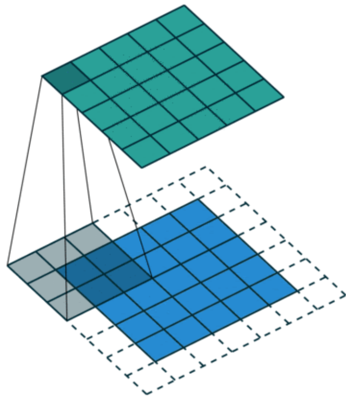
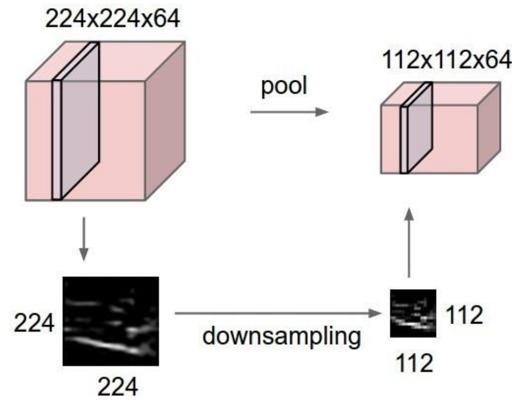
Deep Learning

Components of a Convolutional Network

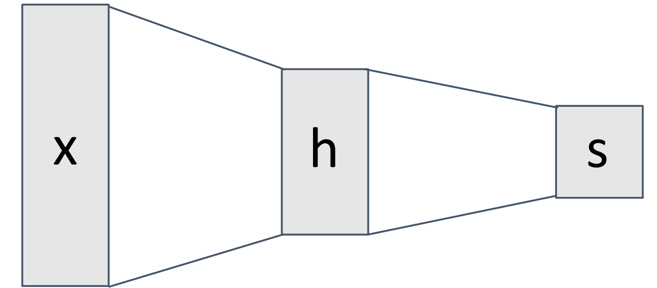
Convolution Layers



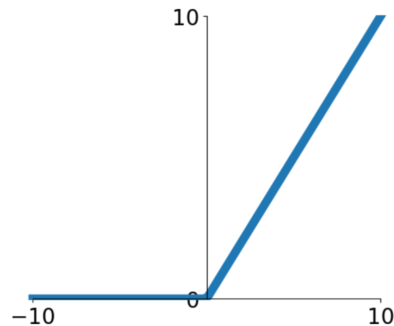
Pooling Layers



Fully-Connected Layers



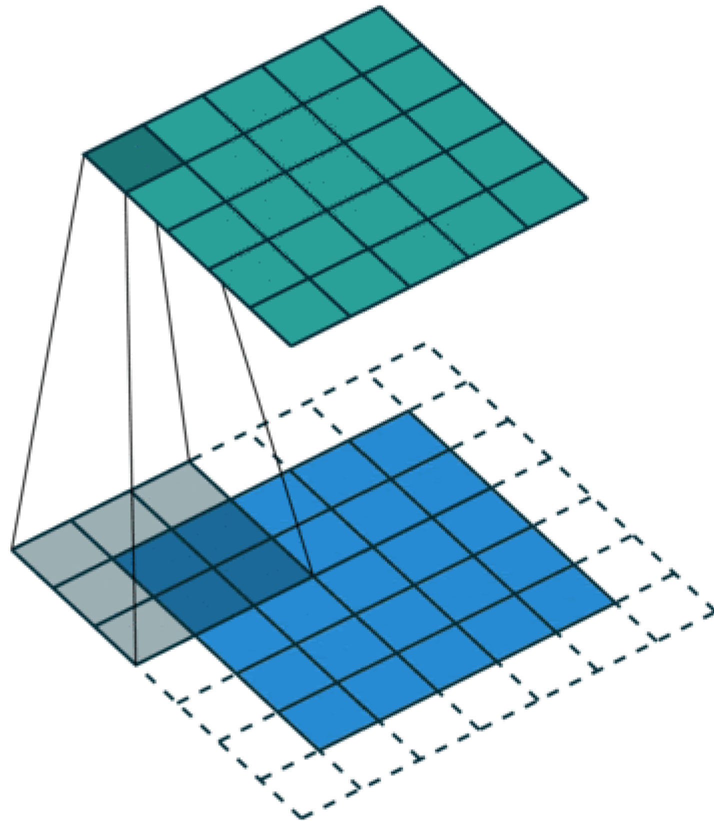
Activation Function



Normalization

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

Convolution Layers



The Convolution operation

Assume a 6x6 matrix **M** as input. The 2D convolution of **M** with *filter (or kernel)* **F** and *stride* 1 is a 4x4 matrix **CM** (sometimes called *feature map*) computed as follows:

3	0	1	2	4	7
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

M

*

1	0	-1
1	0	-1
1	0	-1

F

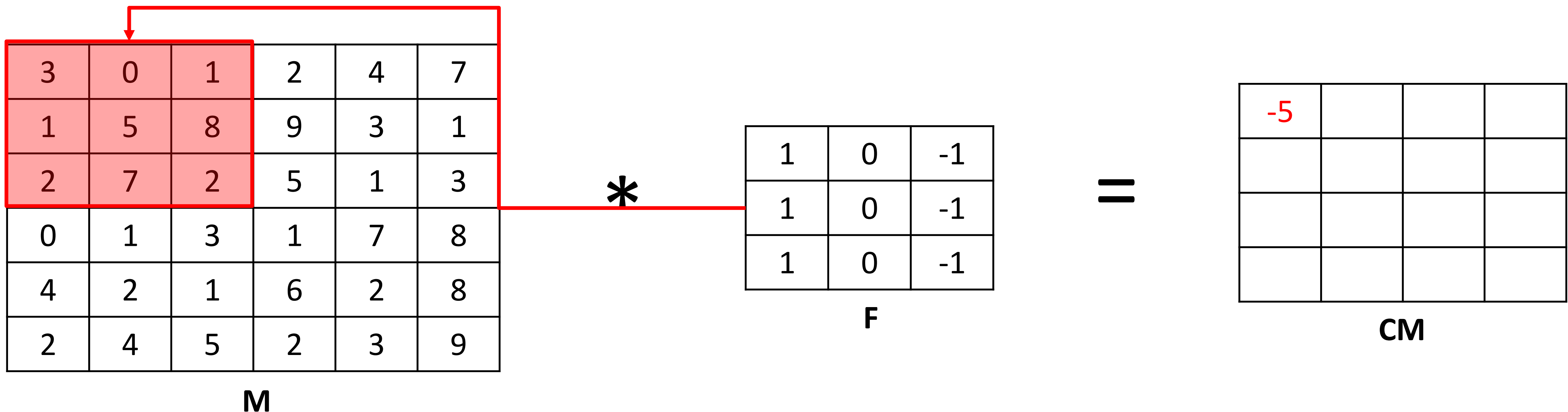
=

CM

convolution operator

The Convolution operation

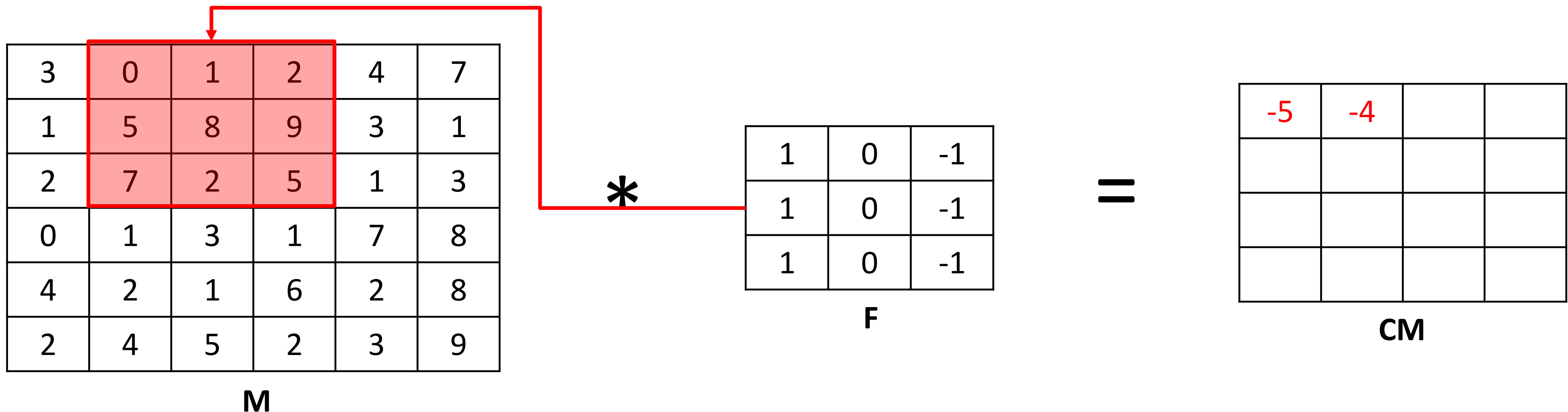
Assume a 6x6 matrix **M** as input. The 2D convolution of **M** with *filter (or kernel)* **F** and *stride* 1 is a 4x4 matrix **CM** (sometimes called *feature map*) computed as follows:



3*1 + 1*1 + 2*1 + 0*0 + 5*0 + 7*0 + 1*(-1) + 8*(-1) + 2*(-1)

The Convolution operation

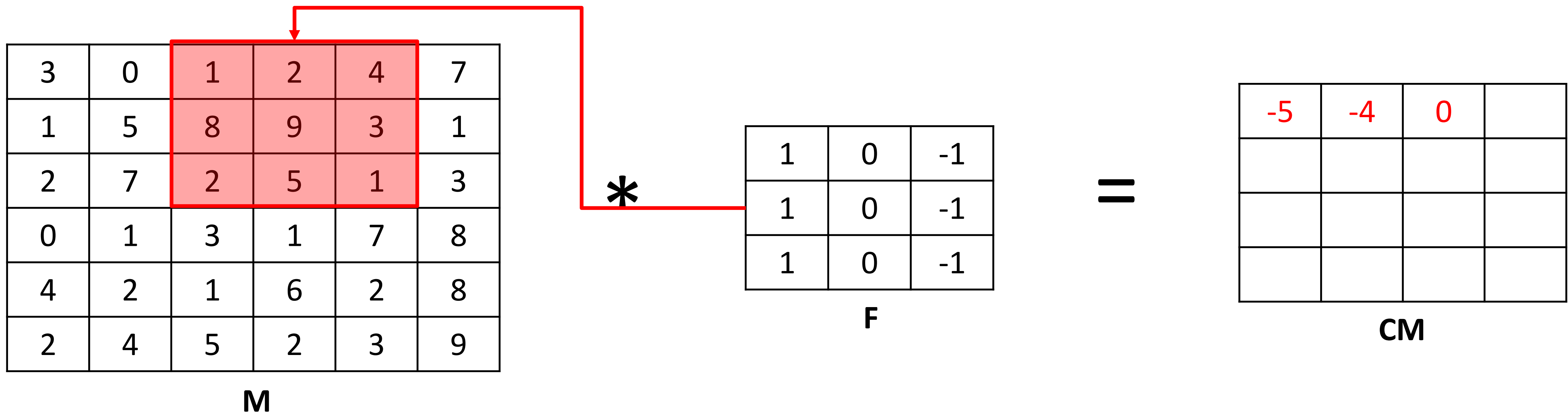
Assume a 6x6 matrix **M** as input. The 2D convolution of **M** with *filter (or kernel)* **F** and *stride* 1 is a 4x4 matrix **CM** (sometimes called *feature map*) computed as follows:



0*1 + 5*1 + 7*1 + 1*0 + 8*0 + 2*0 + 2*(-1) + 9*(-1) + 5*(-1)

The Convolution operation

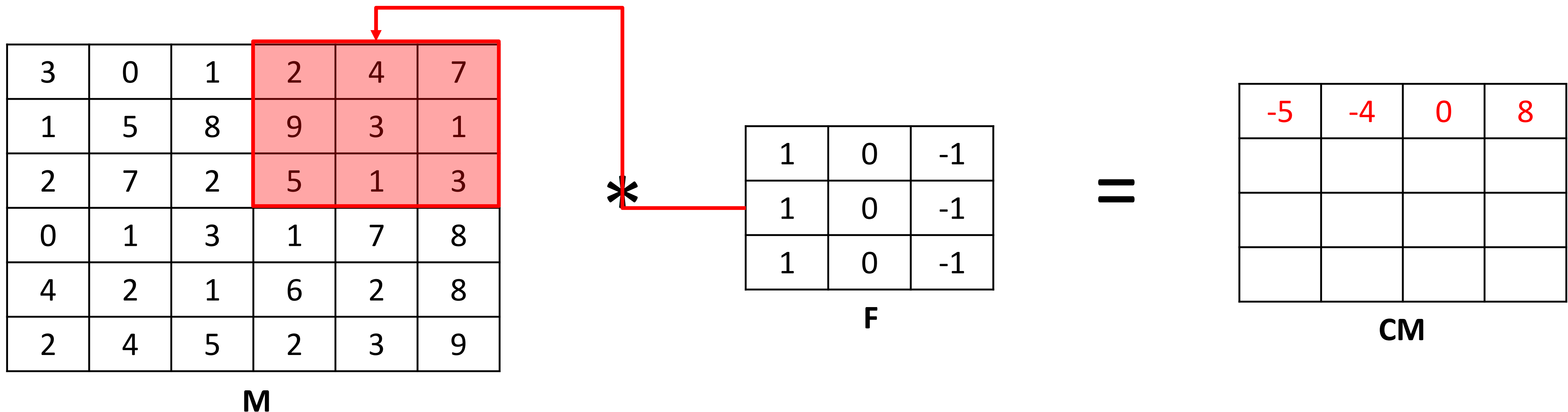
Assume a 6x6 matrix **M** as input. The 2D convolution of **M** with *filter (or kernel)* **F** and *stride* 1 is a 4x4 matrix **CM** (sometimes called *feature map*) computed as follows:



1*1 + 8*1 + 2*1 + 2*0 + 9*0 + 5*0 + 4*(-1) + 3*(-1) + 1*(-1)

The Convolution operation

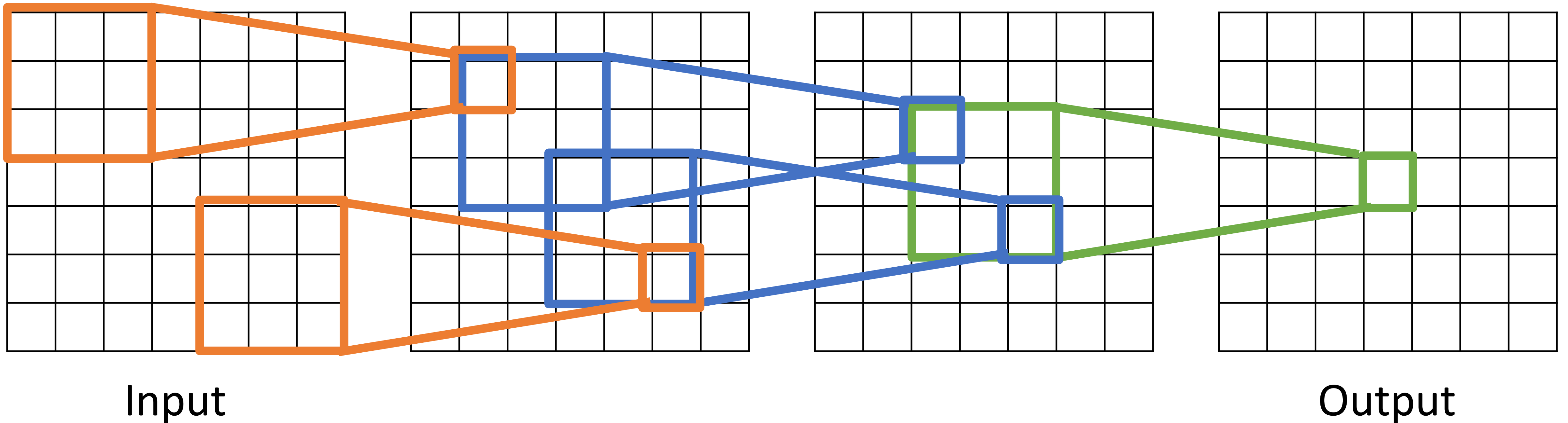
Assume a 6x6 matrix **M** as input. The 2D convolution of **M** with *filter (or kernel)* **F** and *stride* 1 is a 4x4 matrix **CM** (sometimes called *feature map*) computed as follows:



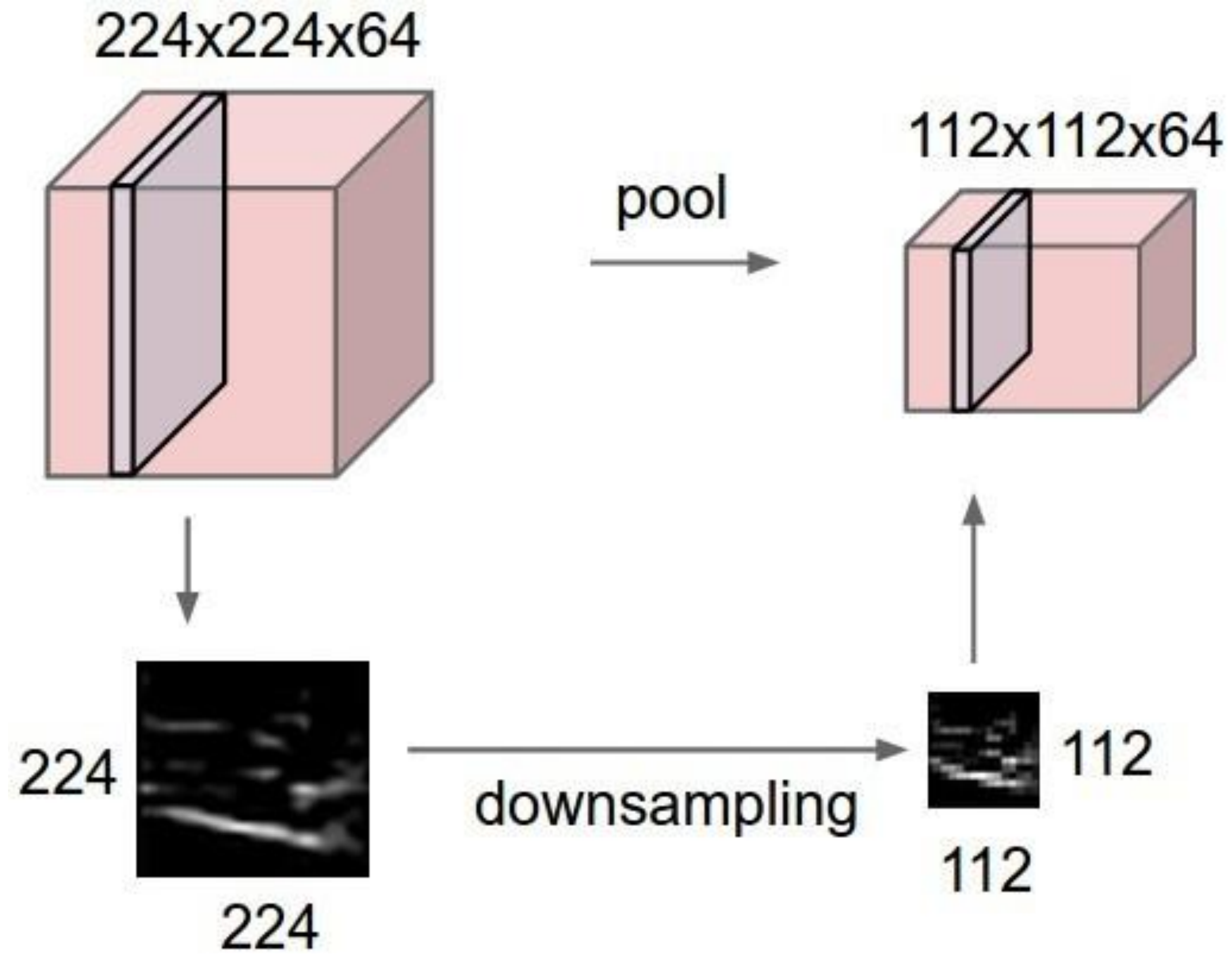
2*1 + 9*1 + 5*1 + 4*0 + 3*0 + 1*0 + 7*(-1) + 1*(-1) + 3*(-1)

Receptive Fields

Each successive convolution adds K – 1 to the receptive field size
With L layers the receptive field size is 1 + L * (K – 1)

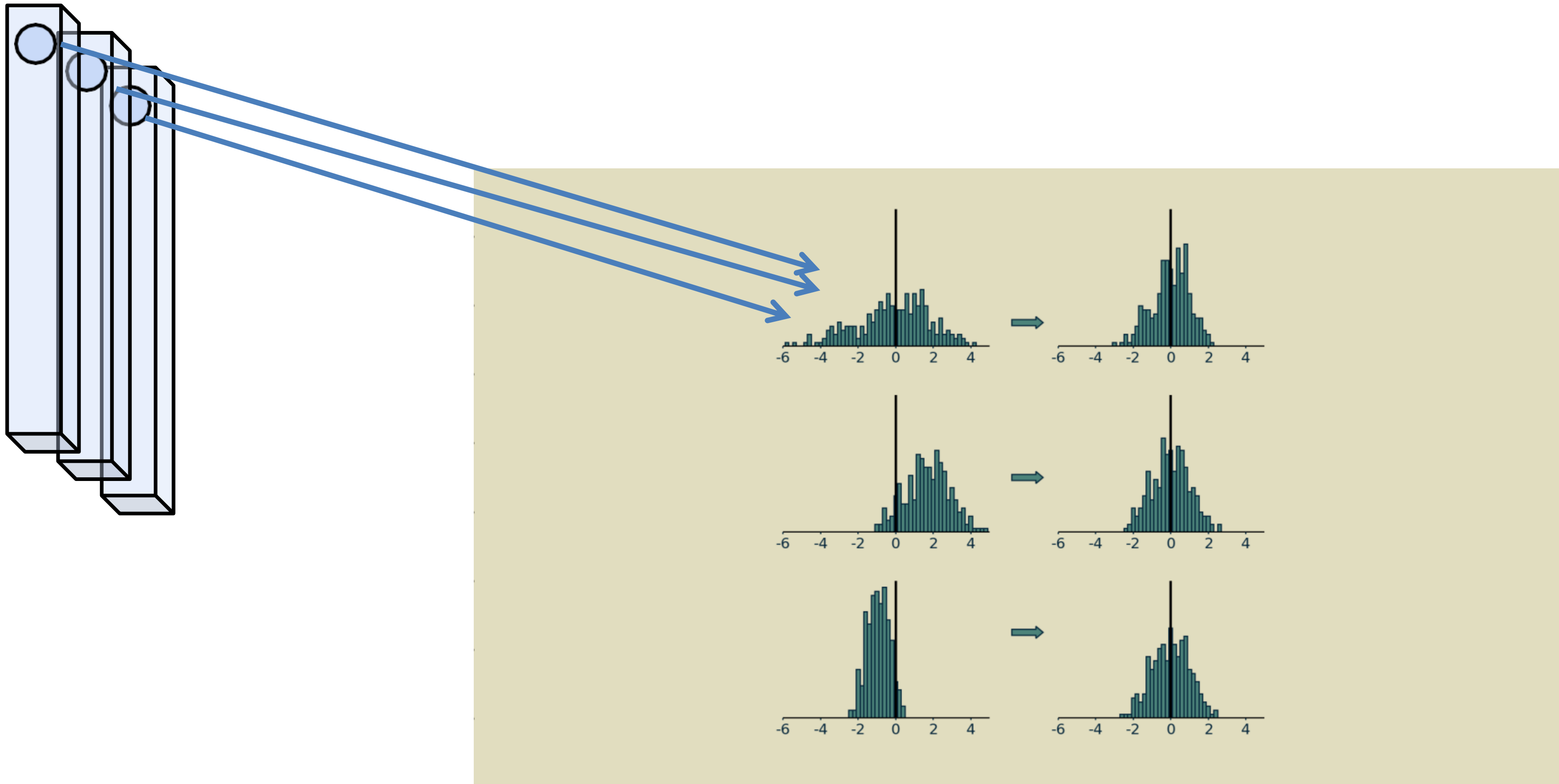


Pooling Layers: Another way to downsample



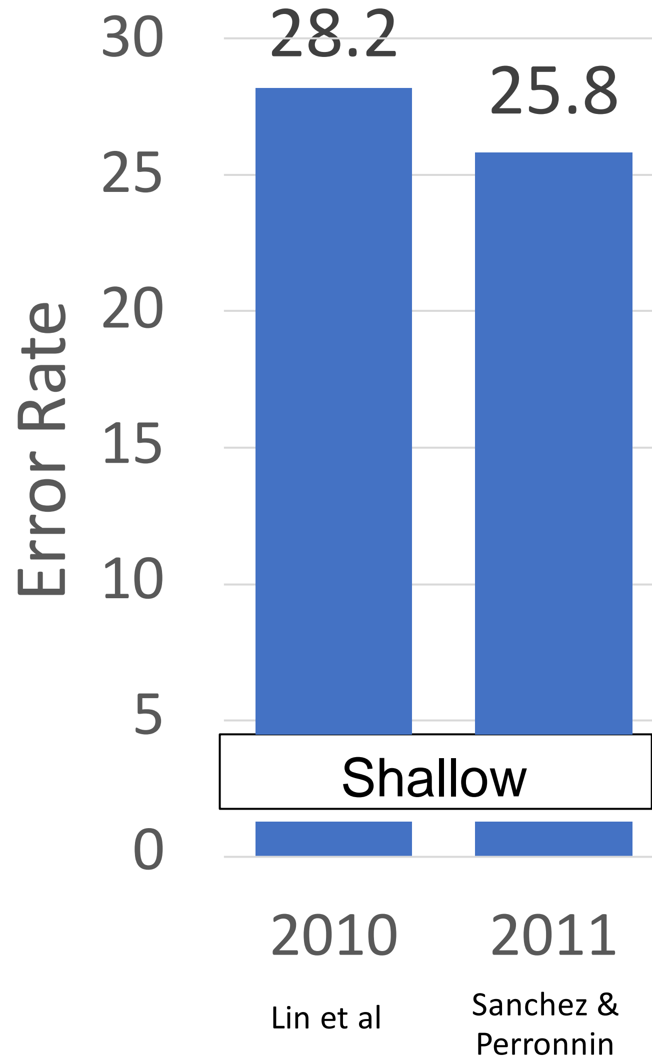
Hyperparameters:
Kernel Size
Stride
Pooling function

