

Lecture 8: CNN Architectures

Assignment 3

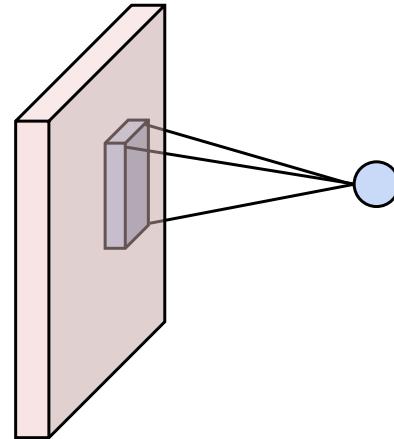
Assignment 3 is released! It covers:

- Fully-connected networks
- Dropout
- Update rules: SGD+Momentum, RMSprop, Adam
- Convolutional networks
- Batch normalization

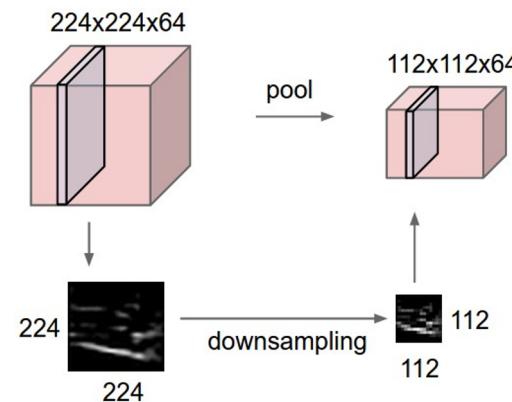
Due **Friday February 11, 11:59pm ET**

Last Time: Components of Convolutional Networks

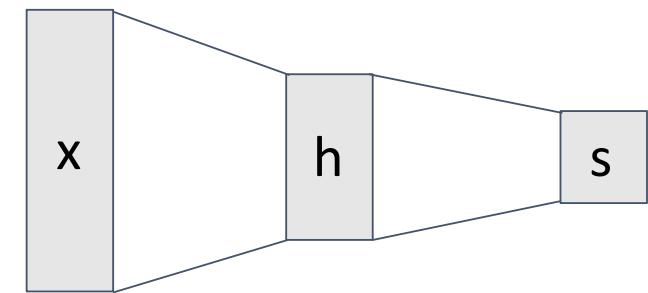
Convolution Layers



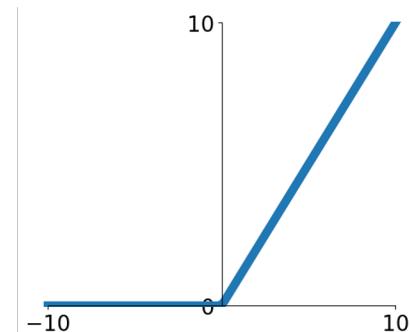
Pooling Layers



Fully-Connected Layers



Activation Function

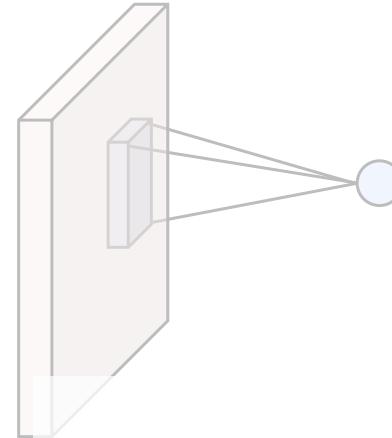


Normalization

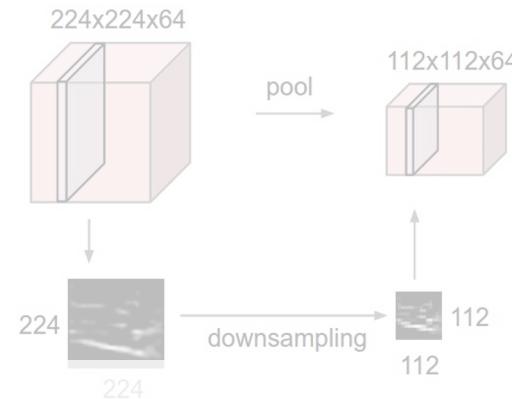
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Last Time: Components of Convolutional Networks

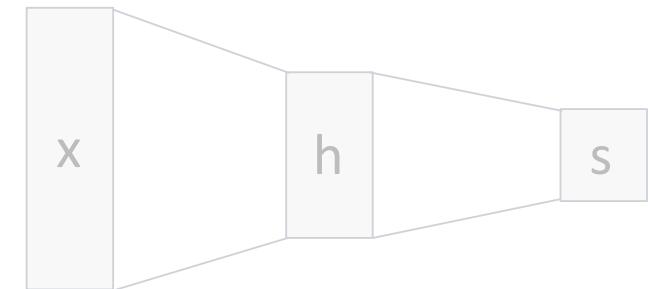
Convolution Layers



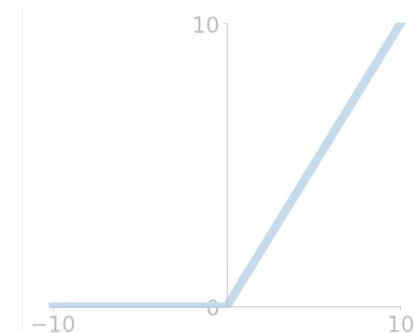
Pooling Layers



Fully-Connected Layers



Activation Function



Normalization

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

Batch Normalization

Consider a single layer $y = Wx$

The following could lead to tough optimization:

- Inputs x are not *centered around zero* (need large bias)
- Inputs x have different scaling per-element
(entries in W will need to vary a lot)

Idea: force inputs to be “nicely scaled” at each layer!

Batch Normalization

Idea: “Normalize” the inputs of a layer so they have zero mean and unit variance

We can normalize a batch of activations like this:

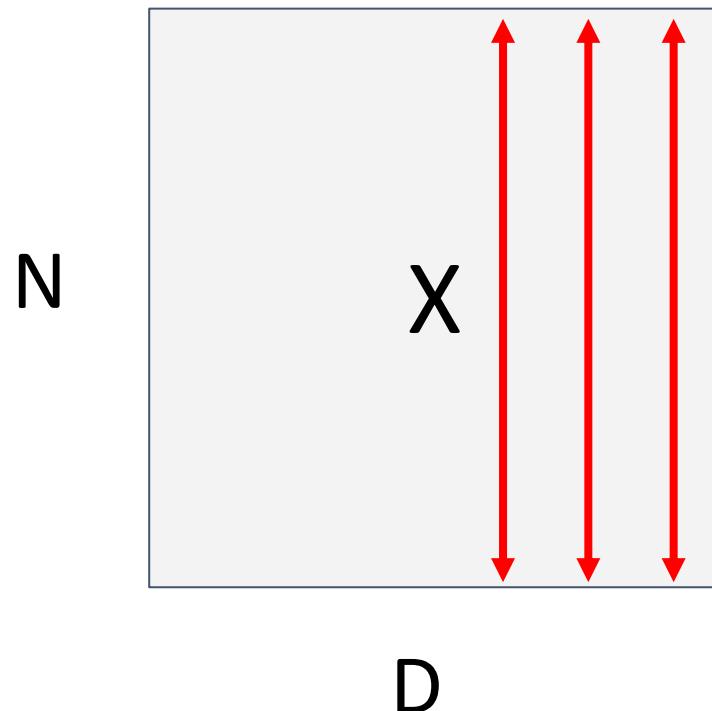
$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x]}}$$

This is a **differentiable function**, so we can use it as an operator in our networks and backprop through it!

Ioffe and Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, ICML 2015

Batch Normalization

Input: $x \in \mathbb{R}^{N \times D}$



$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel
mean, shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel
std, shape is D

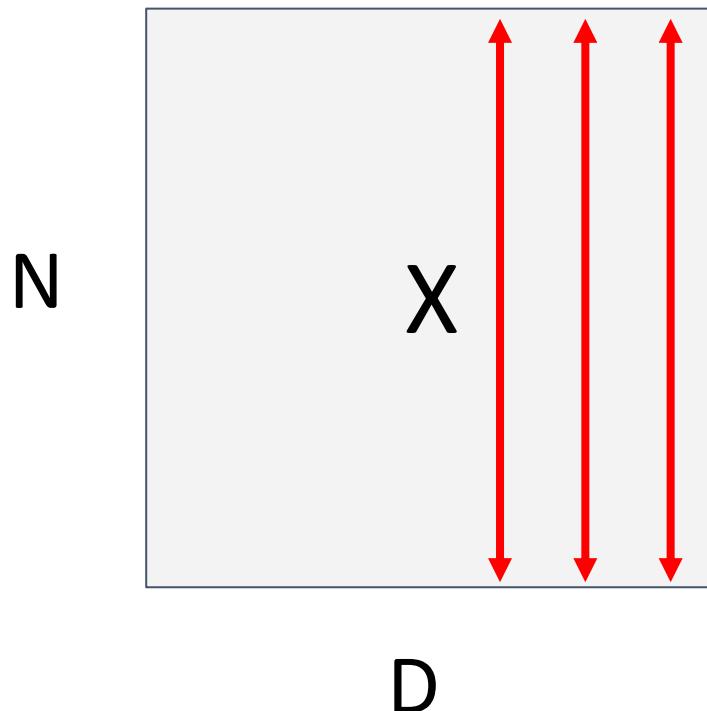
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalized x,
Shape is $N \times D$

Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Batch Normalization

Input: $x \in \mathbb{R}^{N \times D}$



$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel
mean, shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel
std, shape is D

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

Normalized x,
Shape is $N \times D$

Problem: What if zero-mean, unit variance is too hard of a constraint?

Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Batch Normalization

Input: $x \in \mathbb{R}^{N \times D}$

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel
mean, shape is D

Learnable scale and shift parameters:

$$\gamma, \beta \in \mathbb{R}^D$$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function (in expectation)

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel std, shape is D

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

Normalized x,
Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output,
Shape is N x D

Batch Normalization

Problem: Estimates depend on minibatch; can't do this at test-time!

Input: $x \in \mathbb{R}^{N \times D}$

Learnable scale and shift parameters:

$$\gamma, \beta \in \mathbb{R}^D$$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function (in expectation)

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel std, shape is D

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

Normalized x,
Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output,
Shape is N x D

Batch Normalization: Test-Time

Input: $x \in \mathbb{R}^{N \times D}$

μ_j = (Running) average of values seen during training Per-channel mean, shape is D

Learnable scale and shift parameters:

$\gamma, \beta \in \mathbb{R}^D$

σ_j^2 = (Running) average of values seen during training Per-channel std, shape is D

Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function (in expectation)

$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$ Normalized x, Shape is N x D

$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$ Output, Shape is N x D

Batch Normalization: Test-Time

Input: $x \in \mathbb{R}^{N \times D}$

μ_j = (Running) average of values seen during training

Per-channel mean, shape is D

Learnable scale and shift parameters:

$\gamma, \beta \in \mathbb{R}^D$

Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function (in expectation)

$$\mu_j^{test} = 0$$

For each training iteration:

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

$$\mu_j^{test} = 0.99 \mu_j^{test} + 0.01 \mu_j$$

(Similar for σ)

Batch Normalization: Test-Time

Input: $x \in \mathbb{R}^{N \times D}$

μ_j = (Running) average of values seen during training Per-channel mean, shape is D

Learnable scale and shift parameters:

$\gamma, \beta \in \mathbb{R}^D$

σ_j^2 = (Running) average of values seen during training Per-channel std, shape is D

Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function (in expectation)

$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$ Normalized x, Shape is N x D

$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$ Output, Shape is N x D

Batch Normalization: Test-Time

Input: $x \in \mathbb{R}^{N \times D}$

μ_j = (Running) average of values seen during training
Per-channel mean, shape is D

Learnable scale and shift parameters:

$\gamma, \beta \in \mathbb{R}^D$

During testing batchnorm becomes a linear operator!

Can be fused with the previous fully-connected or conv layer

σ_j^2 = (Running) average of values seen during training
Per-channel std, shape is D

$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$
Normalized x,
Shape is N x D

$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$
Output,
Shape is N x D

Batch Normalization for ConvNets

Batch Normalization for
fully-connected networks

$$x : N \times D$$

Normalize

$$\mu, \sigma : 1 \times D$$

$$\gamma, \beta : 1 \times D$$

$$y = \frac{(x - \mu)}{\sigma} \gamma + \beta$$

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

$$x : N \times C \times H \times W$$

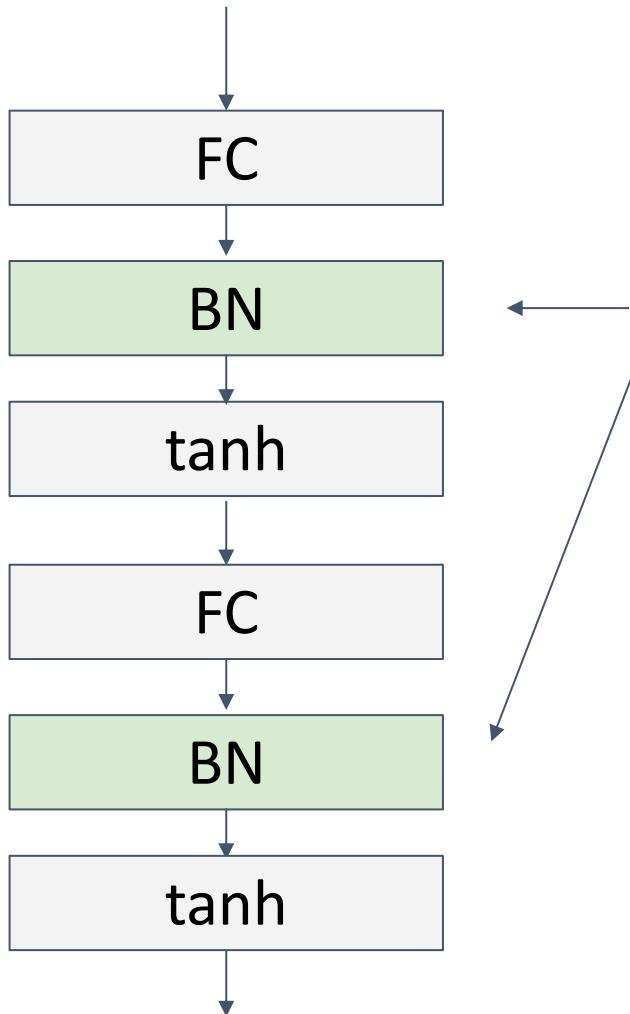
Normalize

$$\mu, \sigma : 1 \times C \times 1 \times 1$$

$$\gamma, \beta : 1 \times C \times 1 \times 1$$

$$y = \frac{(x - \mu)}{\sigma} \gamma + \beta$$

Batch Normalization

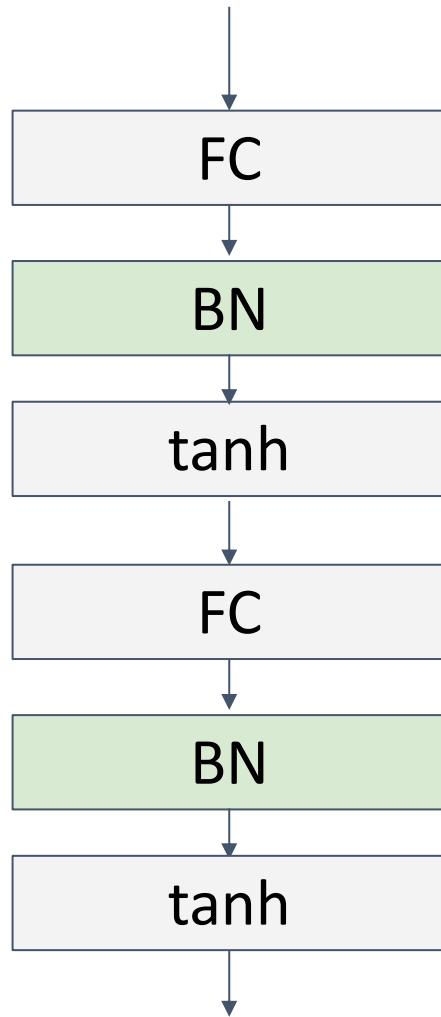


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

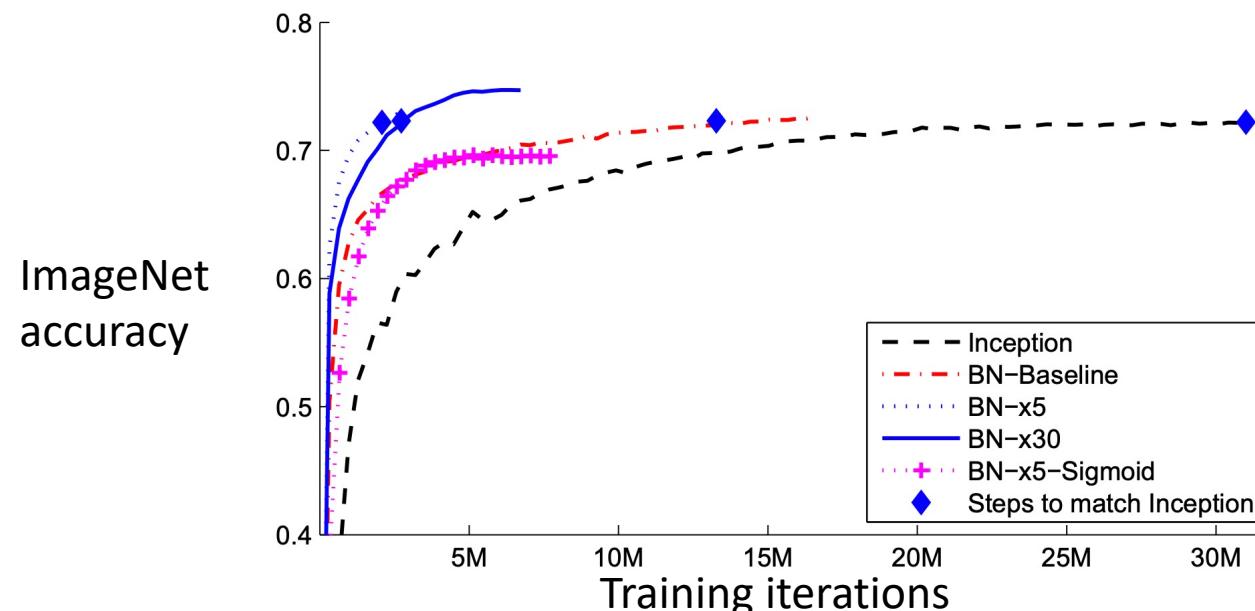
$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x]}}$$

Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Batch Normalization

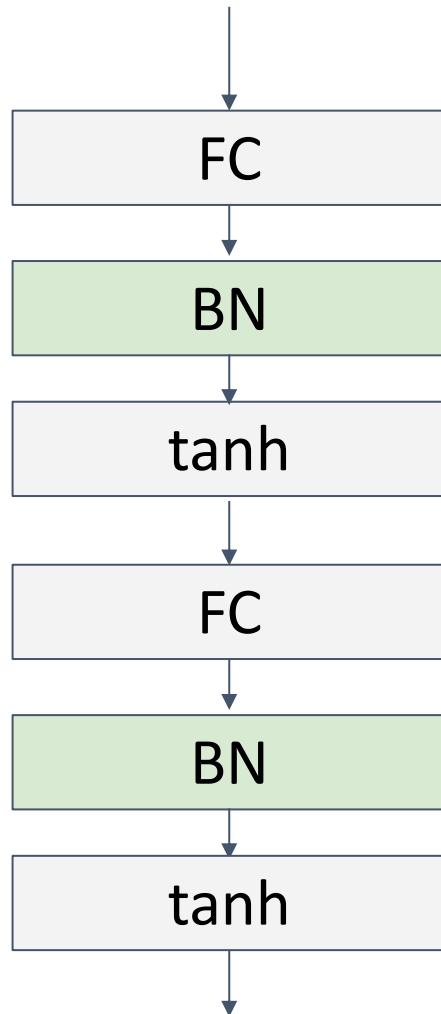


- Makes deep networks **much** easier to train!
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!



Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Batch Normalization



- Makes deep networks **much** easier to train!
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Not well-understood theoretically (yet)
- Behaves differently during training and testing: this is a **very common source of bugs!**

Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Layer Normalization

Batch Normalization for
fully-connected networks

$$x : N \times D$$

Normalize

$$\mu, \sigma : 1 \times D$$

$\gamma, \beta : 1 \times D$

$$y = \frac{(x - \mu)}{\sigma} \gamma + \beta$$

Layer Normalization for fully-

connected networks

Same behavior at train and test!

Used in RNNs, Transformers

$$x : N \times D$$

Normalize

$$\mu, \sigma : N \times 1$$

$\gamma, \beta : 1 \times D$

$$y = \frac{(x - \mu)}{\sigma} \gamma + \beta$$

Instance Normalization

Batch Normalization for convolutional networks

$$x : N \times C \times H \times W$$

Normalize

$$\mu, \sigma : 1 \times C \times 1 \times 1$$

$$\gamma, \beta : 1 \times C \times 1 \times 1$$

$$y = \frac{(x - \mu)}{\sigma} \gamma + \beta$$

Instance Normalization for convolutional networks

$$x : N \times C \times H \times W$$

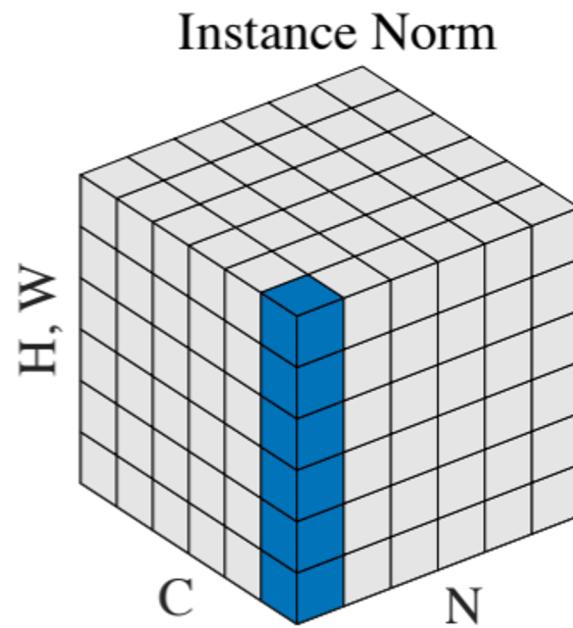
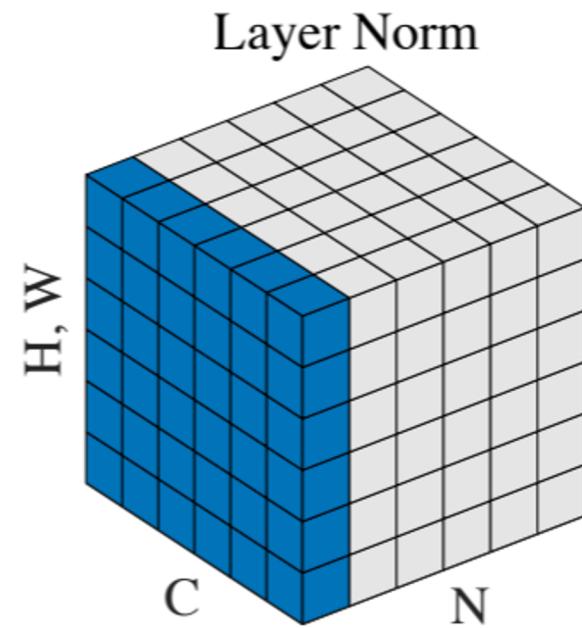
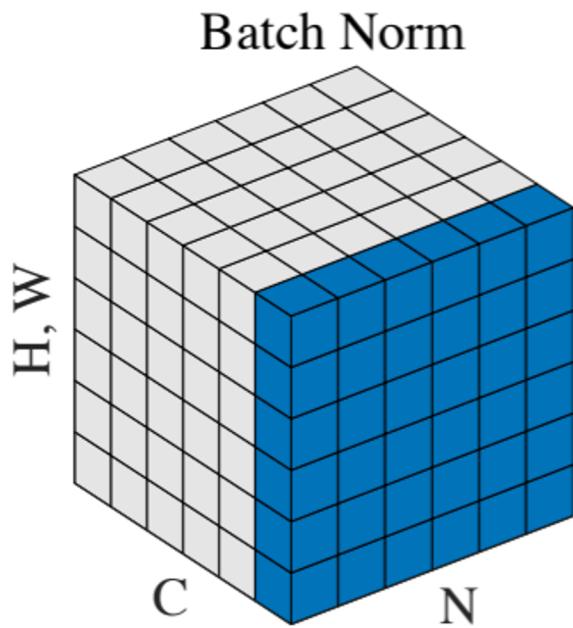
Normalize

$$\mu, \sigma : N \times C \times 1 \times 1$$

$$\gamma, \beta : 1 \times C \times 1 \times 1$$

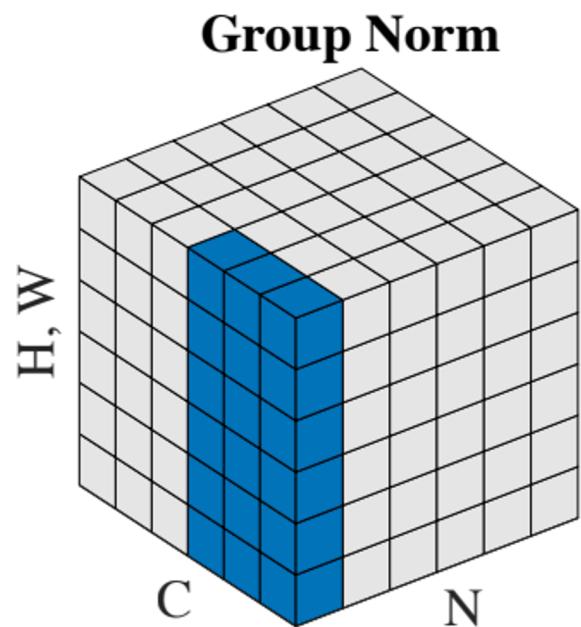
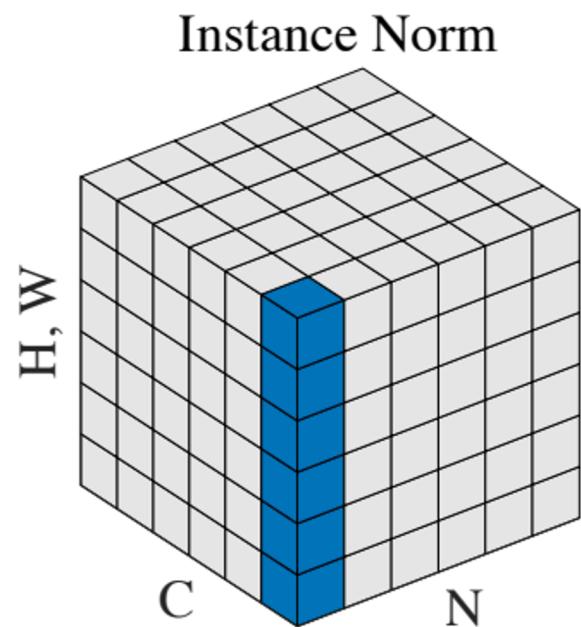
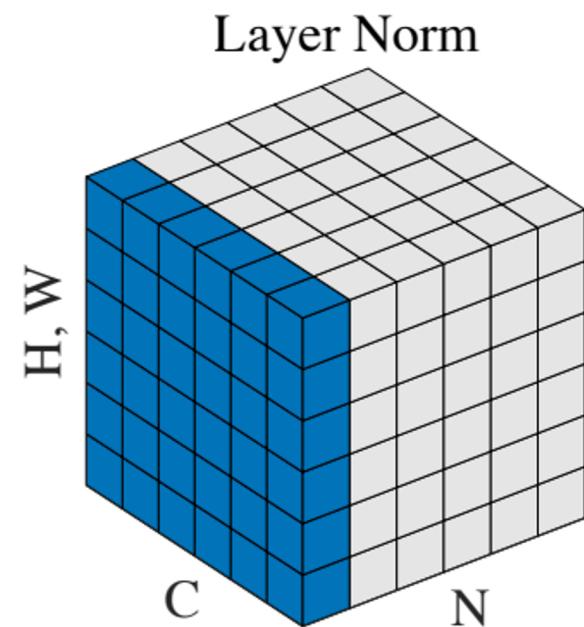
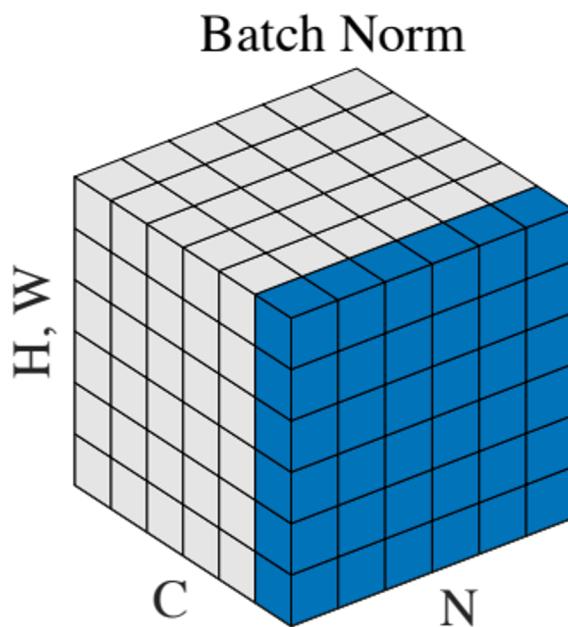
$$y = \frac{(x - \mu)}{\sigma} \gamma + \beta$$

Comparison of Normalization Layers



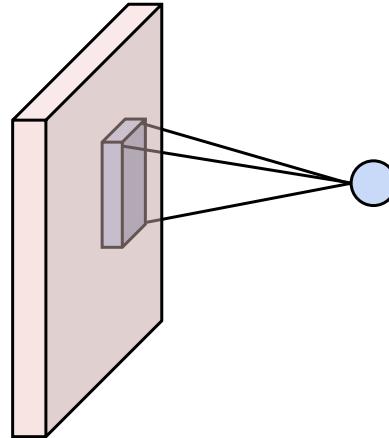
Wu and He, "Group Normalization", ECCV 2018

Group Normalization

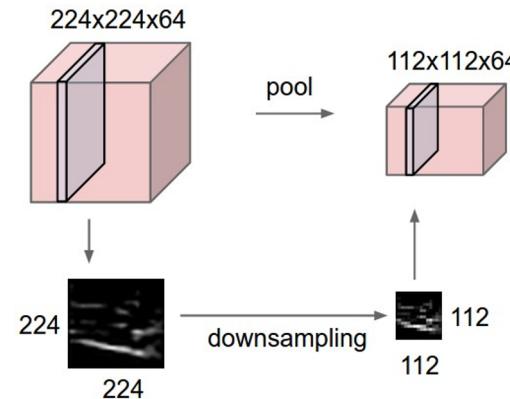


Components of Convolutional Networks

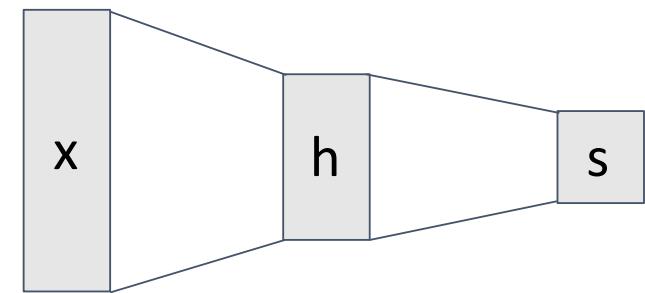
Convolution Layers



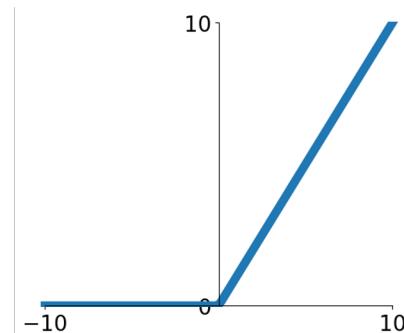
Pooling Layers



Fully-Connected Layers



Activation Function

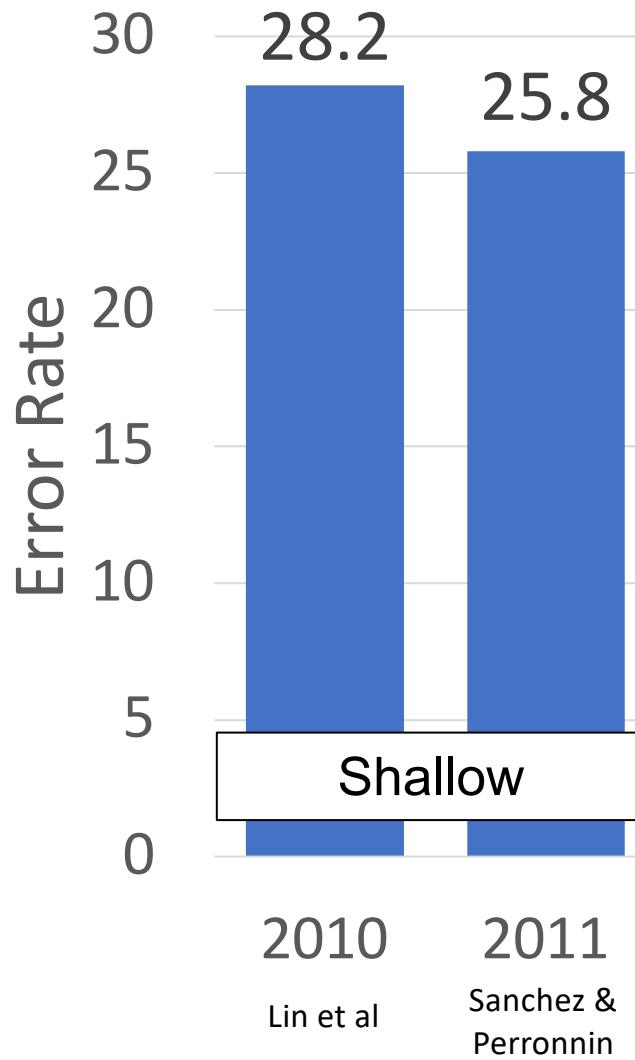


Normalization

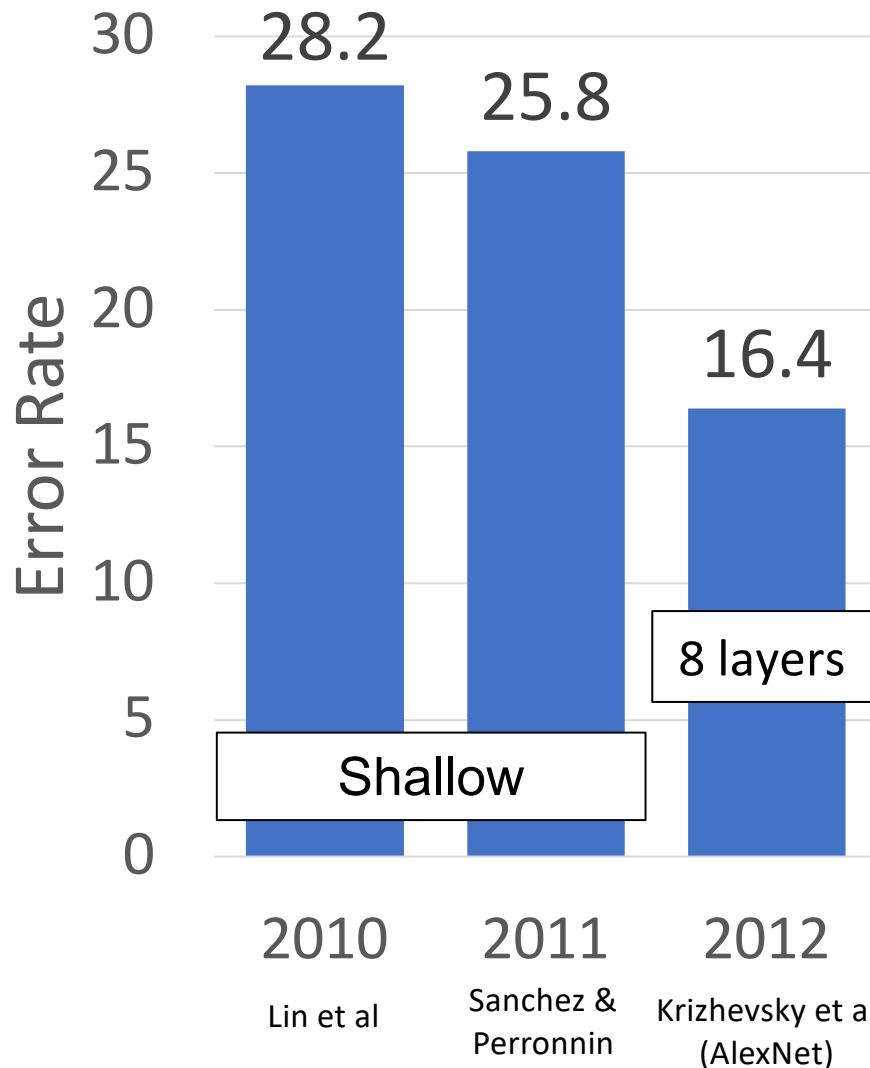
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Question: How should we put them together?

