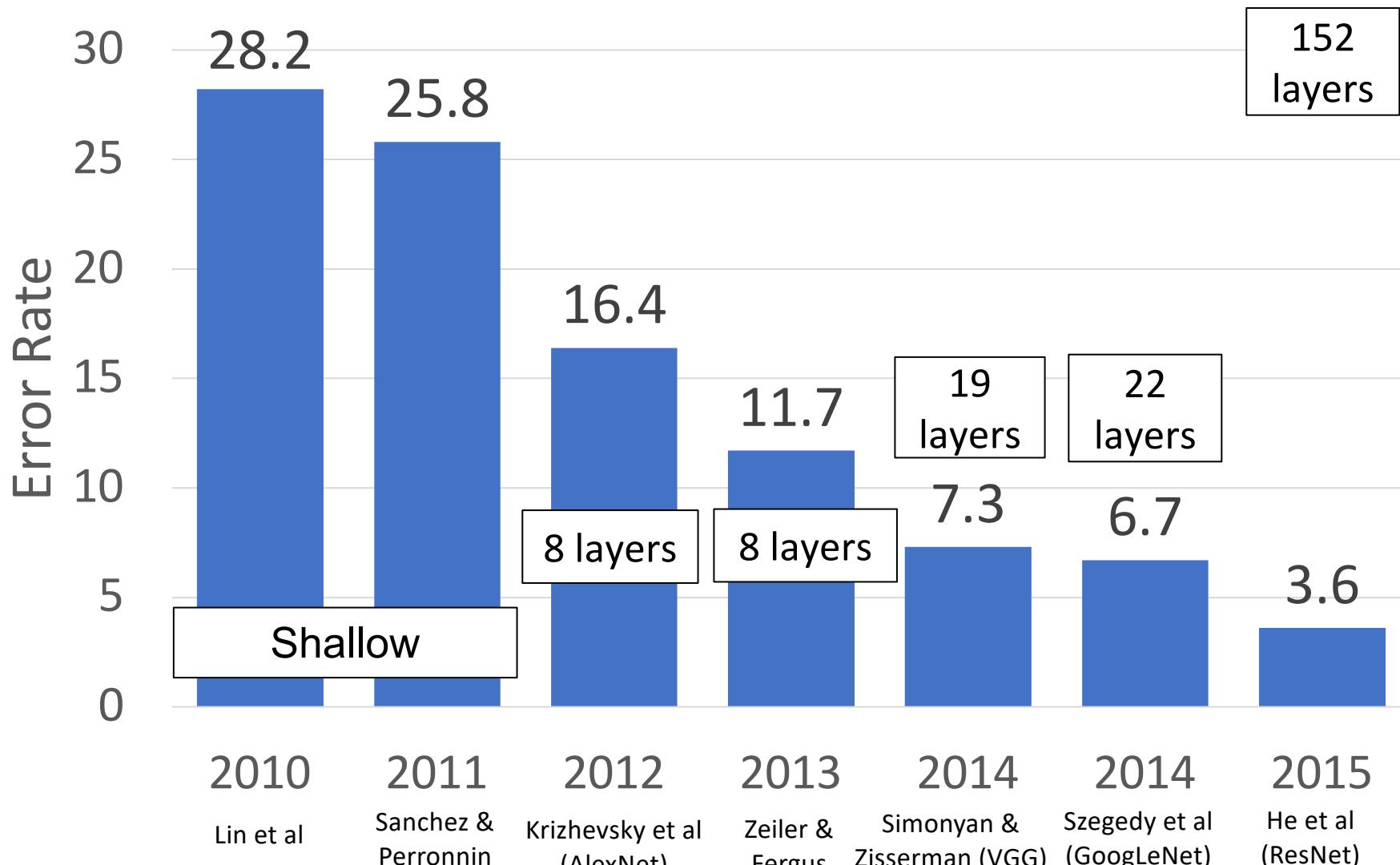
# Comparing Complexity



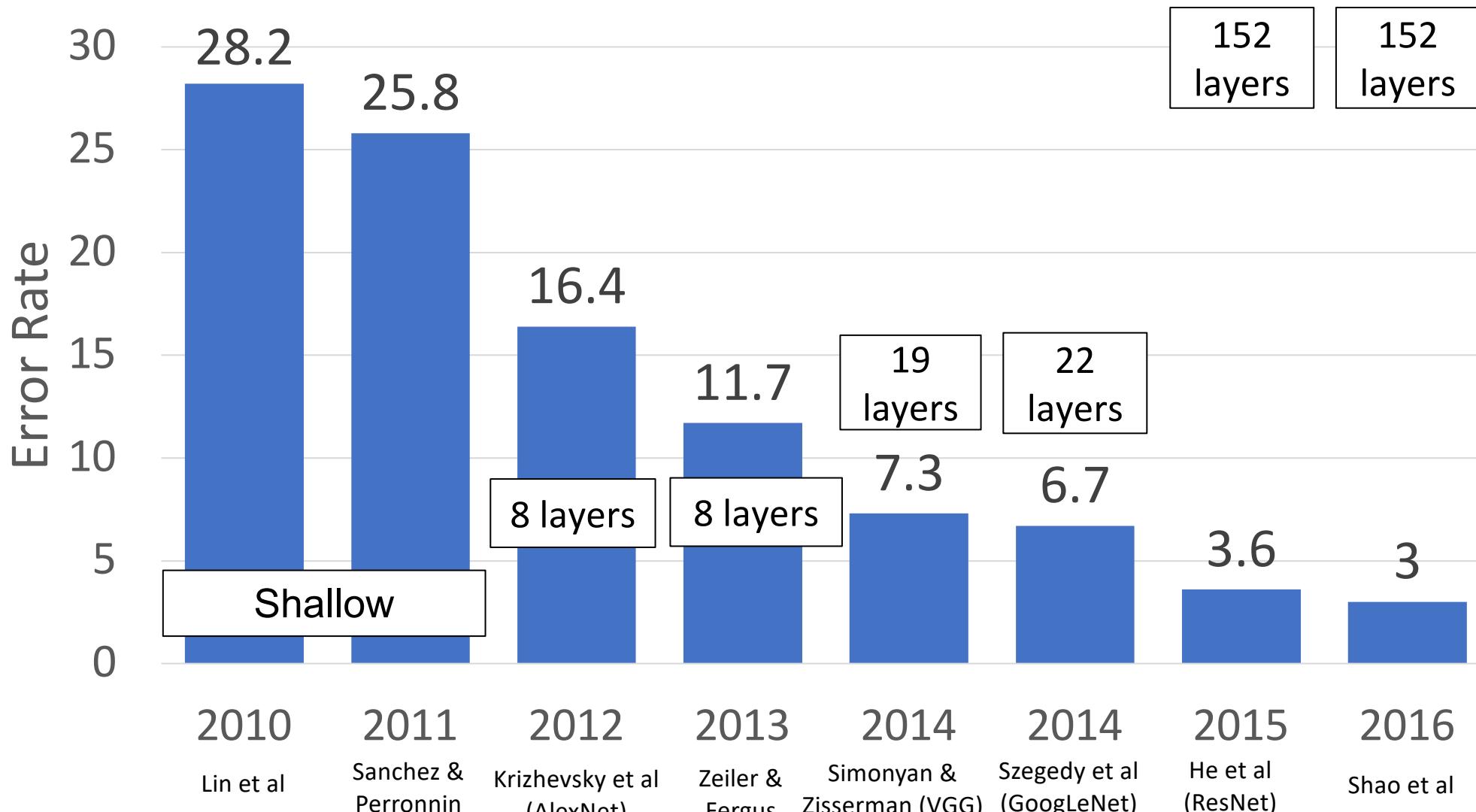
ResNet: Simple design,  
moderate efficiency,  
high accuracy