Batch Normalization

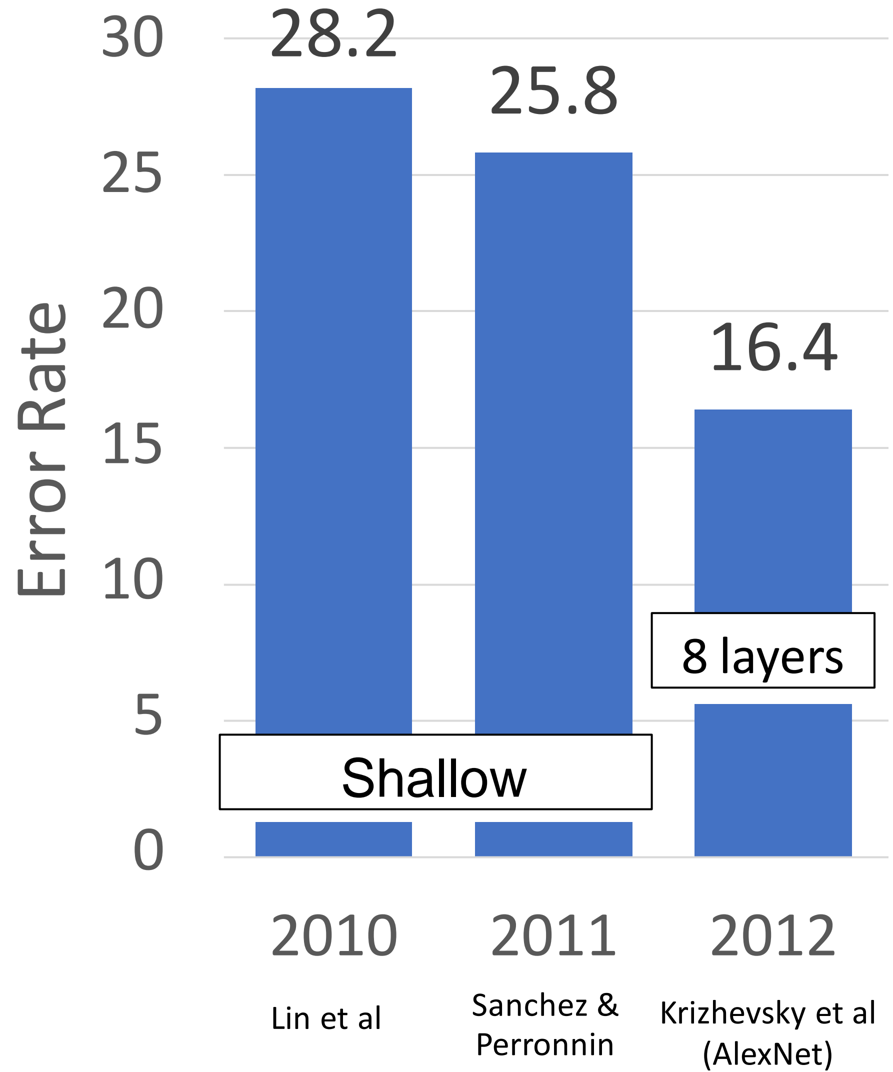


Normalize node activation across a batch

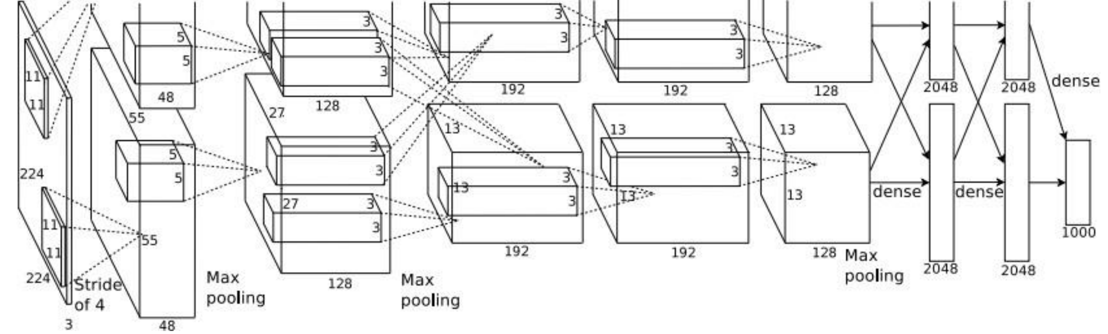
ImageNet Classification Challenge



ImageNet Classification Challenge

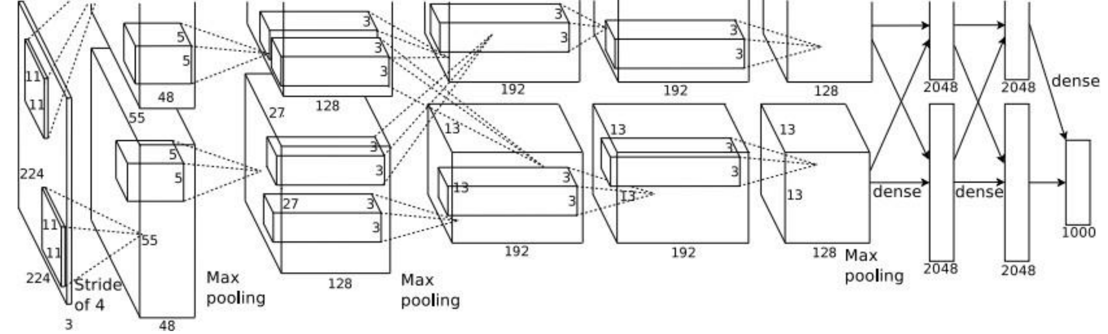


AlexNet



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

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

AlexNet

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

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

Recall: W' = (W – K + 2P) / S + 1
= (227 – 11 + 2*2) / 4 + 1
= 220/4 + 1 = 56

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	

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

Number of output elements = C * H' * W'
= 64*56*56 = 200,704

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

KB = (number of elements) * (bytes per elem) / 1024
= 200704 * 4 / 1024
= **784**

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	

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

Weight shape = C_{out} x C_{in} x K x K
= 64 x 3 x 11 x 11

Bias shape = C_{out} = 64

Number of weights = 64*3*11*11 + 64
= **23,296**

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	

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)
= (number of output elements) * (ops per output elem)
= $(C_{out} \times H' \times W') * (C_{in} \times K \times K)$
= (64 * 56 * 56) * (3 * 11 * 11)
= 200,704 * 363
= **72,855,552**

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					

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:

#output channels = #input channels = 64

W’ = floor((W – K) / S + 1)

= floor(53 / 2 + 1) = floor(27.5) = **27**

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} x H’ x W’

Bytes per elem = 4

KB = C_{out} * H’ * W’ * 4 / 1024

= 64 * 27 * 27 * 4 / 1024

= **182.25**

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!

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

= (number of output positions) * (flops per output position)

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

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

= 419,904

= **0.4 MFLOP**

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

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

Flatten output size = C_{in} x H x W

= 256 * 6 * 6

= **9216**

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

$$\text{FC params} = C_{in} \times C_{out} + C_{out} = 9216 * 4096 + 4096 = 37{,}725{,}832$$

$$\text{FC flops} = C_{in} \times C_{out} = 9216 * 6409 = 37{,}748{,}736$$

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

FC params $= C_{in} \times C_{out} + C_{out}$
$= 9216 * 4096 + 4096$
$= 37{,}725{,}832$

FC flops $= C_{in} \times C_{out}$
$= 9216 * 6409$
$= 37{,}748{,}736$

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

	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

Determined by trial and error

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

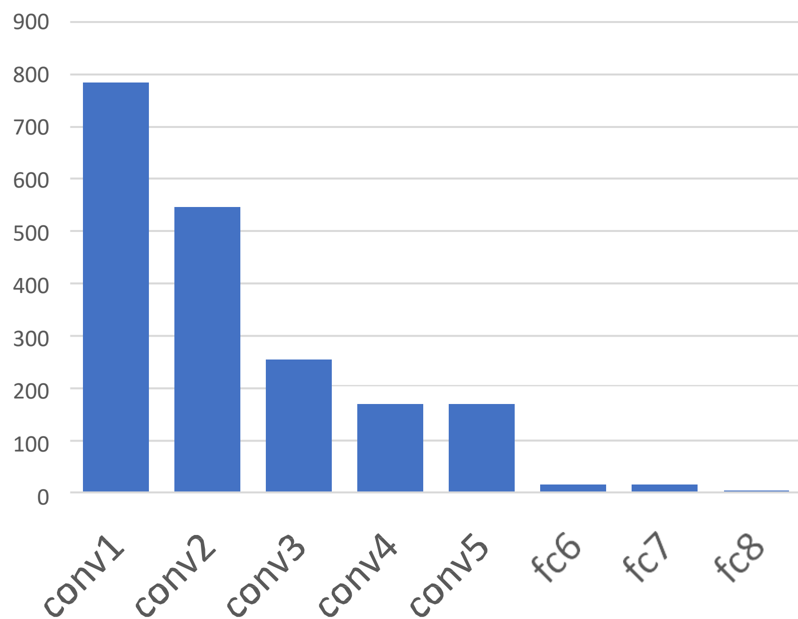
Max pooling inexpensive

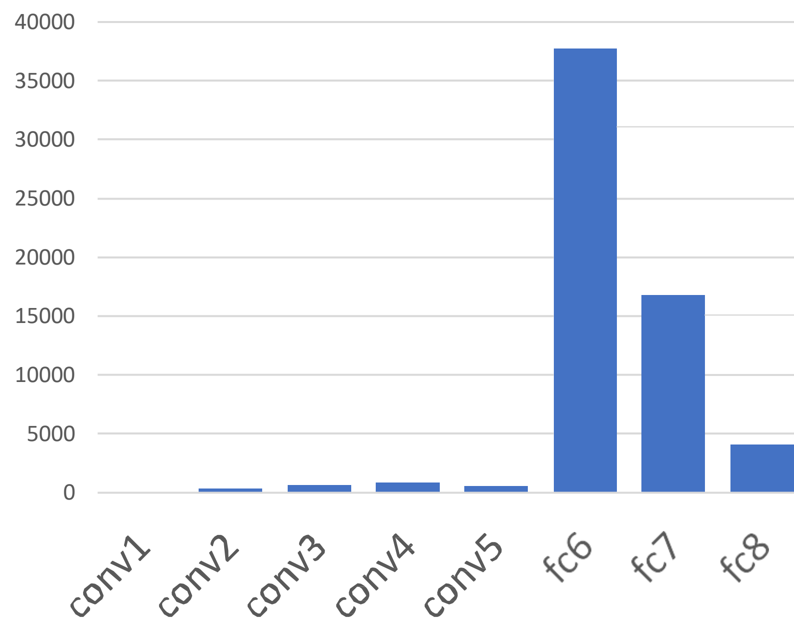
AlexNet

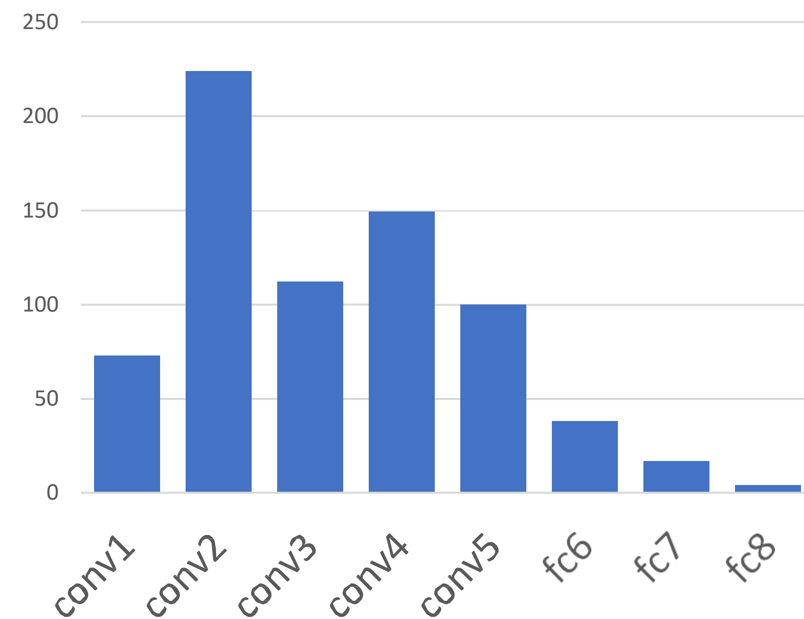
Most of the **memory usage** is in the early convolution layers

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

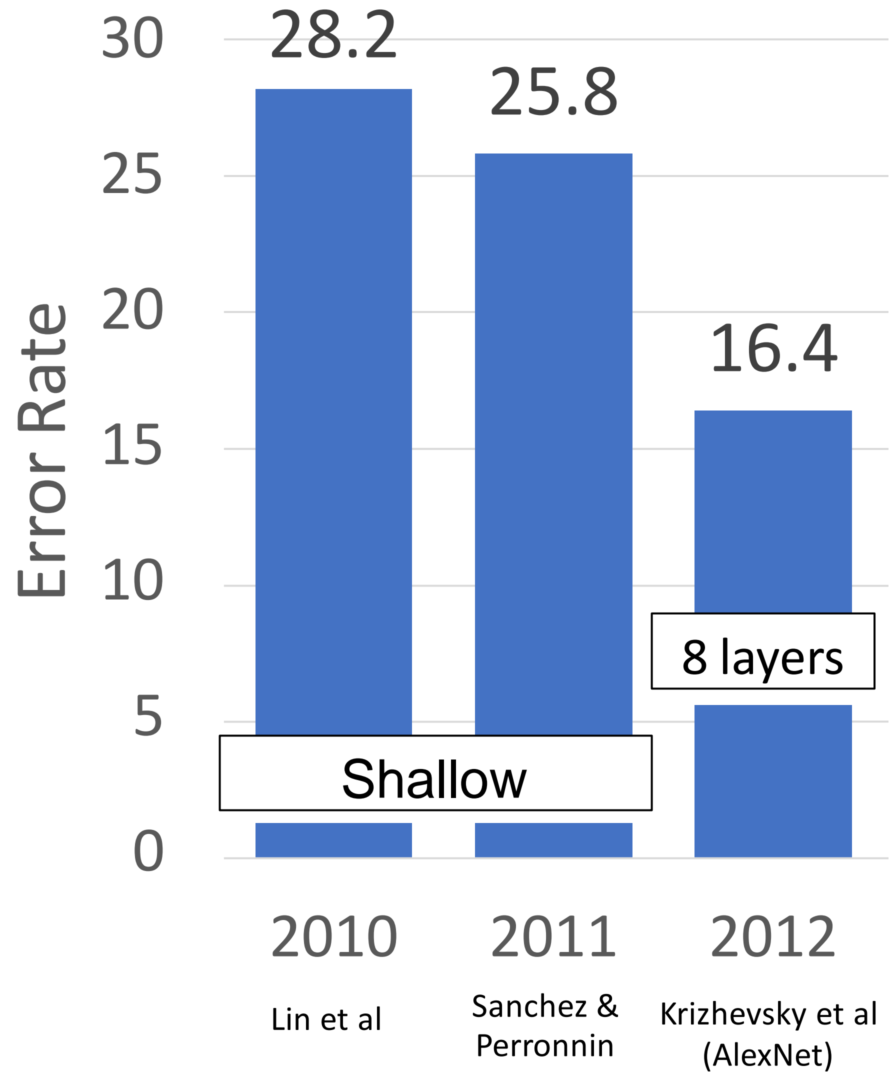
Most **floating-point ops** occur in the convolution layers



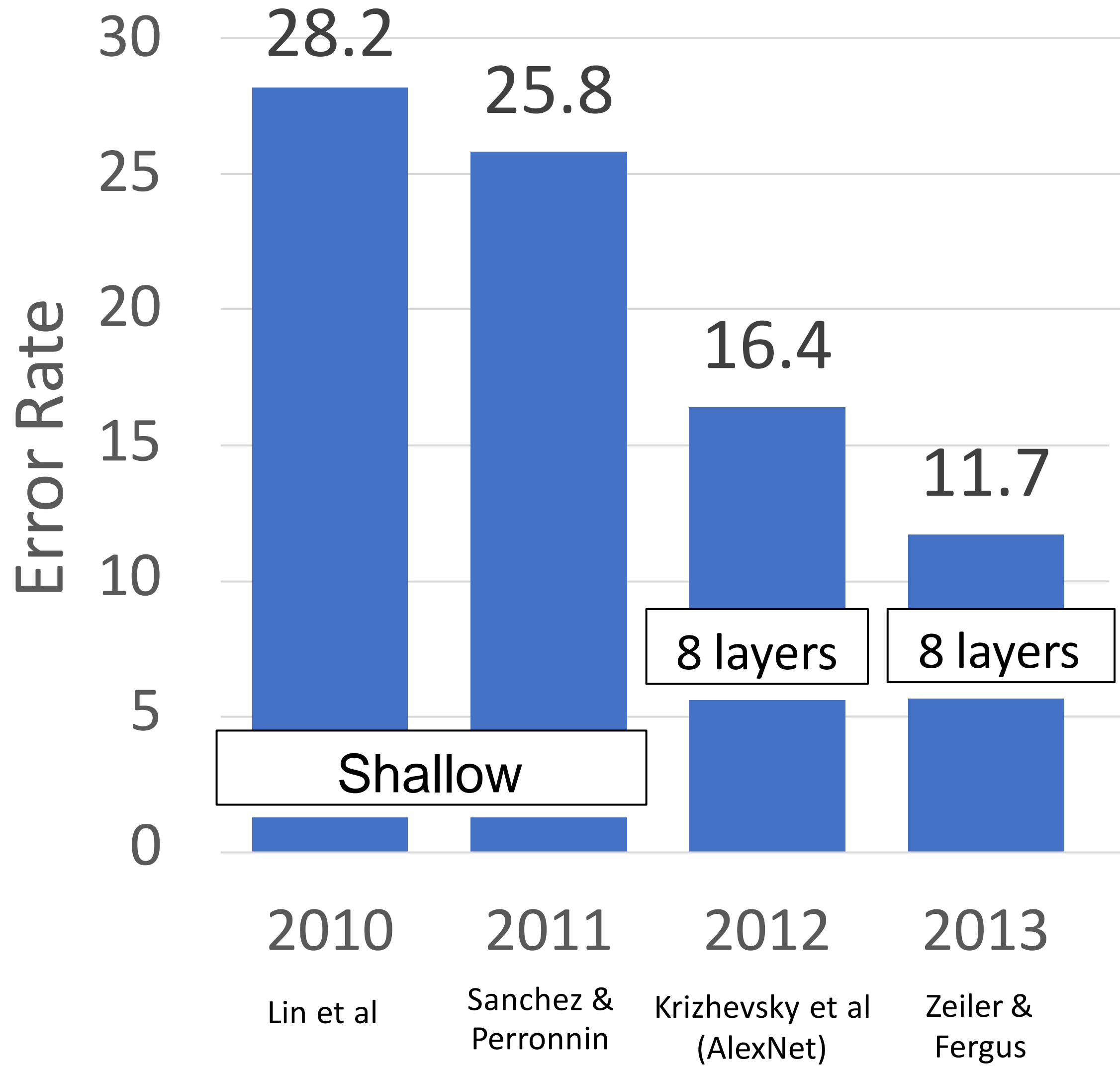




ImageNet Classification Challenge

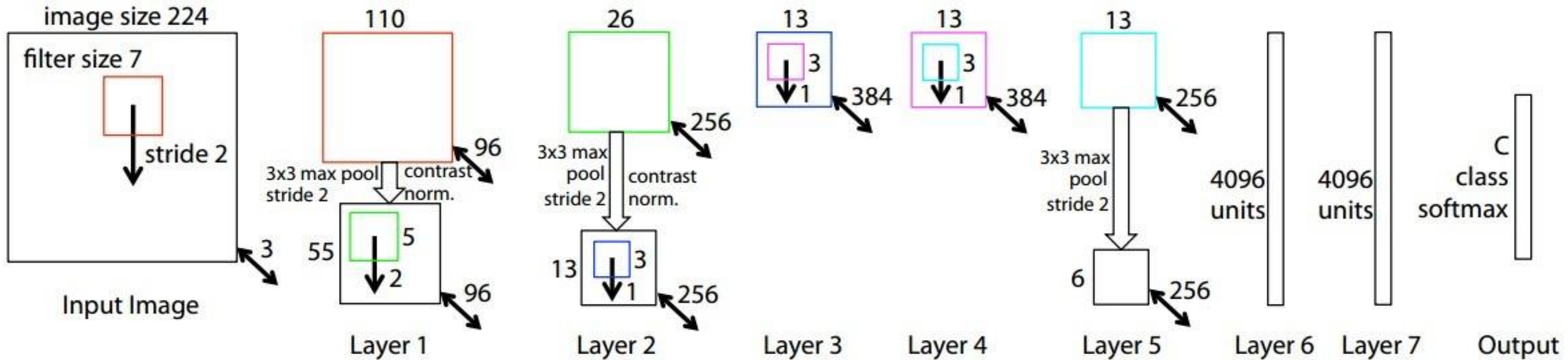


ImageNet Classification Challenge



ZFNet: A Bigger AlexNet

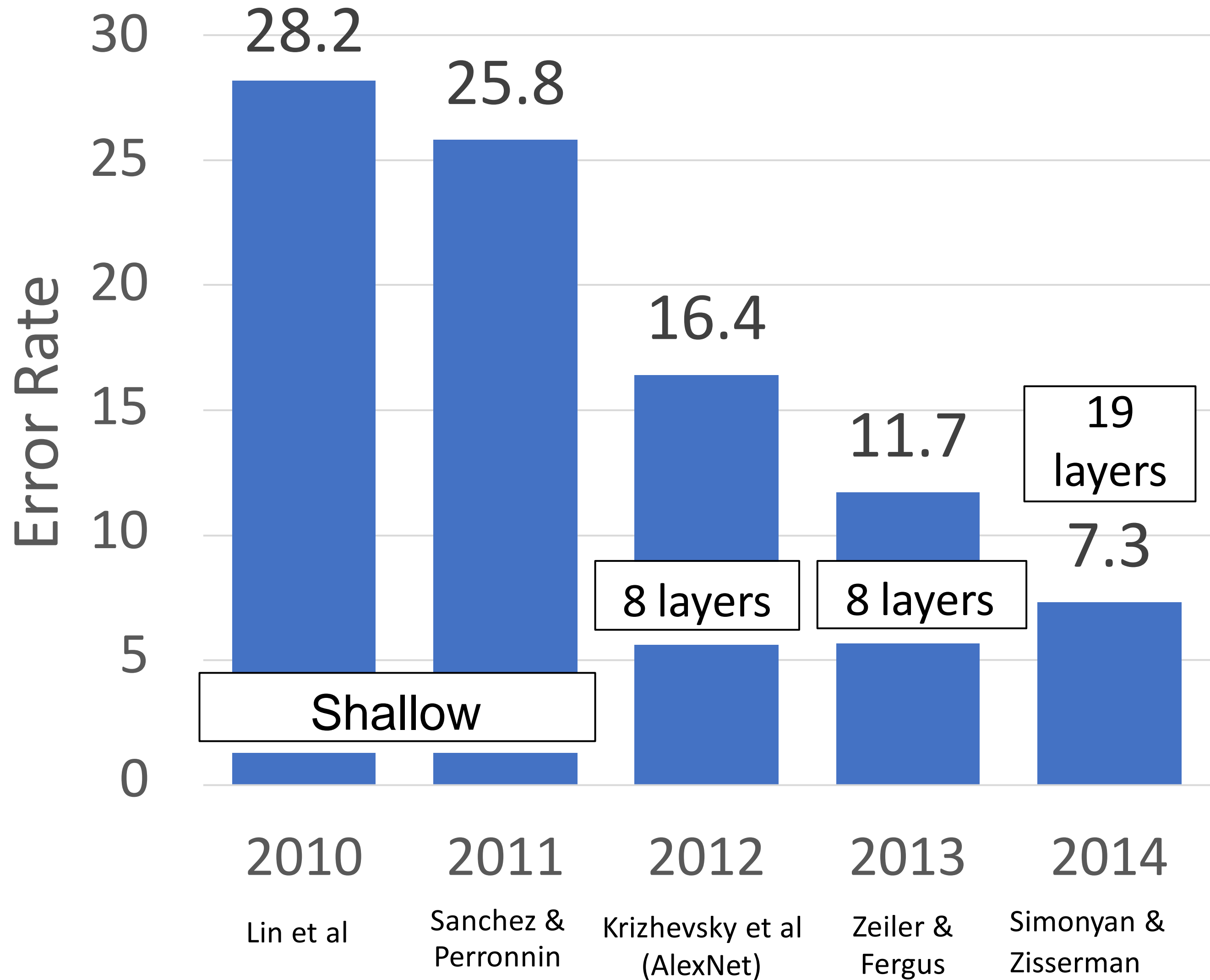
ImageNet top 5 error: 16.4% -> 11.7%



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

Zeiler and Fergus, “Visualizing and Understanding Convolutional Networks”, ECCV 2014