ImageNet Classification Challenge



ImageNet Classification Challenge



AlexNet

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

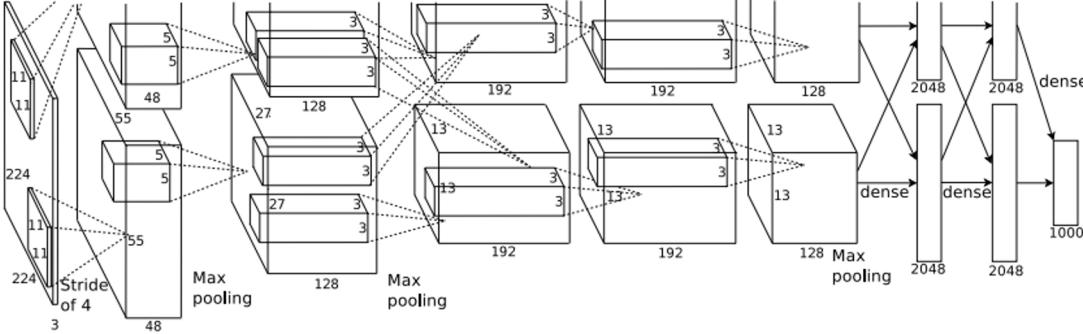
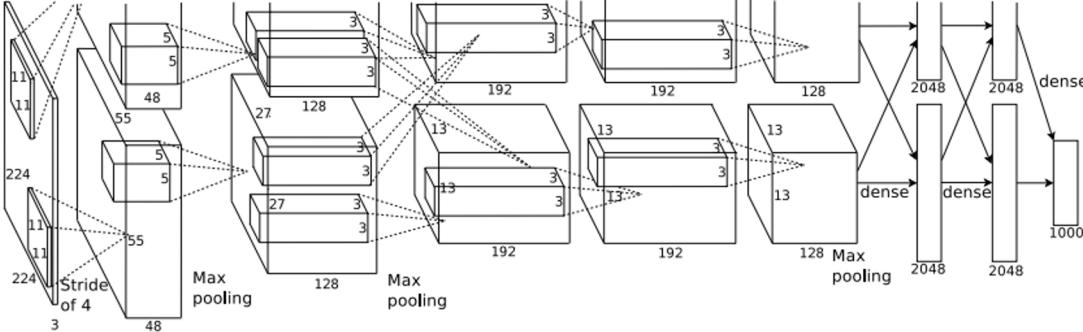


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

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

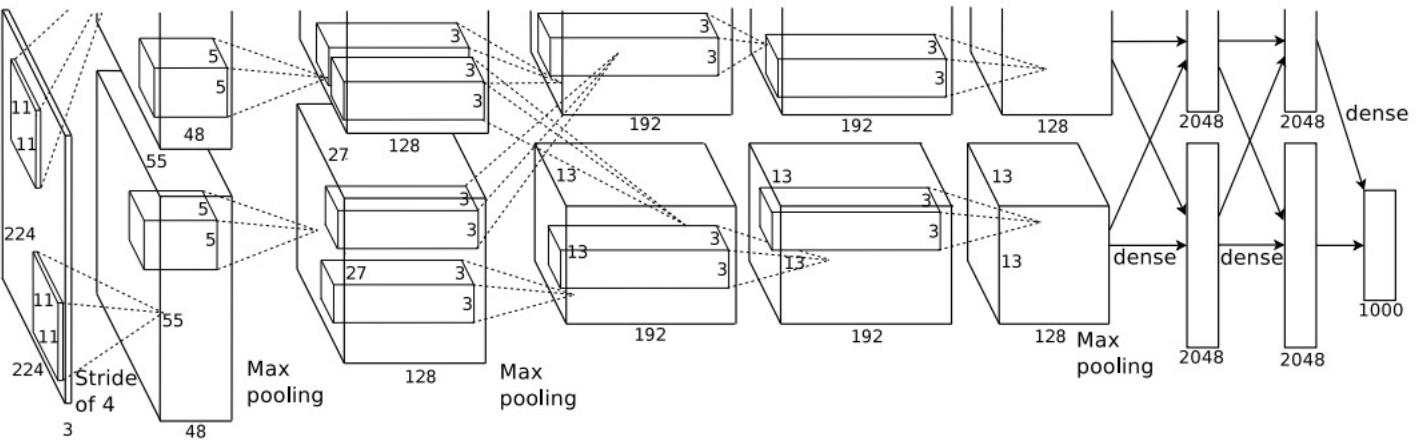


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input 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.

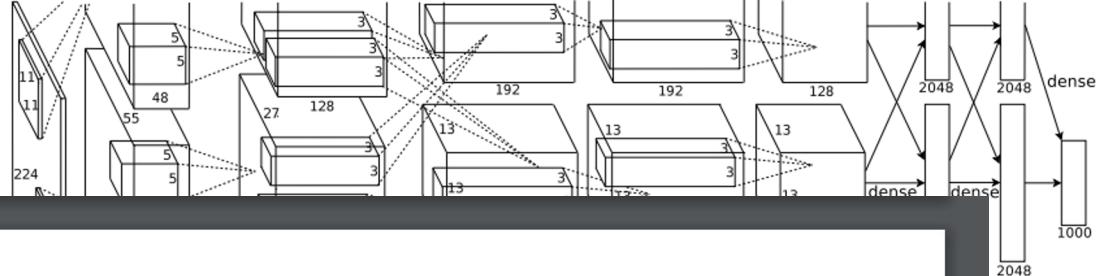
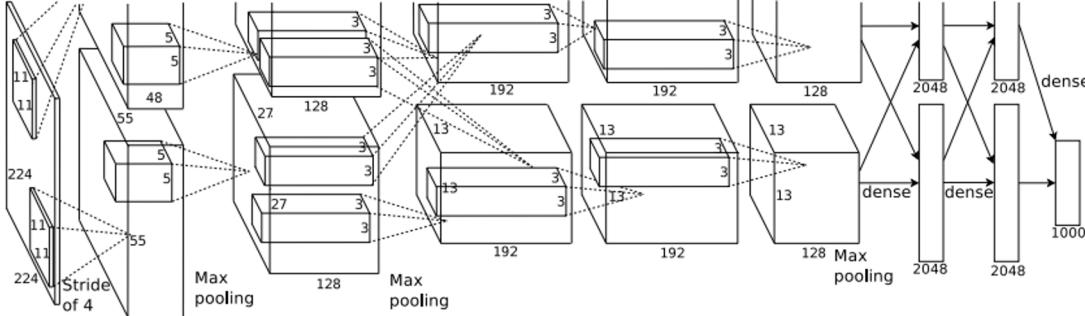
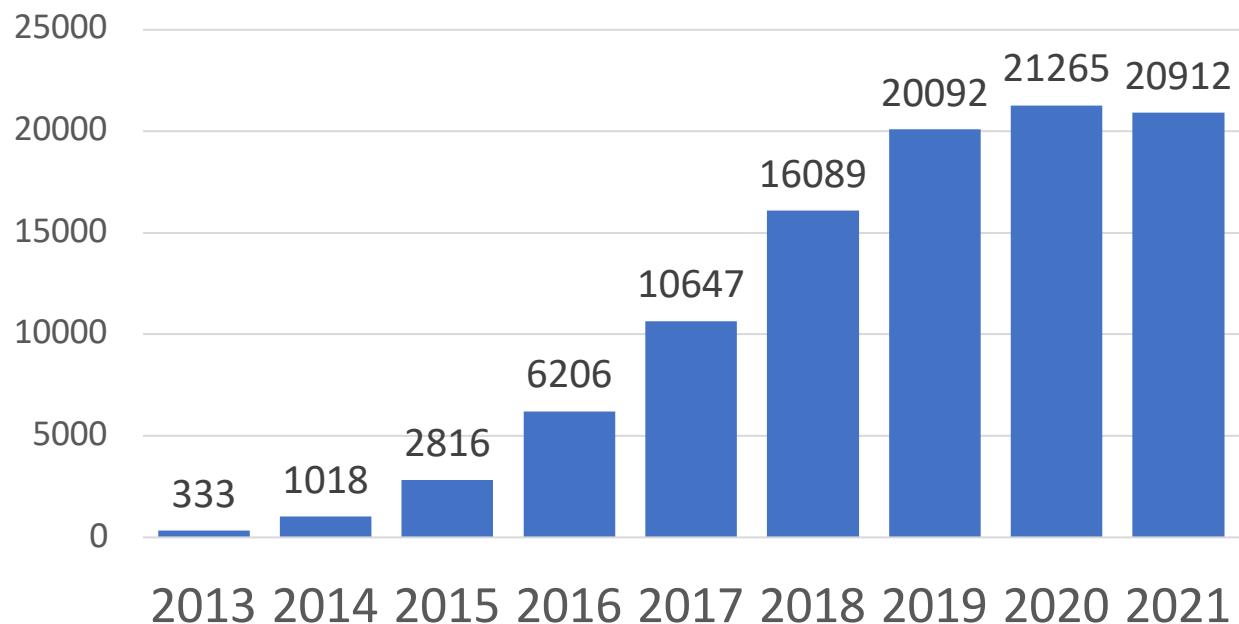


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

AlexNet



AlexNet Citations per year
(as of 2/2/2022)

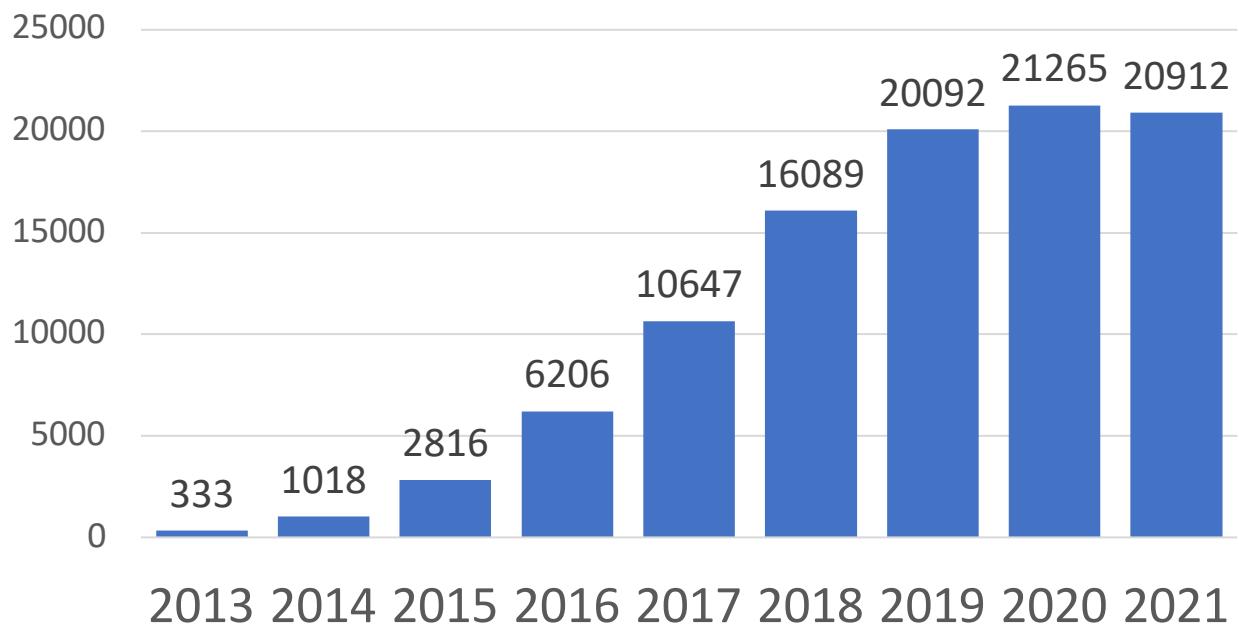


Total Citations: **102,486**

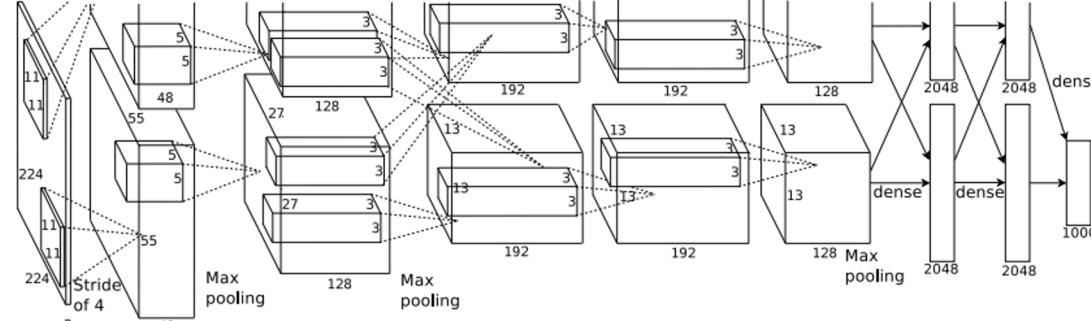
Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

AlexNet

AlexNet Citations per year
(as of 2/2/2022)



Total Citations: **102,486**



Citation Counts

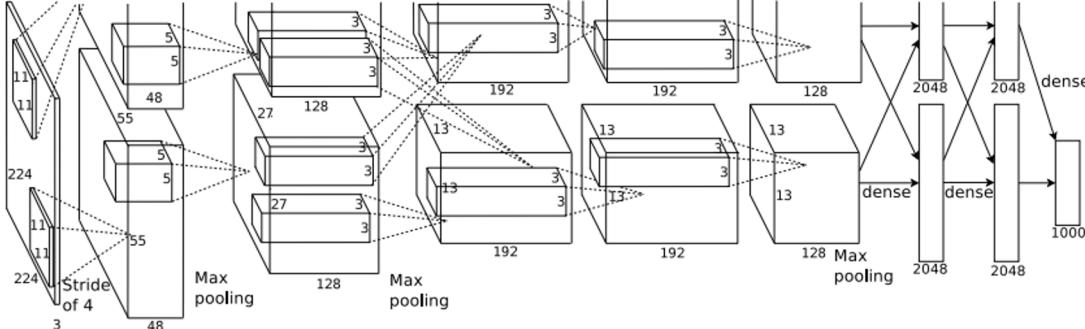
Darwin, "On the origin of species", 1859: **60,117**

Shannon, "A mathematical theory of communication", 1948: **140,459**

Watson and Crick, "Molecular Structure of Nucleic Acids", 1953: **16,298**

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

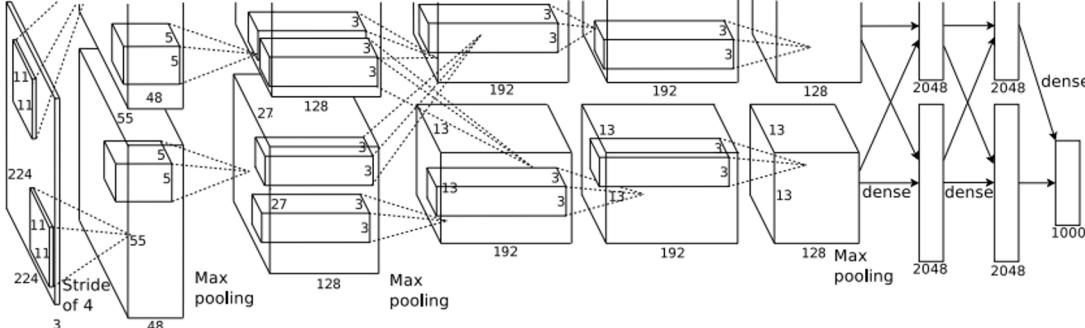
AlexNet



	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

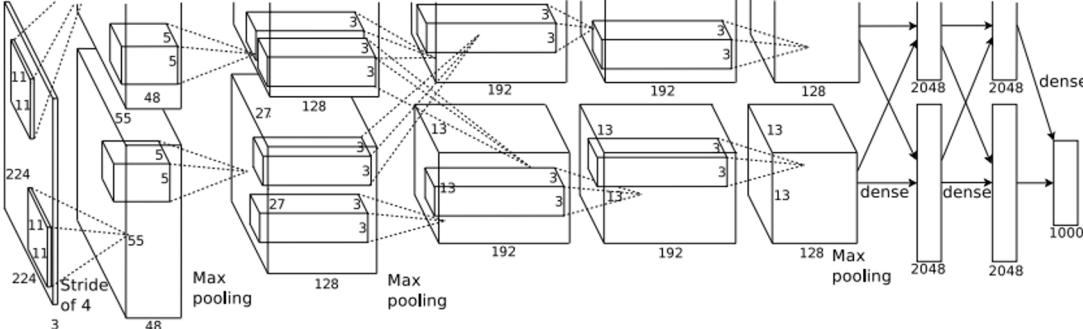


	Input size		Layer					Output size	
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	
conv1	3	227	64	11	4	2	64	?	