# ImageNet Classification Challenge



# ImageNet Classification Challenge

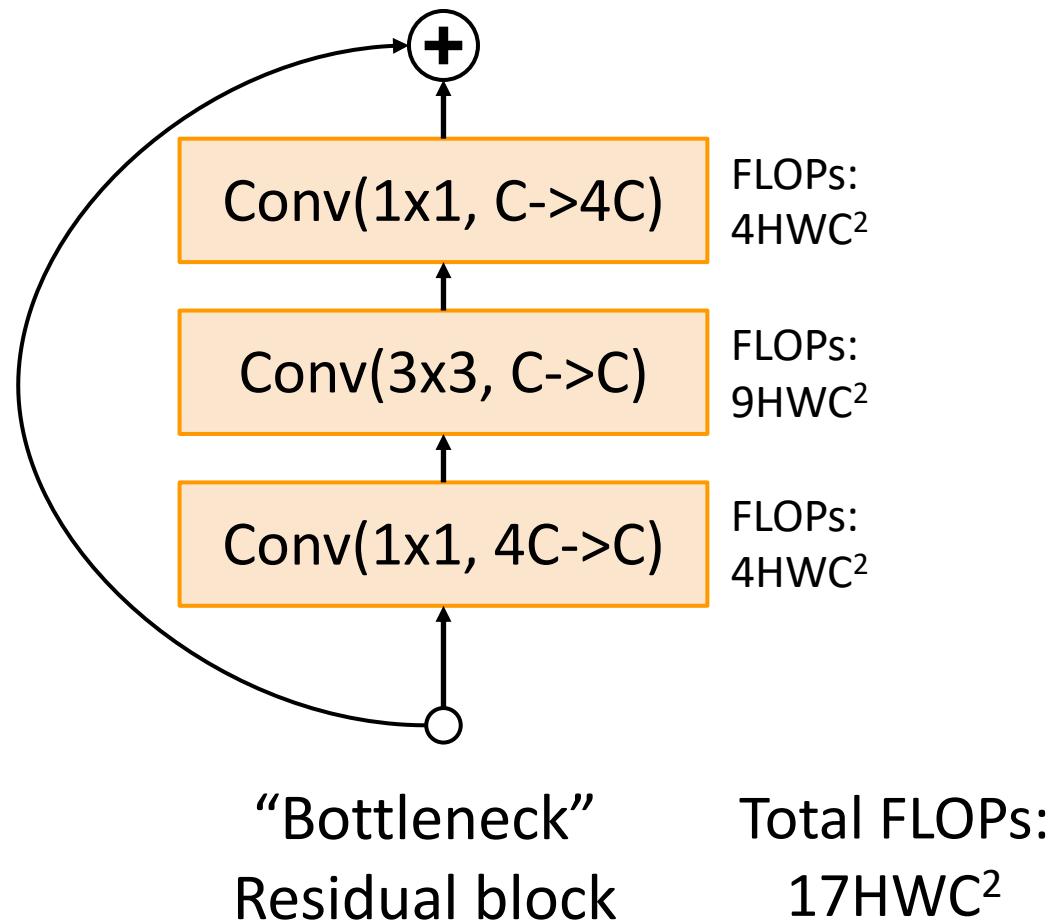


# ImageNet 2016 winner: Model Ensembles

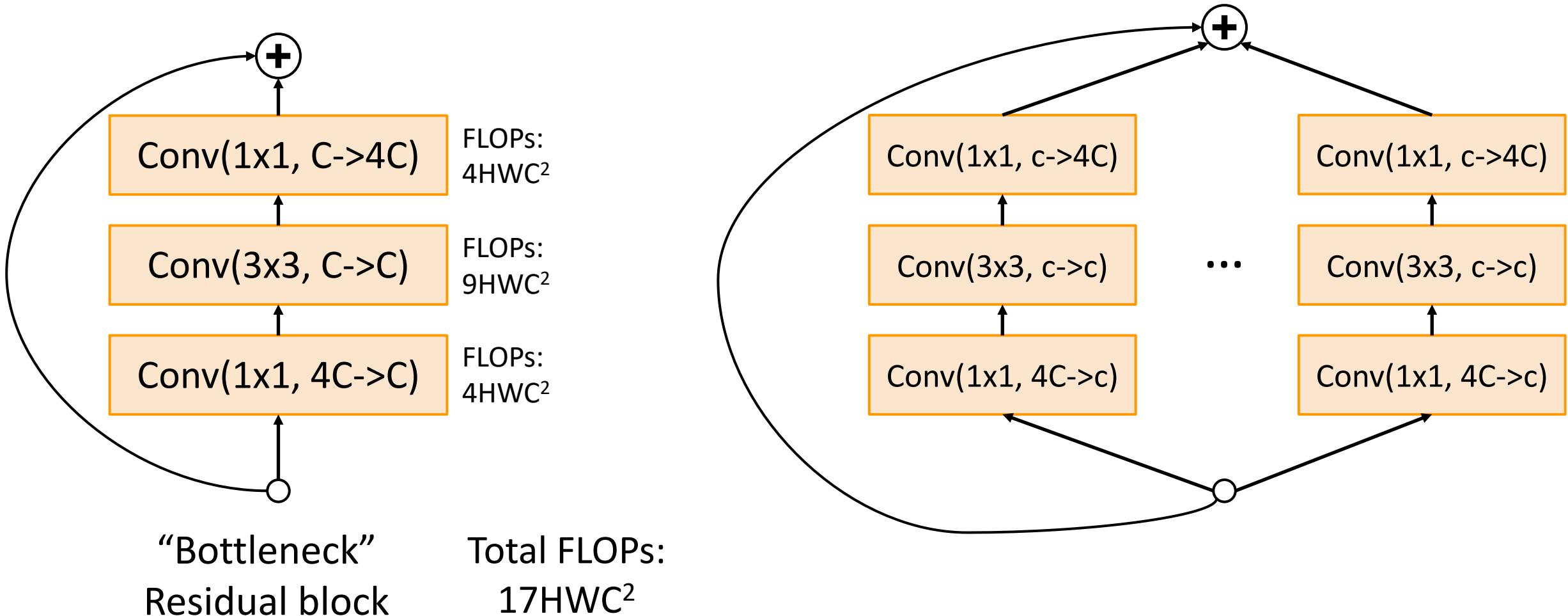
Multi-scale ensemble of Inception, Inception-Resnet,  
Resnet, Wide Resnet models

	Inception-v3	Inception-v4	Inception-Resnet-v2	Resnet-200	Wrn-68-3	Fusion (Val.)	Fusion (Test)
Err. (%)	4.20	4.01	3.52	4.26	4.65	<b>2.92 (-0.6)</b>	<b>2.99</b>

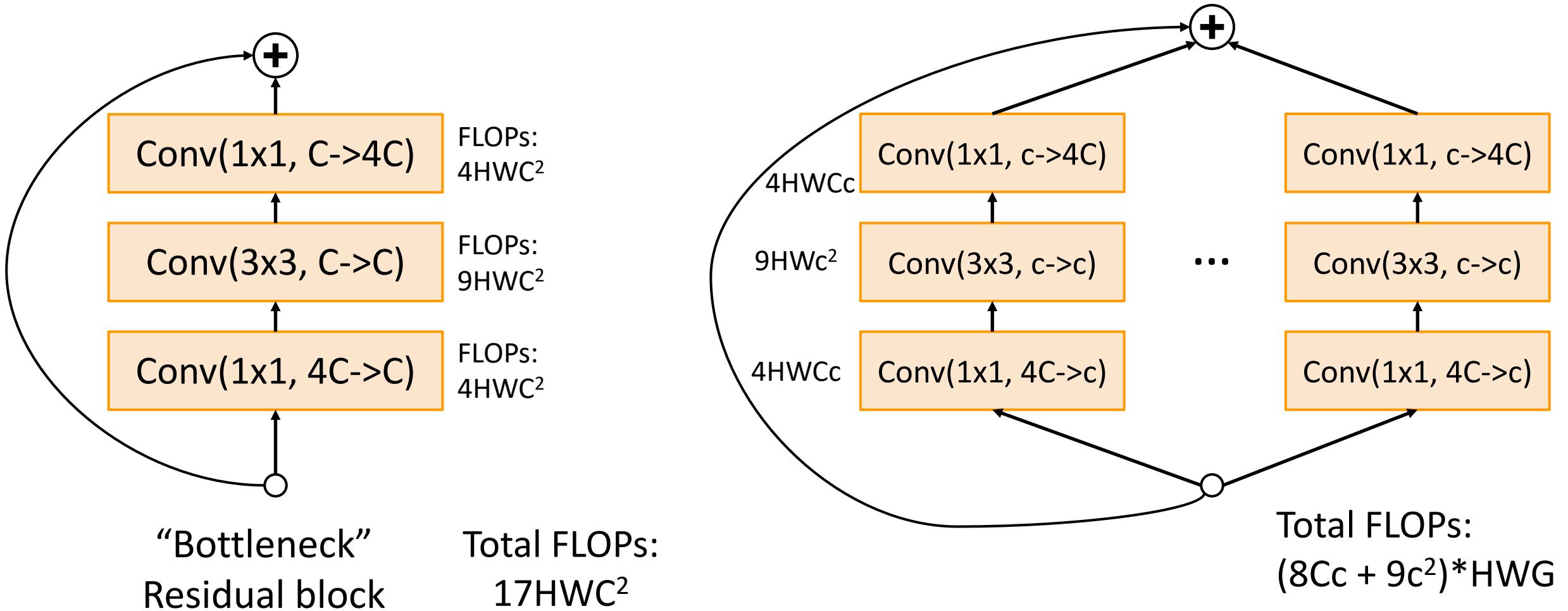
# Improving ResNets



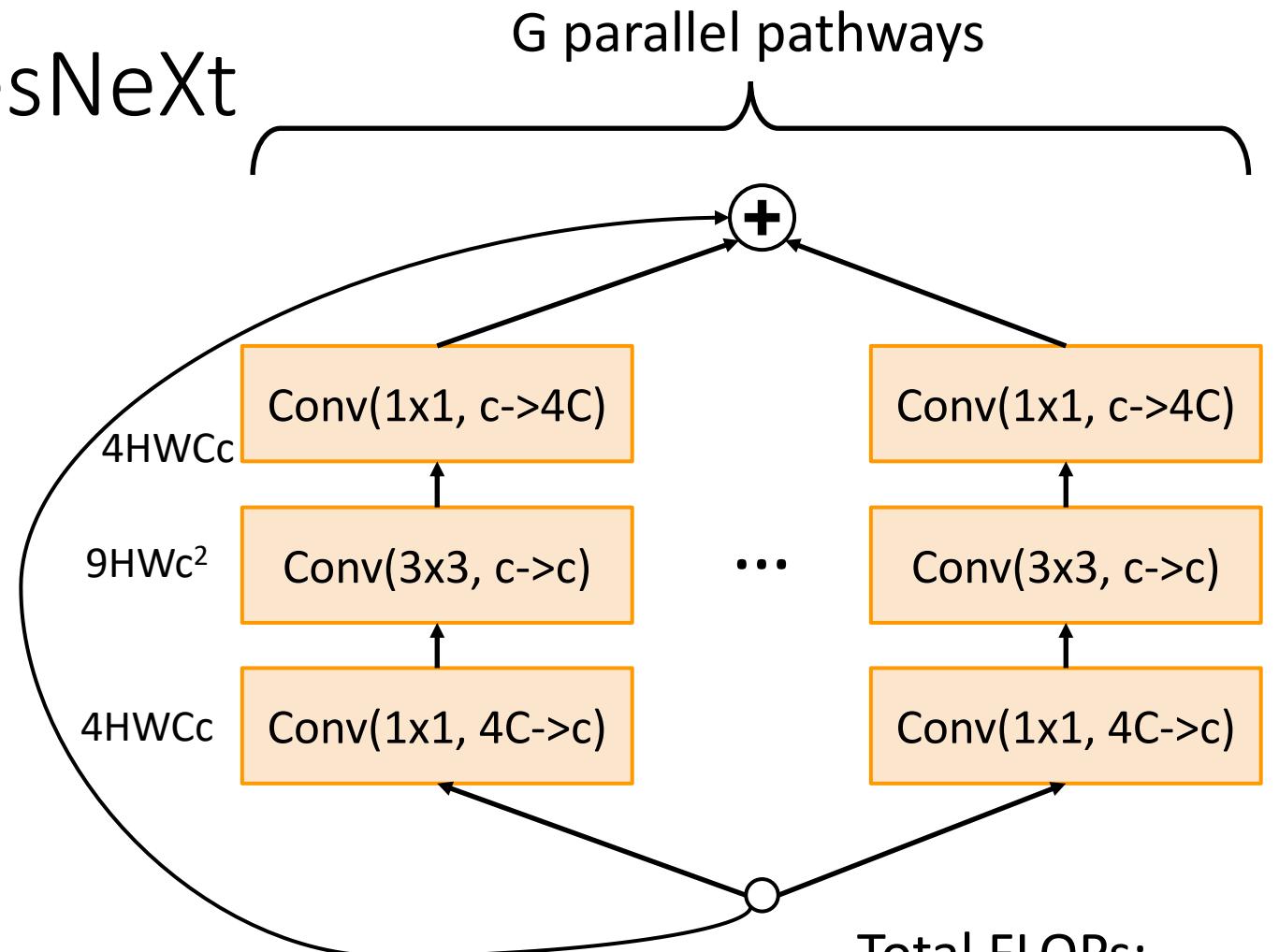
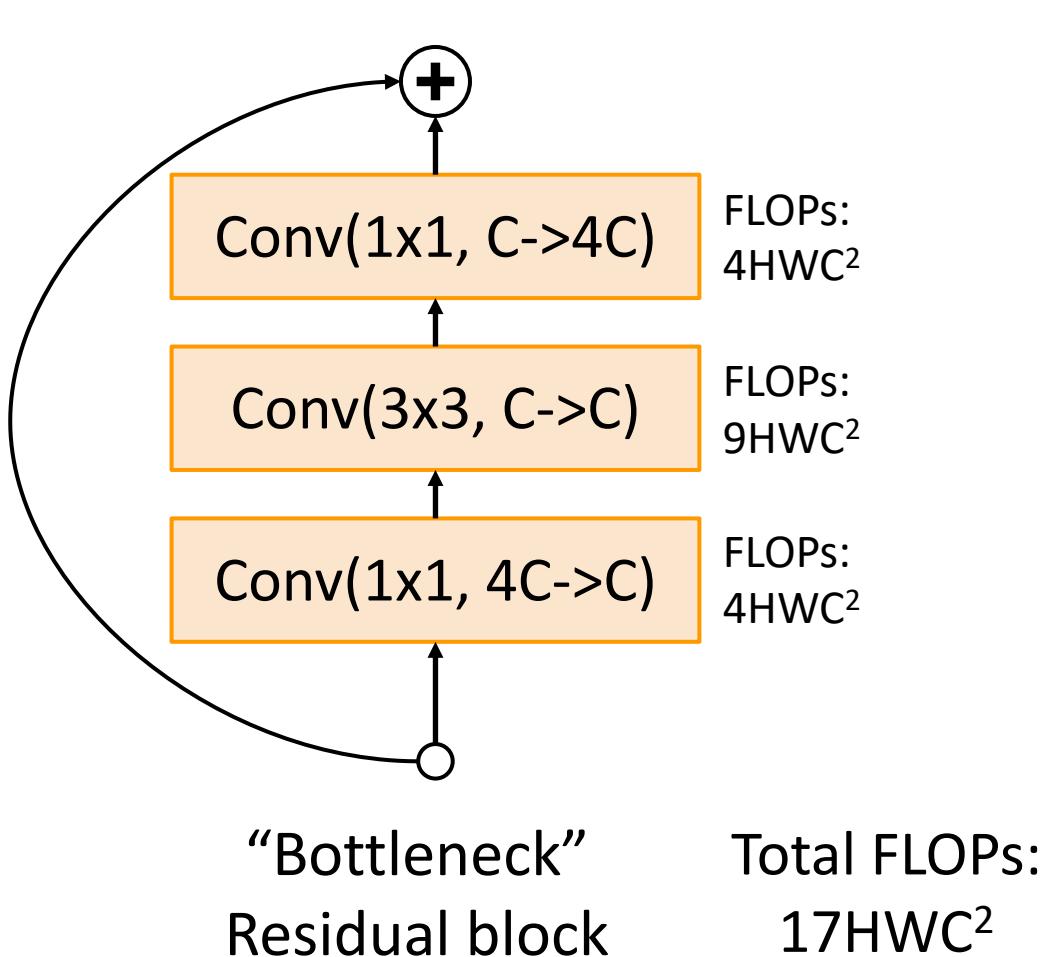
# Improving ResNets: ResNeXt