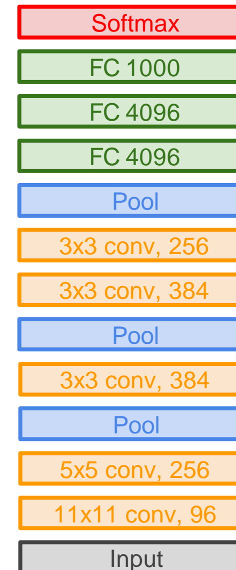
ImageNet Classification Challenge



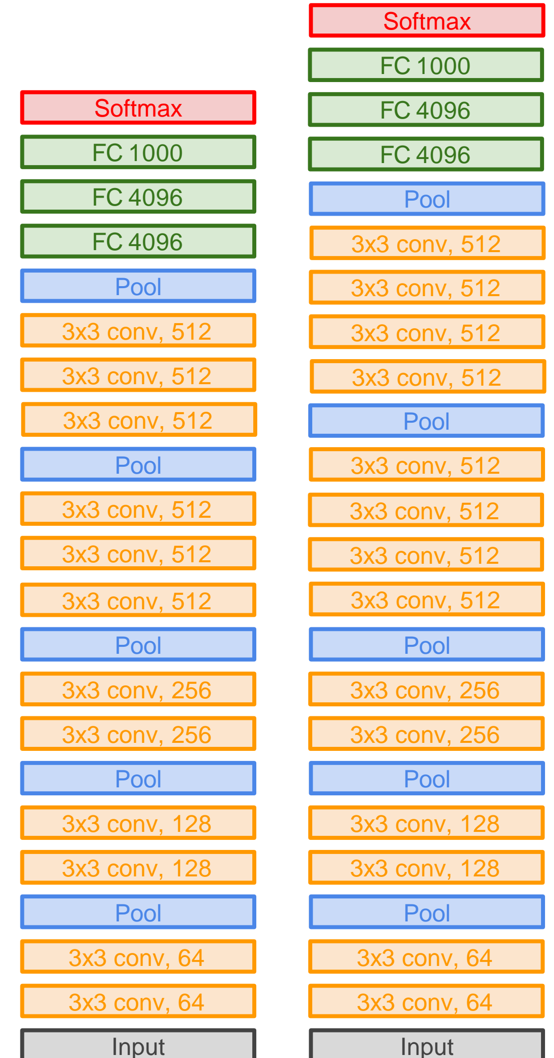
VGG: Deeper Networks, Regular Design

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

<u>VGG Design rules:</u>

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

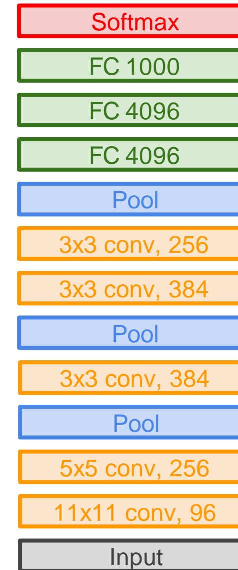
Stage 2: conv-conv-pool

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

<u>VGG Design rules:</u>

All conv are 3x3 stride 1 pad 1

All max pool are 2x2 stride 2

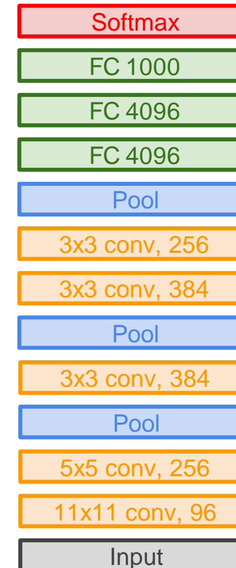
After pool, double #channels

<u>Option 1:</u>

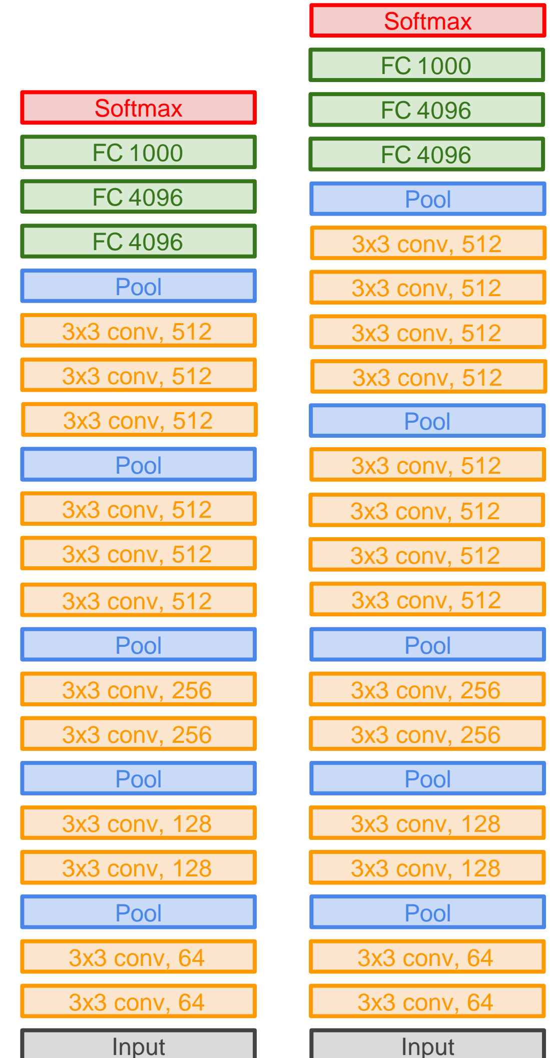
Conv(5x5, C -> C)

Params: $25C^2$

FLOPs: $25C^2HW$



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

Option 1:

Conv(5x5, C -> C)

Params: $25C^2$

FLOPs: $25C^2HW$

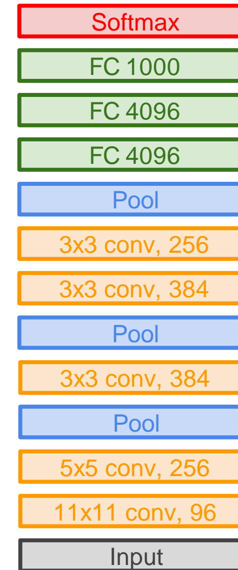
Option 2:

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

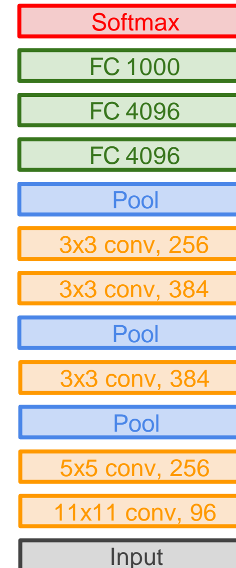
Input: C x 2H x 2W

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

Memory: 4HWC

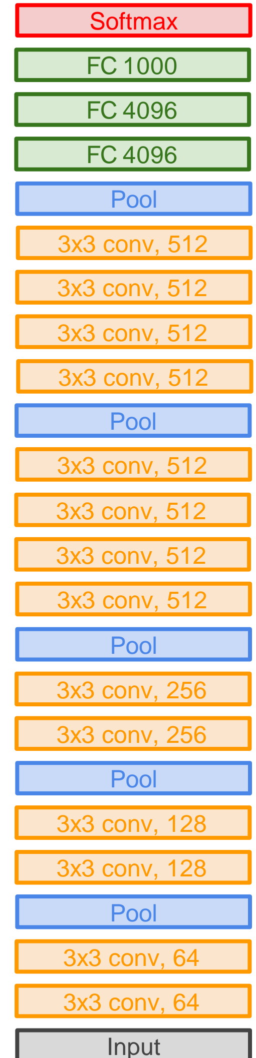
Params: $9C^2$

FLOPs: $36HWC^2$



AlexNet

VGG16



VGG19

VGG: Deeper Networks, Regular Design

<u>VGG Design rules:</u>

All conv are 3x3 stride 1 pad 1

All max pool are 2x2 stride 2

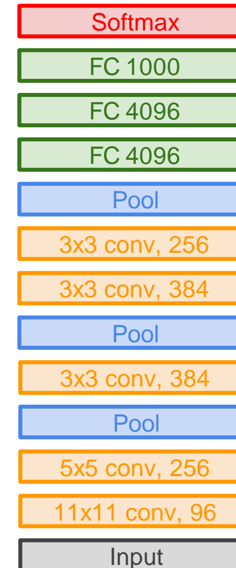
After pool, double #channels

Input: C x 2H x 2W
Layer: Conv(3x3, C->C)

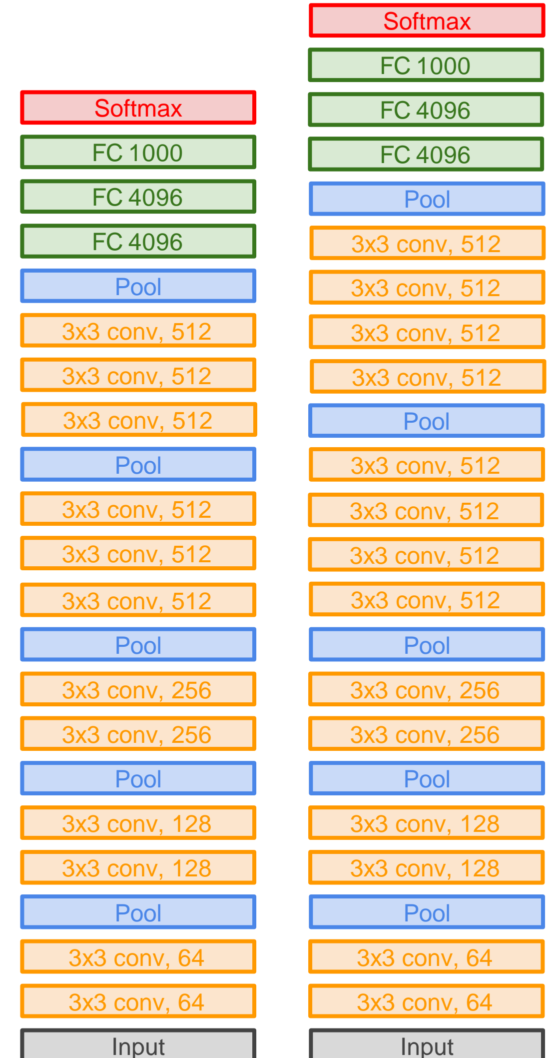
Memory: 4HWC
Params: $9C^2$
FLOPs: $36HWC^2$

Input: 2C x H x W
Conv(3x3, 2C -> 2C)

Memory: 2HWC
Params: $36C^2$
FLOPs: $36HWC^2$



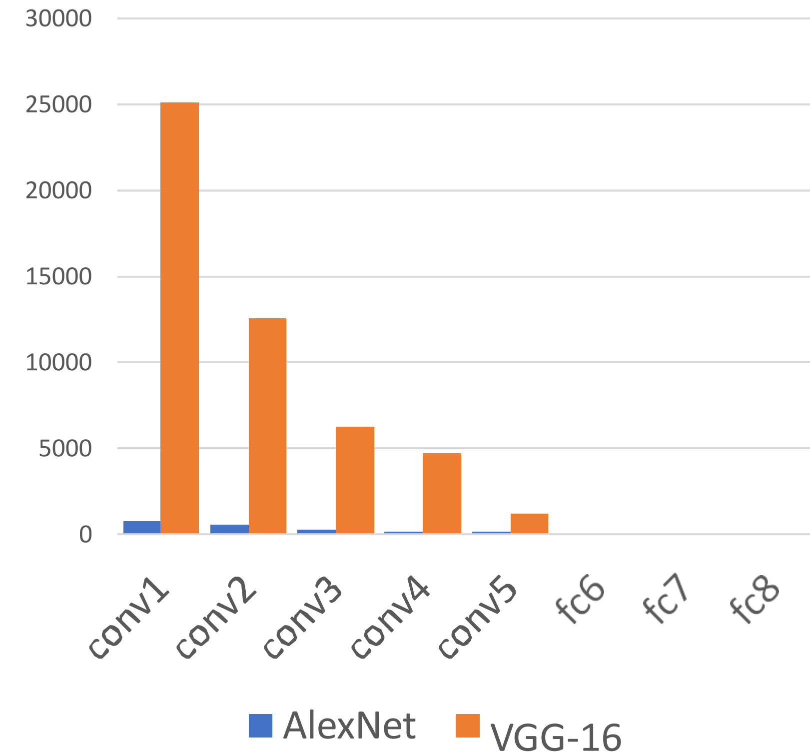
AlexNet

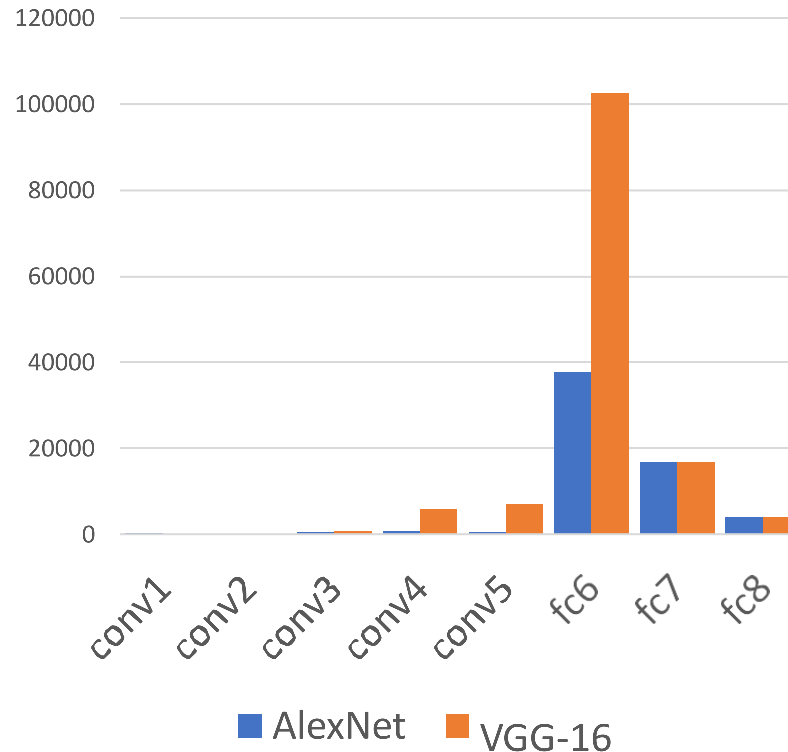


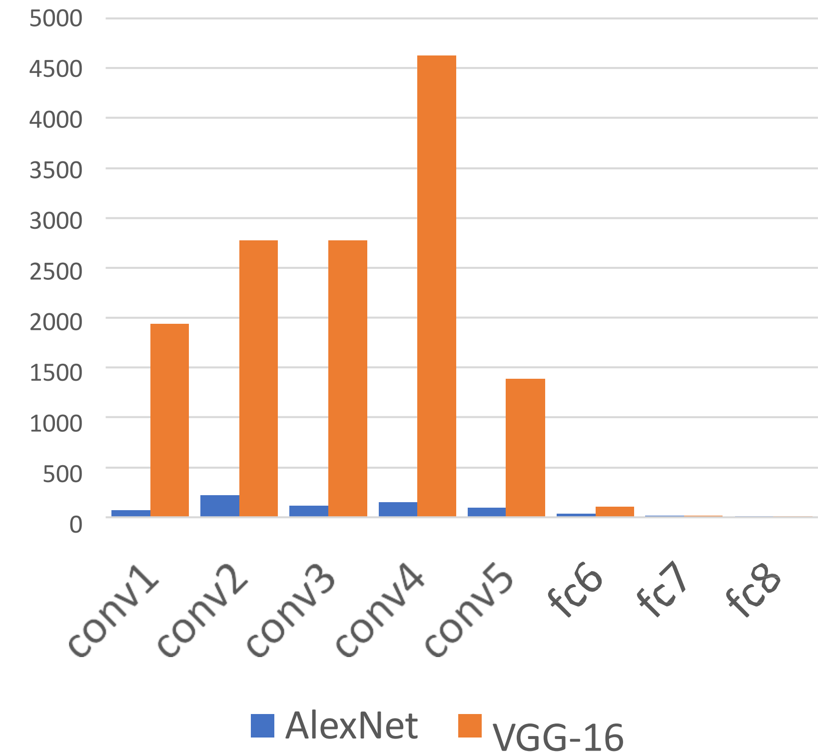
VGG16

VGG19

AlexNet vs VGG-16







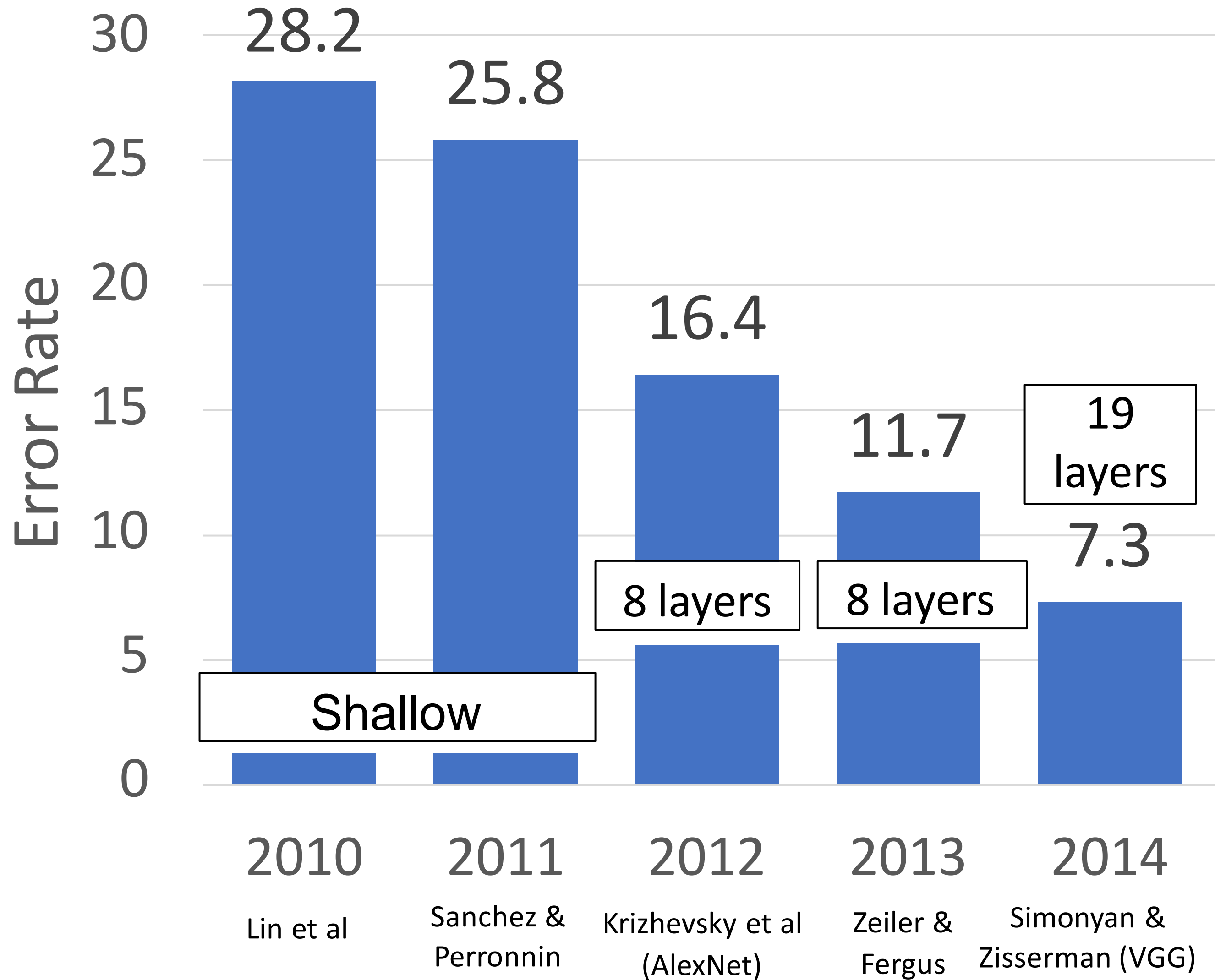
AlexNet total: 1.9 MB
VGG-16 total: 48.6 MB (25x)

AlexNet total: 61M
VGG-16 total: 138M (2.3x)

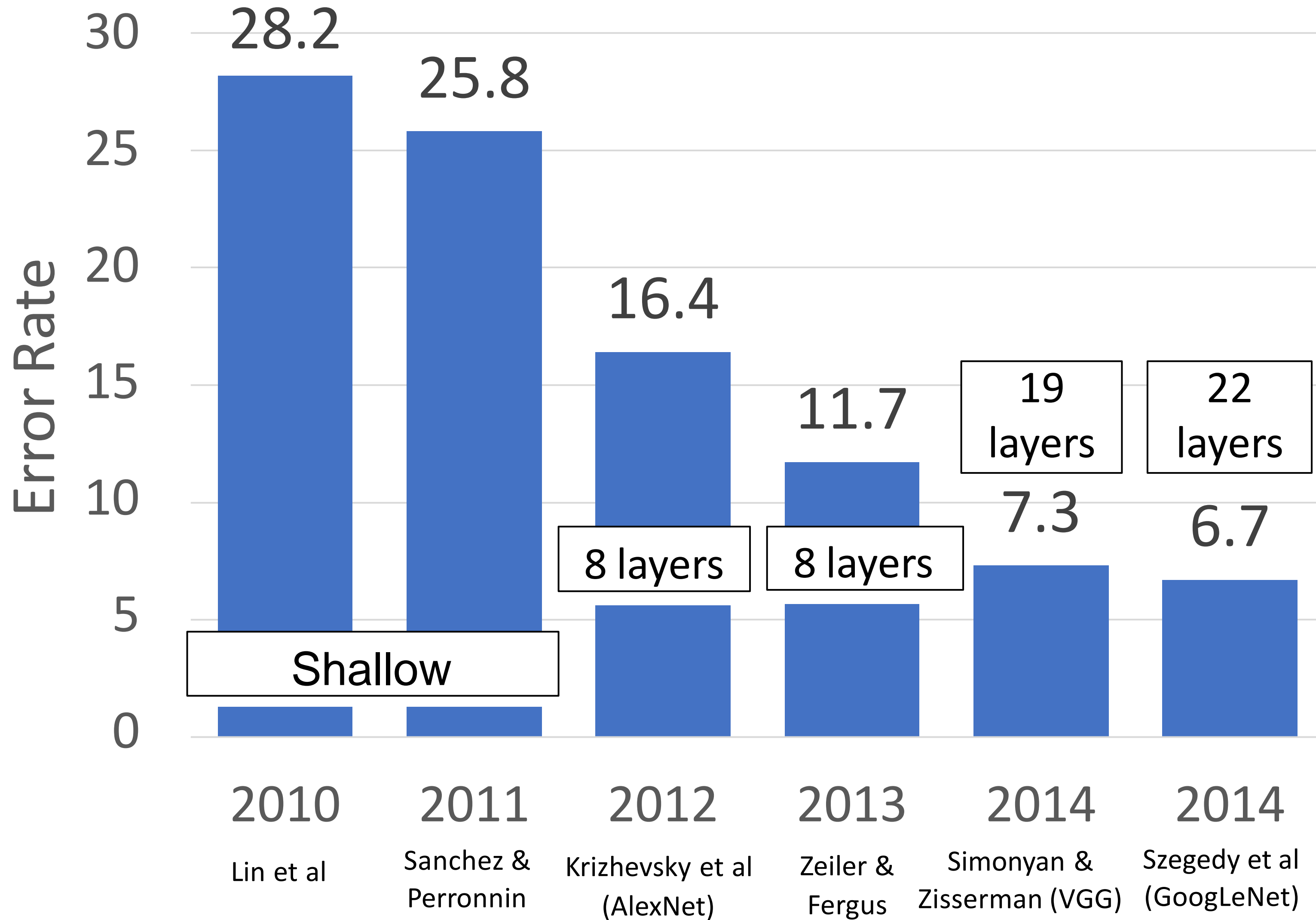
AlexNet total: 0.7 GFLOP
VGG-16 total: 13.6 GFLOP (19.4x)

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

ImageNet Classification Challenge



ImageNet Classification Challenge

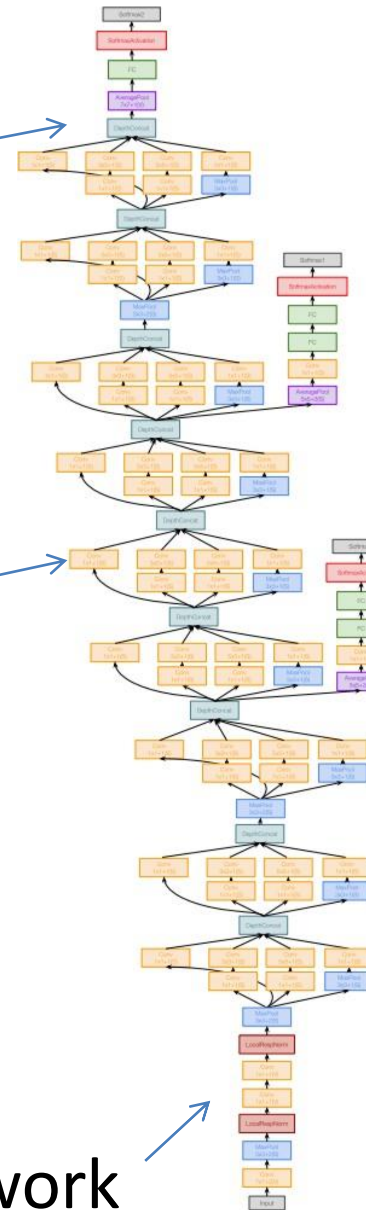


GoogLeNet

Global Average Pooling

Inception Modules

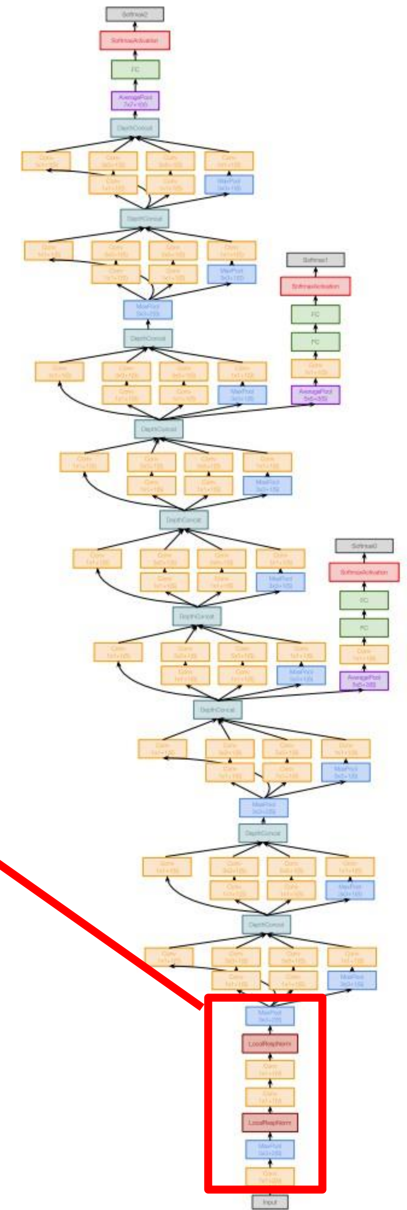
Stem Network



GoogLeNet: Aggressive Stem

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

	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H/W	memory (KB)	params (K)	flop (M)
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