Recall: Output channels = number of filters

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

AlexNet

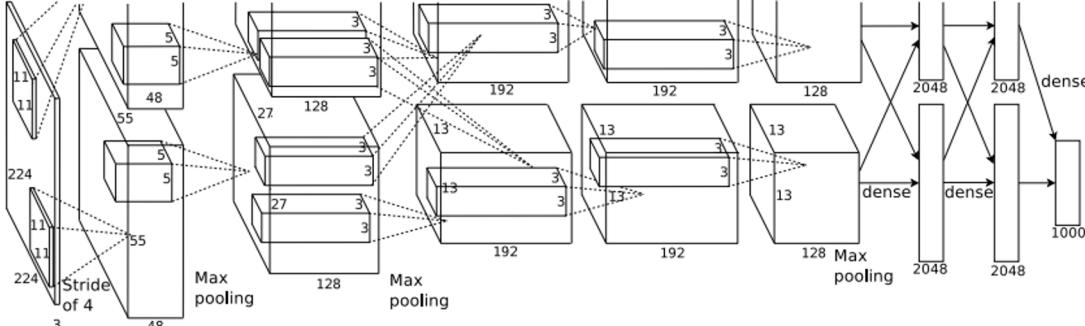


	Input size		Layer					Output size	
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	
conv1	3	227	64	11	4	2	64	56	

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

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

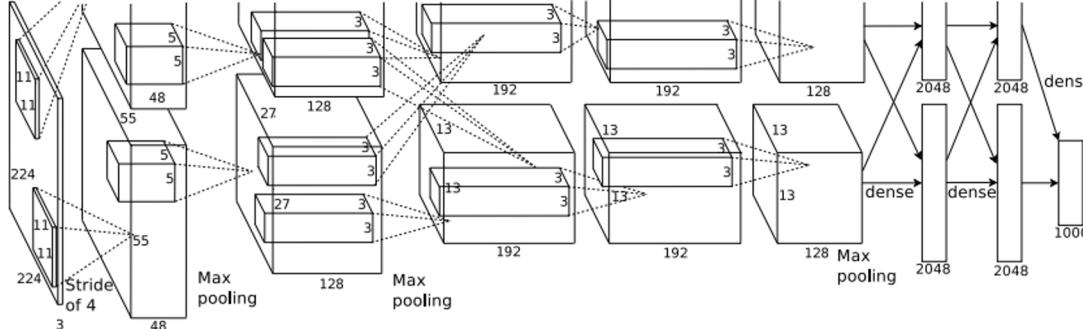
AlexNet



	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	

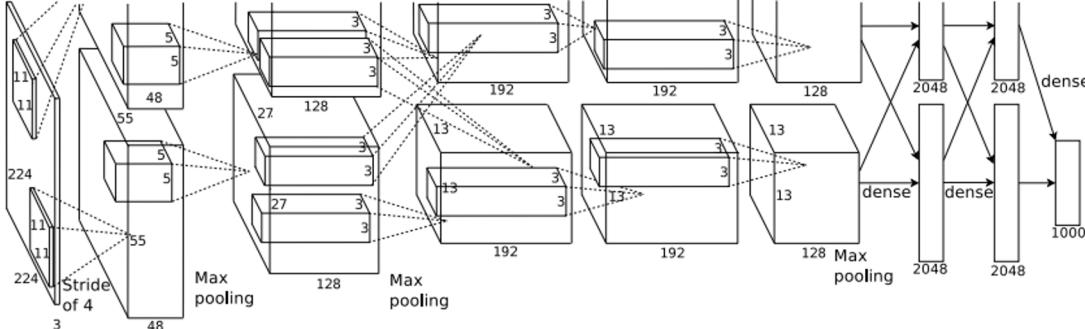
$$\begin{aligned}\text{Number of output elements} &= C * H' * W' \\ &= 64 * 56 * 56 = 200,704\end{aligned}$$

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

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

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

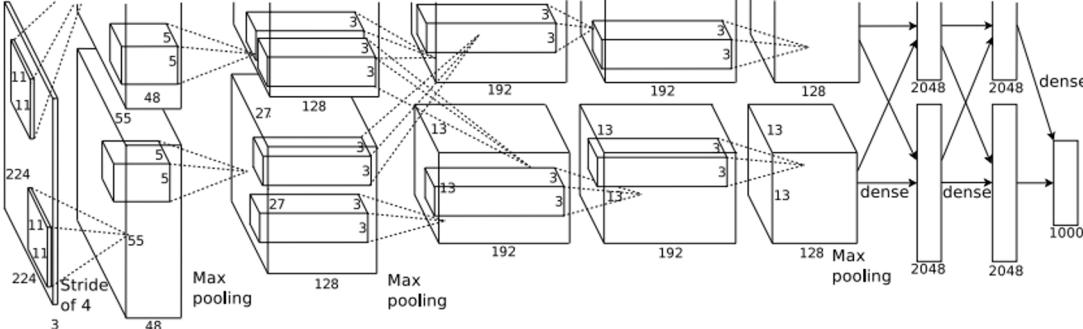
AlexNet



	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	

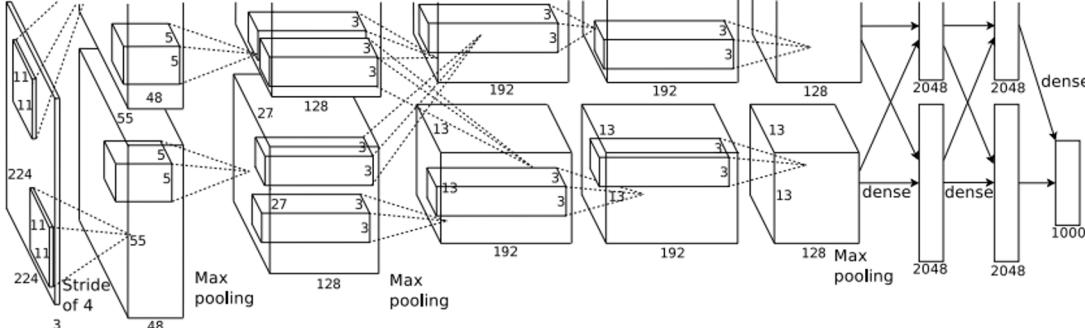
$$\begin{aligned}\text{Weight shape} &= C_{\text{out}} \times C_{\text{in}} \times K \times K \\ &= 64 \times 3 \times 11 \times 11\end{aligned}$$

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

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

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

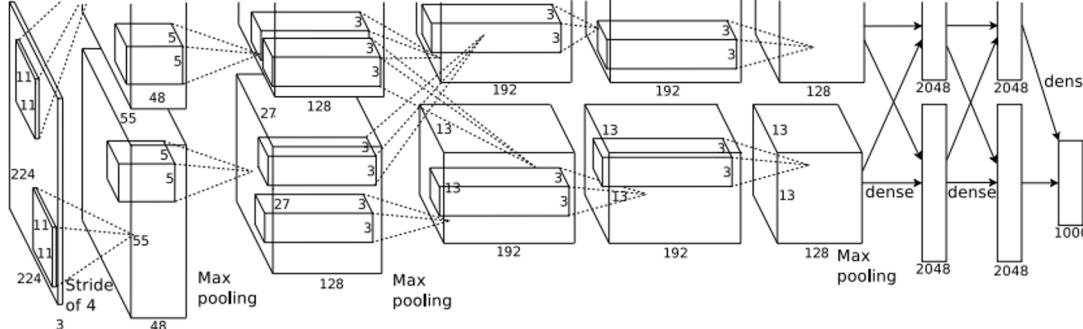
AlexNet



	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

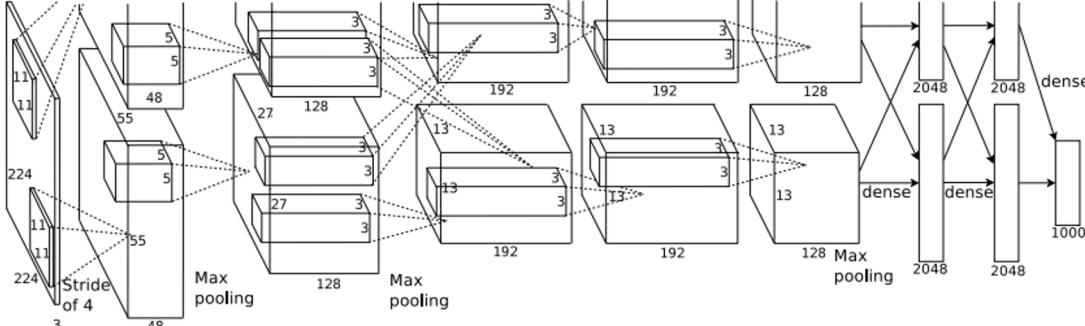


	Input size		Layer					Output size			
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73

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.

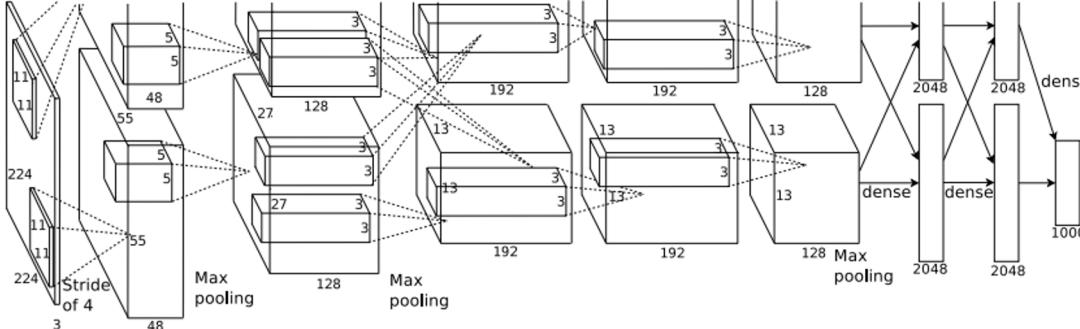
AlexNet



	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			

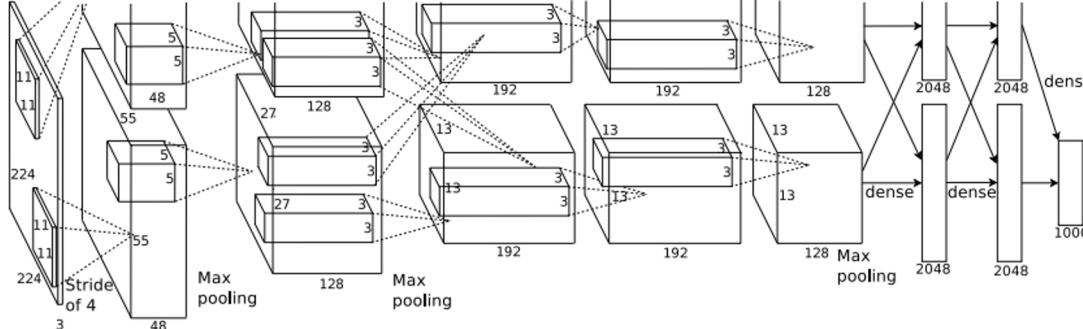
For pooling layer:

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

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

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	?		

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

$$\text{Bytes per elem} = 4$$

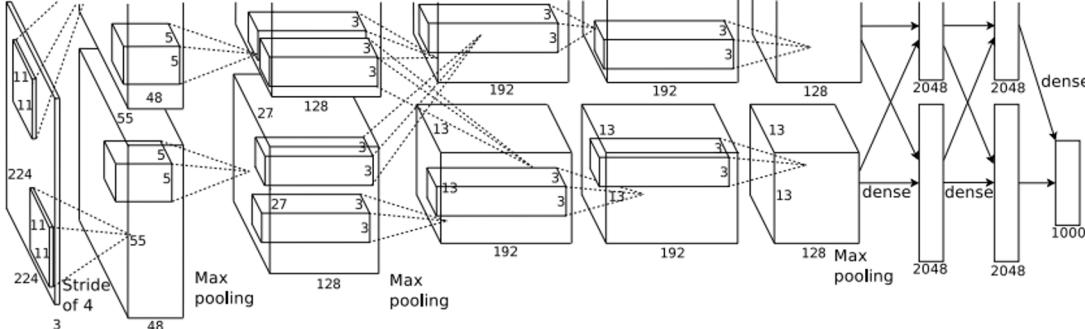
$$KB = C_{\text{out}} * H' * W' * 4 / 1024$$

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

$$= 182.25$$

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

AlexNet

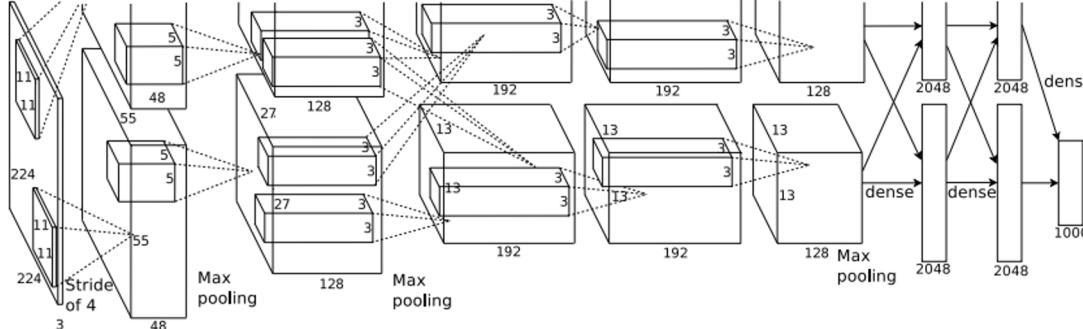


	Input size		Layer					Output size							
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)				
conv1	3	227	64	11	4	2	64	56	784	23	73				
pool1	64	56		3	2	0	64	27	182	0	?				

Pooling layers have no learnable parameters!

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

AlexNet



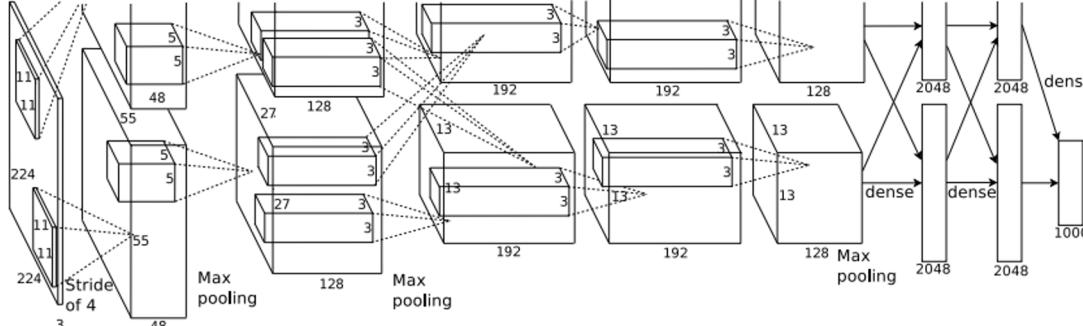
	Input size		Layer					Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)	
conv1	3	227	64	11	4	2	64	56	784	23	73	
pool1	64	56		3	2	0	64	27	182	0	0	

Floating-point ops for pooling layer

$$\begin{aligned}
 &= (\text{number of output positions}) * (\text{flops per output position}) \\
 &= (C_{\text{out}} * H' * W') * (K * K) \\
 &= (64 * 27 * 27) * (3 * 3) \\
 &= 419,904 \\
 &= \mathbf{0.4 \text{ MFLOP}}
 \end{aligned}$$

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

AlexNet

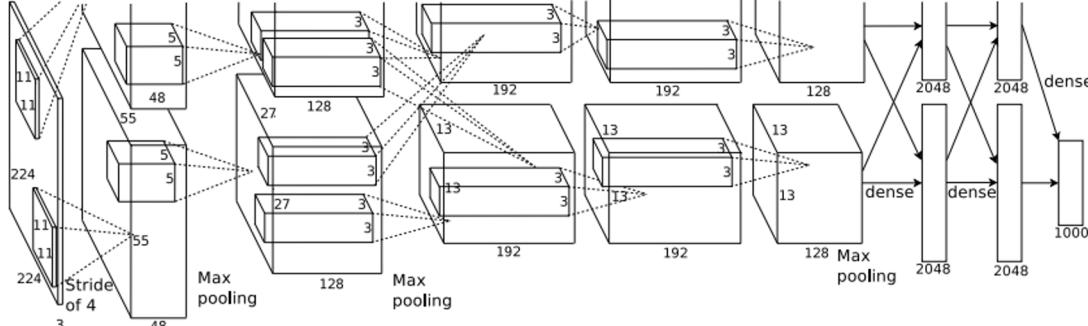


	Input size		Layer					Output size							
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)				
conv1	3	227	64	11	4	2	64	56	784	23	73				
pool1	64	56		3	2	0	64	27	182	0	0				
conv2	64	27	192	5	1	2	192	27	547	307	224				
pool2	192	27		3	2	0	192	13	127	0	0				
conv3	192	13	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				