# Improving ResNets: ResNeXt



# Improving ResNets: ResNeXt



# Grouped Convolution

Convolution with groups=1:

Normal convolution

Input:  $C_{in} \times H \times W$

Weight:  $C_{out} \times C_{in} \times K \times K$

Output:  $C_{out} \times H' \times W'$

FLOPs:  $C_{out}C_{in}K^2HW$

All convolutional kernels touch  
all  $C_{in}$  channels of the input

# Grouped Convolution

## Convolution with groups=1:

Normal convolution

Input:  $C_{in} \times H \times W$

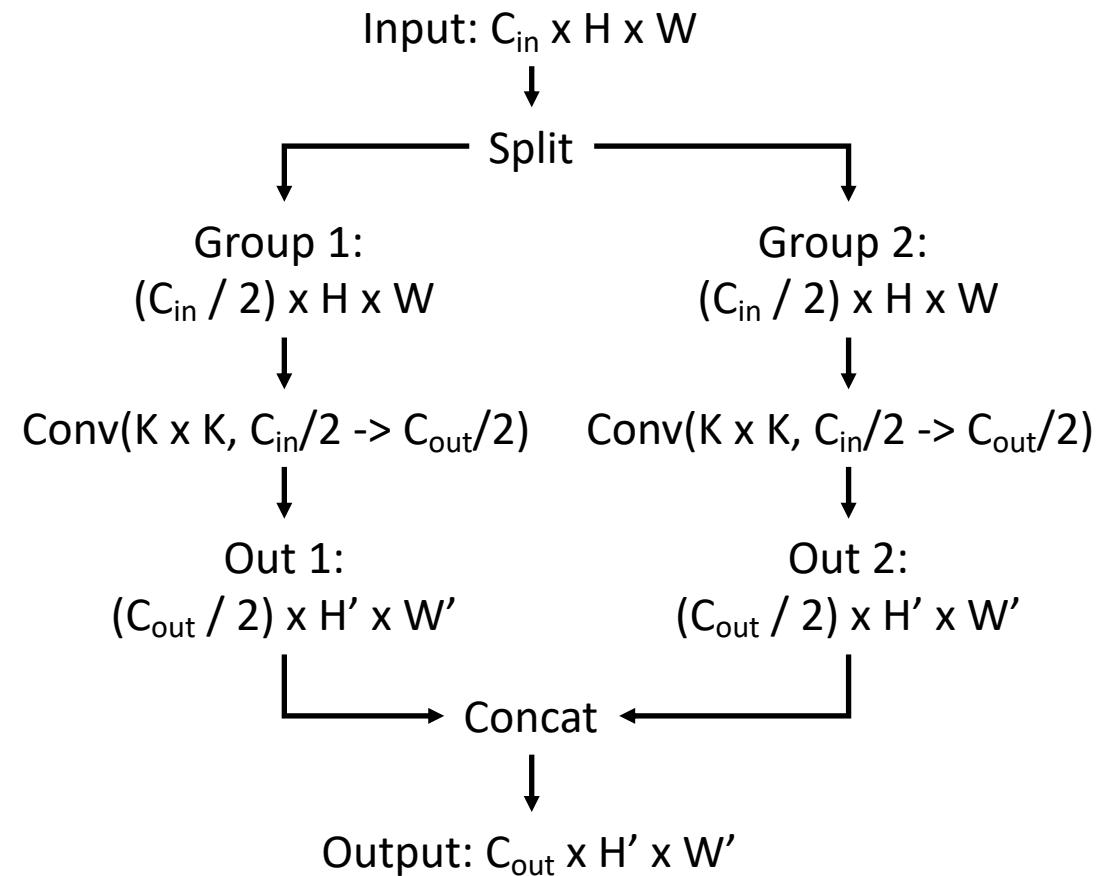
Weight:  $C_{out} \times C_{in} \times K \times K$

Output:  $C_{out} \times H' \times W'$

FLOPs:  $C_{out} C_{in} K^2 HW$

All convolutional kernels touch  
all  $C_{in}$  channels of the input

Convolution with groups=2:  
Two parallel convolution layers that  
work on half the channels



# Grouped Convolution is used in AlexNet!

## Why? GPU RAM limitation in 2012

AlexNet was first trained on two Nvidia 580 GPUs with 3GB per card

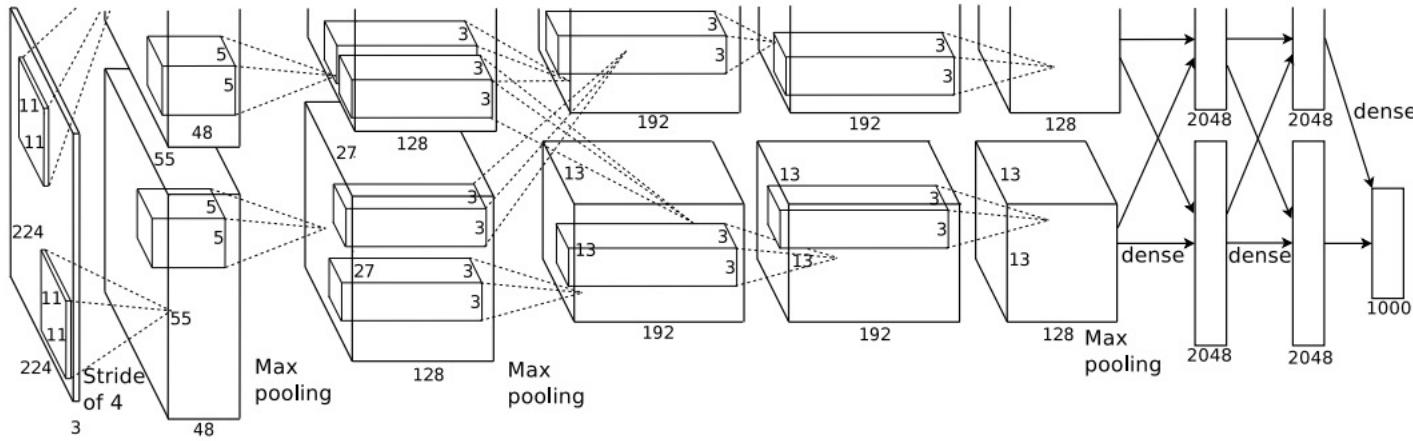


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

# Grouped Convolution

## Convolution with groups=1:

Normal convolution

Input:  $C_{in} \times H \times W$

Weight:  $C_{out} \times C_{in} \times K \times K$

Output:  $C_{out} \times H' \times W'$

FLOPs:  $C_{out} C_{in} K^2 HW$

All convolutional kernels touch  
all  $C_{in}$  channels of the input

## Convolution with groups=G:

G parallel conv layers; each “sees”  
 $C_{in}/G$  input channels and produces  
 $C_{out}/G$  output channels

Input:  $C_{in} \times H \times W$

Split to  $G \times [(C_{in} / G) \times H \times W]$

Weight:  $G \times (C_{out} / G) \times (C_{in} \times G) \times K \times K$

G parallel convolutions

Output:  $G \times [(C_{out} / G) \times H' \times W']$

Concat to  $C_{out} \times H' \times W'$

FLOPs:  $C_{out} C_{in} K^2 HW/G$

# Grouped Convolution

## Convolution with groups=1:

Normal convolution

Input:  $C_{in} \times H \times W$

Weight:  $C_{out} \times C_{in} \times K \times K$

Output:  $C_{out} \times H' \times W'$

FLOPs:  $C_{out} C_{in} K^2 HW$

All convolutional kernels touch  
all  $C_{in}$  channels of the input

## **Depthwise Convolution**

Special case:  $G=C_{in}$ ,  $C_{out} = nC_{in}$

Each input channel is convolved  
with  $n$  different  $K \times K$  filters to  
produce  $n$  output channels

## Convolution with groups=G:

$G$  parallel conv layers; each “sees”  
 $C_{in}/G$  input channels and produces  
 $C_{out}/G$  output channels

Input:  $C_{in} \times H \times W$

Split to  $G \times [(C_{in} / G) \times H \times W]$

Weight:  $G \times (C_{out} / G) \times (C_{in} \times G) \times K \times K$

$G$  parallel convolutions

Output:  $G \times [(C_{out} / G) \times H' \times W']$

Concat to  $C_{out} \times H' \times W'$

FLOPs:  $C_{out} C_{in} K^2 HW/G$

# Grouped Convolution in PyTorch

PyTorch convolution gives an option for groups!