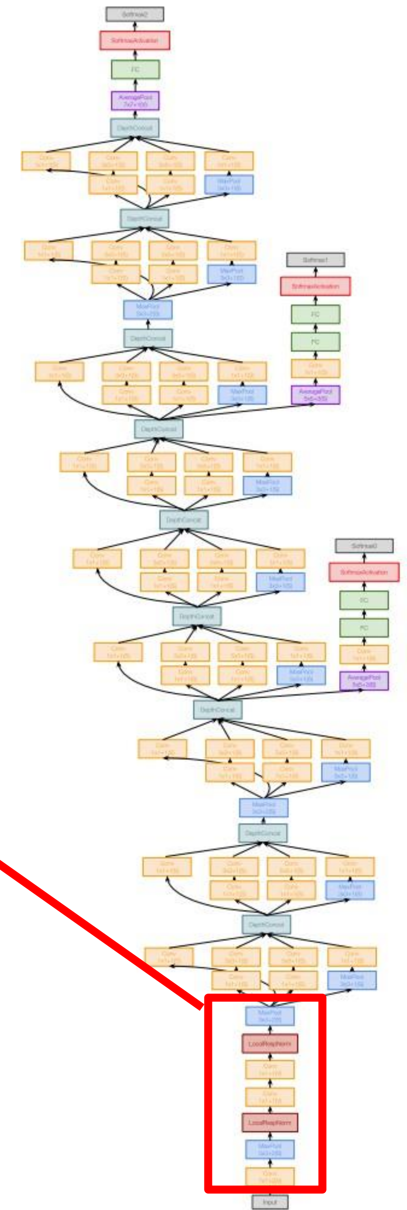


Total from 224 to 28 spatial resolution:
Memory: 7.5 MB
Params: 124K
MFLOP: 418

GoogLeNet: Aggressive Stem

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

	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H/W	memory (KB)	params (K)	flop (M)
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



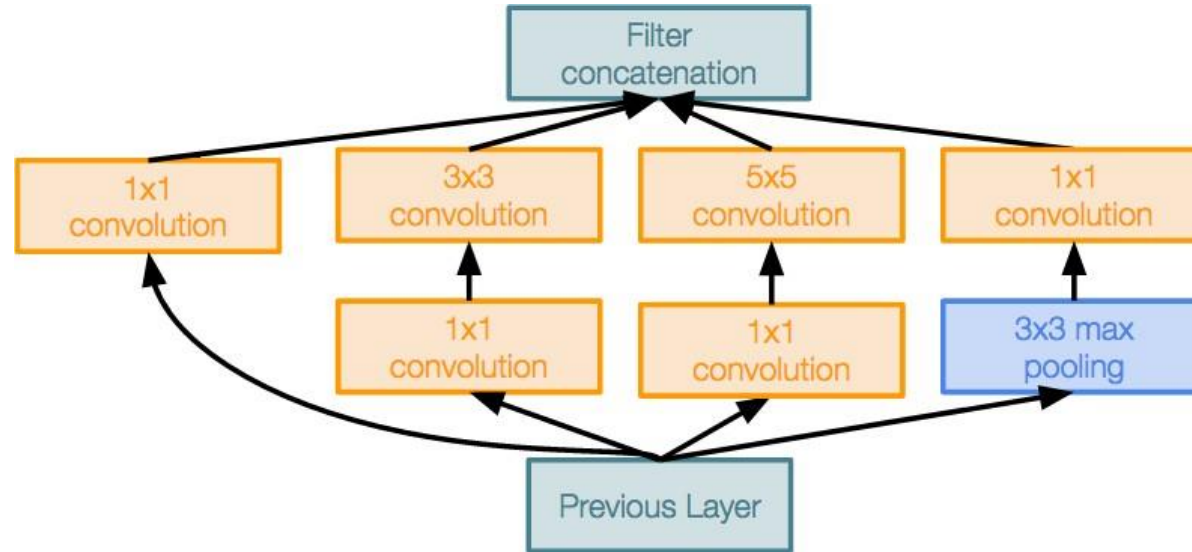
Total from 224 to 28 spatial resolution:
Memory: 7.5 MB
Params: 124K
MFLOP: 418

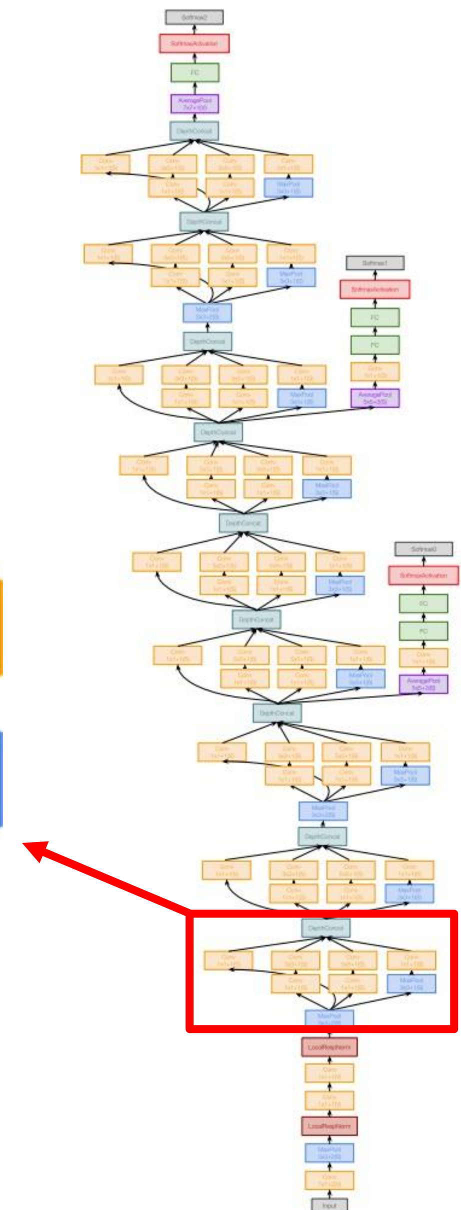
Compare VGG-16:
Memory: 42.9 MB (5.7x)
Params: 1.1M (8.9x)
MFLOP: 7485 (17.8x)

GoogLeNet: Inception Module

Inception module
Local unit with
parallel branches

Local structure repeated
many times throughout the
network





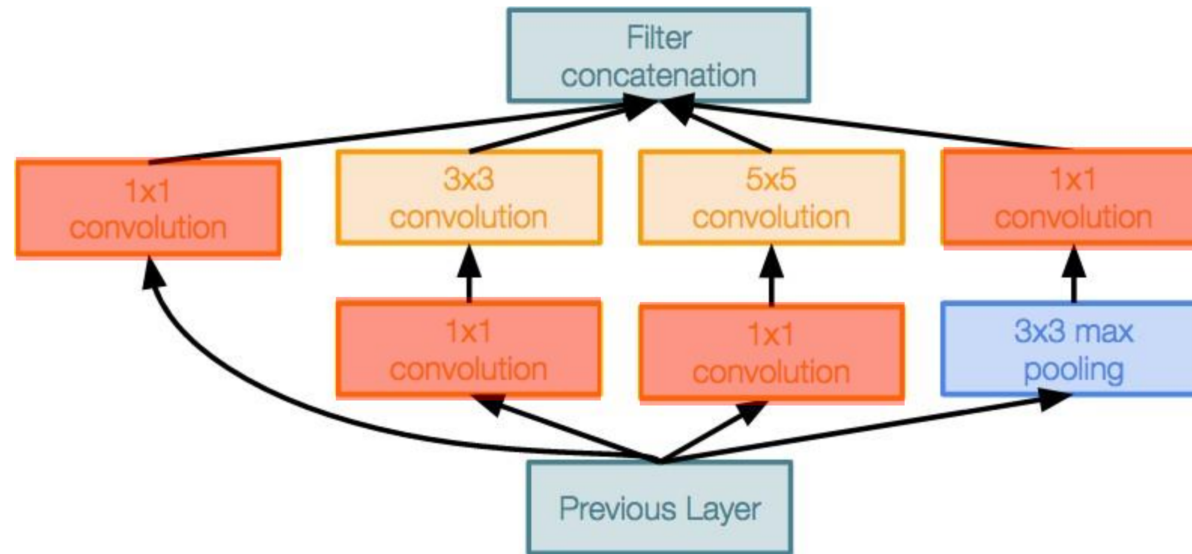
convolution filters of different sizes will handle
objects at multiple scales better

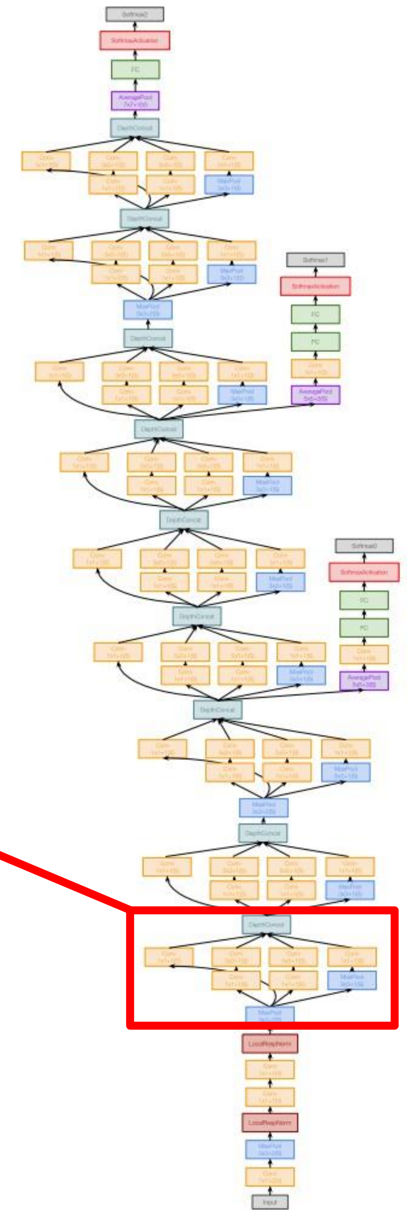
GoogLeNet: Inception Module

Inception module
Local unit with parallel branches

Local structure repeated many times throughout the network

Uses 1x1 “Bottleneck” layers to reduce channel dimension



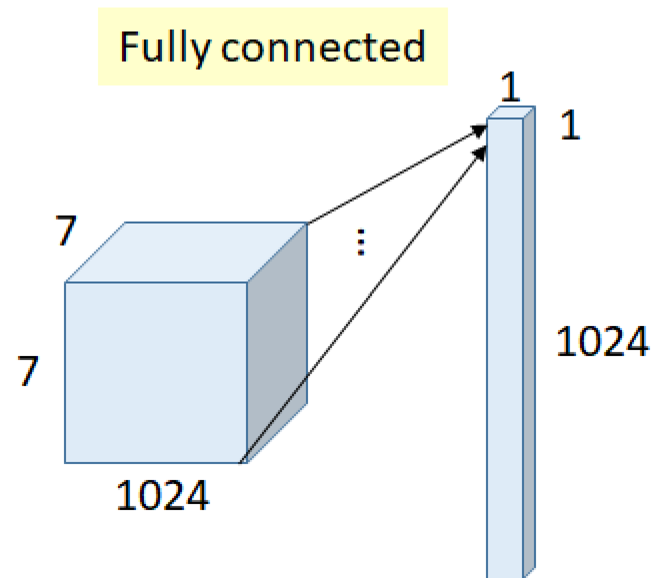


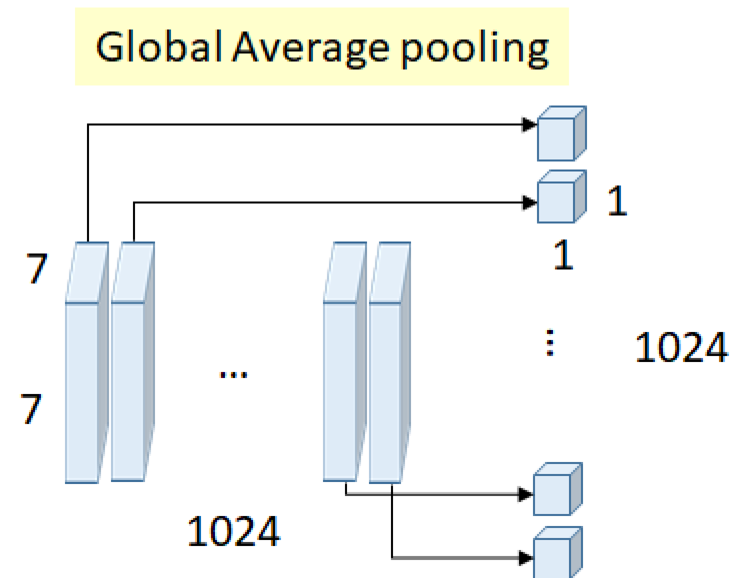
convolution filters of different sizes will handle objects at multiple scales better

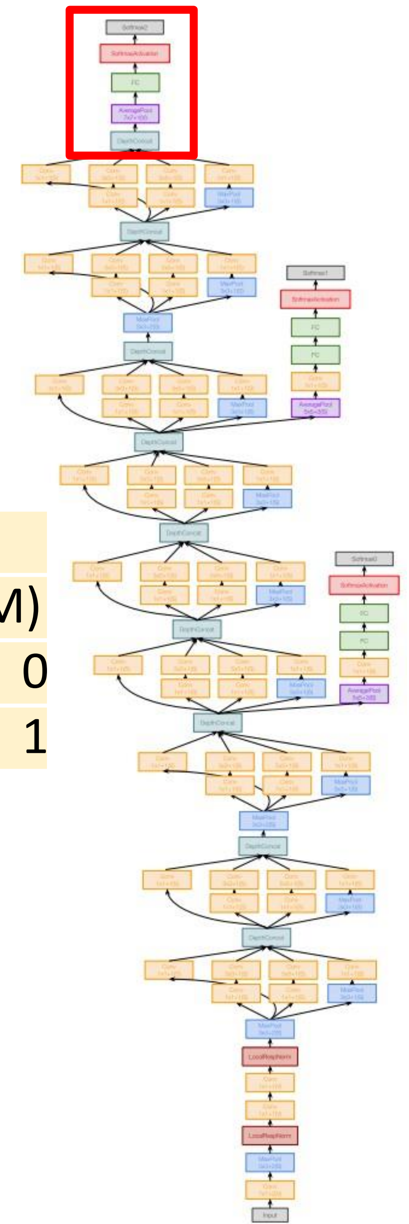
GoogLeNet: Global Average Pooling

Uses **global average pooling** to collapse spatial dimensions, and one linear layer to produce class scores (VGG-16: Most parameters were in 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







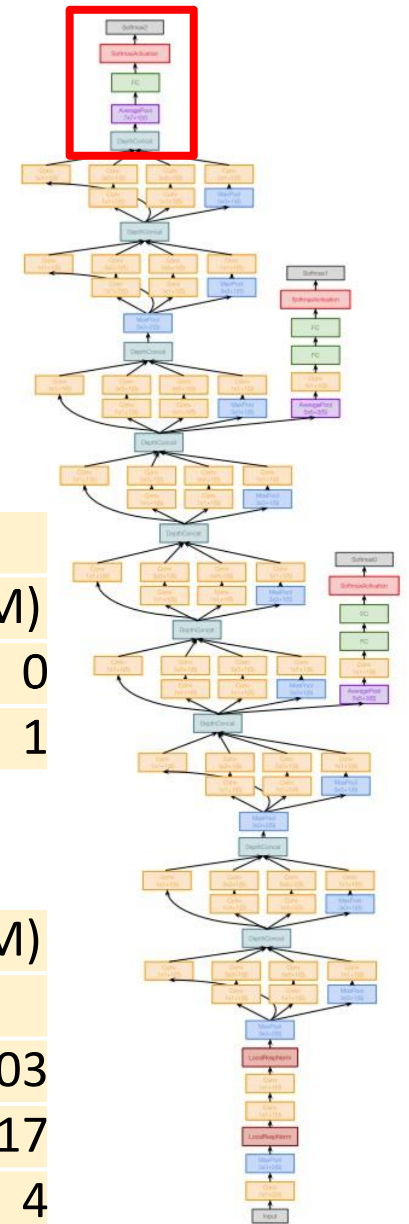
GoogLeNet: Global Average Pooling

Uses **global average pooling** to collapse spatial dimensions, and one linear layer to produce class scores (VGG-16: Most parameters were in 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

Compare with VGG16:

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

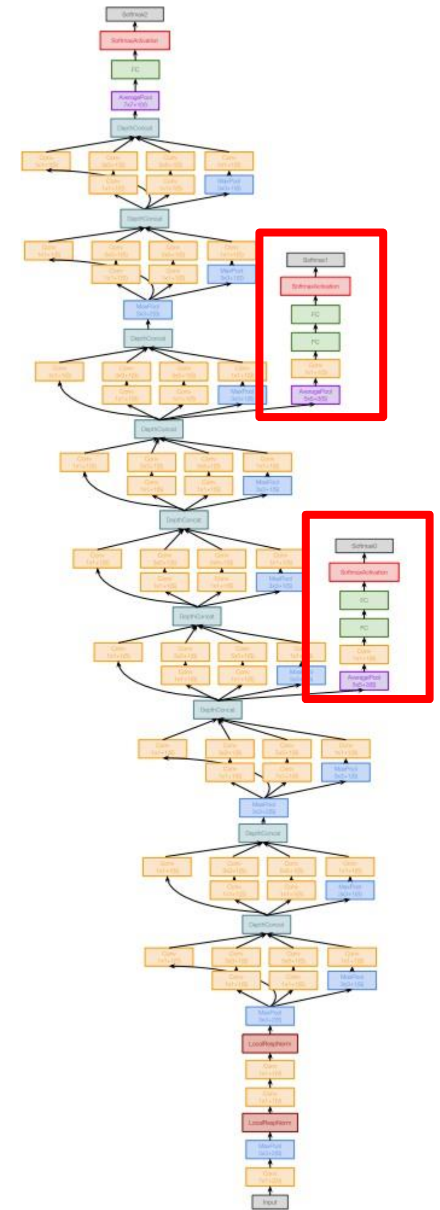


GoogLeNet: Auxiliary Classifiers

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

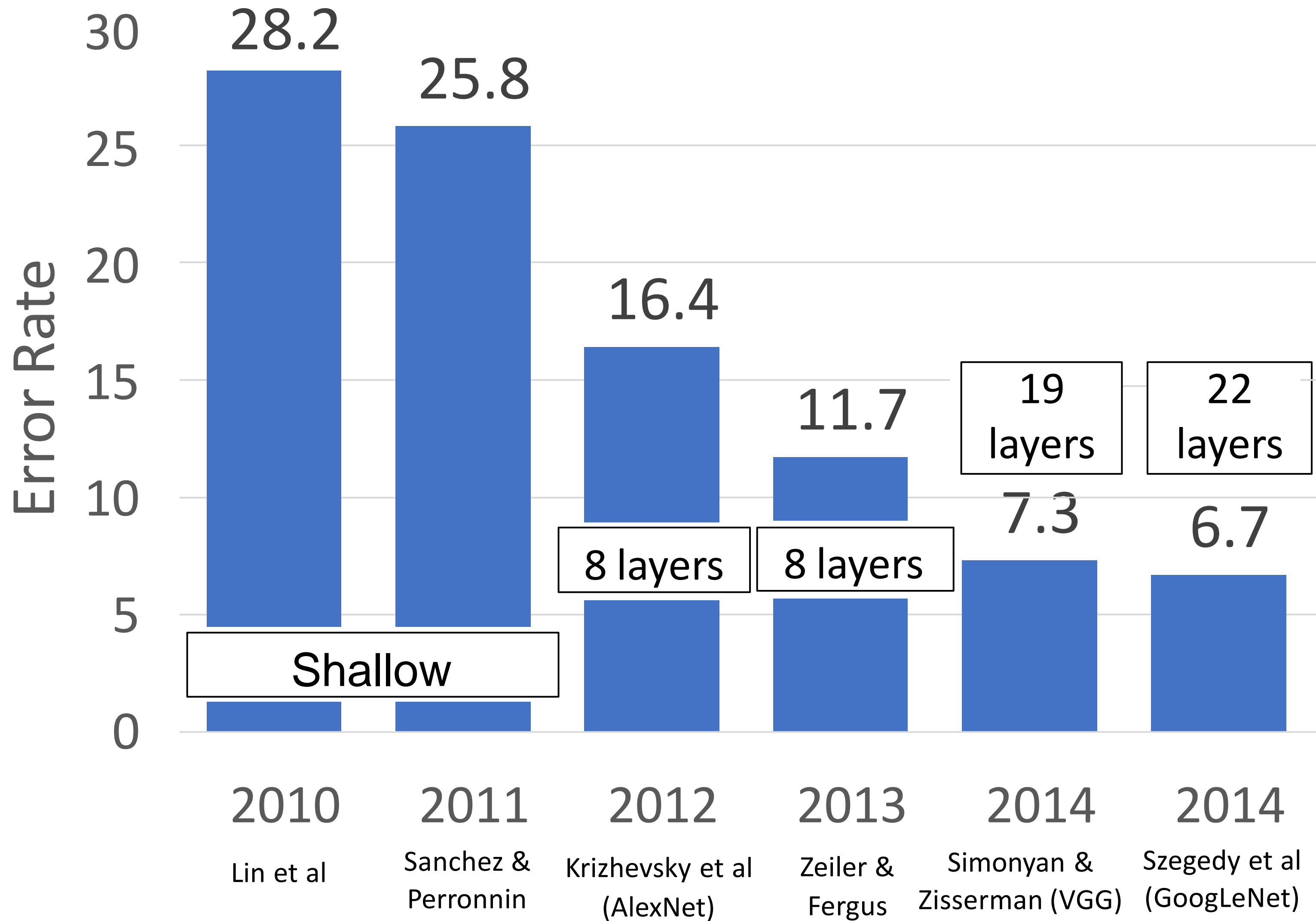
Attach "auxiliary classifiers" at several intermediate points in the network that also try to classify the image and receive loss (weighted with coefficient 0.3 only during training)

GoogLeNet was before batch normalization. With BatchNorm no longer need to use this method

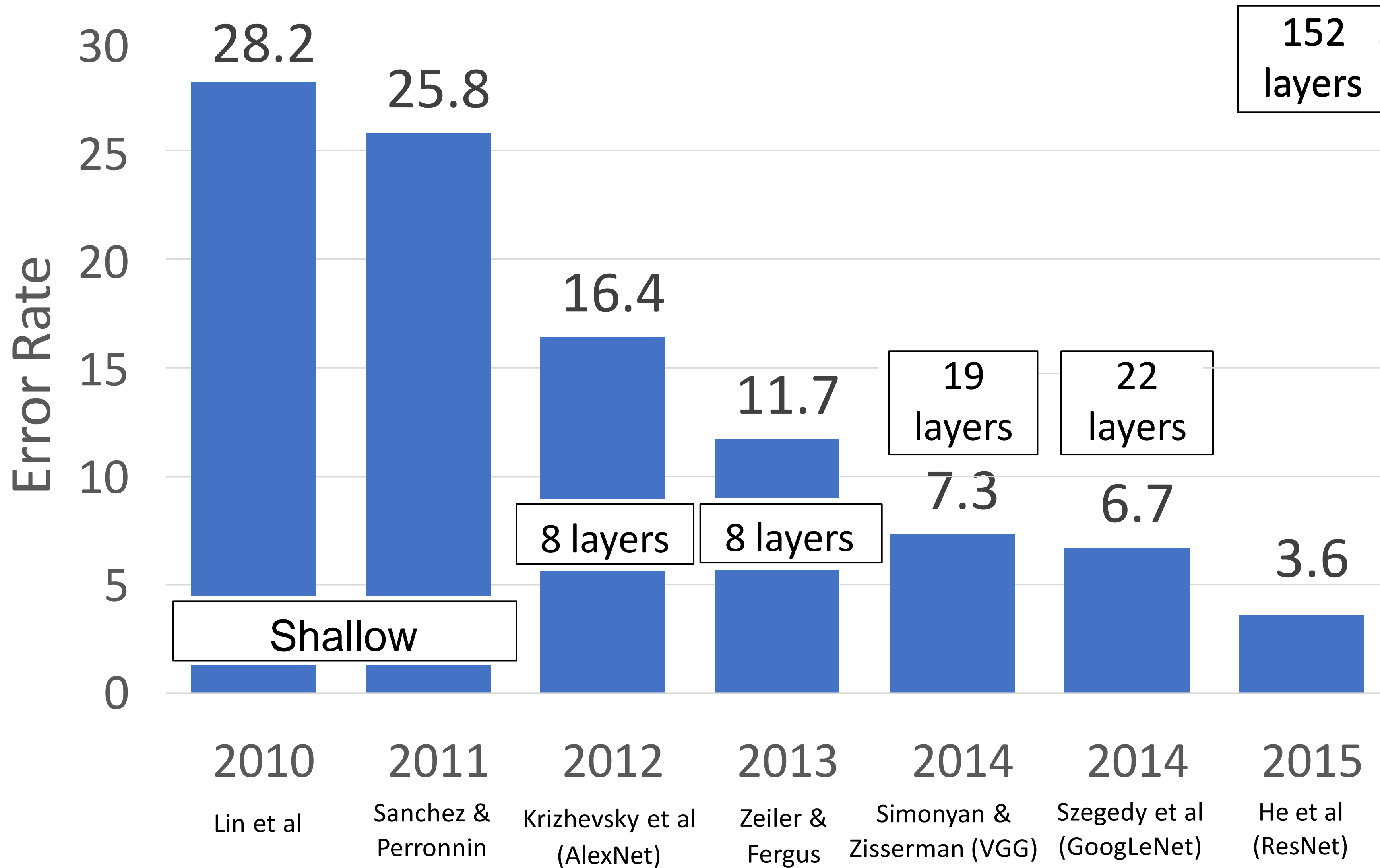


Szegedy et al, "Going deeper with convolutions", CVPR 2015

ImageNet Classification Challenge



ImageNet Classification Challenge



Residual Networks

Once we have Batch Normalization, we can train networks with 10+ layers.
Going deeper leads to **degradation problem**.