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

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

AlexNet

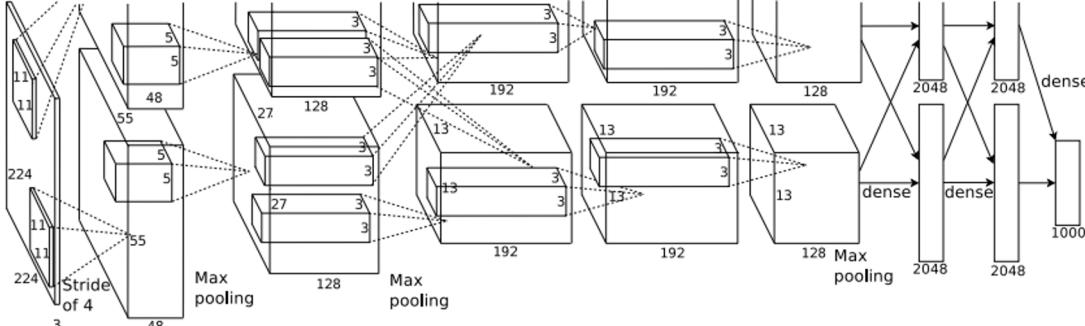


	Input size		Layer					Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)	
conv1	3	227	64	11	4	2	64	56	784	23	73	
pool1	64	56		3	2	0	64	27	182	0	0	
conv2	64	27	192	5	1	2	192	27	547	307	224	
pool2	192	27		3	2	0	192	13	127	0	0	
conv3	192	13	384	3	1	1	384	13	254	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	

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

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

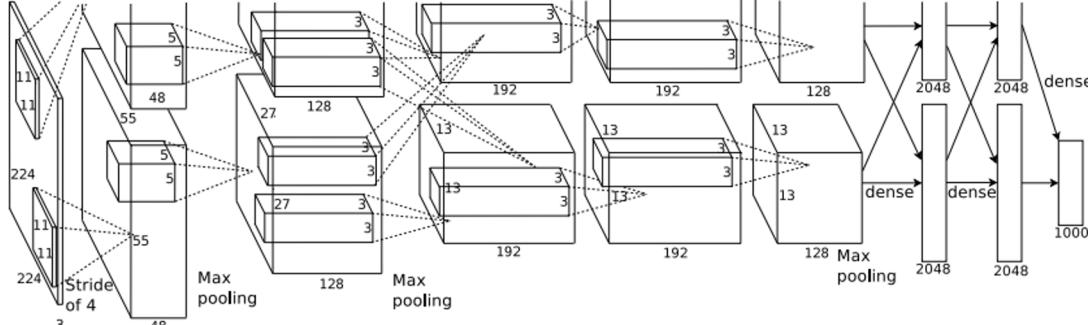
AlexNet



	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

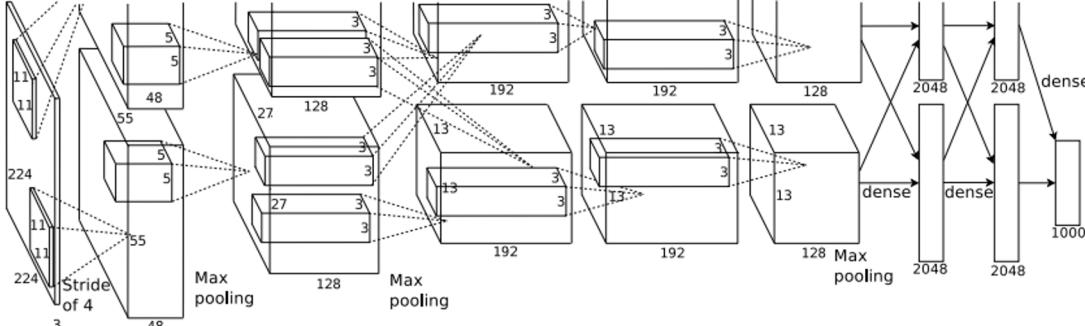
How to choose this?
Trial and error =(



	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

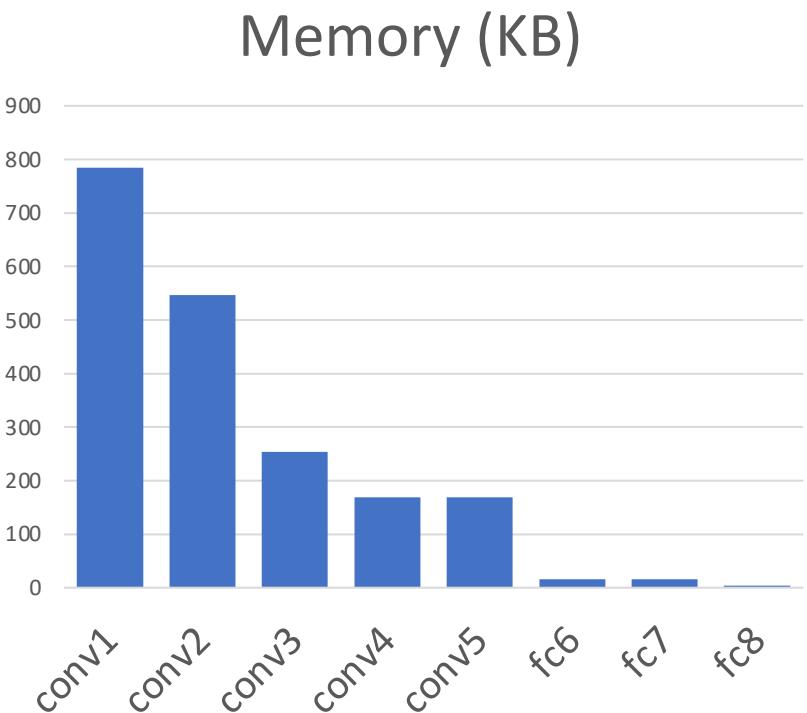
Interesting trends here!



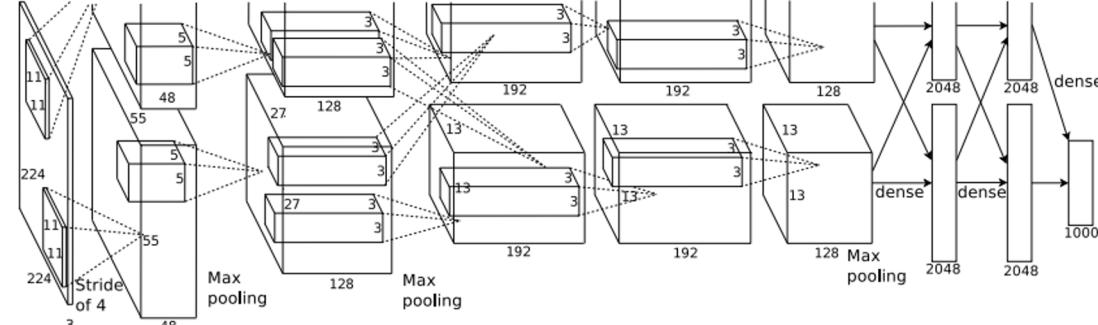
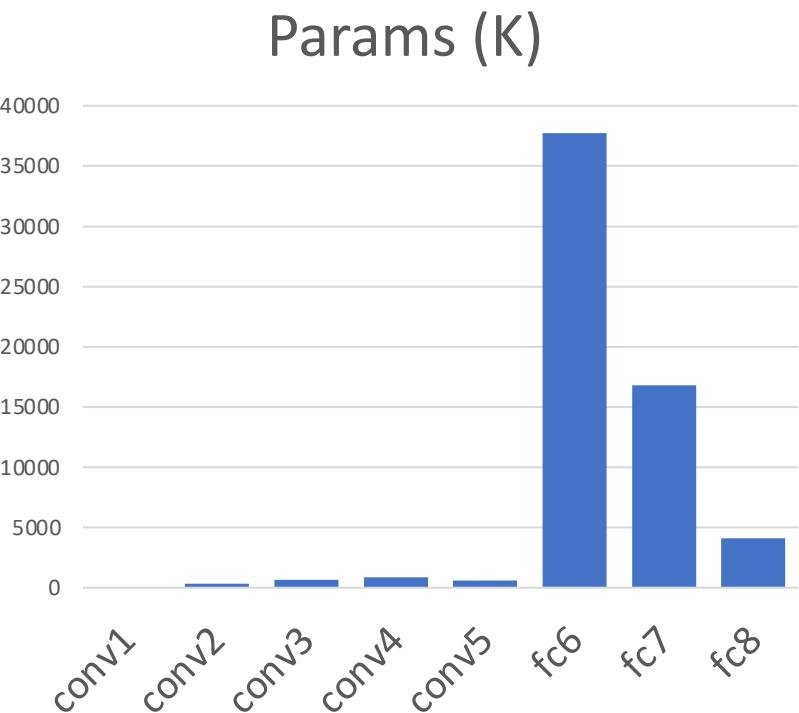
	Input size		Layer					Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)	
conv1	3	227	64	11	4	2	64	56	784	23	73	
pool1	64	56		3	2	0	64	27	182	0	0	
conv2	64	27	192	5	1	2	192	27	547	307	224	
pool2	192	27		3	2	0	192	13	127	0	0	
conv3	192	13	384	3	1	1	384	13	254	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

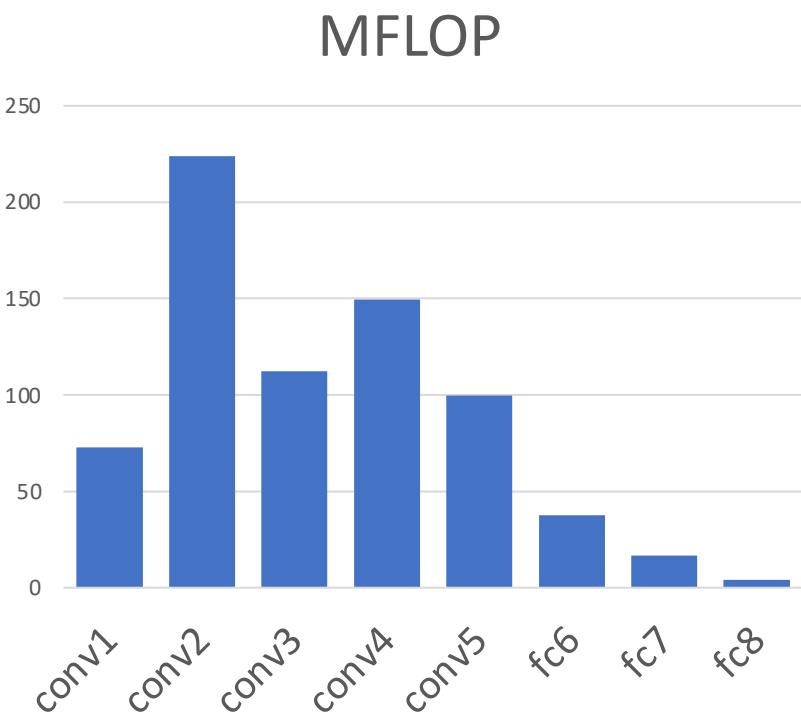
Most of the **memory usage** is in the early convolution layers



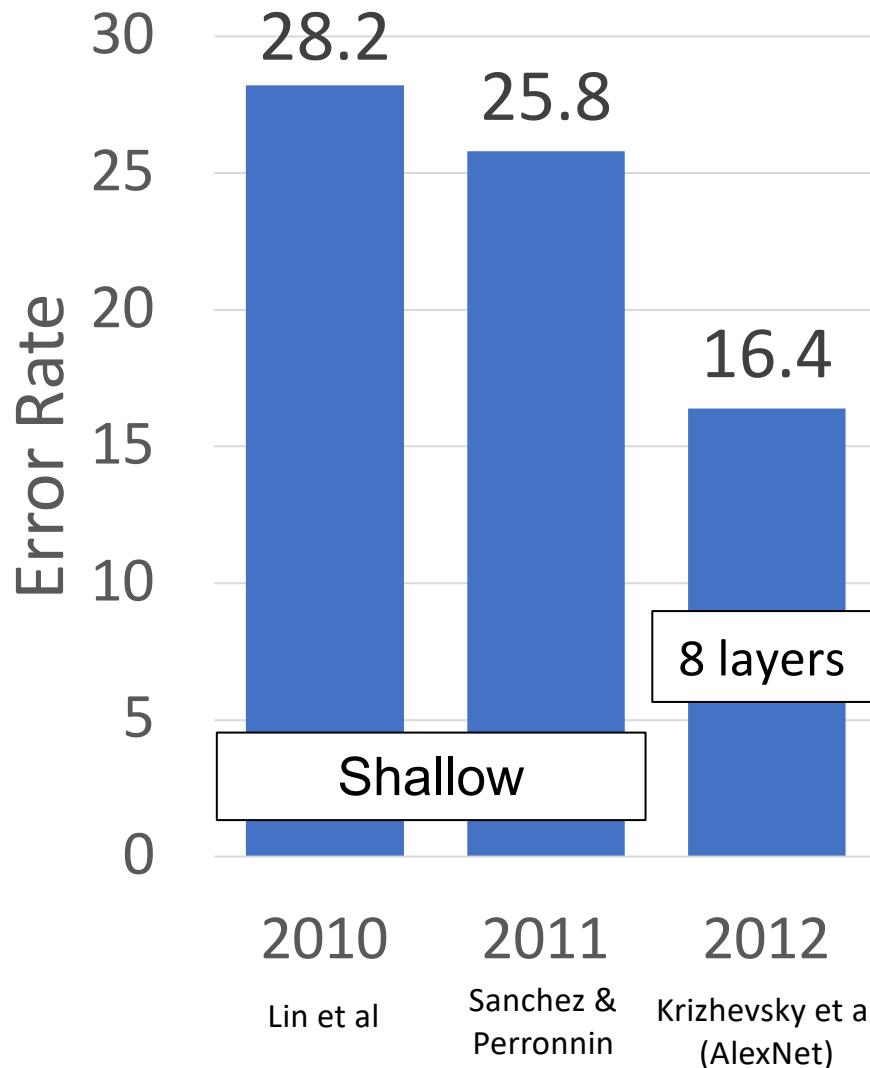
Nearly all **parameters** are in the fully-connected layers



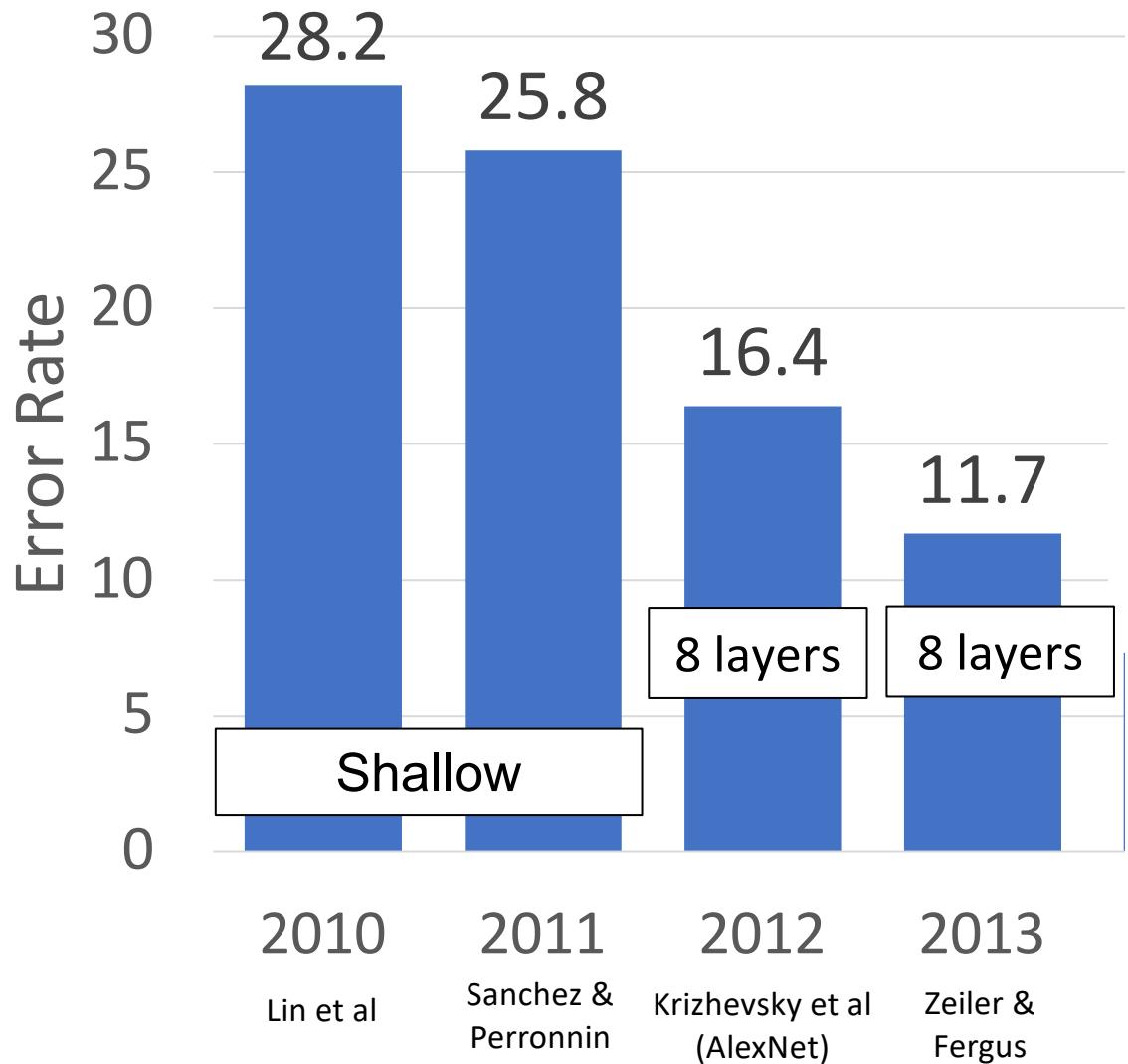
Most **floating-point ops** occur in the convolution layers