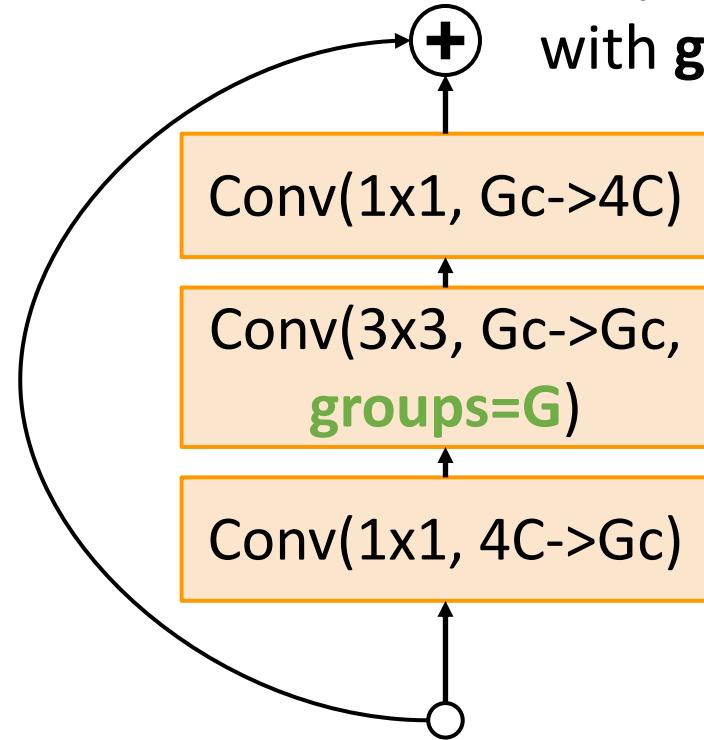
Conv2d

```
CLASS torch.nn.Conv2d(in_channels, out_channels, kernel_size,  
                    stride=1, padding=0, dilation=1, groups=1, bias=True,  
                    padding_mode=‘zeros’)
```

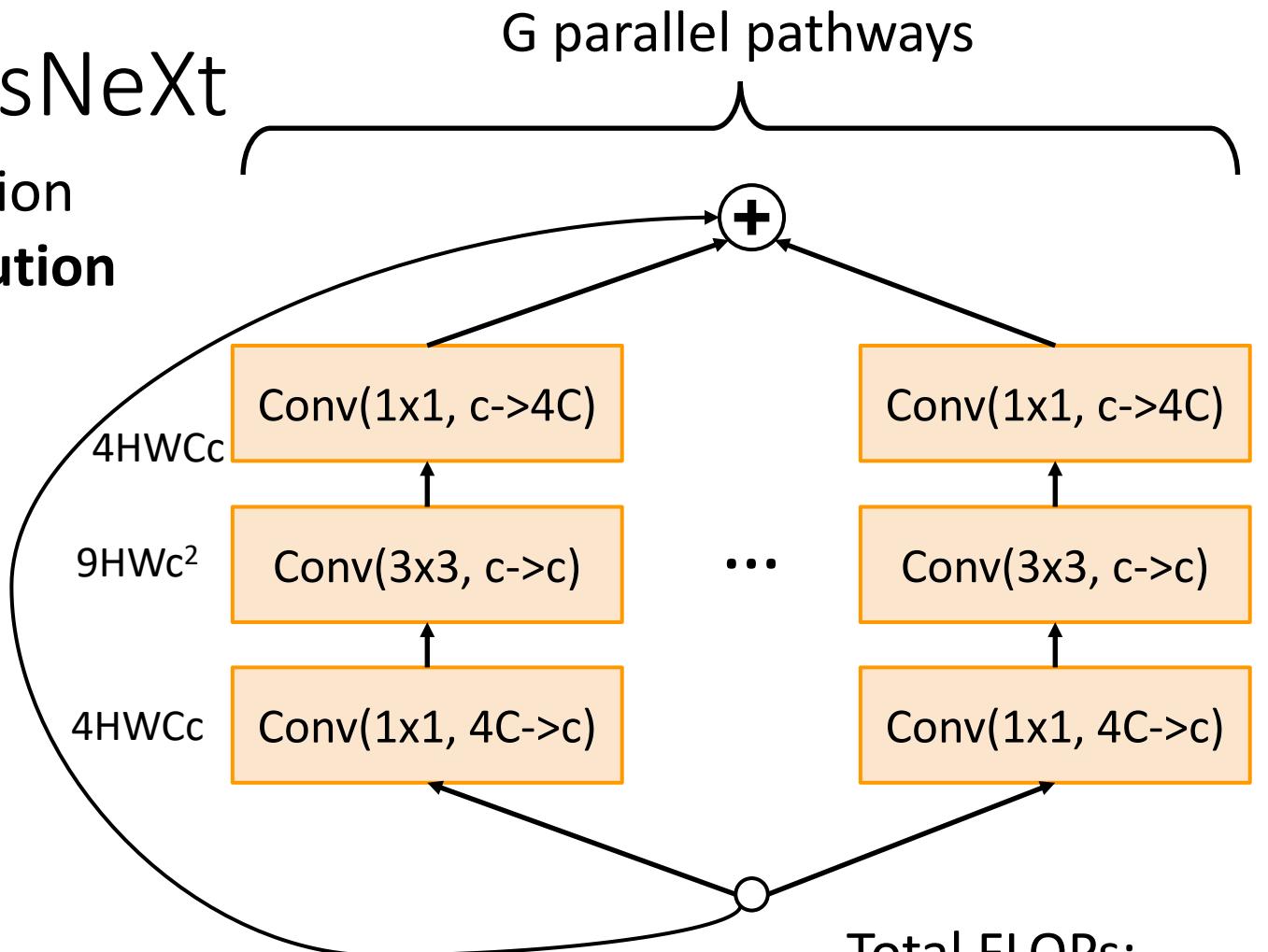
[SOURCE]

# Improving ResNets: ResNeXt

Equivalent formulation  
with **grouped convolution**



ResNeXt block:  
Grouped convolution



Equal cost when  
 $9Gc^2 + 8GCc - 17C^2 = 0$

Example: C=64, G=4, c=24; C=64, G=32, c=4

Total FLOPs:  
 $(8Cc + 9c^2) * HWG$

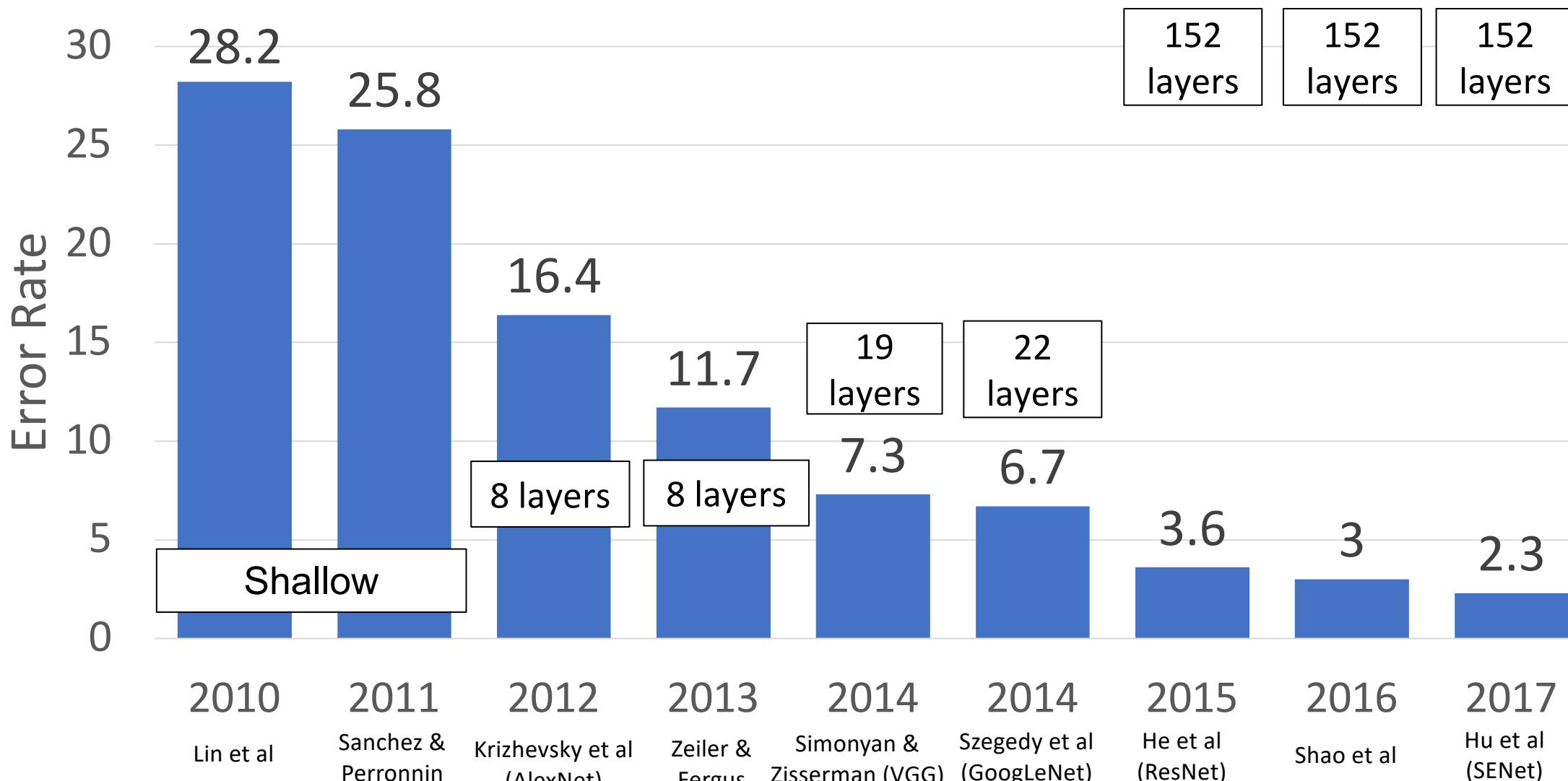
# ResNeXt: Maintain computation by adding groups!