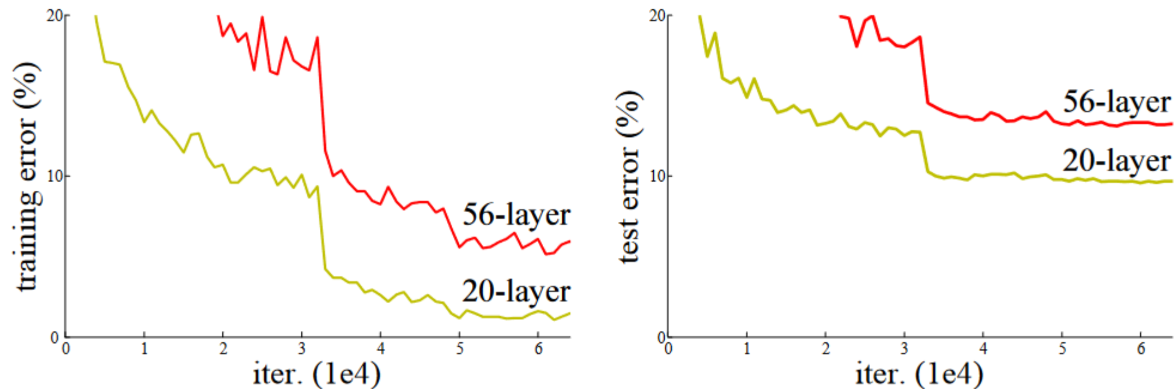
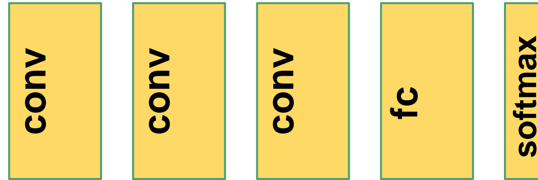


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

Degradation Problem

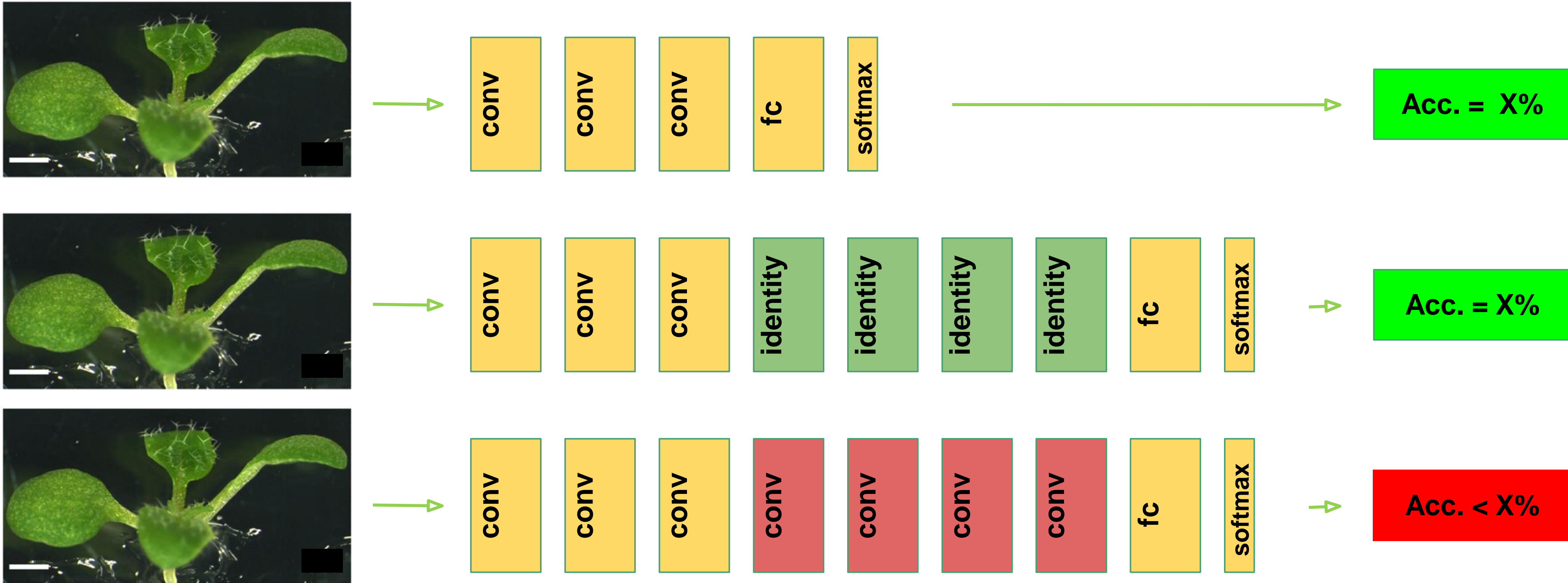
Degradation Problem



Acc. = X%

Acc. = X%

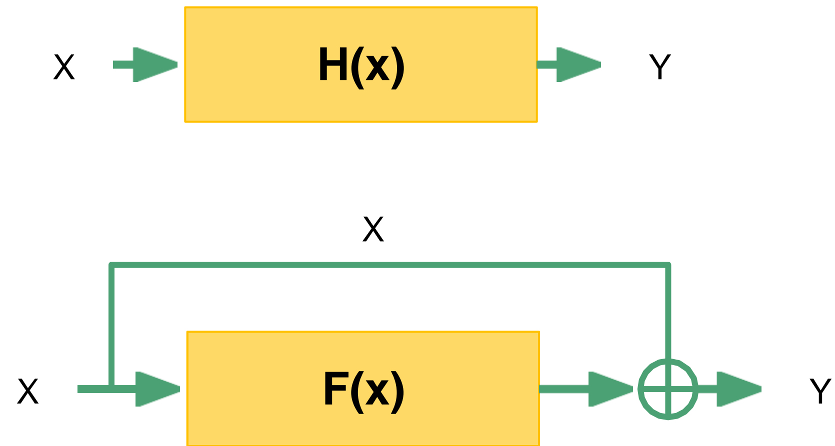
Degradation Problem



Residual Networks

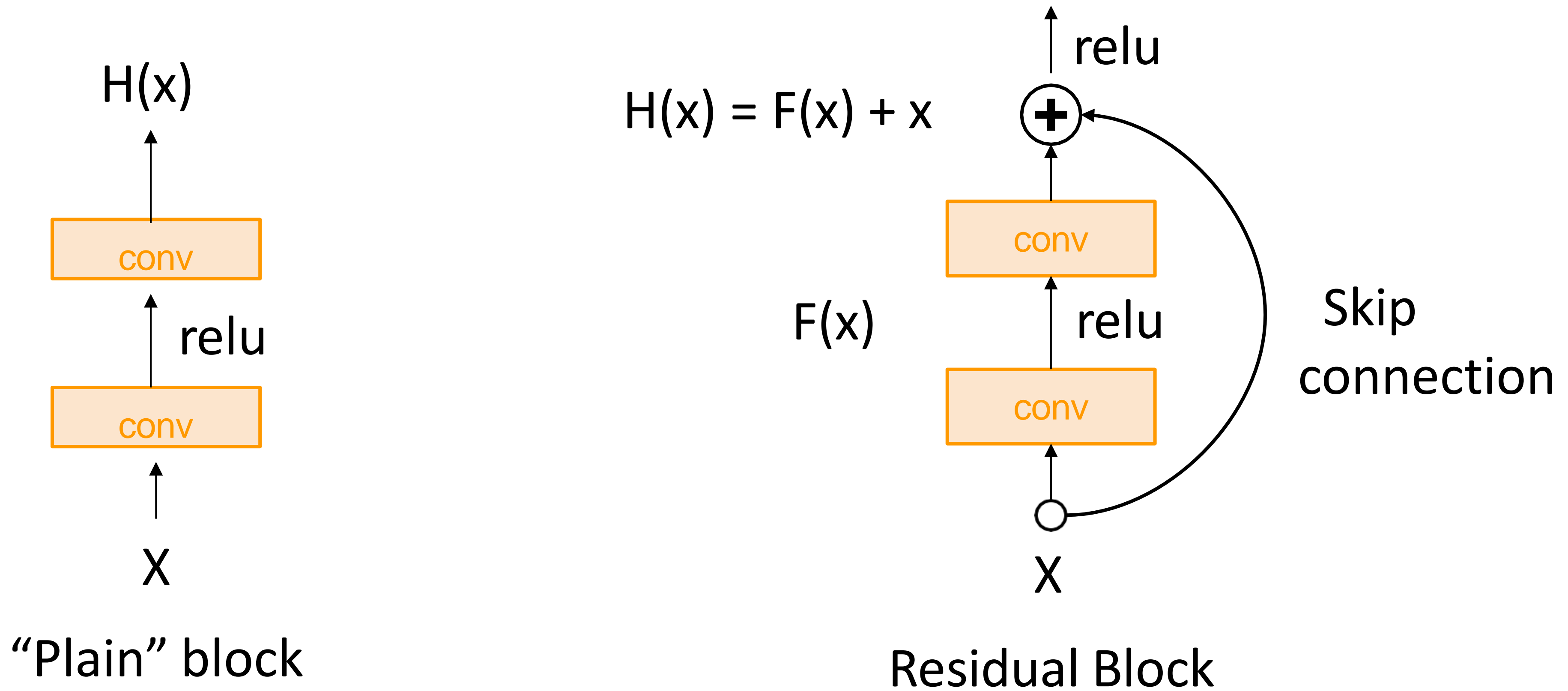
H(x) is the true mapping function we want to learn
Let's define a function F(x), and learn it instead of H(x)

$$F(x) := H(x) - x$$



Residual Networks

Solution: Change the network to easier learn identity functions



"Plain" block

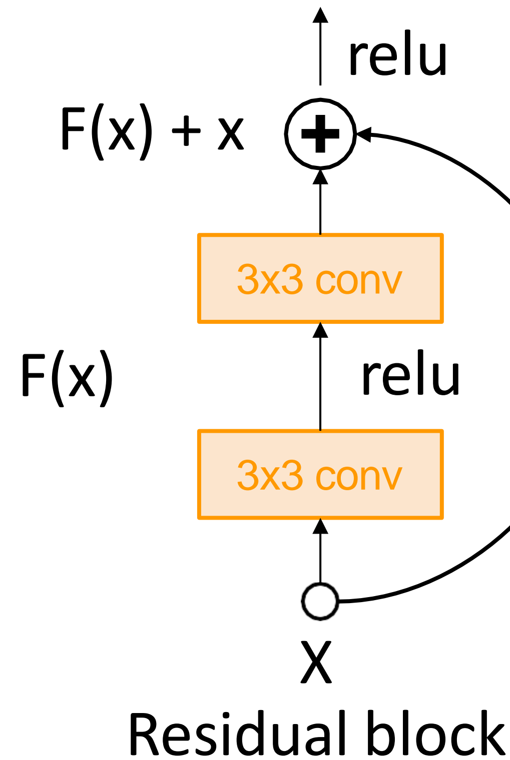
Residual Block

Residual Networks

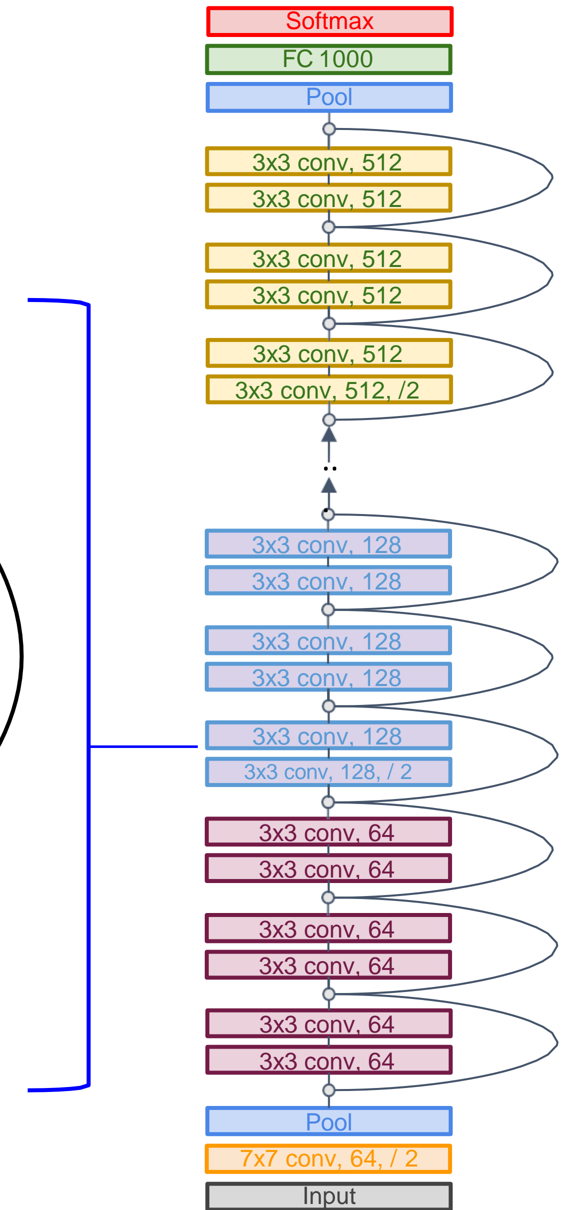
A residual network is a stack of many residual blocks

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



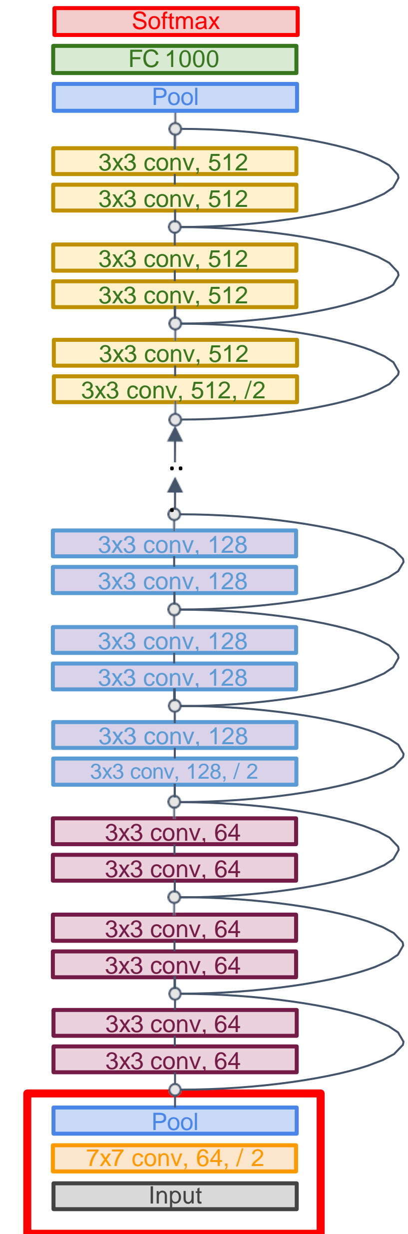
Residual block



Residual Networks

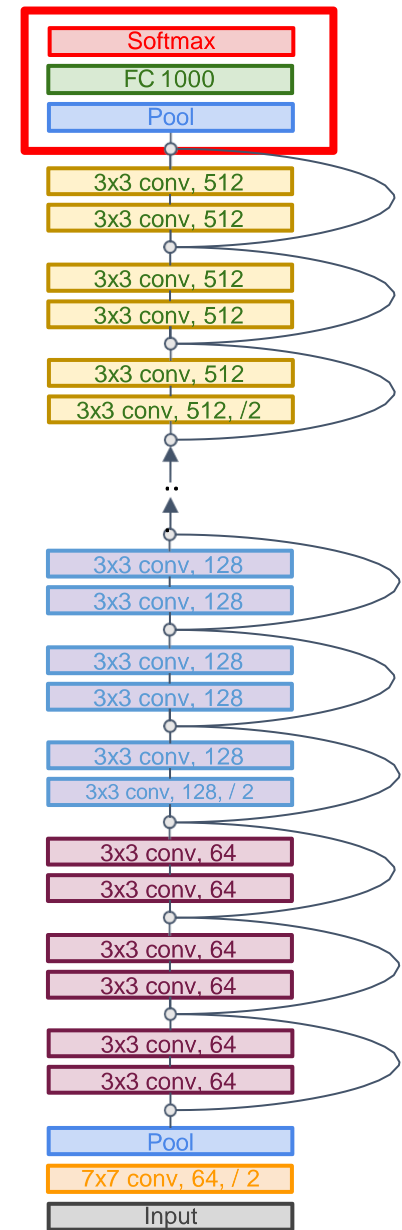
Uses the same **aggressive stem** as GoogleNet to downsample the input 4x before applying residual blocks:

	Input size		Layer				Output size				
Layer	C	H/W	filters	kernel	stride	pad	C	H/W	memory (KB)	params (k)	flop (M)
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

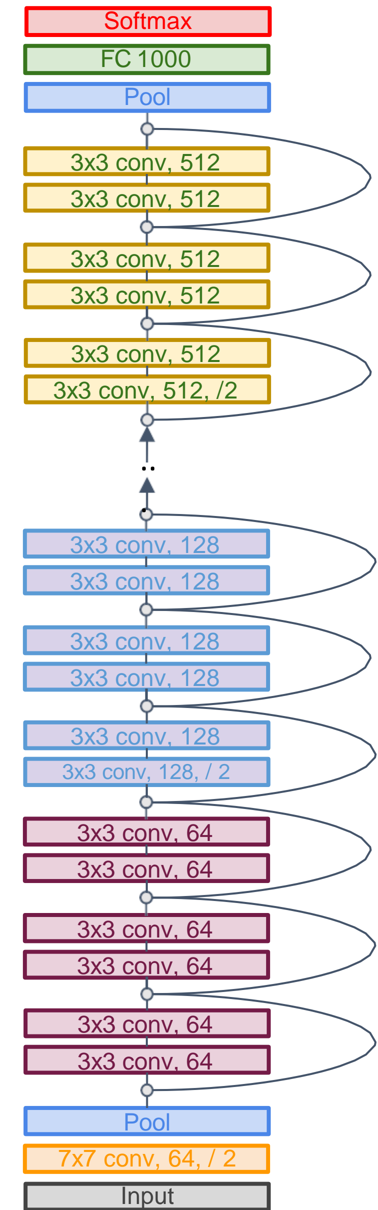
Like GoogLeNet, no big fully-connected-layers: instead use **global average pooling** and a single linear layer at the end



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



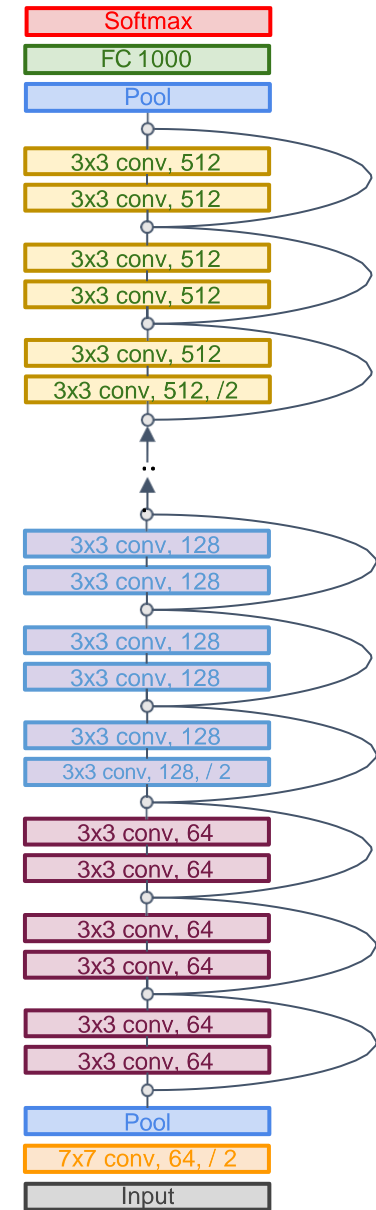
Residual Networks

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



Residual Networks

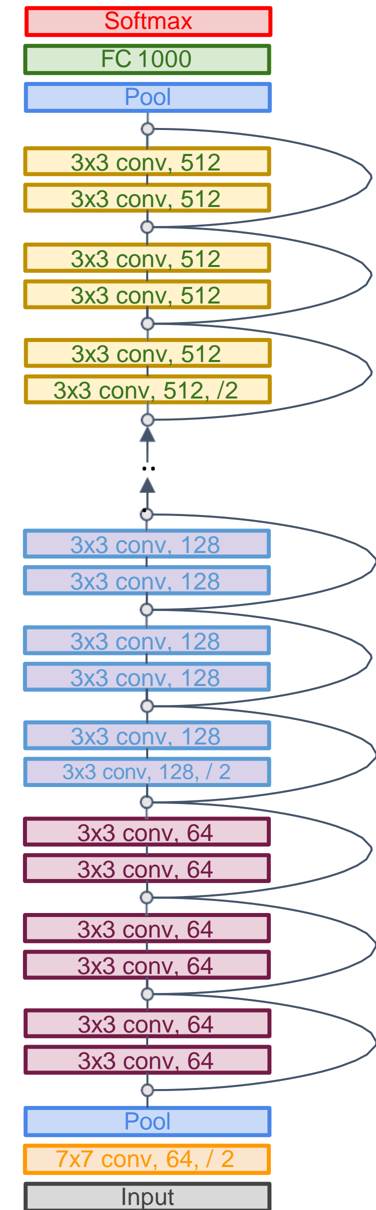
ResNet-18:
Stem: 1 conv layer
Stage 1 (C=64): 2 res. block = 4 conv
Stage 2 (C=128): 2 res. block = 4 conv
Stage 3 (C=256): 2 res. block = 4 conv
Stage 4 (C=512): 2 res. block = 4 conv
Linear

ImageNet top-5 error: 10.92
GFLOP: 1.8

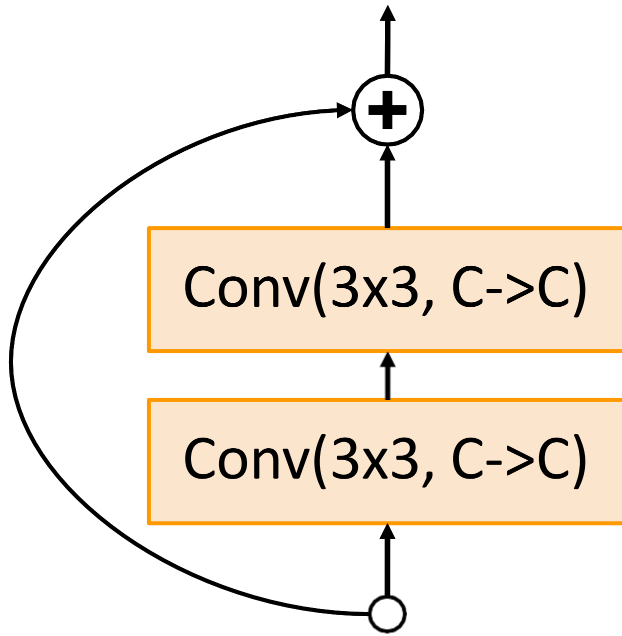
ResNet-34:
Stem: 1 conv layer
Stage 1: 3 res. block = 6 conv
Stage 2: 4 res. block = 8 conv
Stage 3: 6 res. block = 12 conv
Stage 4: 3 res. block = 6 conv
Linear