ImageNet Classification Challenge

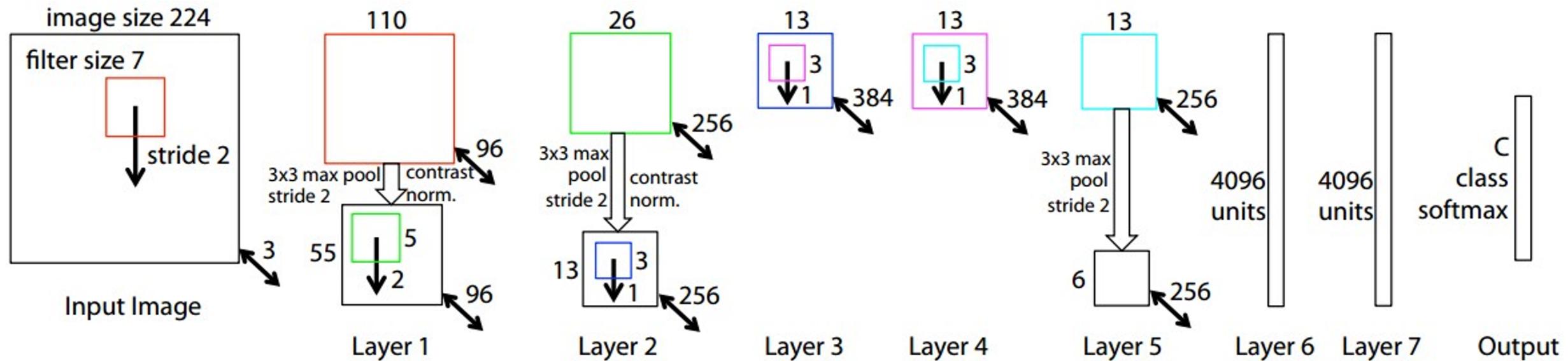


ImageNet Classification Challenge



ZFNet: A Bigger AlexNet

ImageNet top 5 error: 16.4% → 11.7%



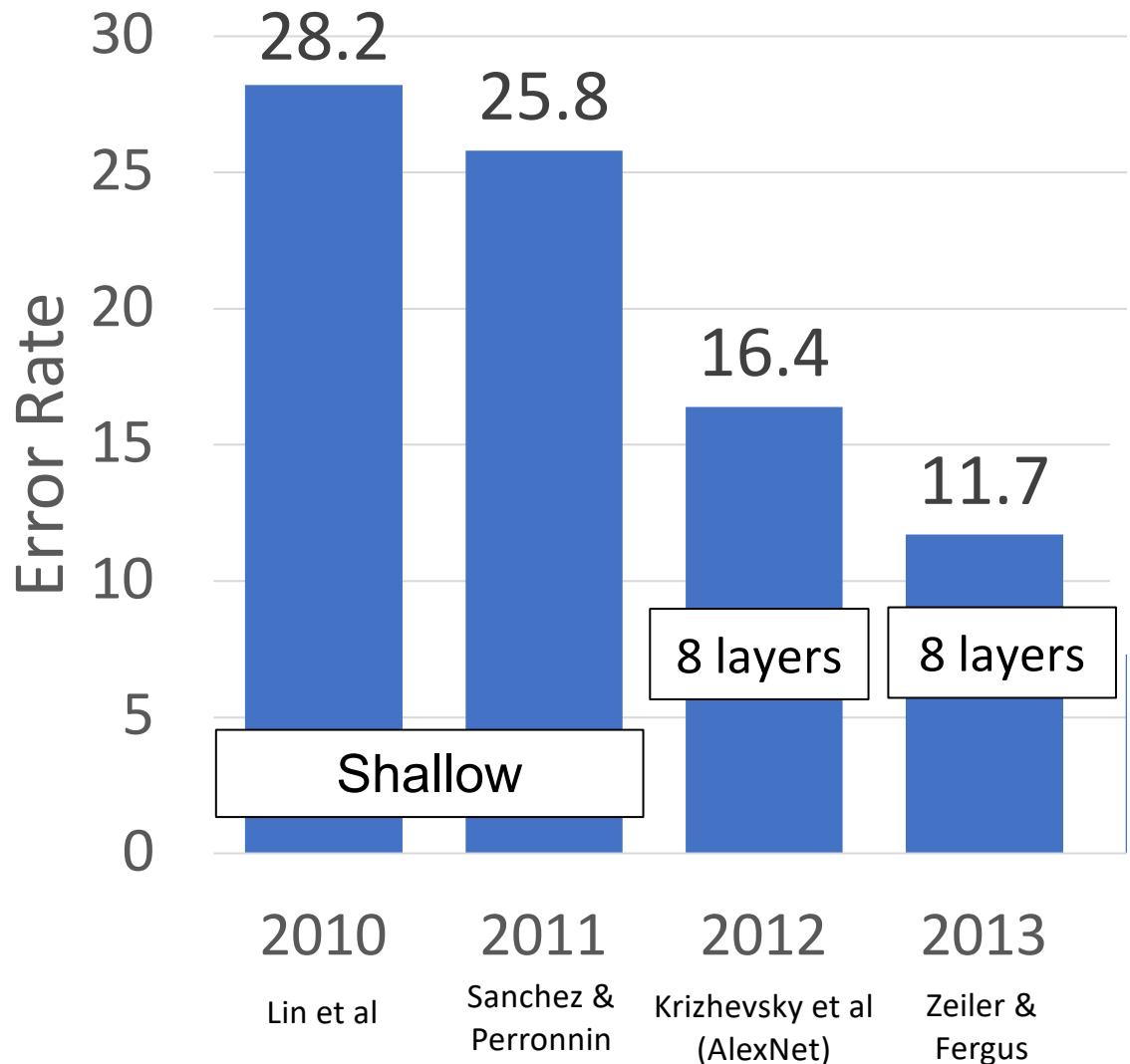
AlexNet but:

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

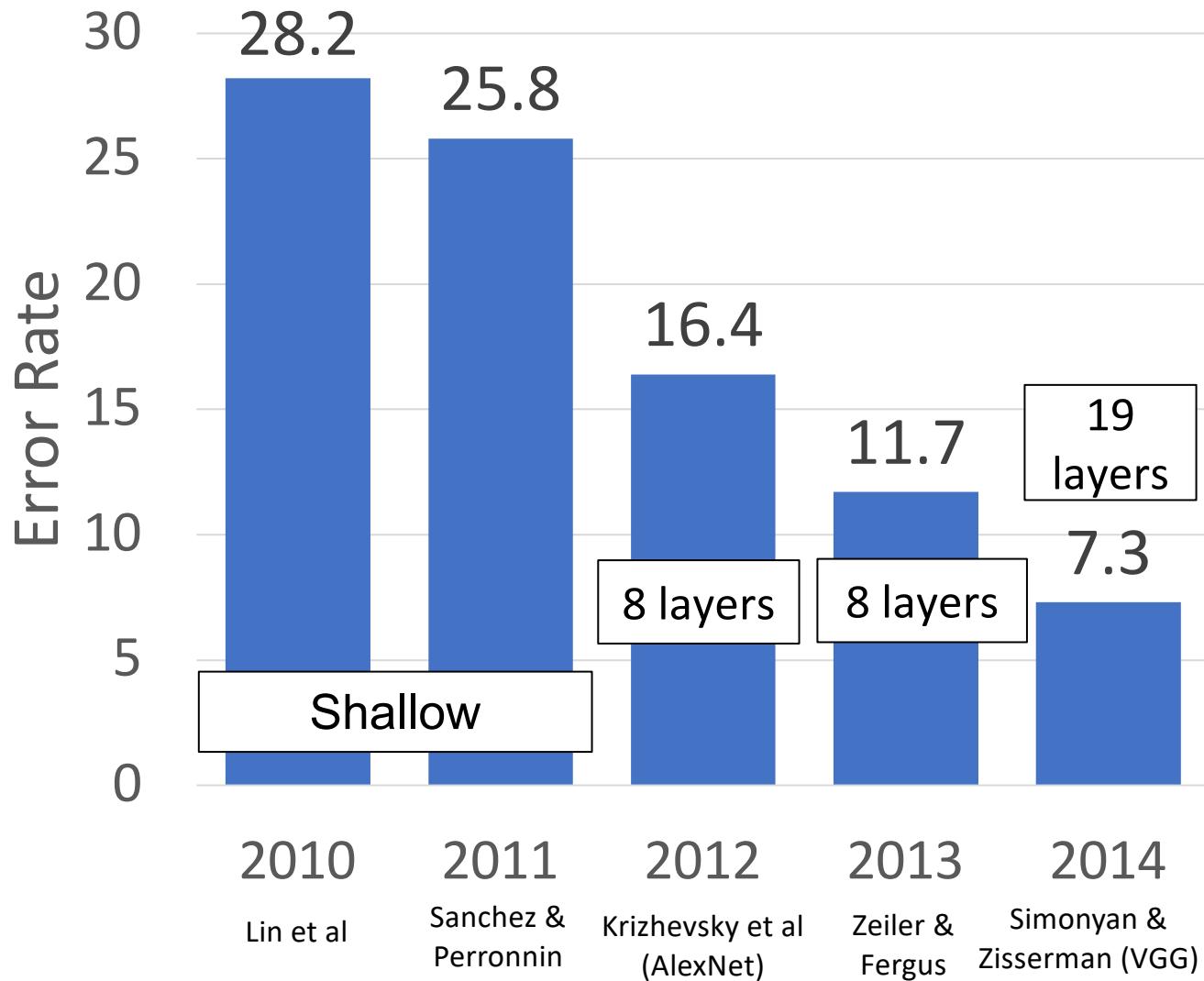
CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512

More trial and error = (

ImageNet Classification Challenge



ImageNet Classification Challenge



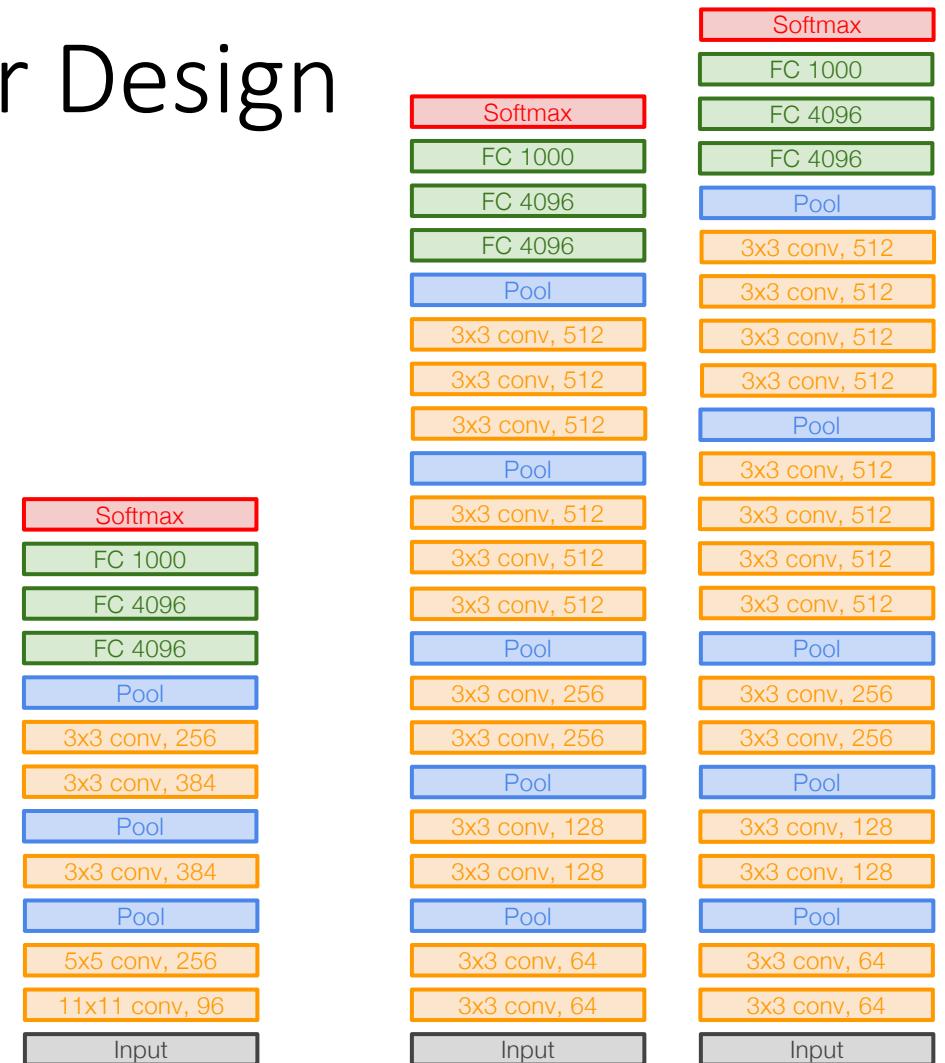
VGG: Deeper Networks, Regular Design

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

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

Network has 5 convolutional **stages**:

Stage 1: conv-conv-pool

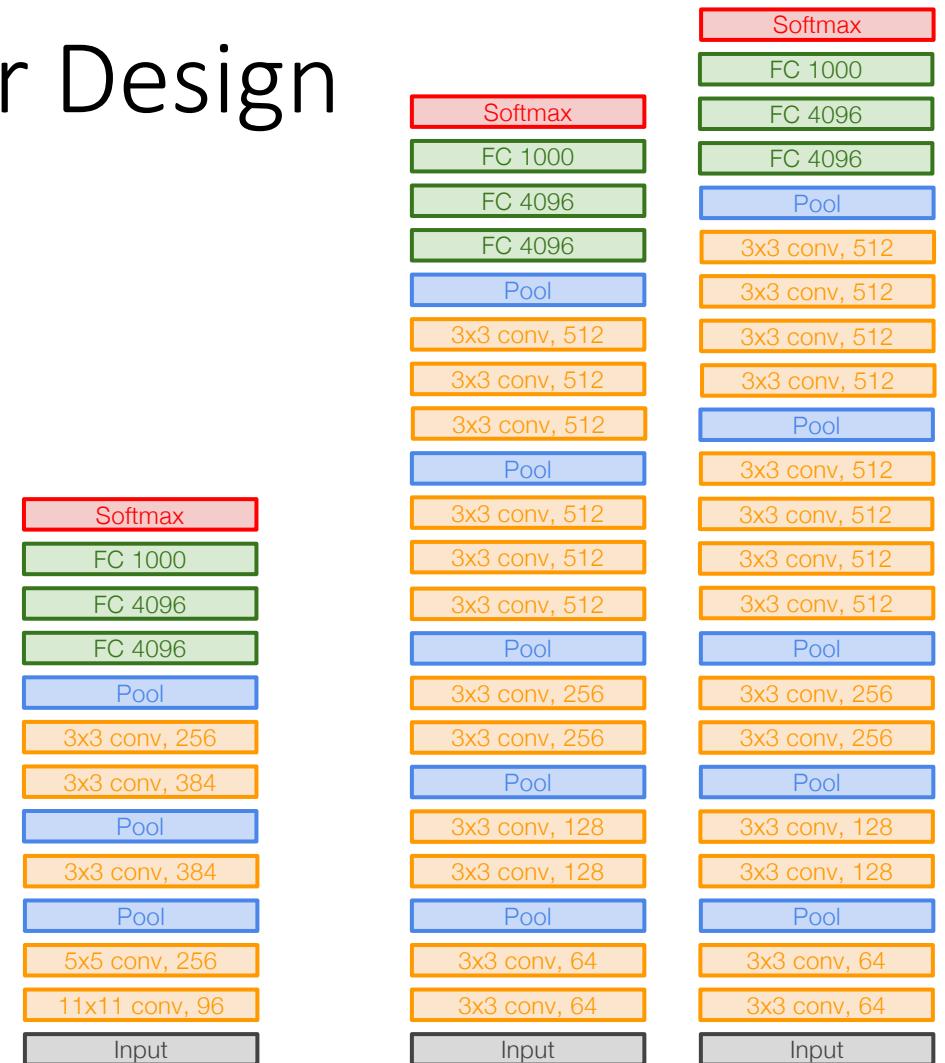
Stage 2: conv-conv-pool

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

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

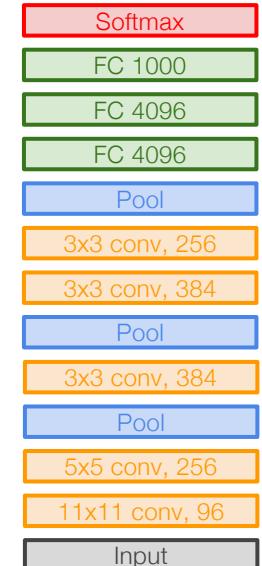
After pool, double #channels

Option 1:

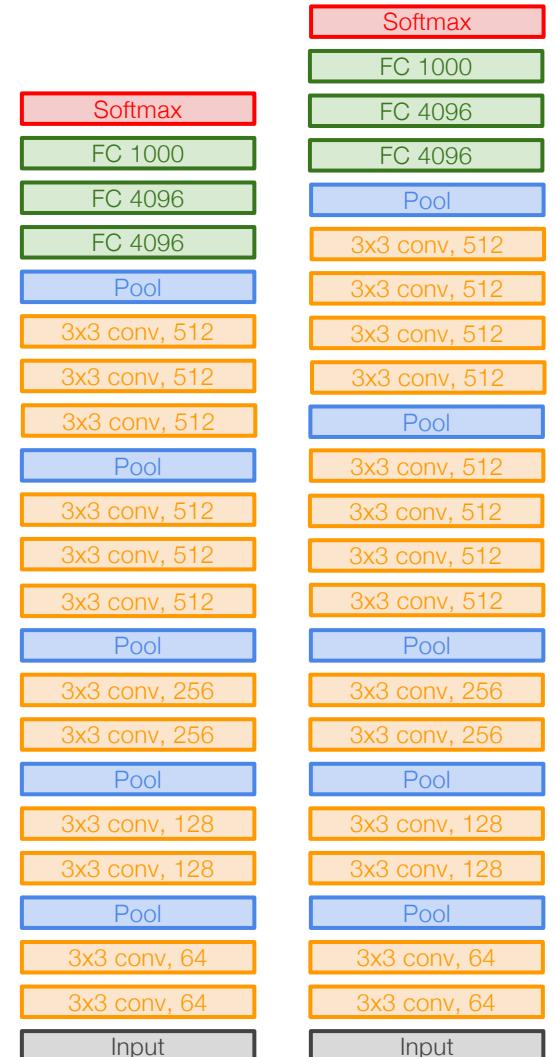
Conv(5x5, C -> C)

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: $25C^2$

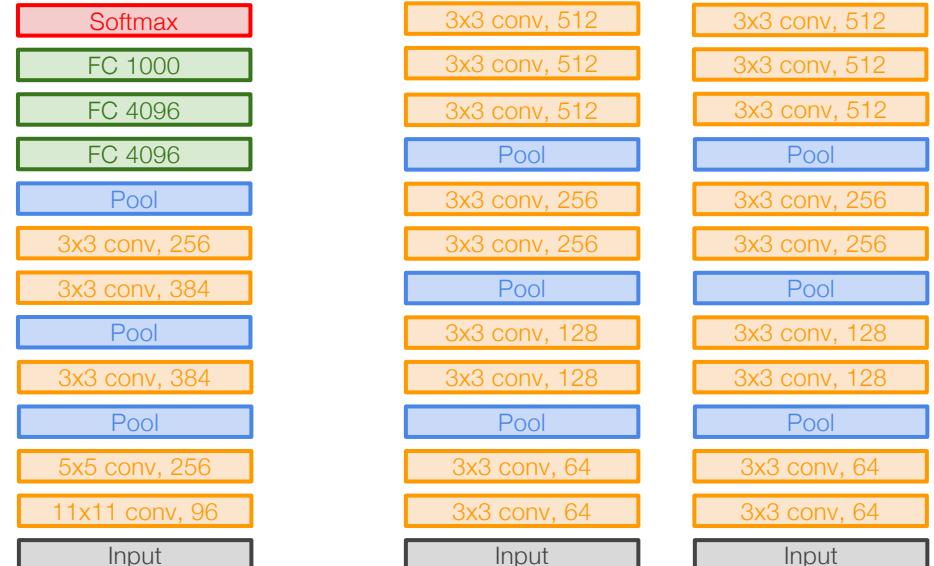
FLOPs: $25C^2HW$

Option 2:

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

Params: $18C^2$

FLOPs: $18C^2HW$



AlexNet

VGG16

VGG19

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

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: $25C^2$

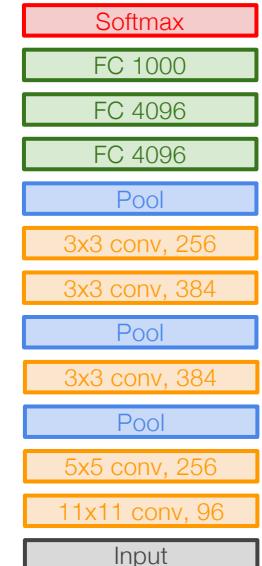
FLOPs: $25C^2HW$

Option 2:

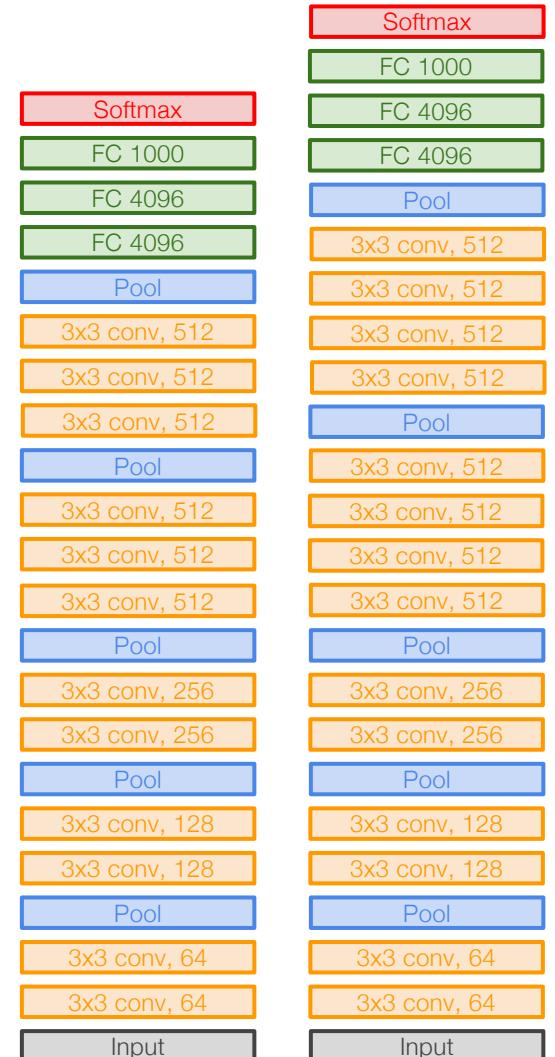
Conv(3x3, C -> C)
Conv(3x3, C -> C)

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

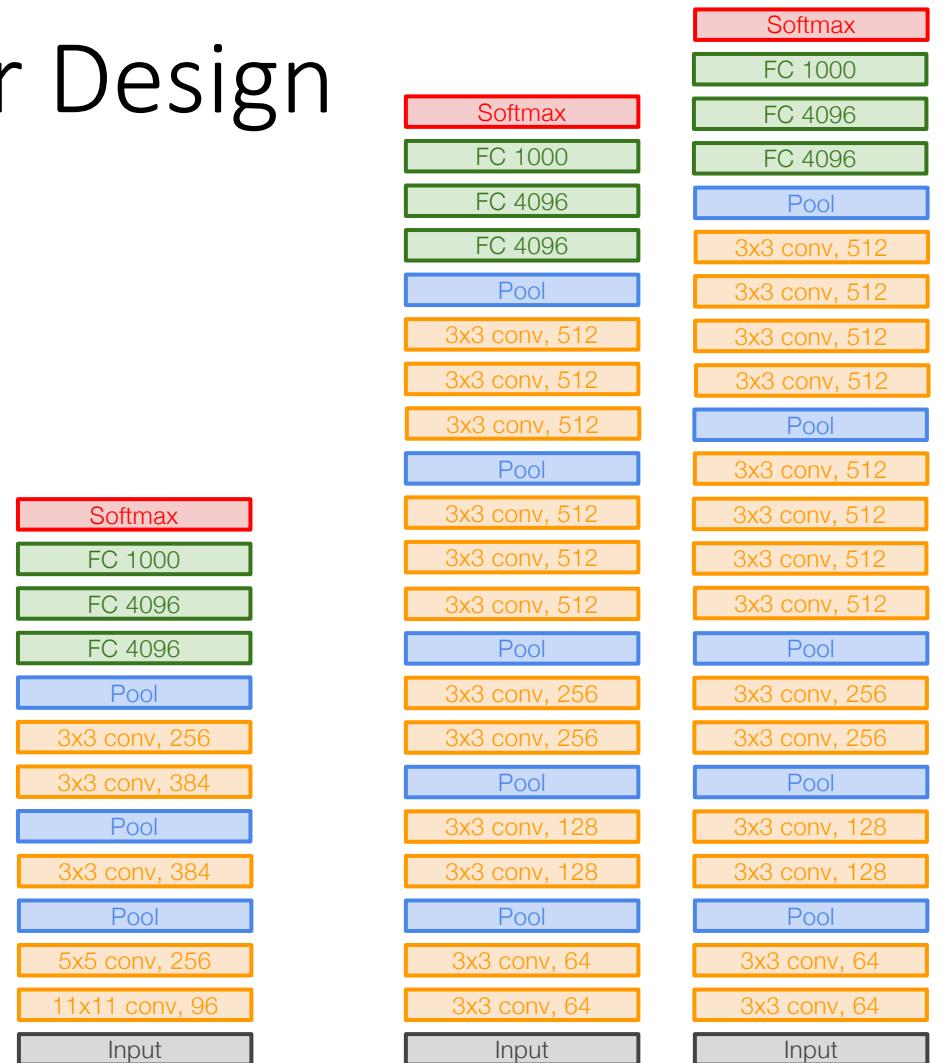
Input: $C \times 2H \times 2W$

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

Memory: 4HWC

Params: $9C^2$

FLOPs: $36HWC^2$



AlexNet

VGG16

VGG19

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 x 2H x 2W

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

Memory: 4HWC

Params: $9C^2$

FLOPs: 36HWC^2

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

Conv(3x3, 2C -> 2C)

Memory: 2HWC

Params: 36C²

FLOPs: $36HWC^2$

Diagram illustrating a neural network design, showing two parallel architectures:

- Left Column (Input to Softmax):**
 - Input
 - 11x11 conv, 96
 - 5x5 conv, 256
 - Pool
 - 3x3 conv, 384
 - Pool
 - 3x3 conv, 384
 - Pool
 - FC 1000
 - Softmax
- Right Column (Input to Softmax):**
 - Input
 - 3x3 conv, 64
 - 3x3 conv, 64
 - 3x3 conv, 64
 - Pool
 - 3x3 conv, 512
 - 3x3 conv, 512
 - 3x3 conv, 512
 - Pool
 - 3x3 conv, 512
 - 3x3 conv, 512
 - 3x3 conv, 512
 - Pool
 - 3x3 conv, 512
 - 3x3 conv, 512
 - 3x3 conv, 512
 - FC 4096
 - FC 4096
 - FC 4096
 - Softmax

AlexNet

VGG16

VGG19

Simonyan and Zisserman, "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 x 2H x 2W

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

Memory: 4HWC

Params: $9C^2$

FLOPs: 36HWC^2

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

Conv(3x3, 2C -> 2C)

Memory: 2HWC

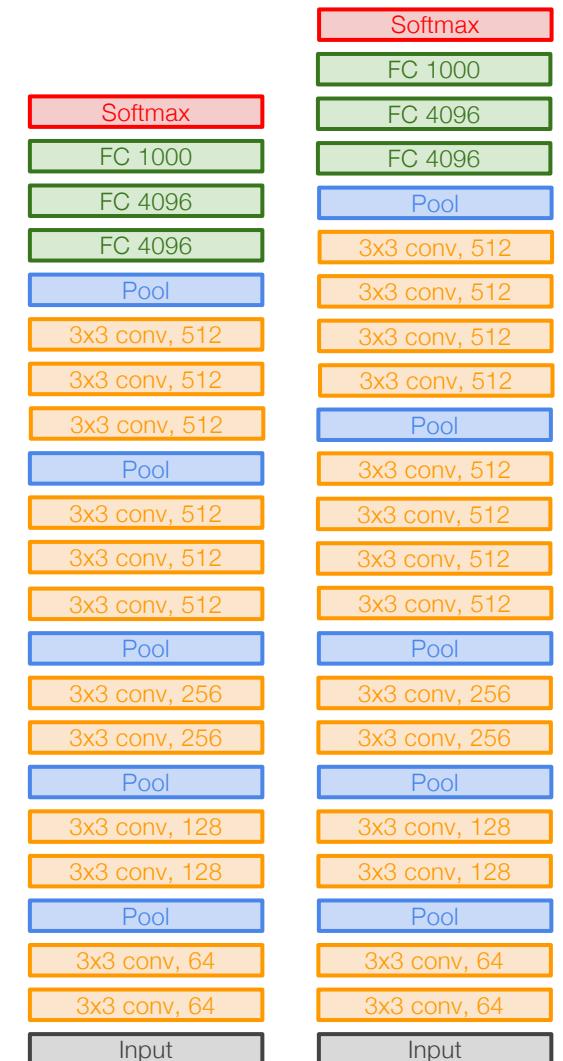
Params: $36C^2$

FLOPs: $36HWC^2$

AlexNet

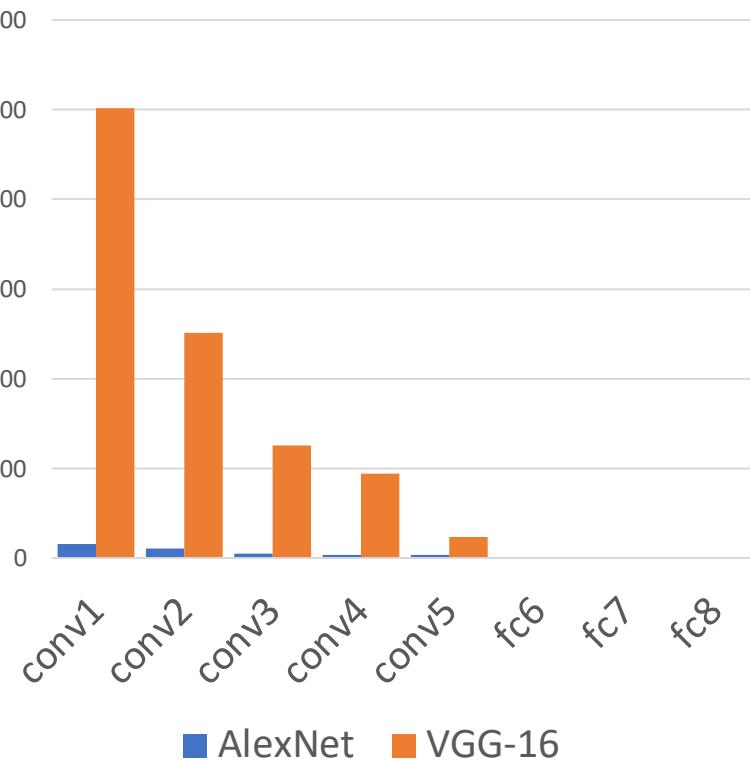
VGG16

VGG19

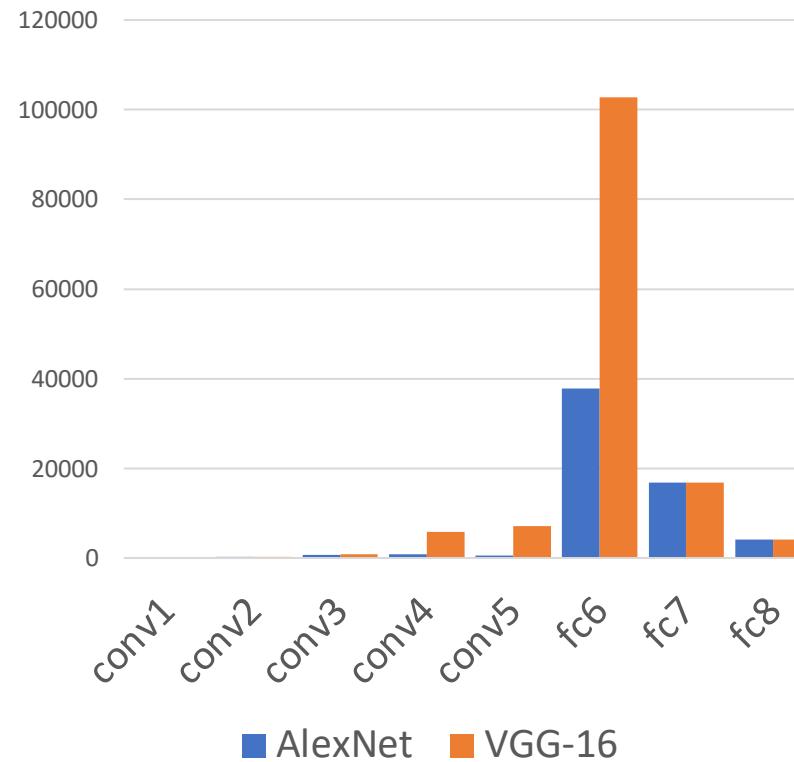


AlexNet vs VGG-16: Much bigger network!