Model	Groups	Group width	Top-1 Error
ResNet-50	1	64	23.9
ResNeXt-50	2	40	23
ResNeXt-50	4	24	22.6
ResNeXt-50	8	14	22.3
ResNeXt-50	32	4	22.2

Model	Groups	Group width	Top-1 Error
ResNet-101	1	64	22.0
ResNeXt-101	2	40	21.7
ResNeXt-101	4	24	21.4
ResNeXt-101	8	14	21.3
ResNeXt-101	32	4	21.2

Adding groups improves performance **with same computational complexity!**

# ImageNet Classification Challenge

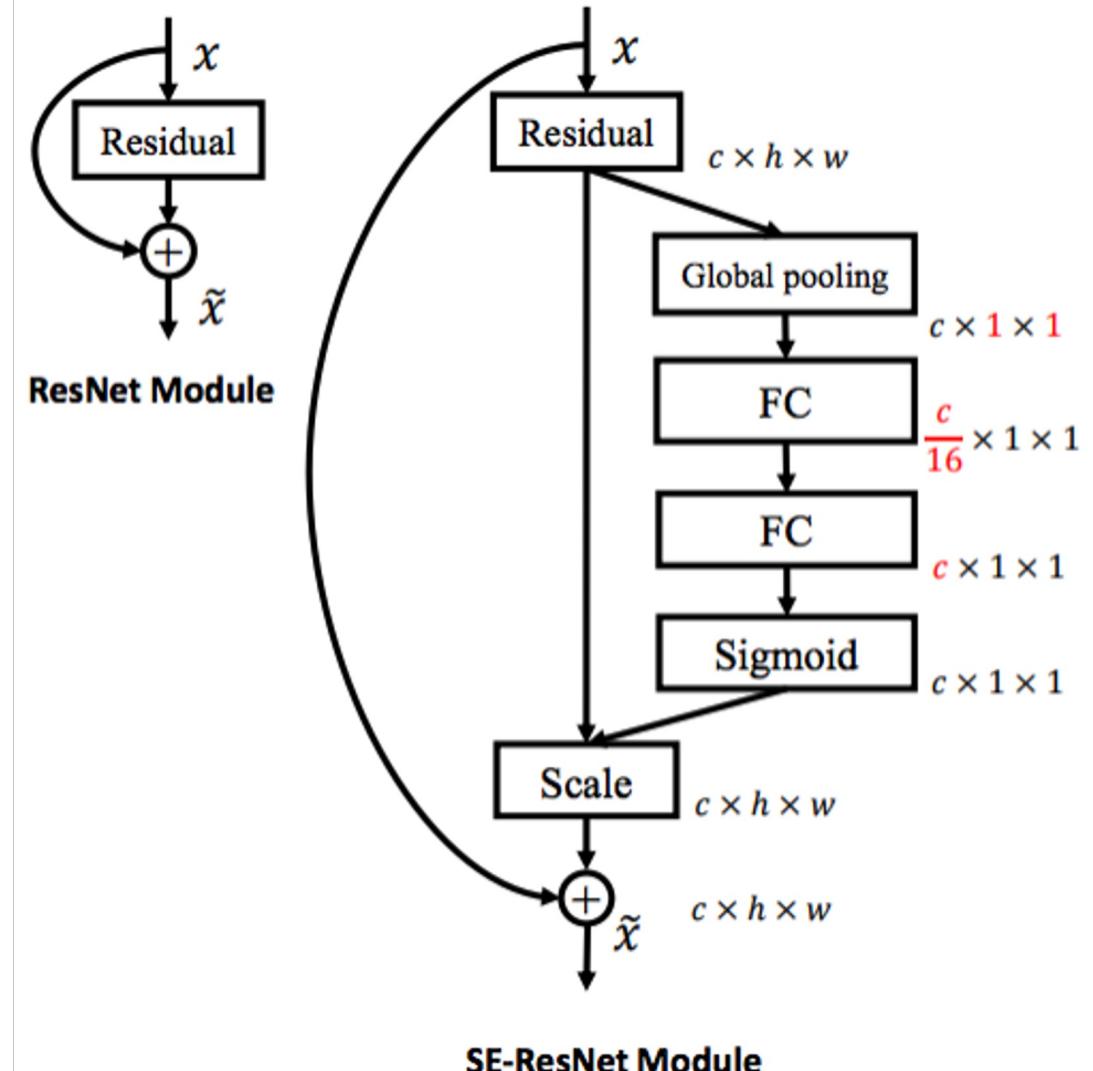


# Squeeze-and-Excitation Networks

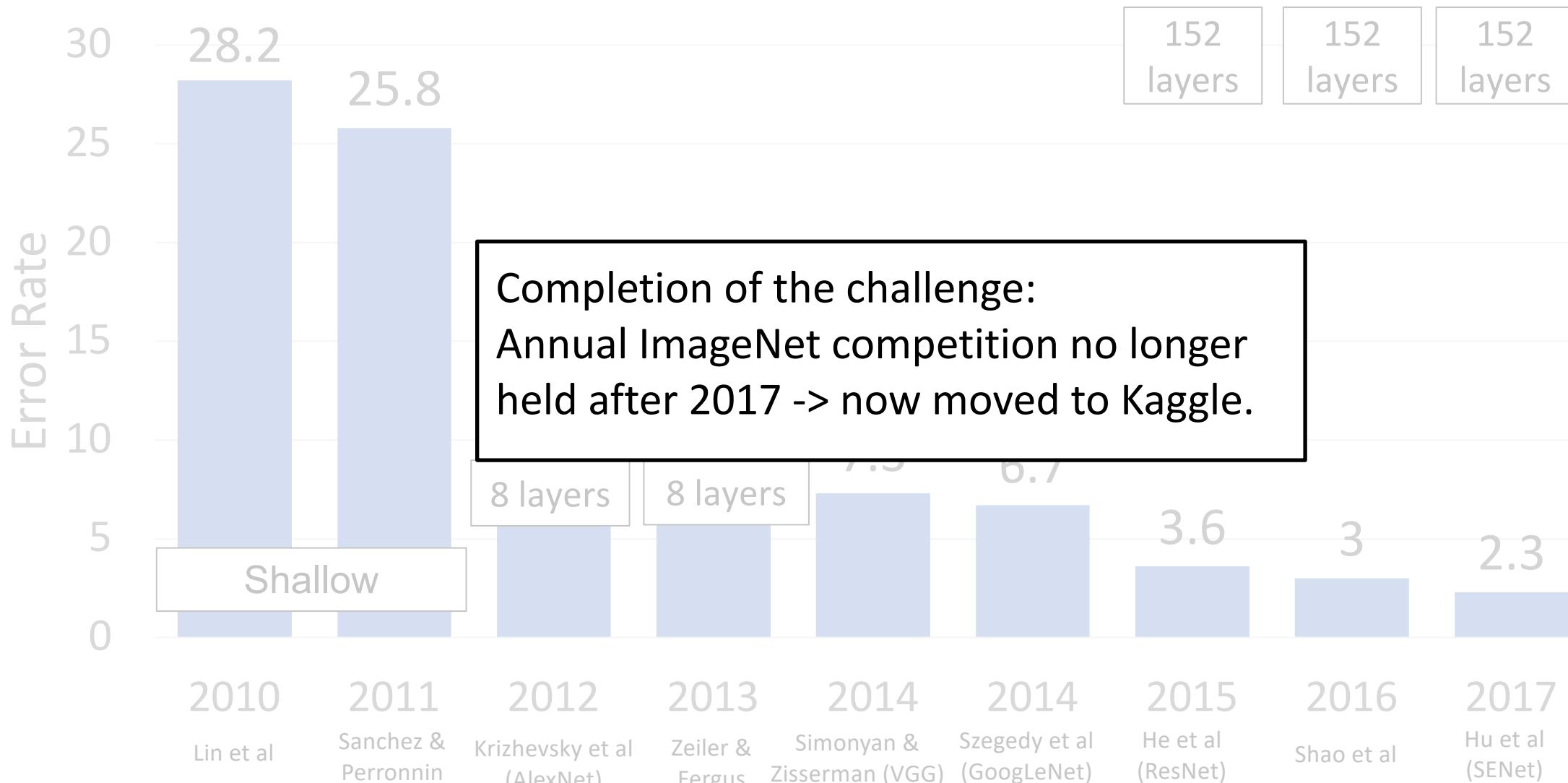
Adds a "Squeeze-and-excite" branch to each residual block that performs global pooling, full-connected layers, and multiplies back onto feature map

Adds **global context** to each residual block!

Won ILSVRC 2017 with ResNeXt-152-SE



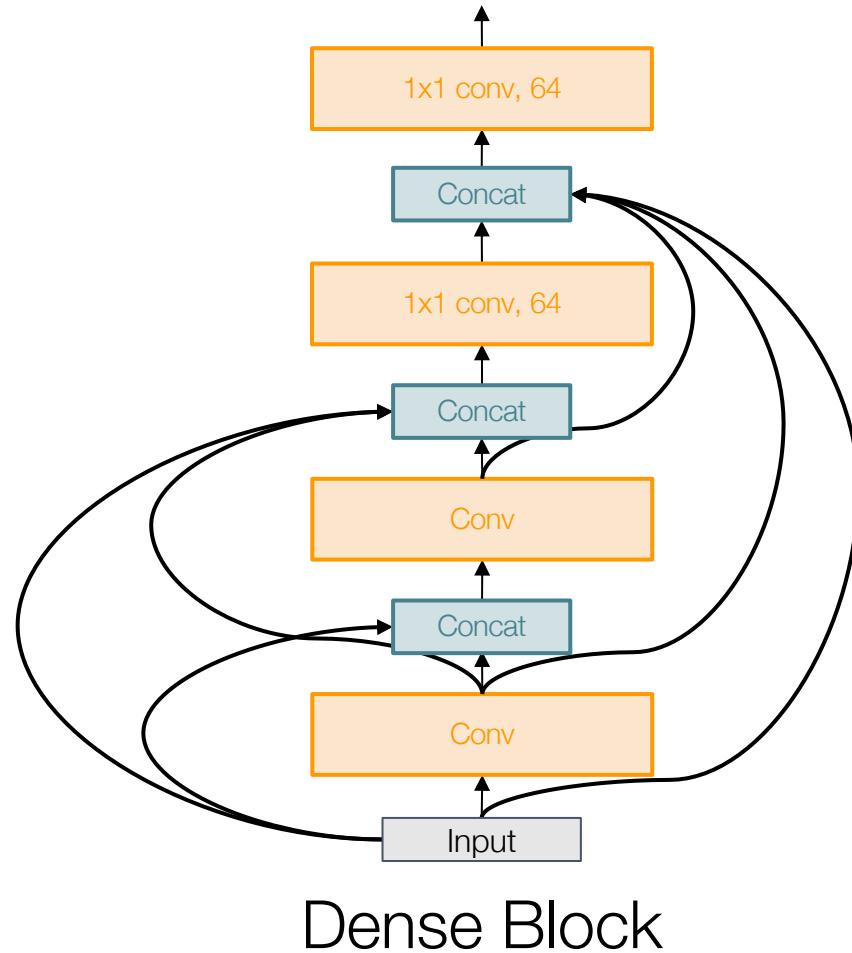
# ImageNet Classification Challenge



# Densely Connected Neural Networks

Dense blocks where each layer is connected to every other layer in feedforward fashion