ImageNet top-5 error: 8.58
GFLOP: 3.6

VGG-16:
ImageNet top-5 error: 9.62
GFLOP: 13.6

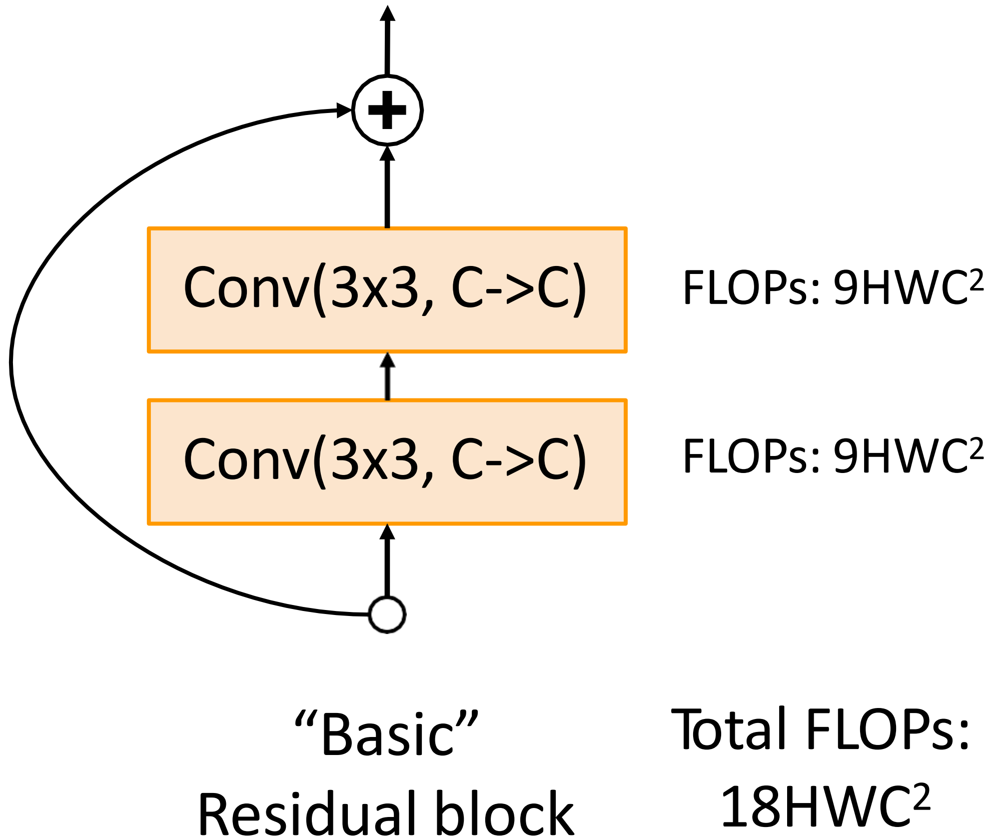


Residual Networks: Basic Block

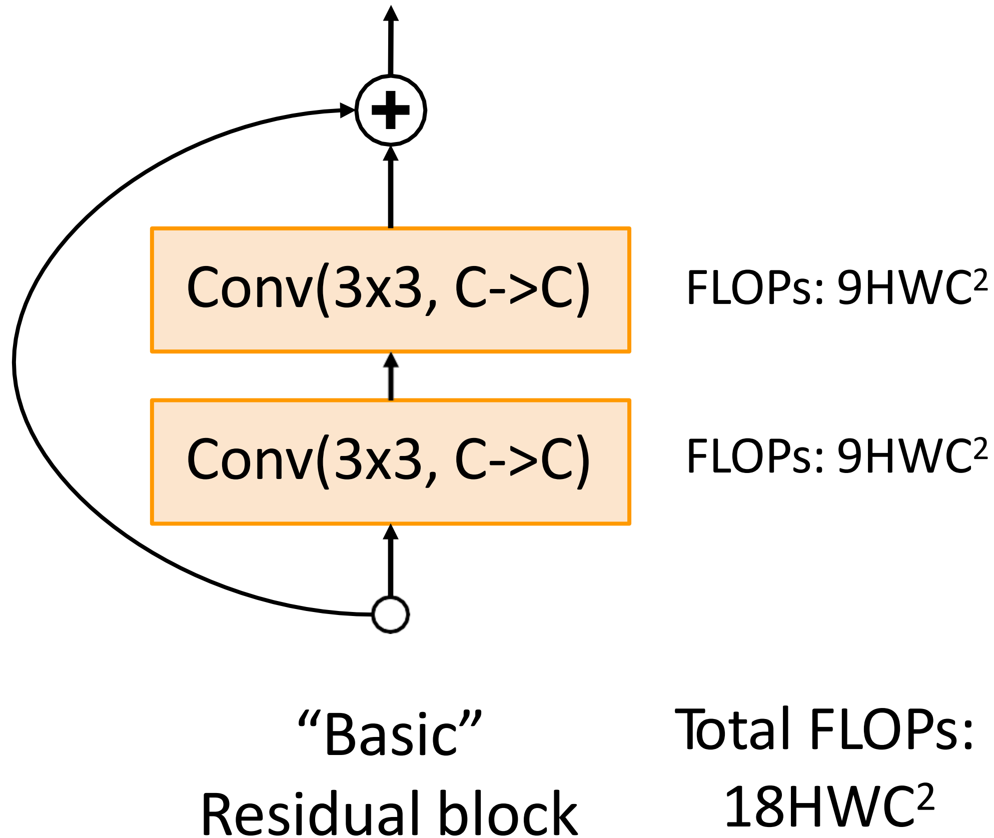


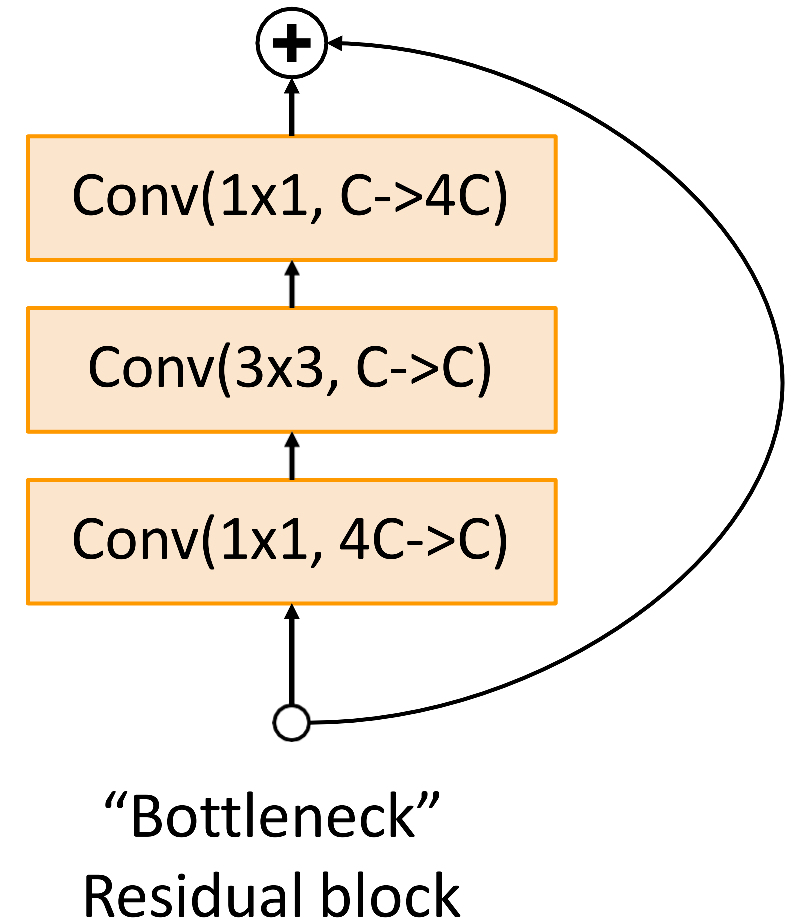
"Basic"
Residual block

Residual Networks: Basic Block



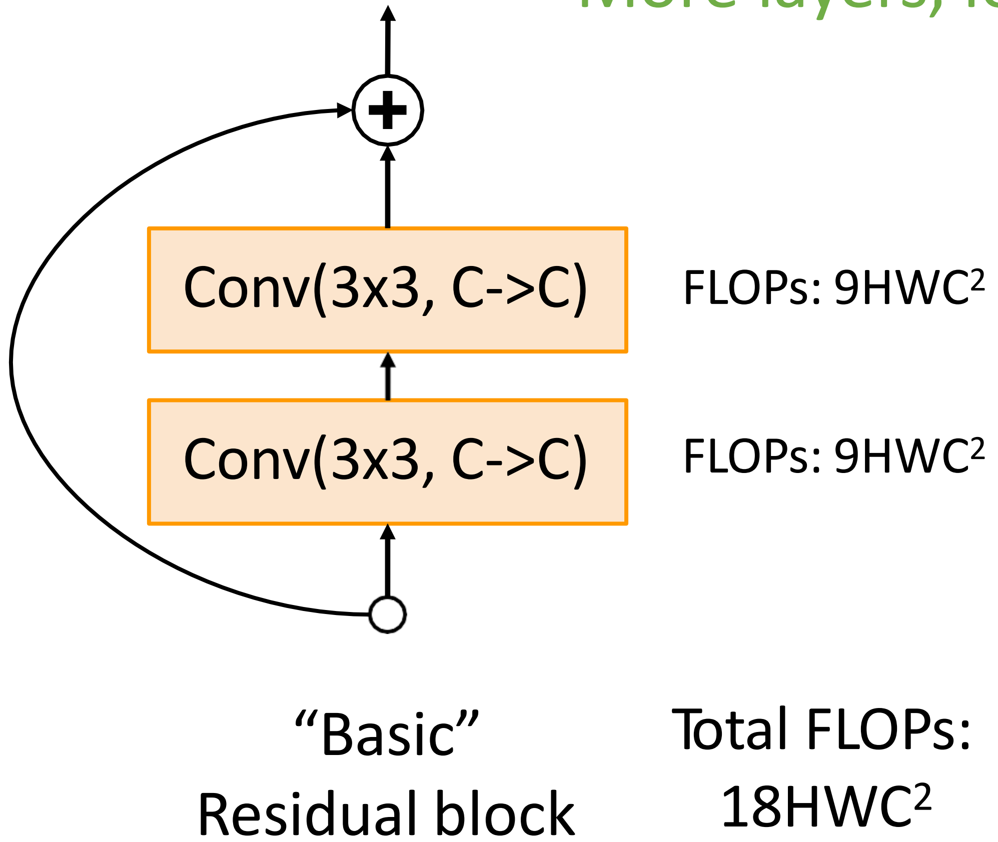
Residual Networks: Bottleneck Block

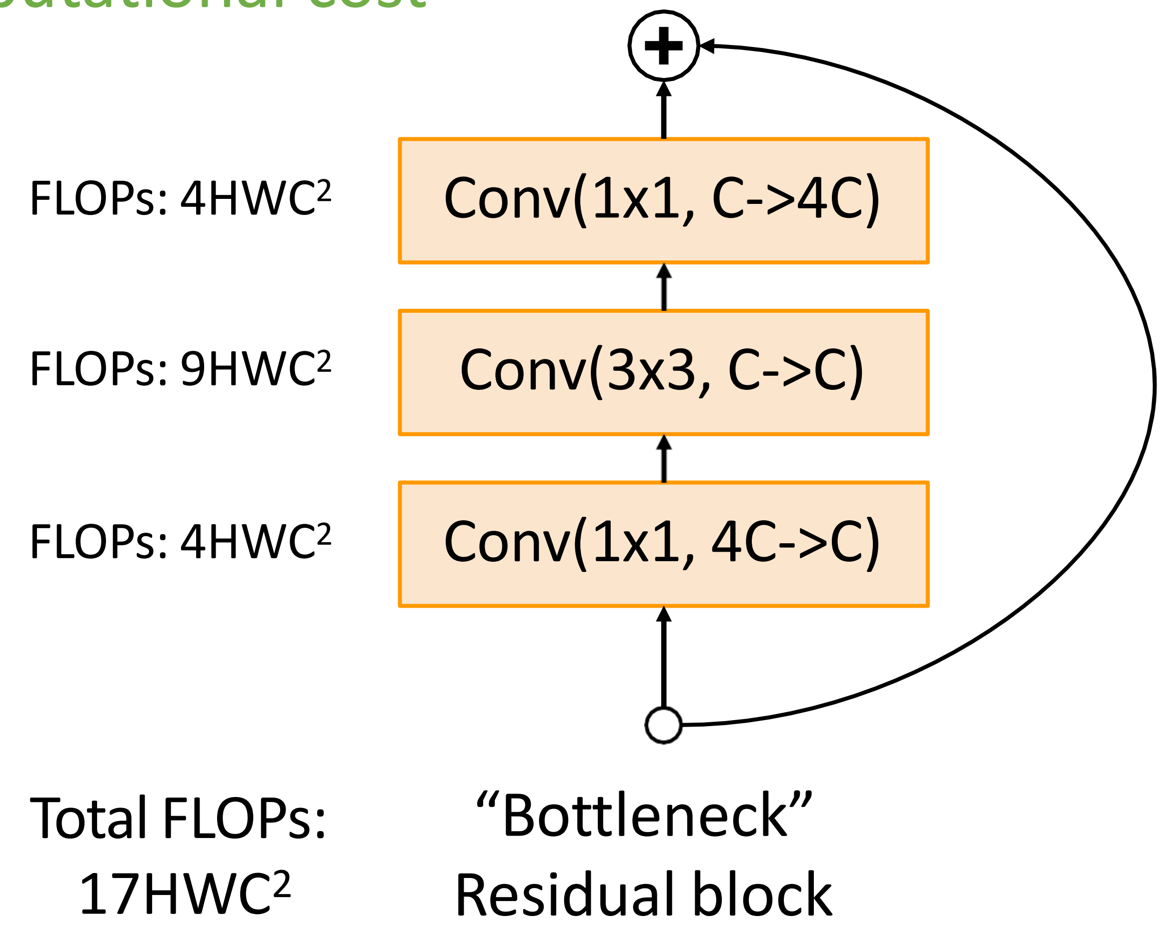




Residual Networks: Bottleneck Block

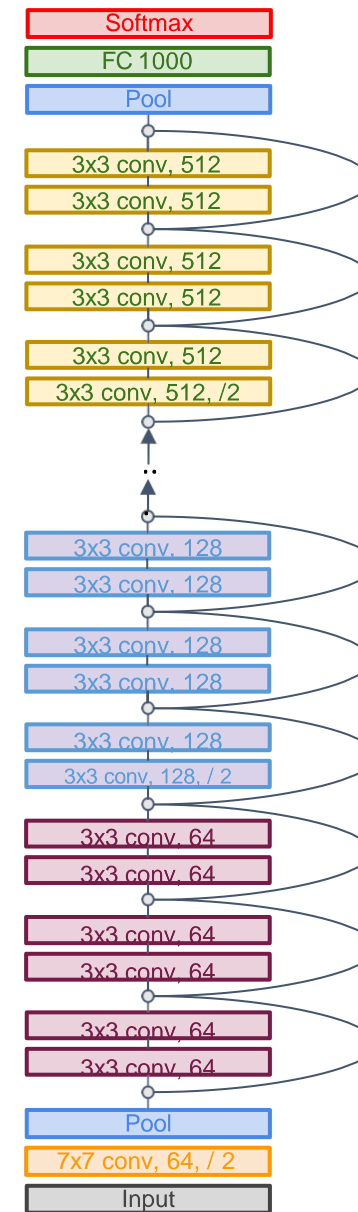
More layers, less computational cost





Residual Networks

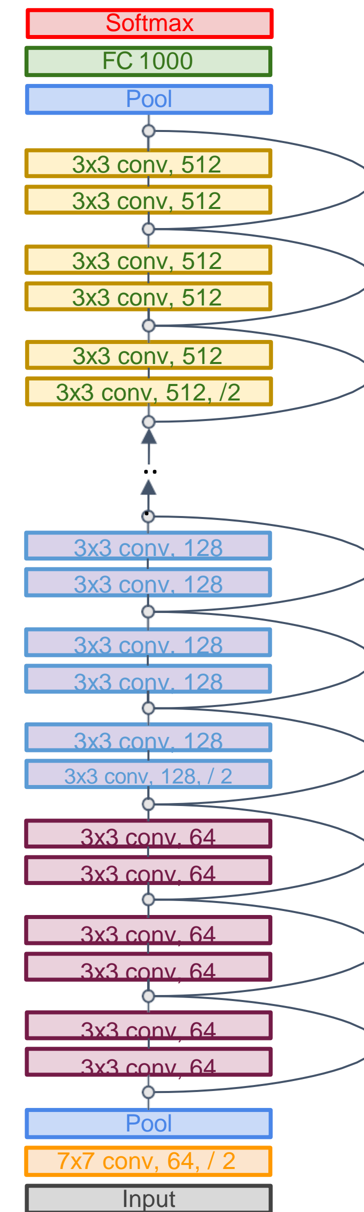
			Stage 1		Stage 2		Stage 3		Stage 4				
	Block type	Stem layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	FC layers	GFLOP	ImageNet top-5 error
ResNet-18	Basic	1	2	4	2	4	2	4	2	4	1	1.8	10.92
ResNet-34	Basic	1	3	6	4	8	6	12	3	6	1	3.6	8.58



Residual Networks

ResNet-50 is the same as ResNet-34, but replaces Basic blocks with Bottleneck Blocks.

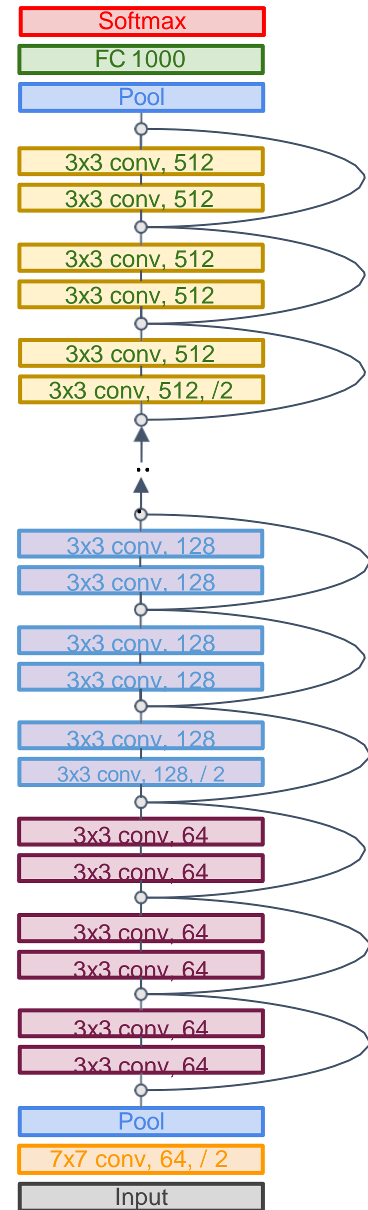
			Stage 1		Stage 2		Stage 3		Stage 4				
	Block type	Stem layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	FC layers	GFLOP	ImageNet top-5 error
ResNet-18	Basic	1	2	4	2	4	2	4	2	4	1	1.8	10.92
ResNet-34	Basic	1	3	6	4	8	6	12	3	6	1	3.6	8.58
ResNet-50	Bottle	1	3	9	4	12	6	18	3	9	1	3.8	7.13



Residual Networks

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

			Stage 1		Stage 2		Stage 3		Stage 4				
	Block type	Stem layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	FC layers	GFLOP	ImageNet 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



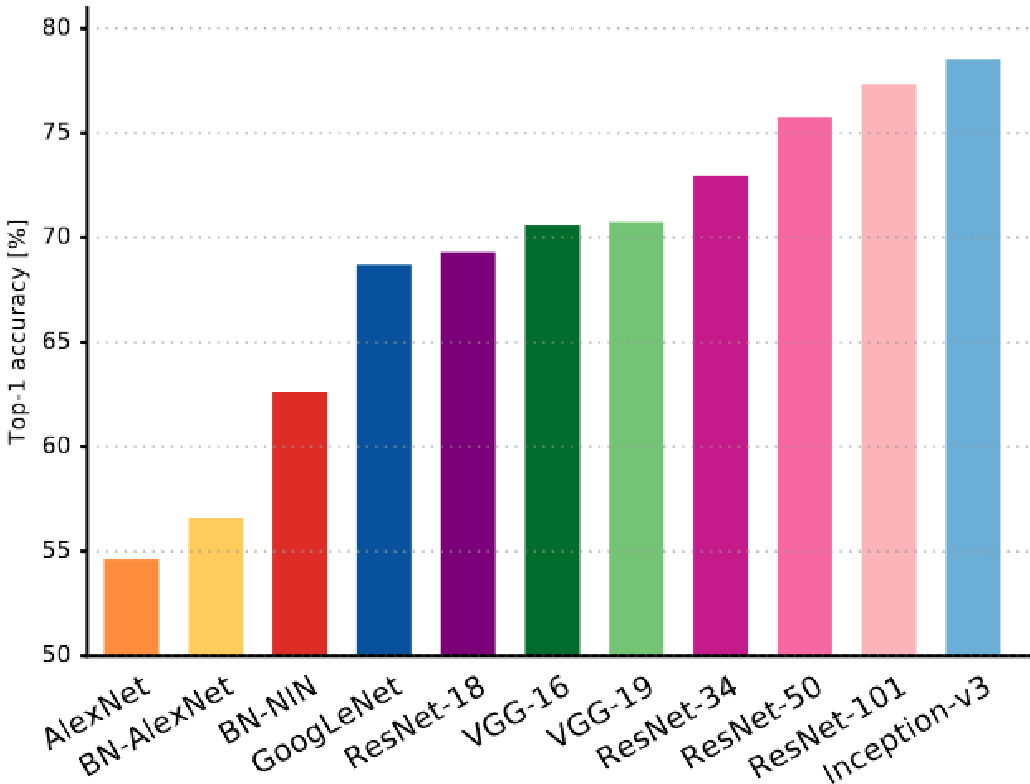
Residual Networks

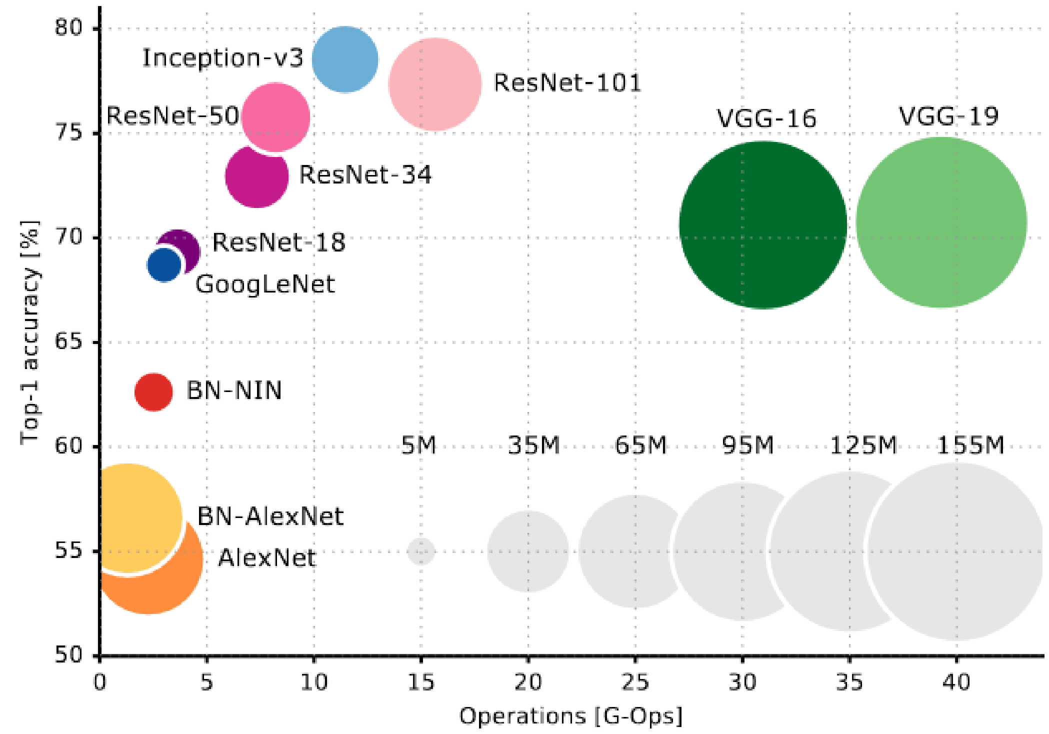
- Able to train very deep networks
- Deeper networks do better than shallow networks (as expected)
- Swept 1st place in all ILSVRC and COCO 2015 competitions
- Widely used today
- Over 100,000 citations

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

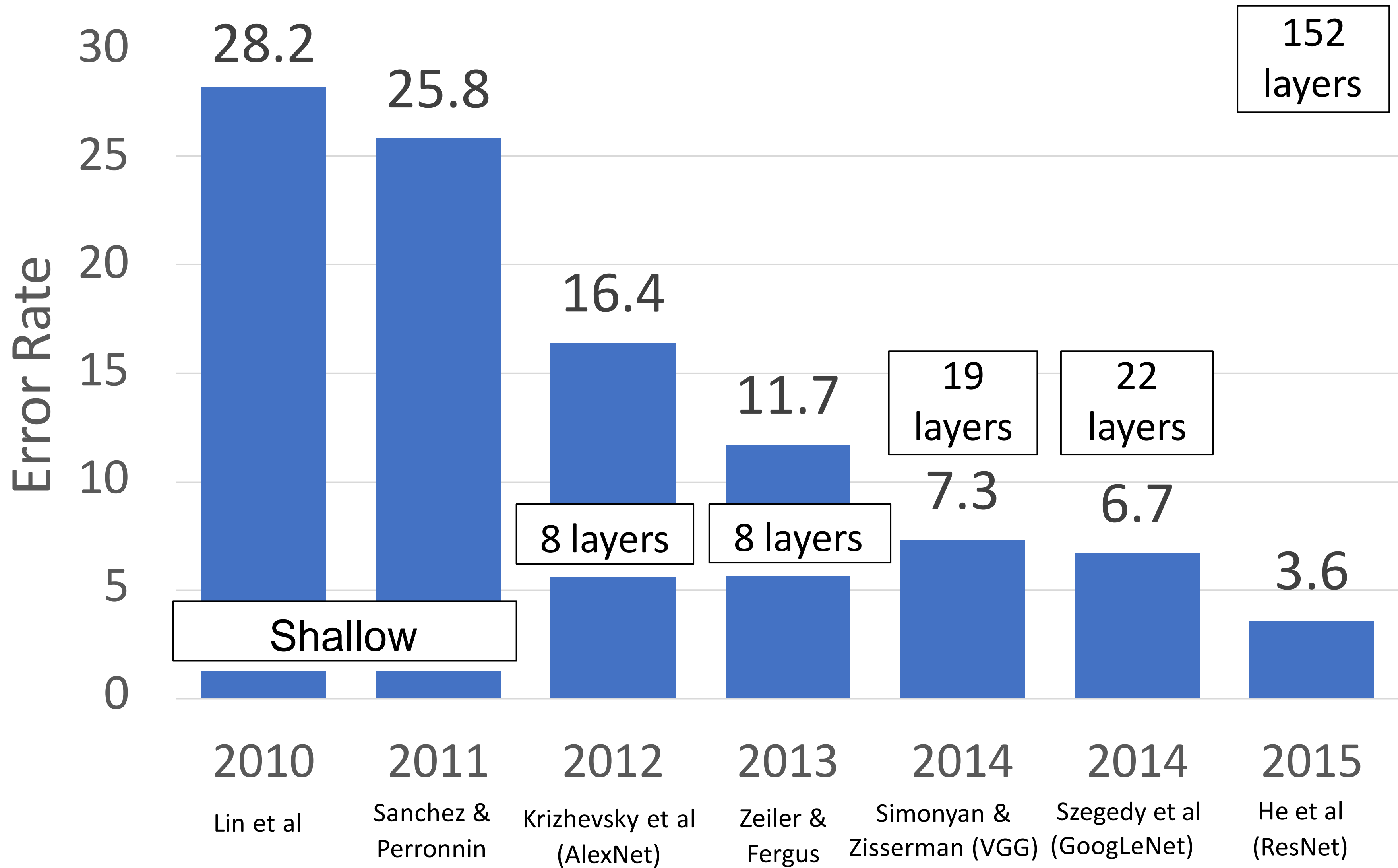
Comparing Complexity



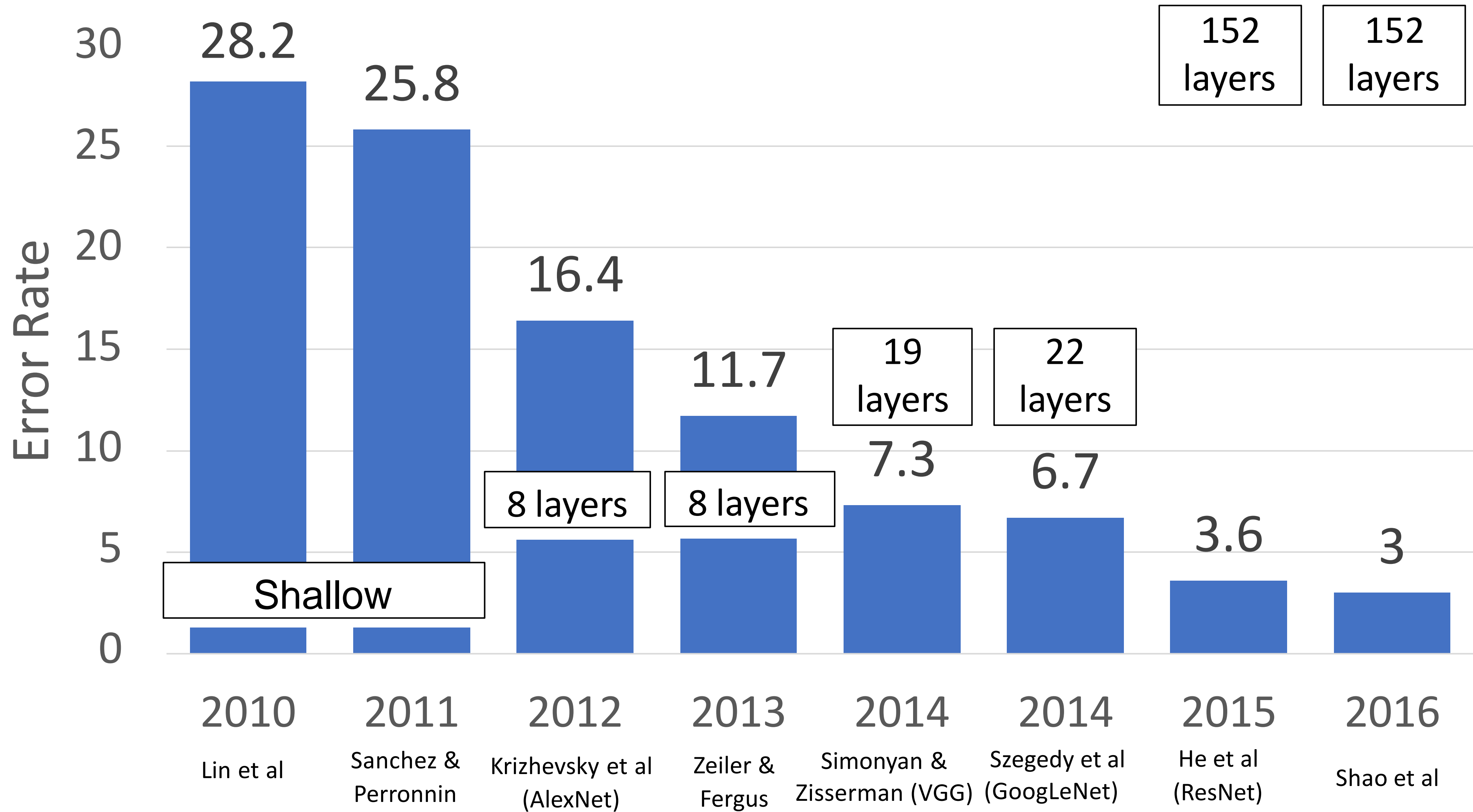


Canziani et al, "An analysis of deep neural network models for practical applications", 2017

ImageNet Classification Challenge



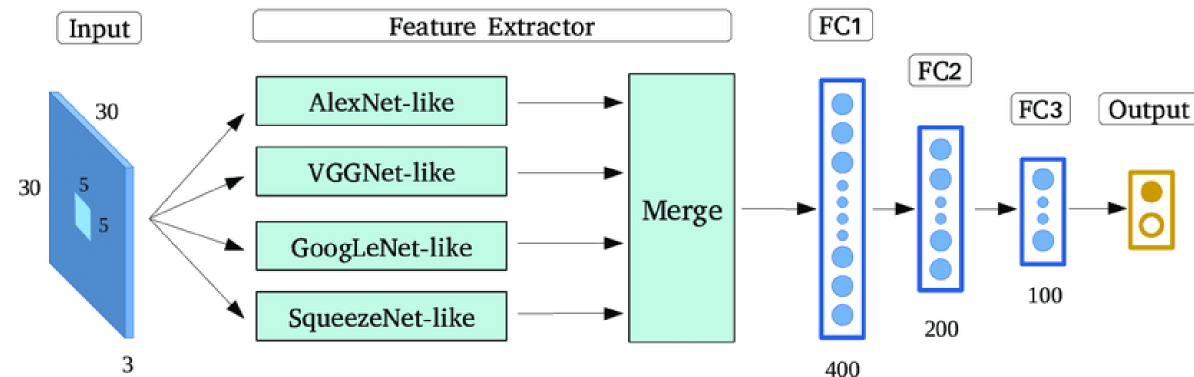
ImageNet Classification Challenge



ImageNet 2016 winner: Model Ensembles

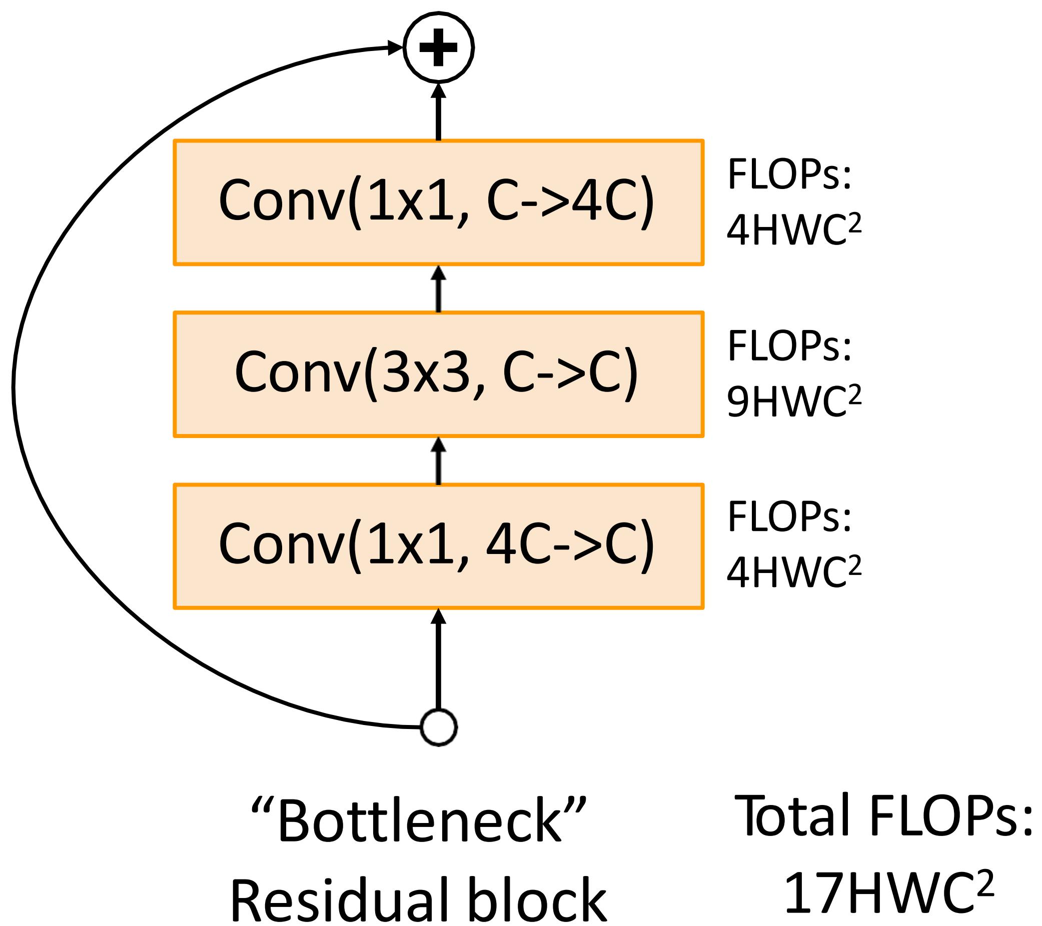
Multi-scale ensemble of Inception, Inception-Resnet, Resnet, Wide Resnet models

	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	2.92 (-0.6)	2.99



Shao et al, 2016

Improving ResNets: ResNeXt



G parallel pathways

Conv(1x1, c->4C)

Conv(3x3, c->c)

Conv(1x1, 4C->c)

...

Conv(1x1, c->4C)

Conv(3x3, c->c)

Conv(1x1, 4C->c)

Xie et al, "Aggregated residual transformations for deep neural networks", CVPR 2017

ResNeXt: Maintain computation by adding groups

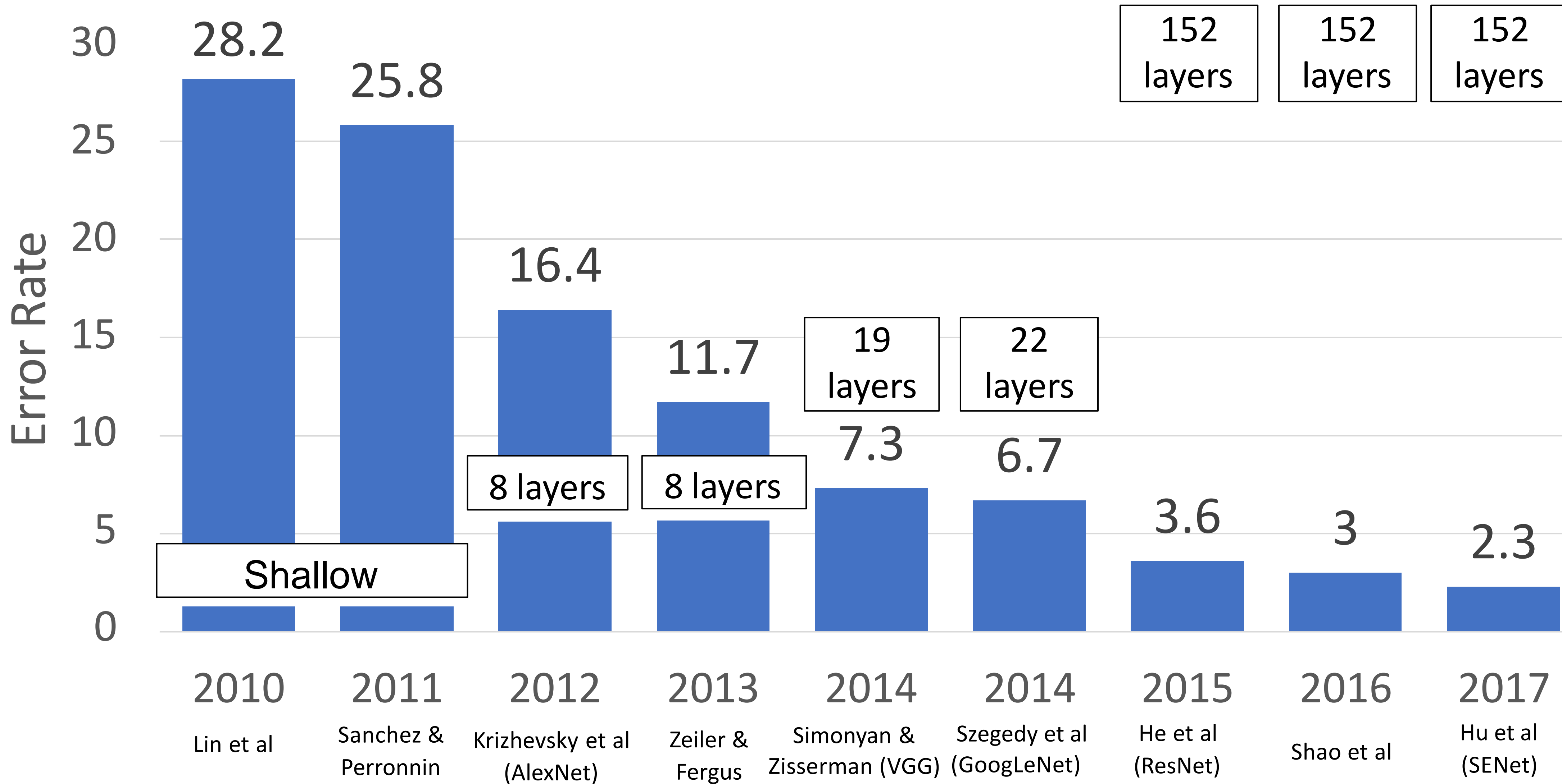
Model	Groups	Group width	Top-1 Error
ResNet-50	1	64	23.9
ResNeXt-50	2	40	23
ResNeXt-50	4	24	22.6
ResNeXt-50	8	14	22.3
ResNeXt-50	32	4	22.2

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**

Xie et al, "Aggregated residual transformations for deep neural networks", CVPR 2017

ImageNet Classification Challenge



Squeeze-and-Excitation Networks

Use channel-wise as opposed to spatial information to emphasize certain features (colored in red).

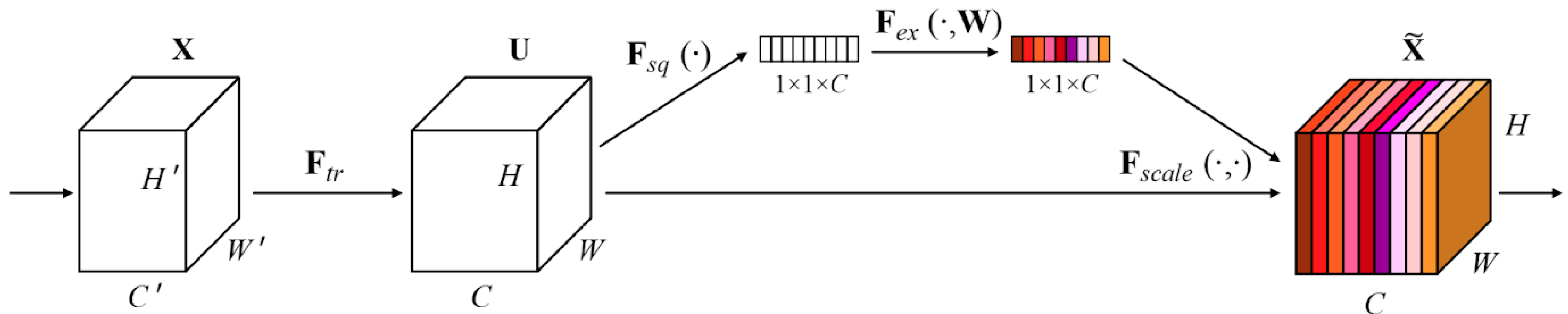


Fig. 1. A Squeeze-and-Excitation block.

Squeeze-and-Excitation Networks

Use channel-wise as opposed to spatial information to emphasize certain features (colored in red).

$$z_c = \mathbf{F}_{sq}(\mathbf{u}_c) = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} u_c(i, j).$$

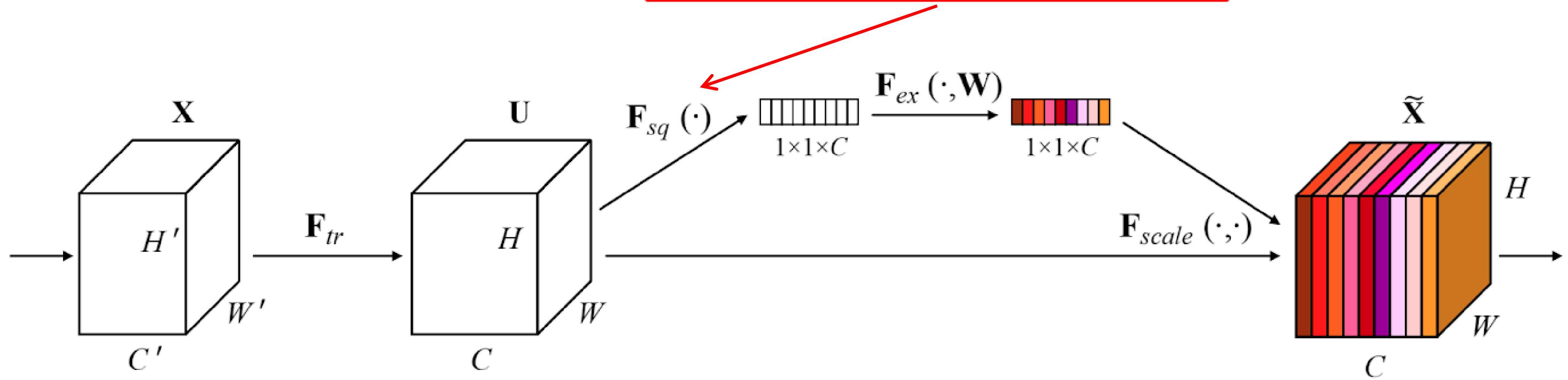


Fig. 1. A Squeeze-and-Excitation block.

Squeeze-and-Excitation Networks

Use channel-wise as opposed to spatial information to emphasize certain features (colored in red).

$$\mathbf{s} = \mathbf{F}_{ex}(\mathbf{z}, \mathbf{W}) = \sigma(g(\mathbf{z}, \mathbf{W})) = \sigma(\mathbf{W}_2\delta(\mathbf{W}_1\mathbf{z})),$$

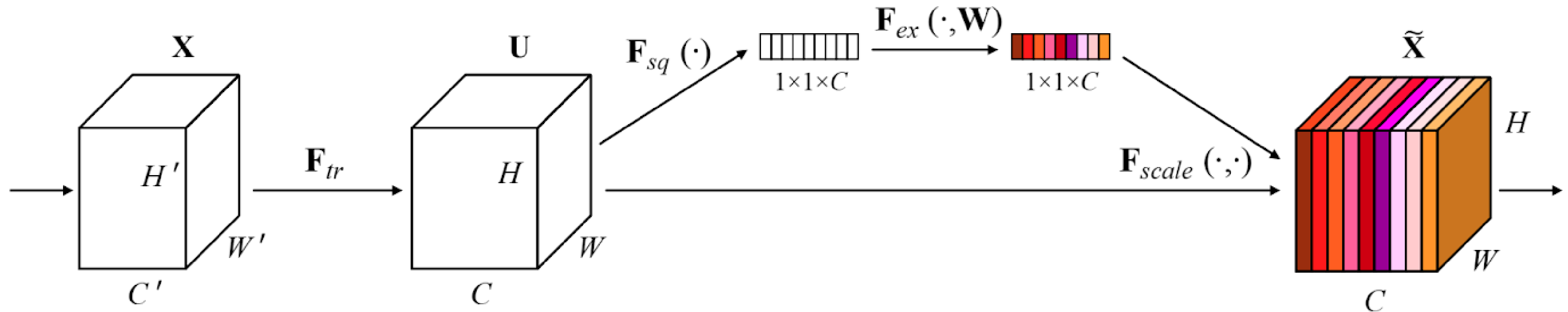
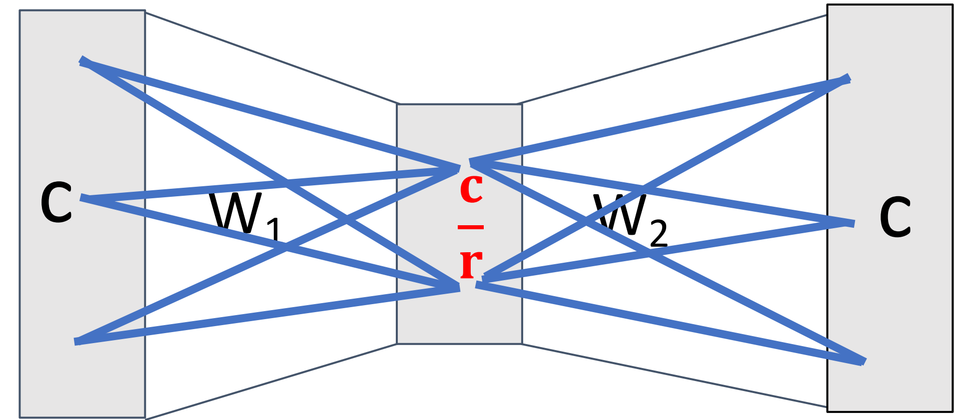


Fig. 1. A Squeeze-and-Excitation block.

Squeeze-and-Excitation Networks

Use channel-wise as opposed to spatial information to emphasize certain features (colored in red).



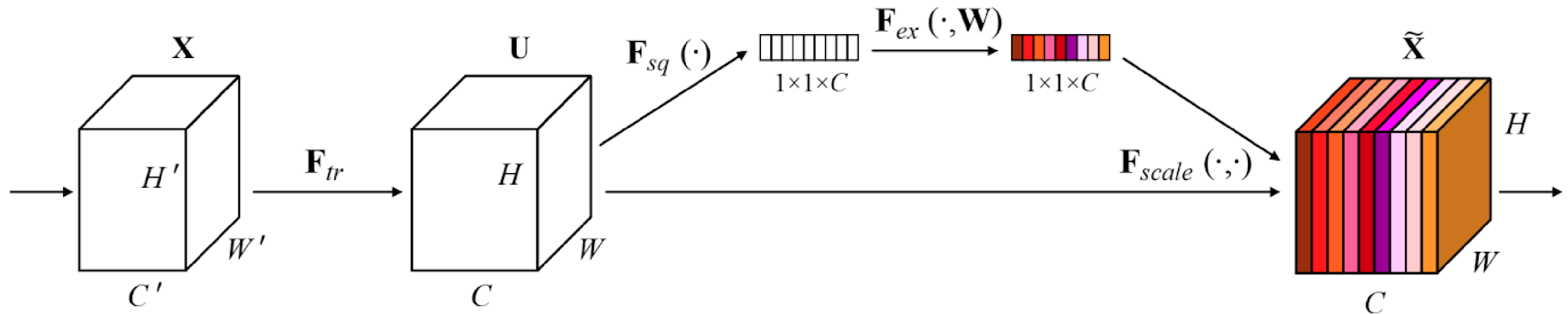


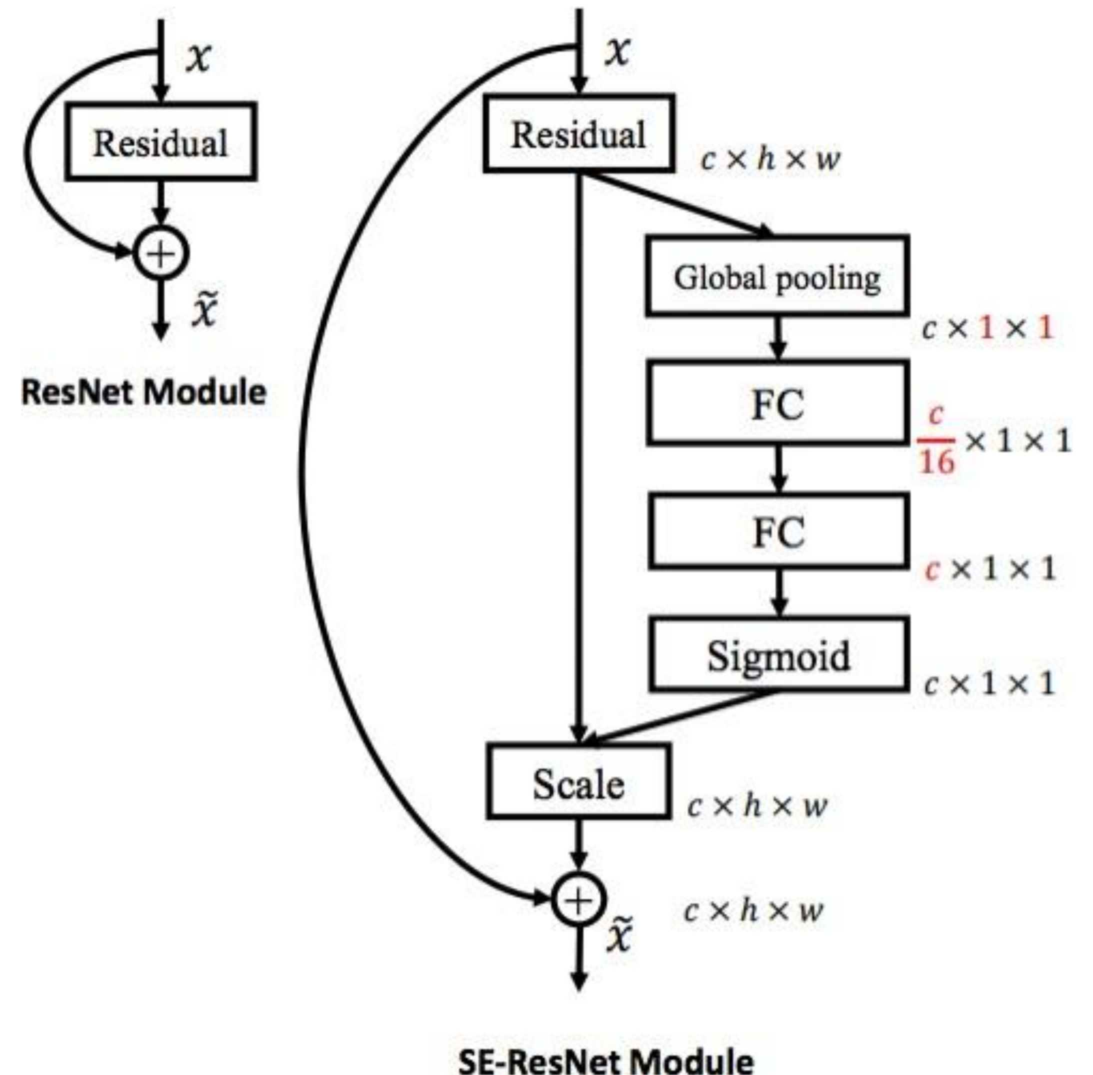
Fig. 1. A Squeeze-and-Excitation block.