Alleviates vanishing gradient, strengthens feature propagation, encourages feature reuse

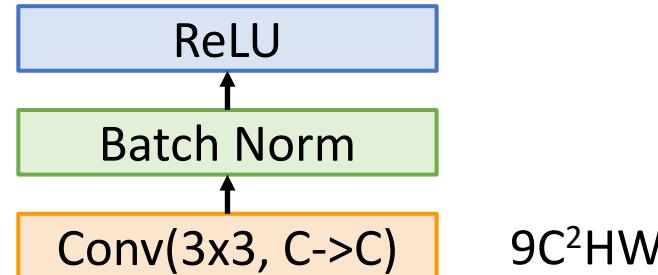


Softmax
FC
Pool
Dense Block 3
Conv
Pool
Conv
Dense Block 2
Conv
Pool
Conv
Dense Block 1
Conv
Input

# MobileNets: Tiny Networks (For Mobile Devices)

## Standard Convolution Block

Total cost:  $9C^2HW$

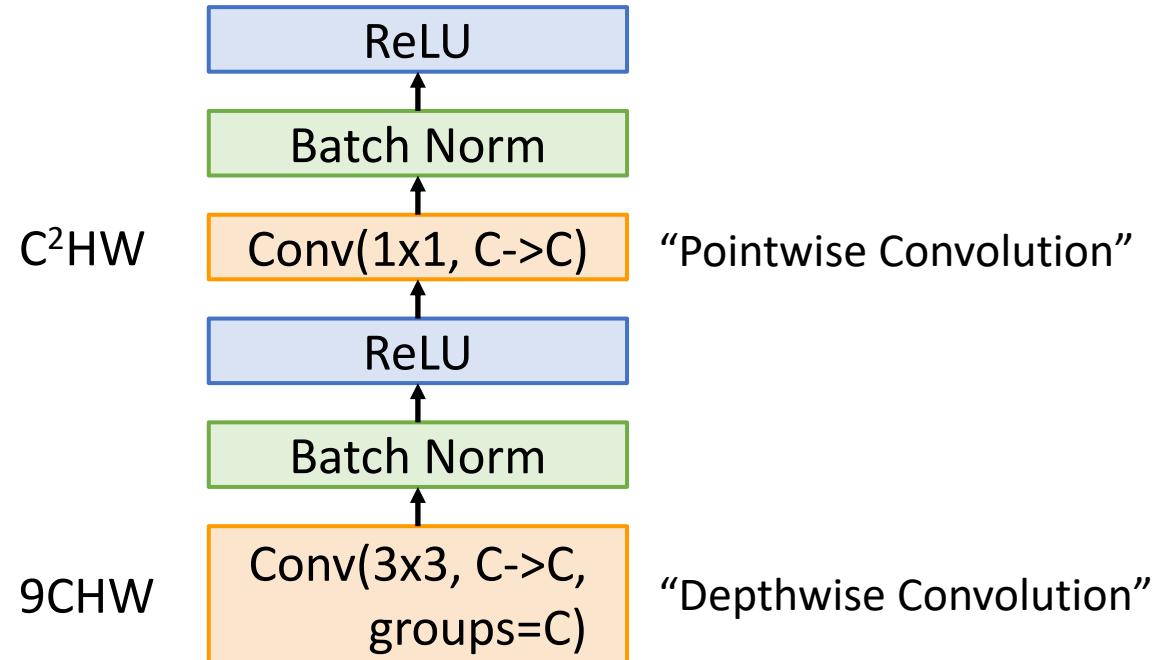


$9C^2HW$

$$\begin{aligned} \text{Speedup} &= 9C^2/(9C+C^2) \\ &= 9C/(9+C) \\ &\Rightarrow 9 \text{ (as } C \text{ is very large)} \end{aligned}$$

## Depthwise Separable Convolution

Total cost:  $(9C + C^2)HW$



# MobileNets: Tiny Networks (For Mobile Devices)

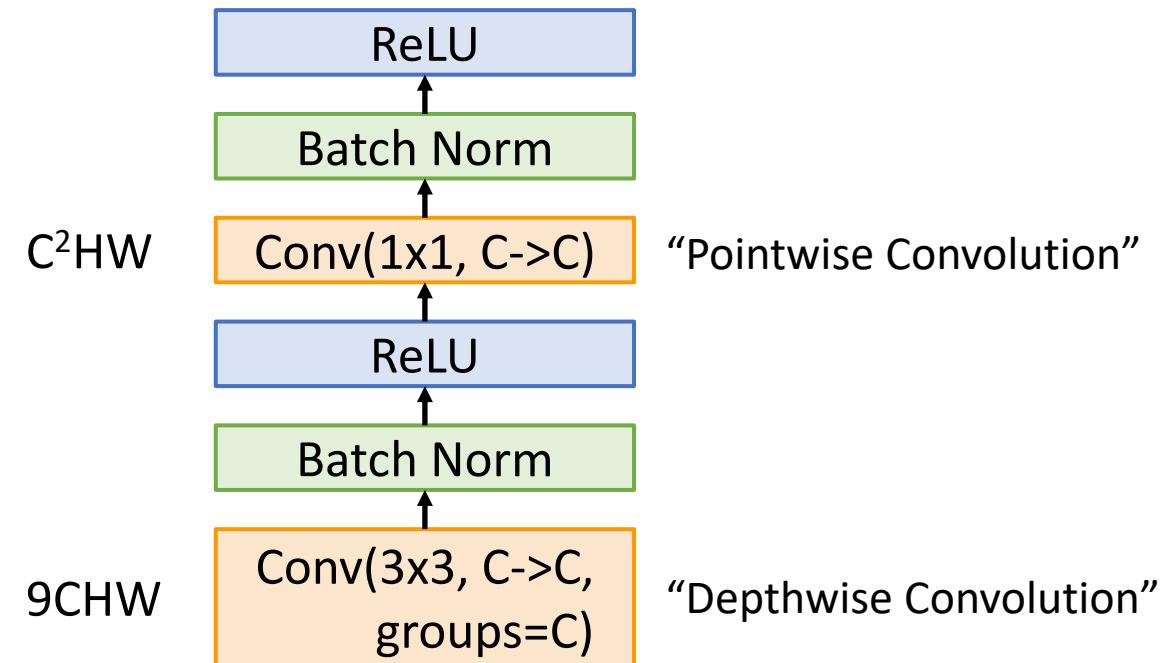
**Depthwise Separable Convolution**  
Total cost:  $(9C + C^2)HW$

Also related:

ShuffleNet: Zhang et al, CVPR 2018

MobileNetV2: Sandler et al, CVPR 2018

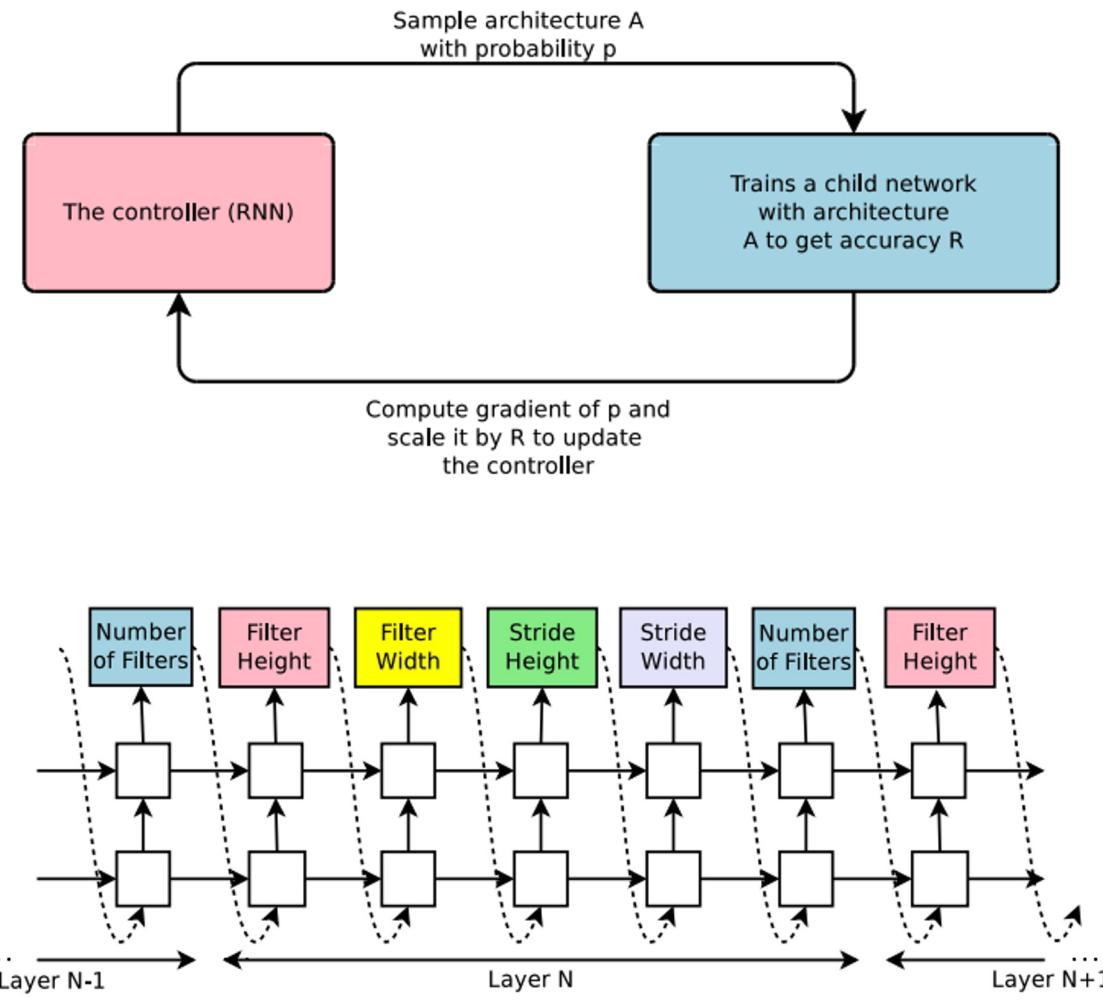
ShuffleNetV2: Ma et al, ECCV 2018



# Neural Architecture Search

Designing neural network architectures is hard – let's automate it!

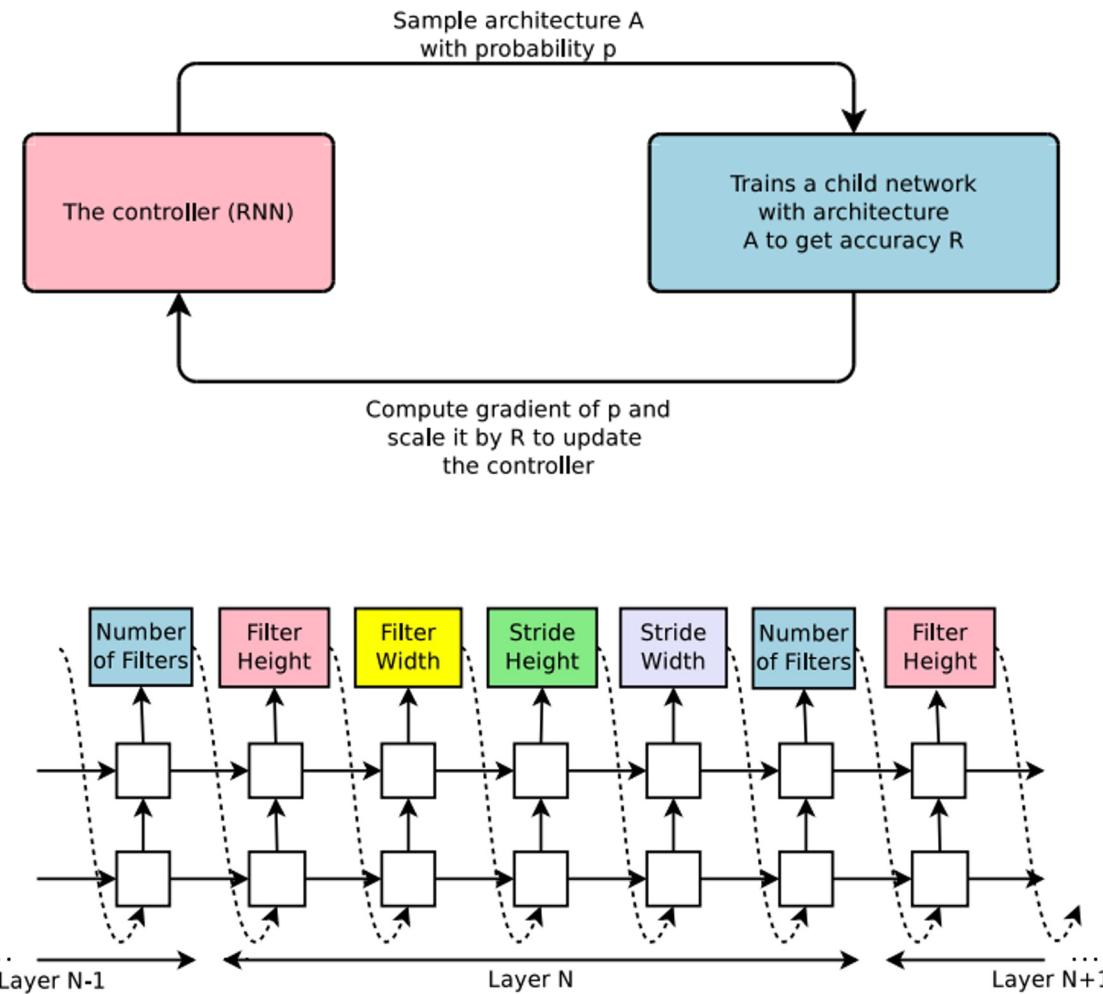
- One network (**controller**) outputs network architectures
- Sample **child networks** from controller and train them
- After training a batch of child networks, make a gradient step on controller network (Using **policy gradient**)
- Over time, controller learns to output good architectures!



# Neural Architecture Search

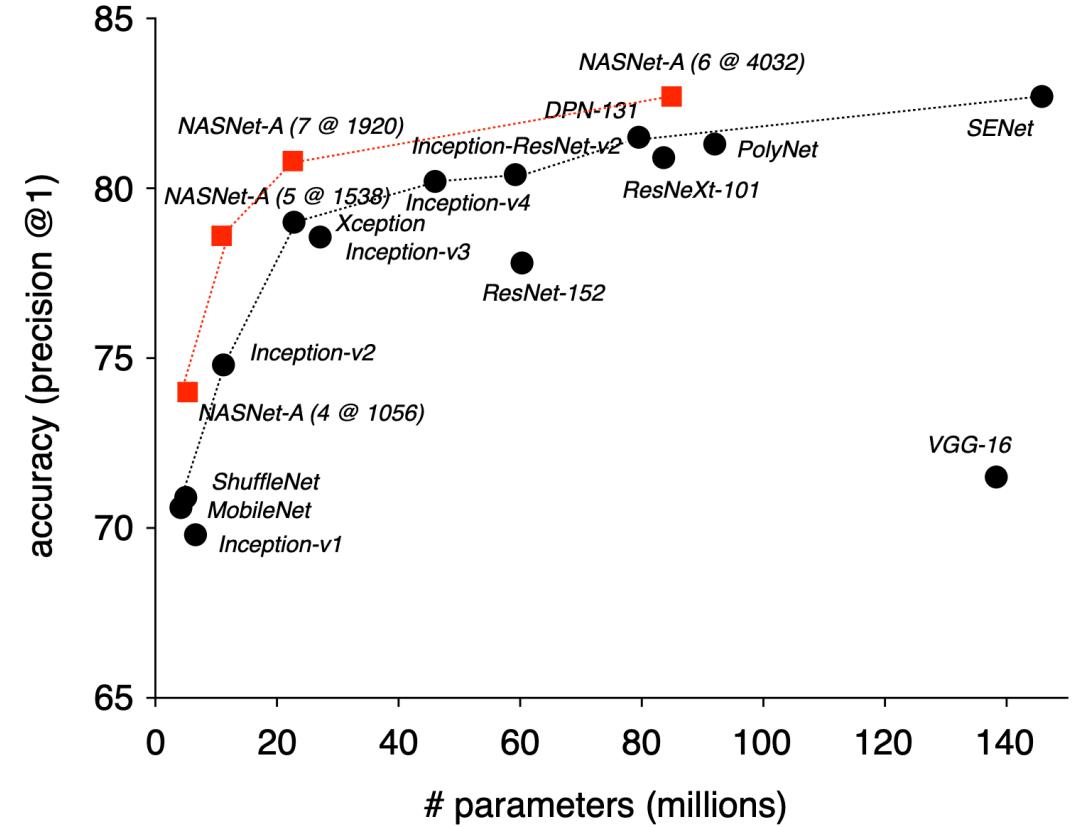
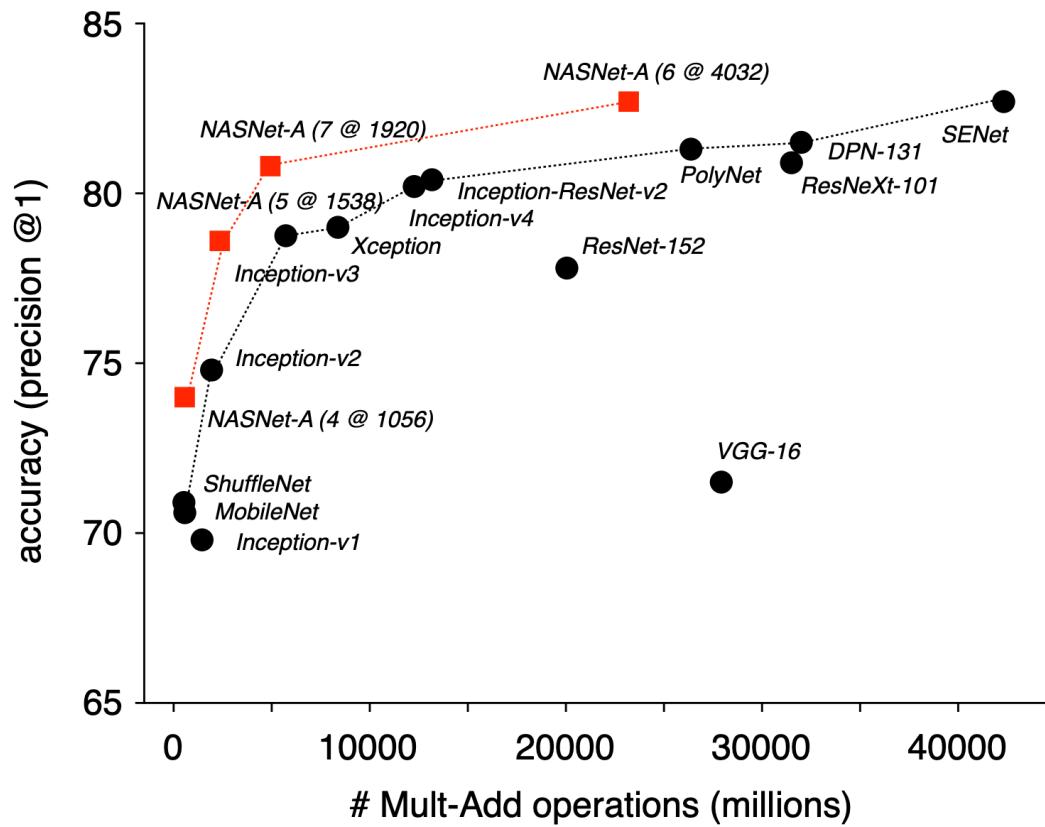
Designing neural network architectures is hard – let's automate it!

- One network (**controller**) outputs network architectures
- Sample **child networks** from controller and train them
- After training a batch of child networks, make a gradient step on controller network (Using **policy gradient**)
- Over time, controller learns to output good architectures!
- **VERY EXPENSIVE!!** Each gradient step on controller requires training a batch of child models!
- Original paper trained on 800 GPUs for 28 days!
- Followup work has focused on efficient search



# Neural Architecture Search

Neural architecture search can be used to find efficient CNN architectures!



# CNN Architectures Summary

Early work (AlexNet -> ZFNet -> VGG) shows that **bigger networks work better**

GoogLeNet one of the first to focus on **efficiency** (aggressive stem, 1x1 bottleneck convolutions, global avg pool instead of FC layers)

ResNet showed us how to train extremely deep networks – limited only by GPU memory! Started to show diminishing returns as networks got bigger

After ResNet: **Efficient networks** became central: how can we improve the accuracy without increasing the complexity?