Squeeze-and-Excitation Networks

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

Adds **global context** to each residual block

Won ILSVRC 2017 with ResNeXt-152-SE



Squeeze-and-Excitation Networks

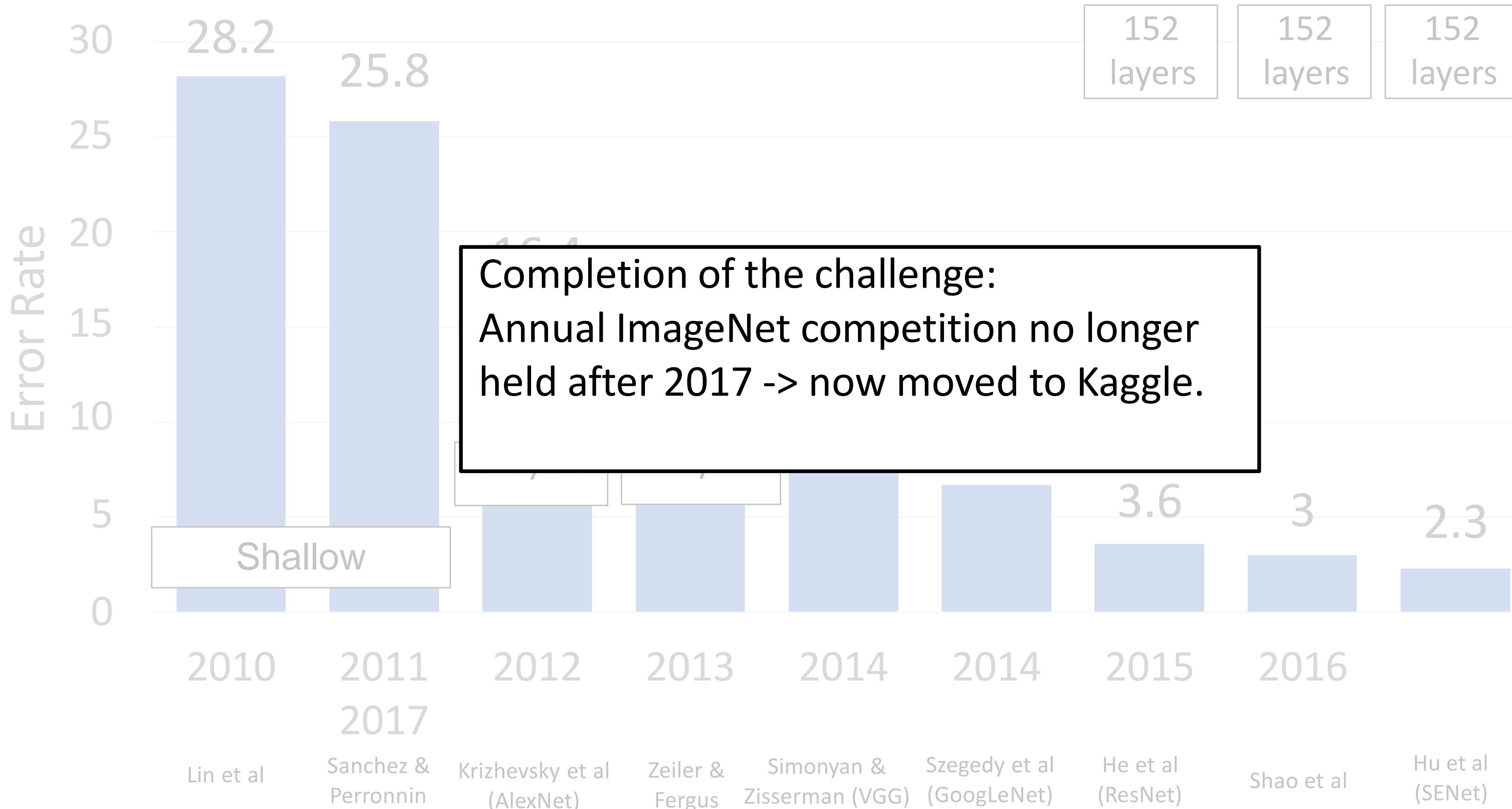
TABLE 2

Single-crop error rates (%) on the ImageNet validation set and complexity comparisons. The *original* column refers to the results reported in the original papers (the results of ResNets are obtained from the website: https://github.com/Kaiminghe/deep-residual-networks). To enable a fair comparison, we re-train the baseline models and report the scores in the *re-implementation* column. The *SENet* column refers to the corresponding architectures in which SE blocks have been added. The numbers in brackets denote the performance improvement over the re-implemented baselines. † indicates that the model has been evaluated on the non-blacklisted subset of the validation set (this is discussed in more detail in [21]), which may slightly improve results. VGG-16 and SE-VGG-16 are trained with batch normalization.

	original		re-implementation			SENet		
	top-1 err.	top-5 err.	top-1 err.	top-5 err.	GFLOPs	top-1 err.	top-5 err.	GFLOPs
ResNet-50 [13]	24.7	7.8	24.80	7.48	3.86	23.29(1.51)	6.62(0.86)	3.87
ResNet-101 [13]	23.6	7.1	23.17	6.52	7.58	22.38(0.79)	6.07(0.45)	7.60
ResNet-152 [13]	23.0	6.7	22.42	6.34	11.30	21.57(0.85)	5.73(0.61)	11.32
ResNeXt-50 [19]	22.2	-	22.11	5.90	4.24	21.10(1.01)	5.49(0.41)	4.25
ResNeXt-101 [19]	21.2	5.6	21.18	5.57	7.99	20.70(0.48)	5.01(0.56)	8.00
VGG-16 [11]	-	-	27.02	8.81	15.47	25.22(1.80)	7.70(1.11)	15.48
BN-Inception [6]	25.2	7.82	25.38	7.89	2.03	24.23(1.15)	7.14(0.75)	2.04
Inception-ResNet-v2 [21]	19.9†	4.9†	20.37	5.21	11.75	19.80(0.57)	4.79(0.42)	11.76

Hu et al, “Squeeze-and-Excitation networks”, CVPR 2018

ImageNet Classification Challenge



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)

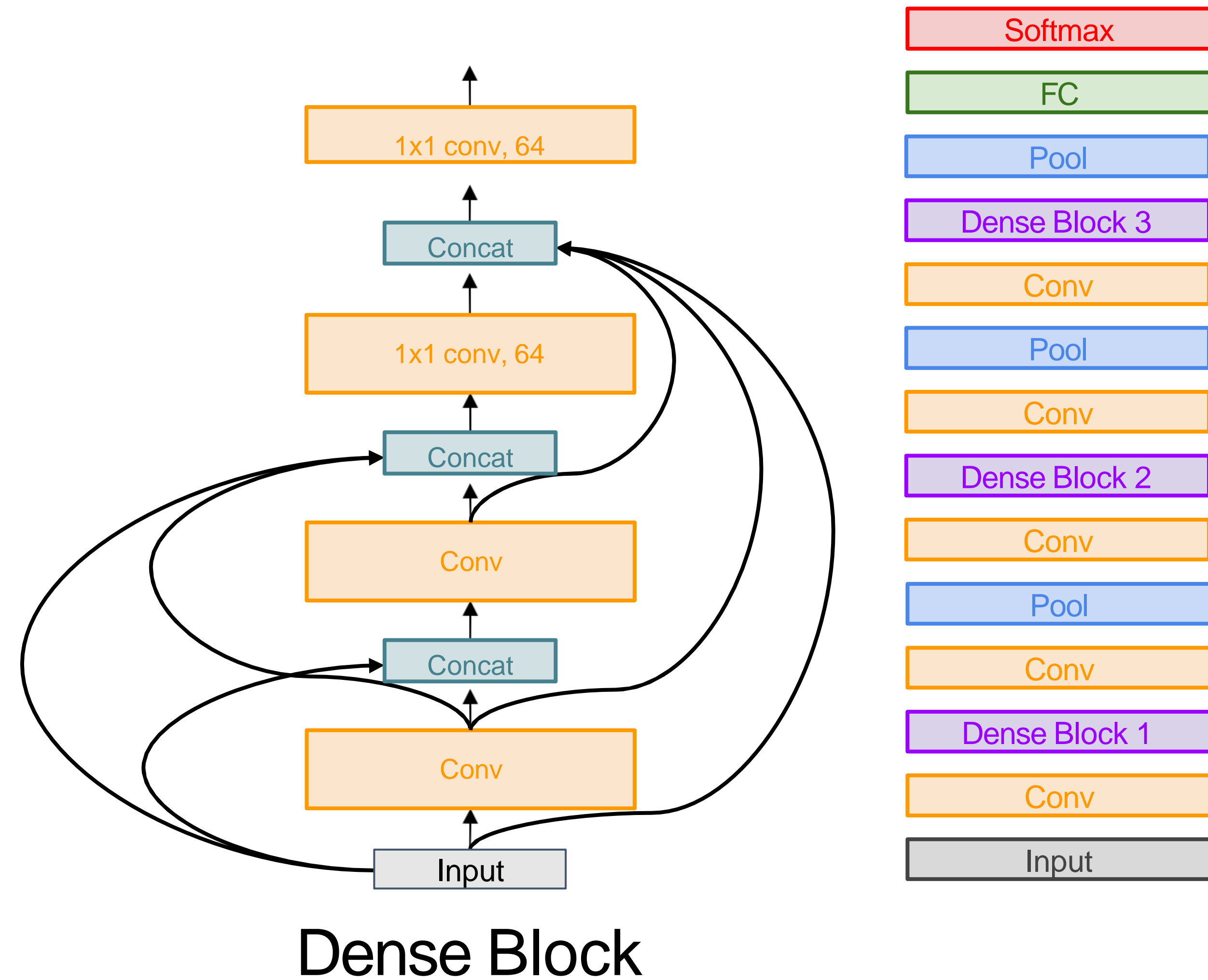
ResNet showed us how to train **very deep** networks – limited only by GPU memory. Started to show diminishing returns as networks got bigger

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

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

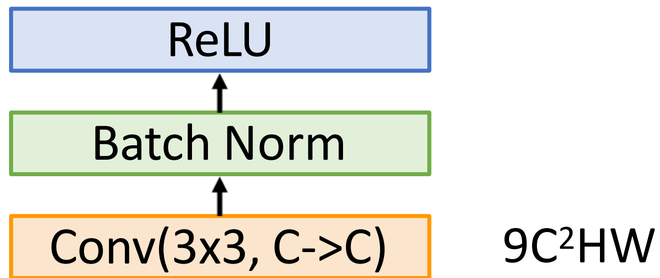


Huang et al, “Densely connected neural networks”, CVPR 2017

MobileNets: Tiny Networks (For Mobile Devices)

Standard Convolution Block

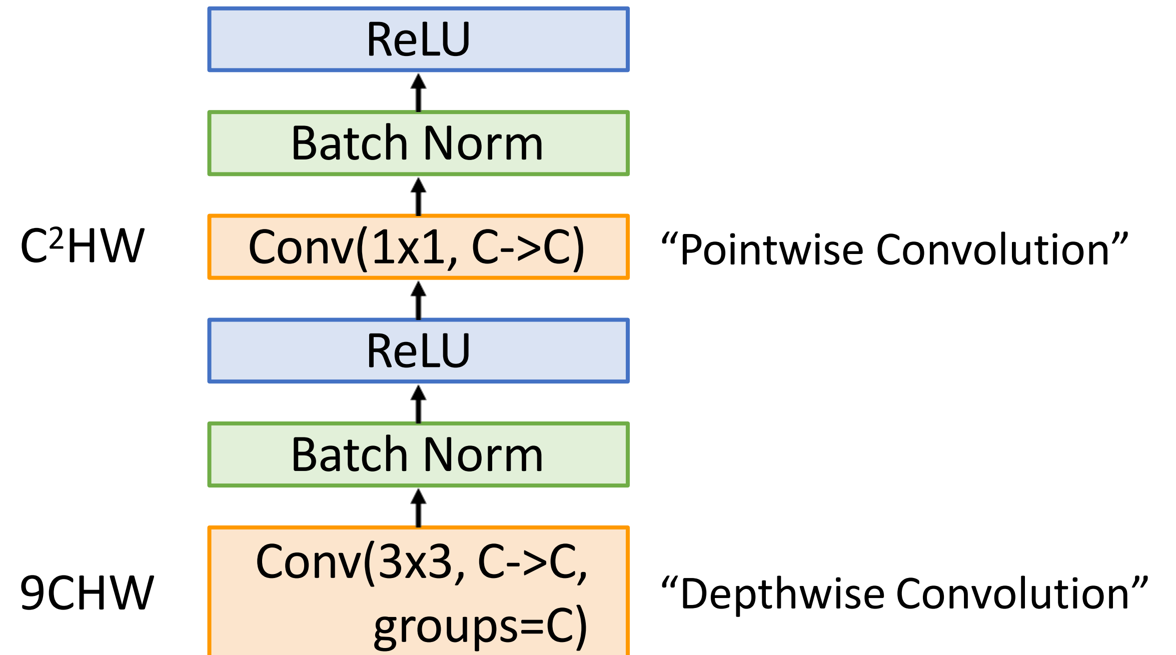
Total cost: $9C^2HW$



$$\text{Speedup} = 9C^2/(9C+C^2) = 9C/(9+C) \Rightarrow 9 \text{ (as } C\to\text{inf)}$$

Depthwise Separable Convolution

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



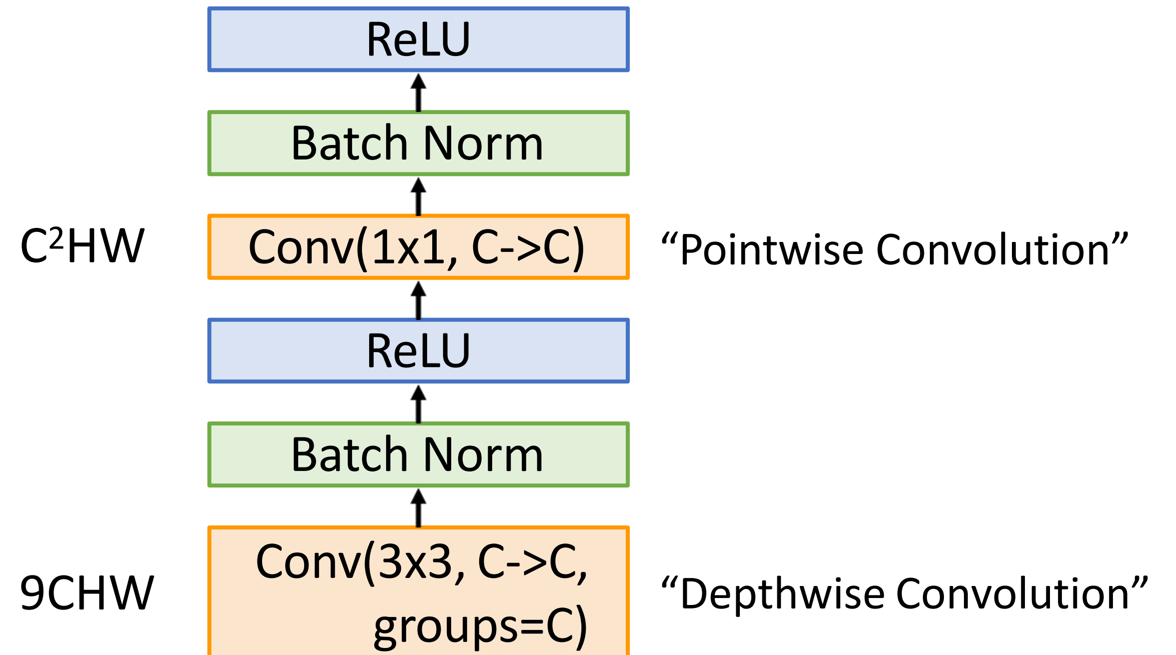
Howard et al, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications", 2017

MobileNets: Tiny Networks (For Mobile Devices)

Depthwise Separable Convolution

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

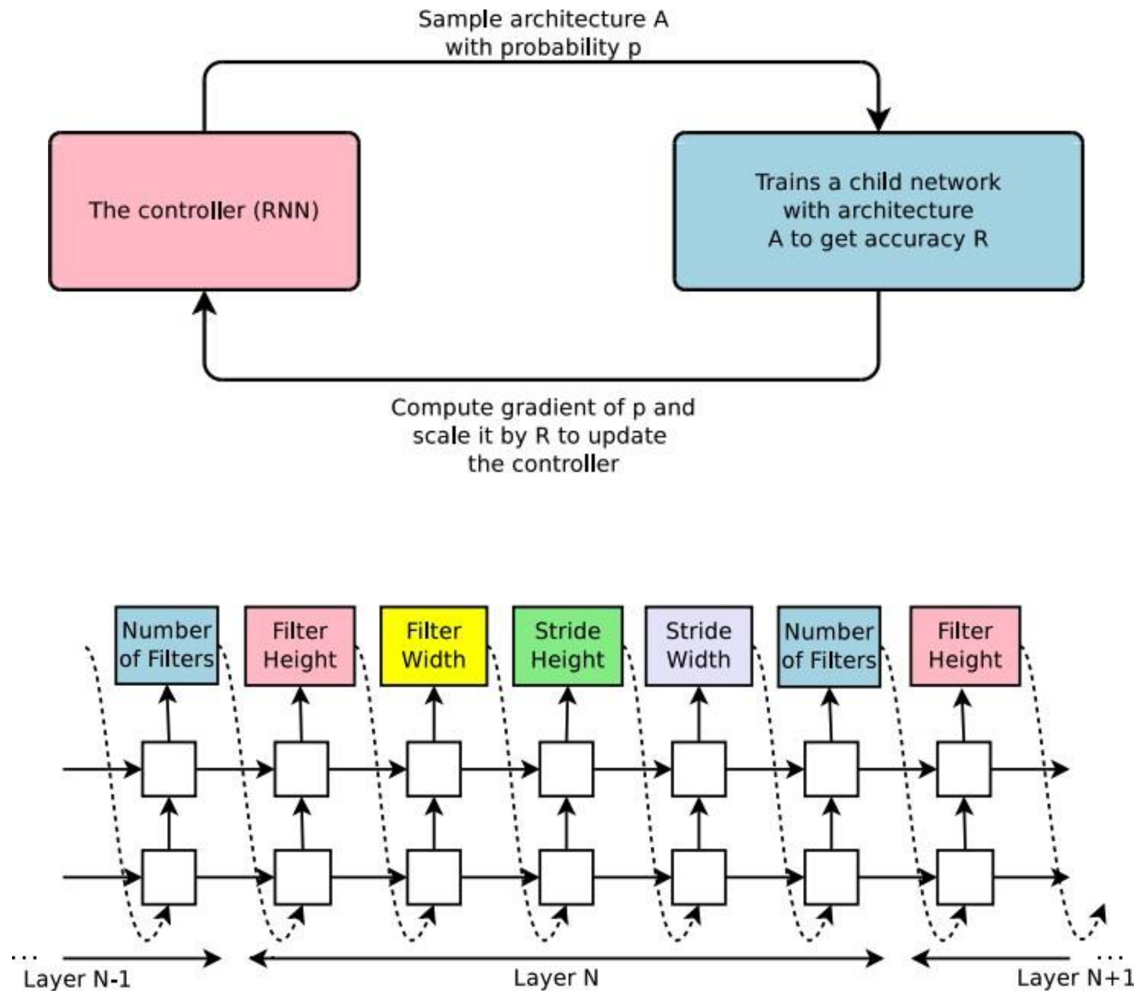
SqueezeNet
https://arxiv.org/abs/1602.07360
MobileNet
https://arxiv.org/abs/1704.04861
ShuffleNet
https://arxiv.org/abs/1707.01083
Xception
https://arxiv.org/abs/1610.02357
GhostNet
https://arxiv.org/pdf/1911.11907



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!

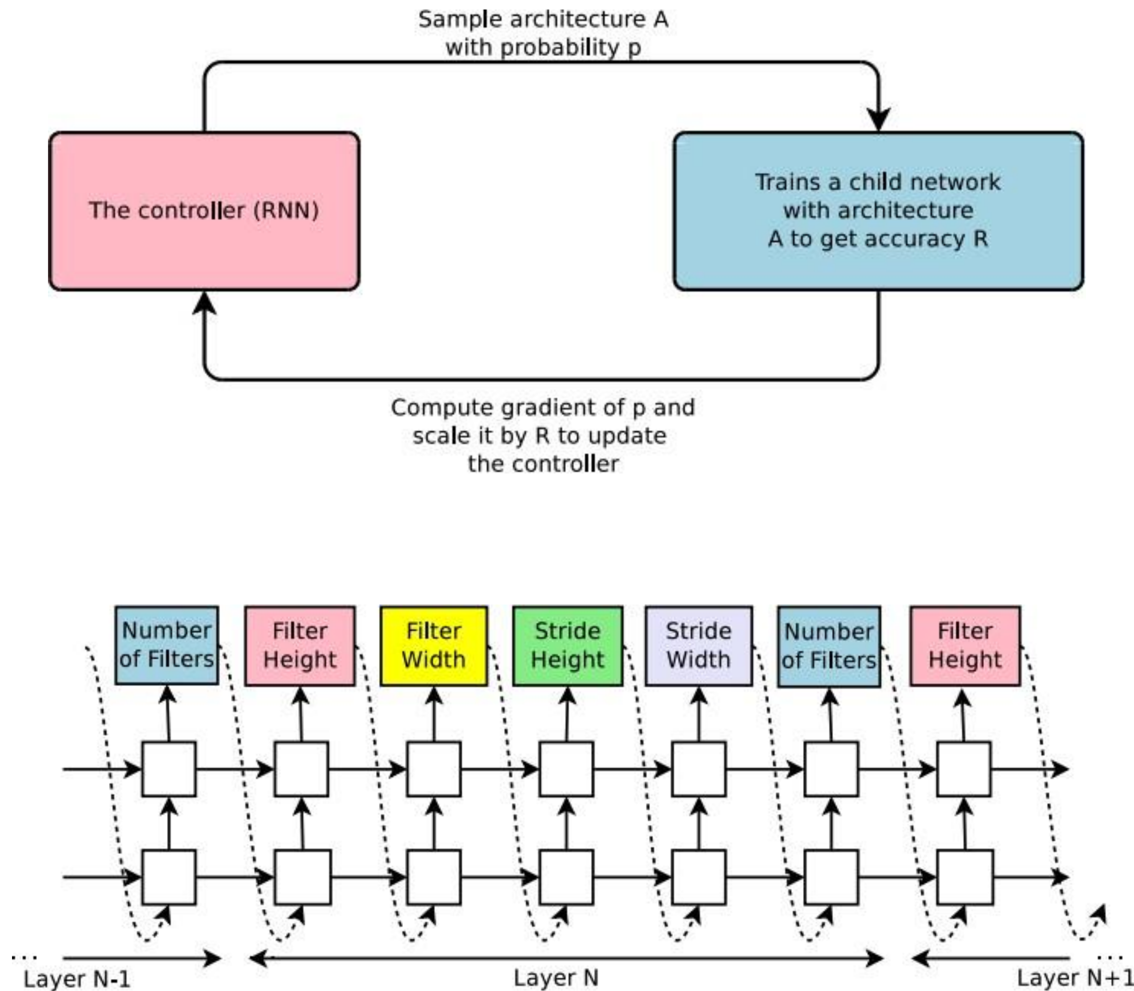


Zoph and Le, "Neural Architecture Search with Reinforcement Learning", ICLR 2017

Neural Architecture Search

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

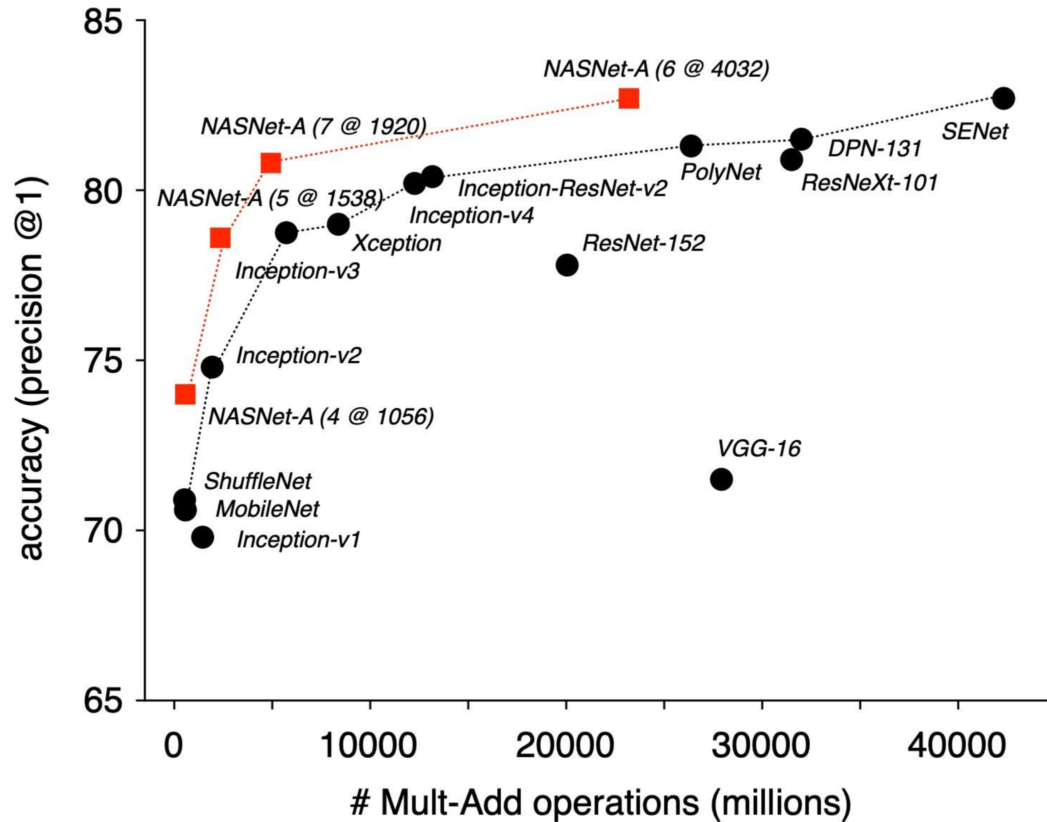
- One network (**controller**) outputs network architectures
- Sample **child networks** from controller and train them
- After training a batch of child networks, make a gradient step on controller network (Using **policy gradient**)
- Over time, controller learns to output good architectures!
- VERY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

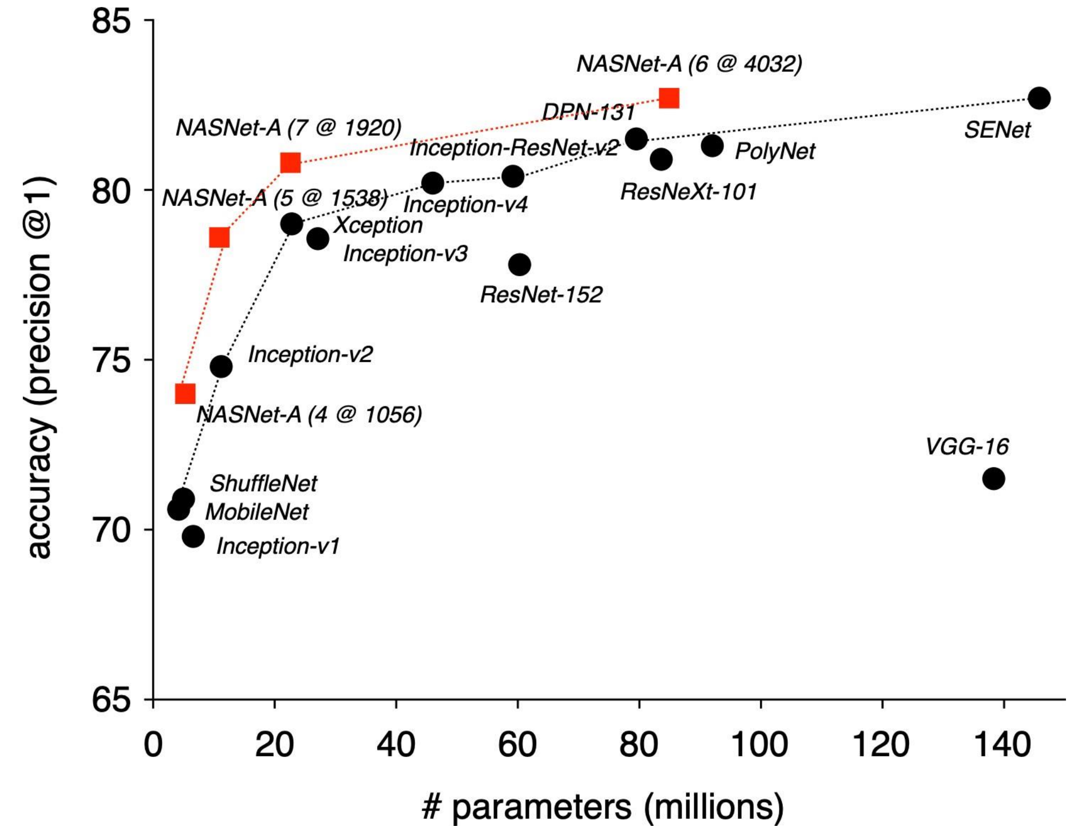


Zoph and Le, "Neural Architecture Search with Reinforcement Learning", ICLR 2017

Neural Architecture Search

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





Zoph et al, “Learning Transferable Architectures for Scalable Image Recognition”, CVPR 2018