Lots of **tiny networks** aimed at mobile devices: MobileNet, ShuffleNet, etc

**Neural Architecture Search** promises to automate architecture design

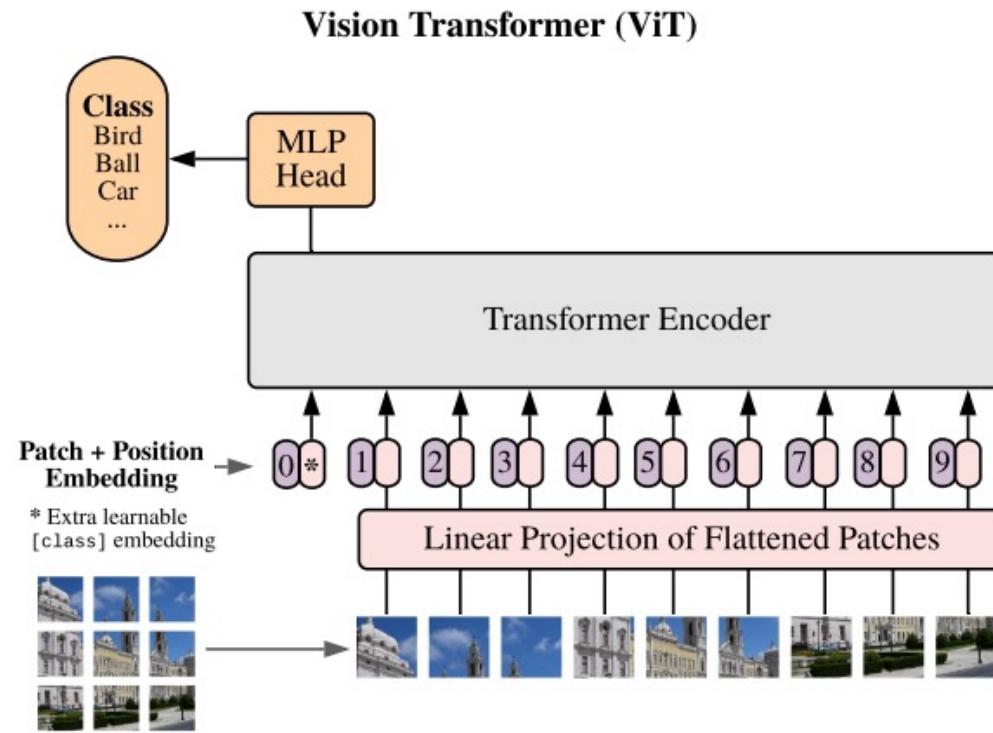
# Which Architecture should I use?

**Don't be a hero.** For most problems you should use an off-the-shelf architecture; don't try to design your own!

If you just care about accuracy, **ResNet-50** or **ResNet-101** are great choices

If you want an efficient network (real-time, run on mobile, etc) try  
**MobileNets** and **ShuffleNets**

# Uprising of Vision Transformer (ViT) in 2021



Published as a conference paper at ICLR 2021

## AN IMAGE IS WORTH 16x16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE

Alexey Dosovitskiy\*,†, Lucas Beyer\*, Alexander Kolesnikov\*, Dirk Weissenborn\*,  
Xiaohua Zhai\*, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer,  
Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, Neil Houlsby\*,†

\*equal technical contribution, †equal advising

Google Research, Brain Team

{adosovitskiy, neilhoulsby}@google.com

An Image is Worth 16x16 Words

by A Dosovitskiy · 2020 · Cited by 2144 ·

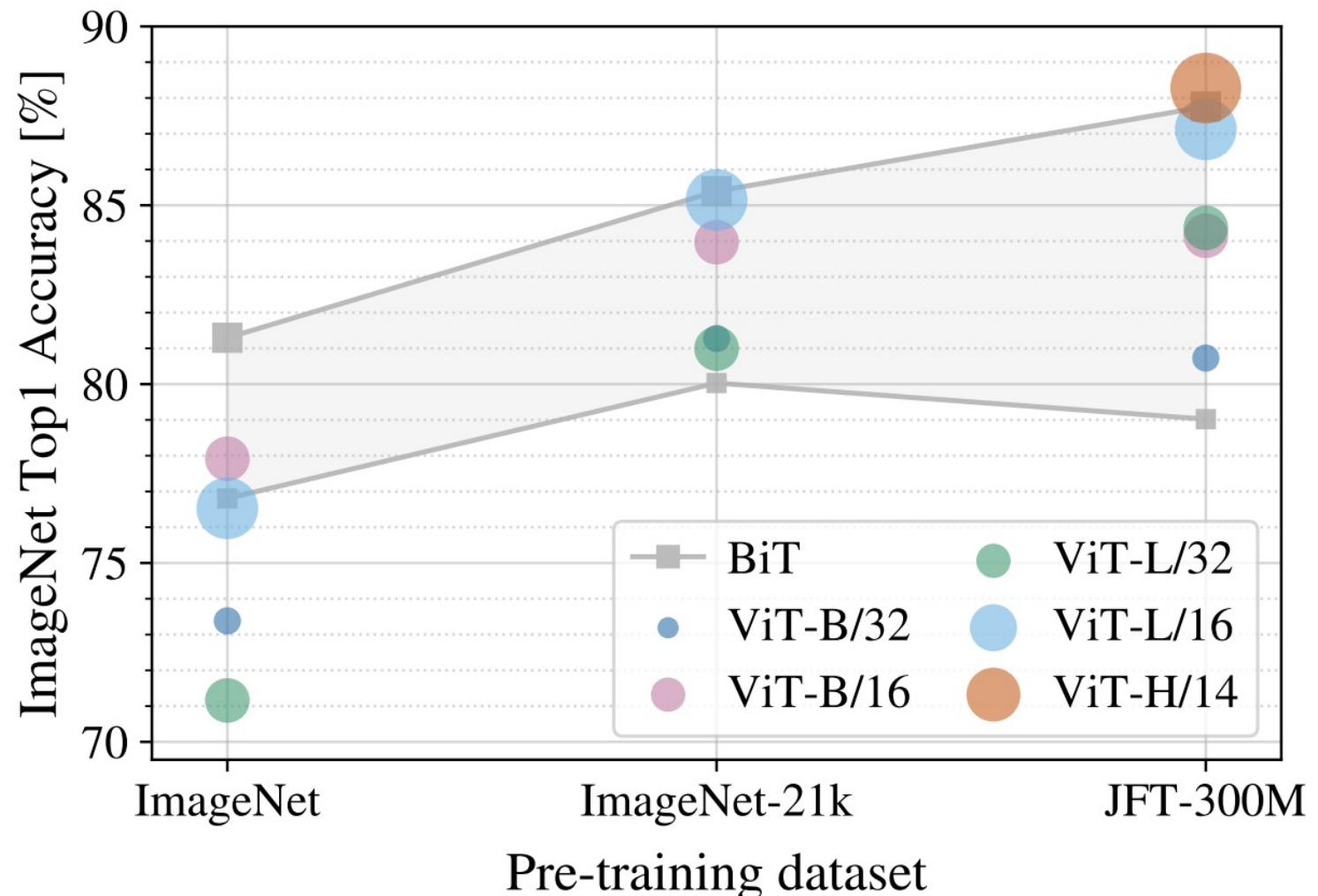
# Uprising of Vision Transformer (ViT) in 2021

ImageNet-21k has 14M images with 21k categories

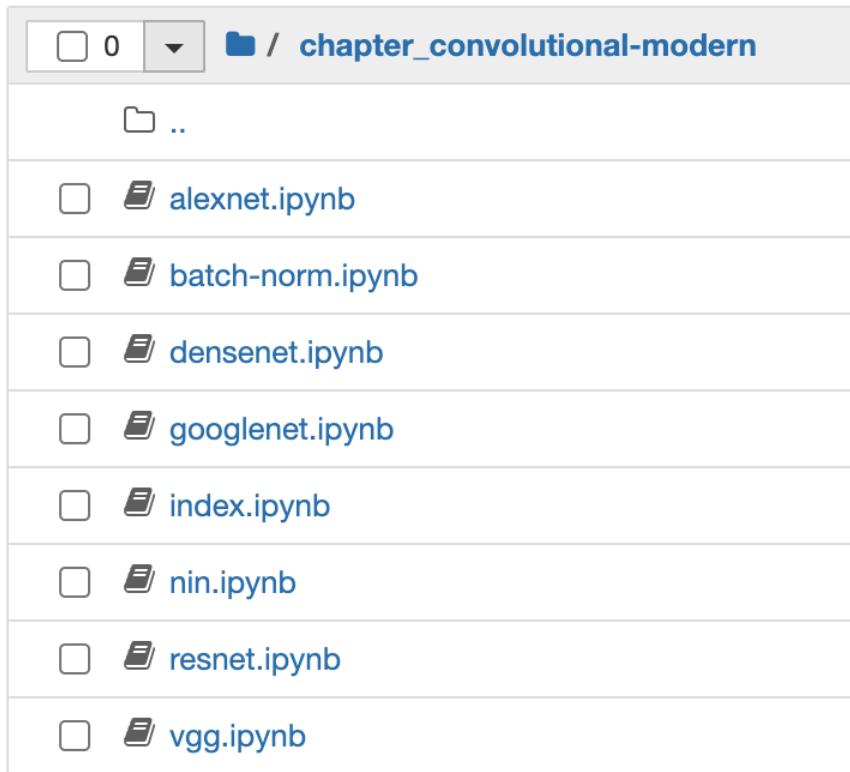
If you pretrain on ImageNet-21k and fine-tune on ImageNet, ViT does better: big ViTs match big ResNets

JFT-300M is an internal Google dataset with 300M labeled images

If you pretrain on JFT and finetune on ImageNet, large ViTs outperform large ResNets



# Reading and Practice



PyTorch official model zoo

<https://github.com/pytorch/vision/tree/main/torchvision/models>

D2L: Chapter 7. Convolutional Neural Networks

D2L: Chapter 8. Modern Convolutional Neural Networks

```
import torch
from torch import nn
from d2l import torch as d2l

def vgg_block(num_convs, in_channels, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
        in_channels = out_channels
    layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
    return nn.Sequential(*layers)

def vgg(conv_arch):
    conv_bcls = []
    in_channels = 1
    # The convolutional part
    for (num_convs, out_channels) in conv_arch:
        conv_bcls.append(vgg_block(num_convs, in_channels, out_channels))
        in_channels = out_channels

    return nn.Sequential(
        *conv_bcls, nn.Flatten(),
        # The fully-connected part
        nn.Linear(out_channels * 7 * 7, 4096), nn.ReLU(), nn.Dropout(0.5),
        nn.Linear(4096, 4096), nn.ReLU(), nn.Dropout(0.5),
        nn.Linear(4096, 10))

net = vgg(conv_arch)
```

Next Time:  
Training NNs