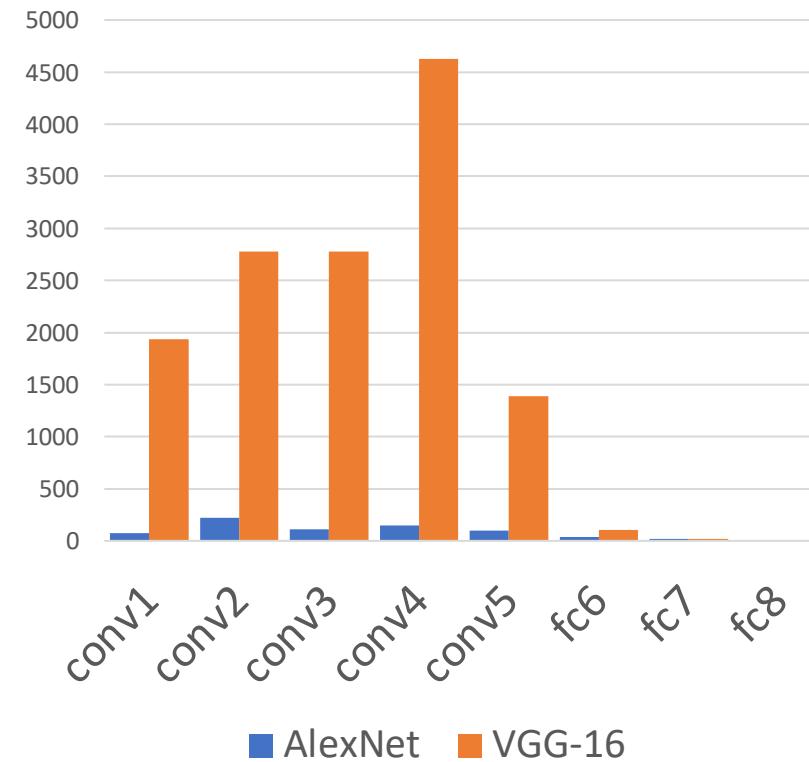
AlexNet vs VGG-16
(Memory, KB)



AlexNet vs VGG-16
(Params, M)



AlexNet vs VGG-16
(MFLOPs)



AlexNet total: 1.9 MB

VGG-16 total: 48.6 MB (25x)

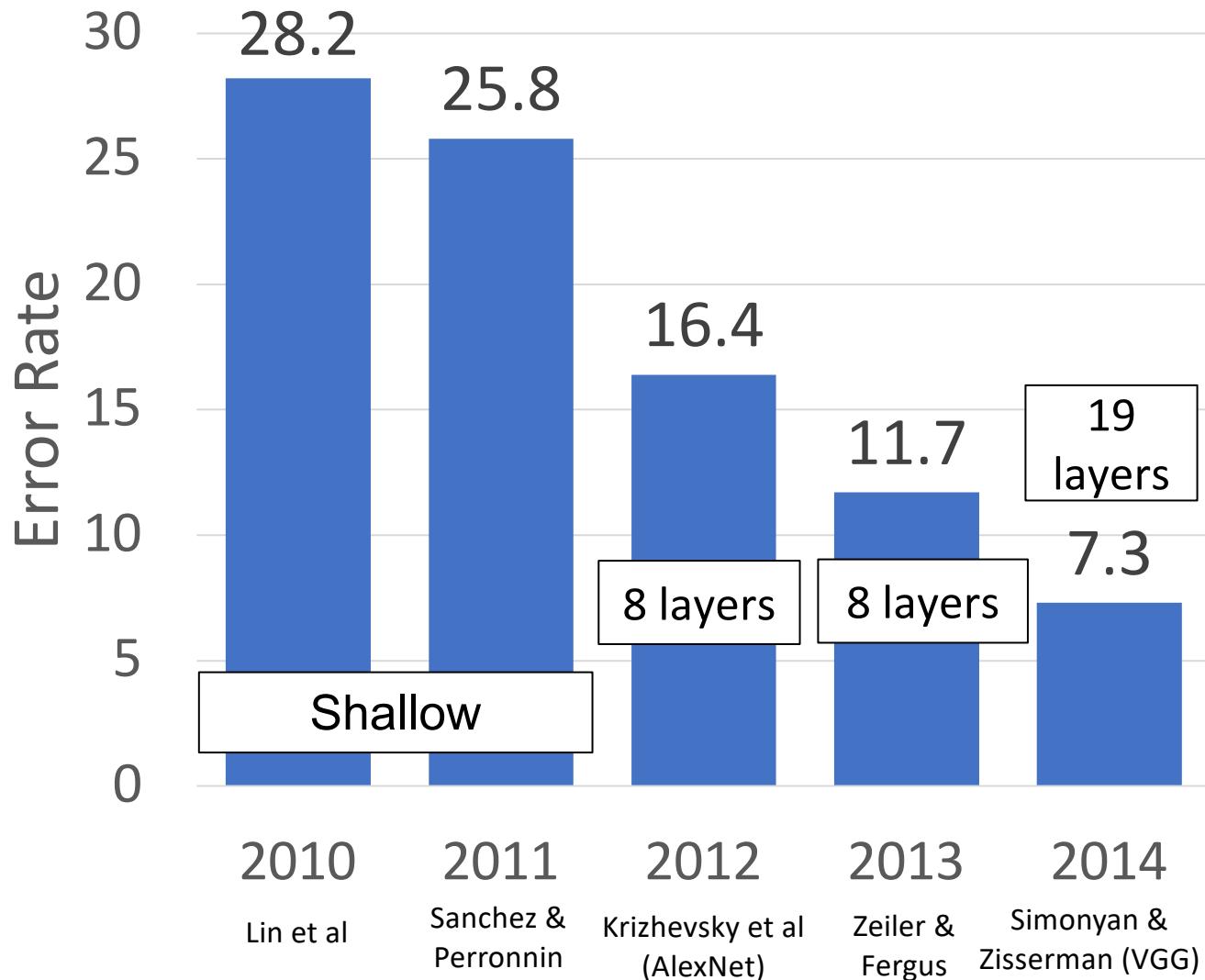
AlexNet total: 61M

VGG-16 total: 138M (2.3x)

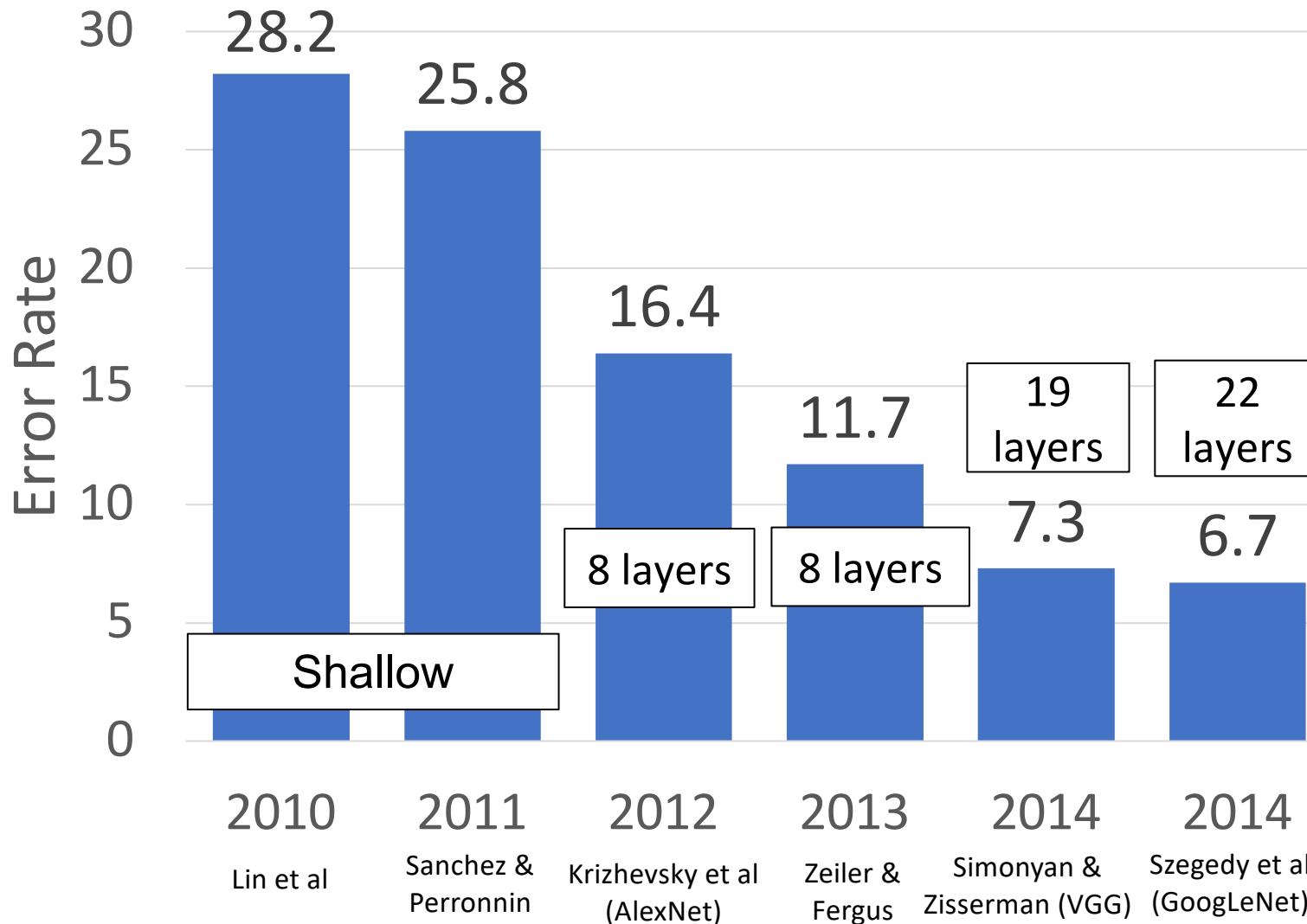
AlexNet total: 0.7 GFLOP

VGG-16 total: 13.6 GFLOP (19.4x)

ImageNet Classification Challenge

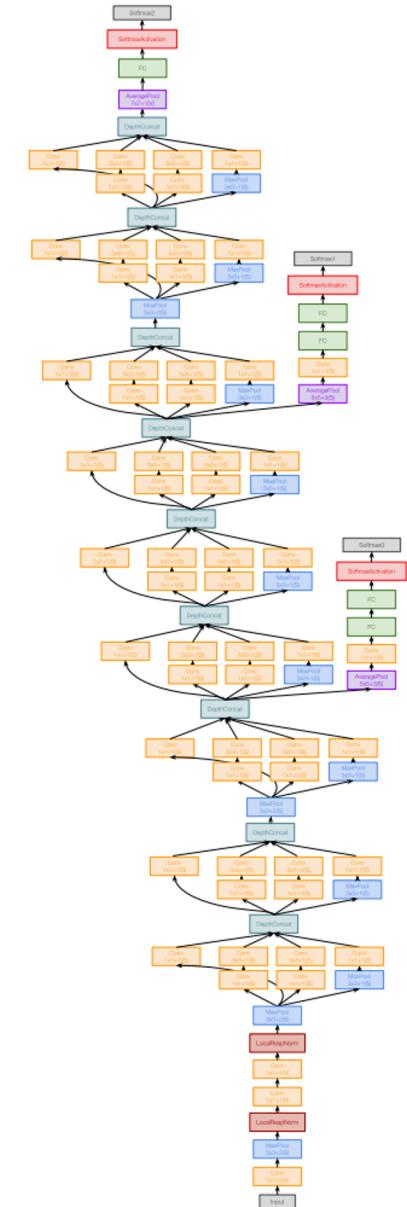


ImageNet Classification Challenge



GoogLeNet: Focus on Efficiency

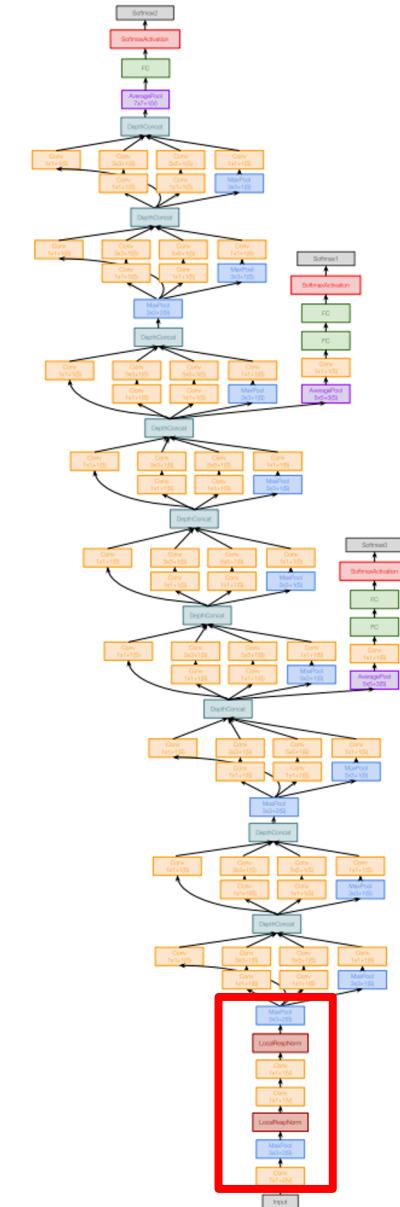
Many innovations for efficiency: reduce parameter count, memory usage, and computation



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)



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)

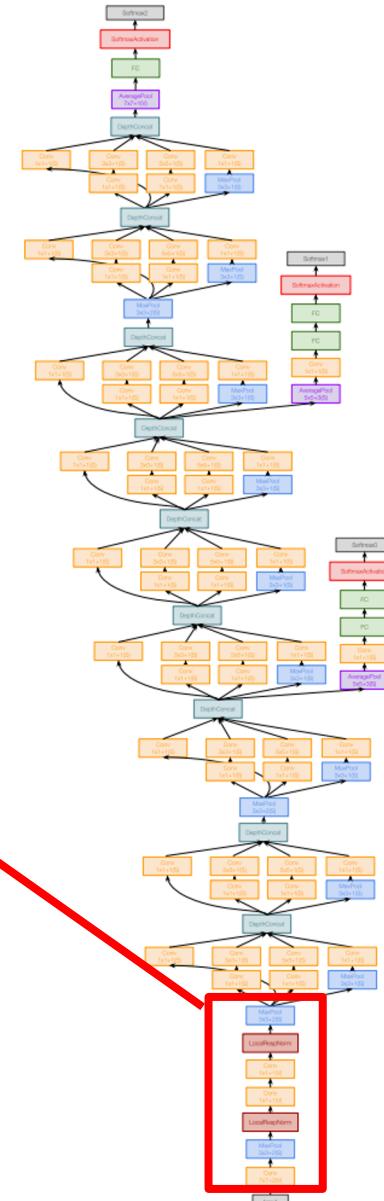
Layer	Input size		Layer				Output size		memory (KB)	params (K)	flop (M)
	C	H / W	filters	kernel	stride	pad	C	H/W			
conv	3	224	64	7	2	3	64	112	3136	9	118
max-pool	64	112		3	2	1	64	56	784	0	2
conv	64	56	64	1	1	0	64	56	784	4	13
conv	64	56	192	3	1	1	192	56	2352	111	347
max-pool	192	56		3	2	1	192	28	588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

Total from 224 to 28 spatial resolution:

Memory: 7.5 MB

Params: 124K

MFLOP: 418



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

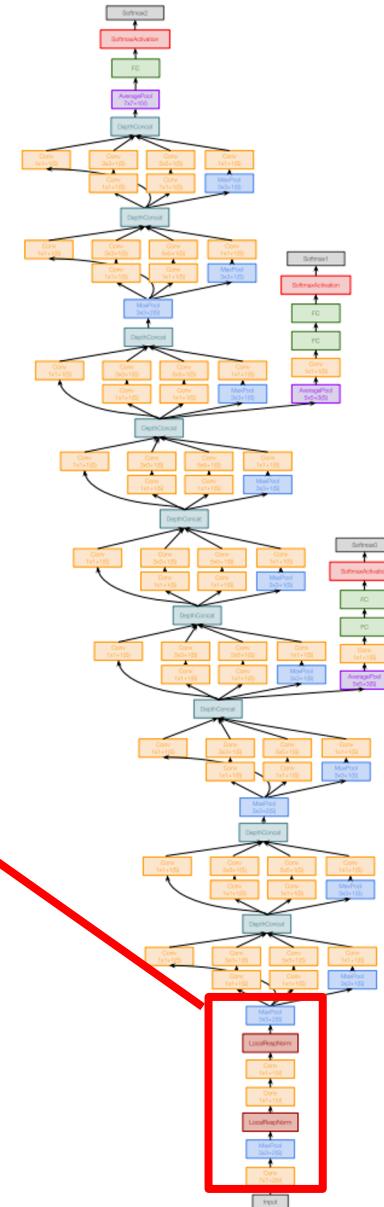
GoogLeNet: Aggressive Stem

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

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

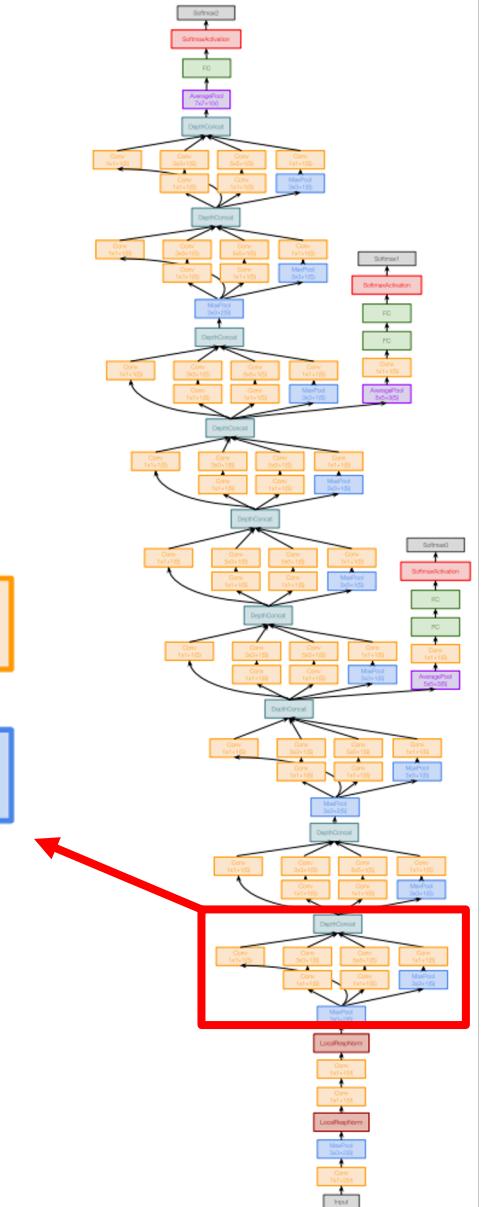
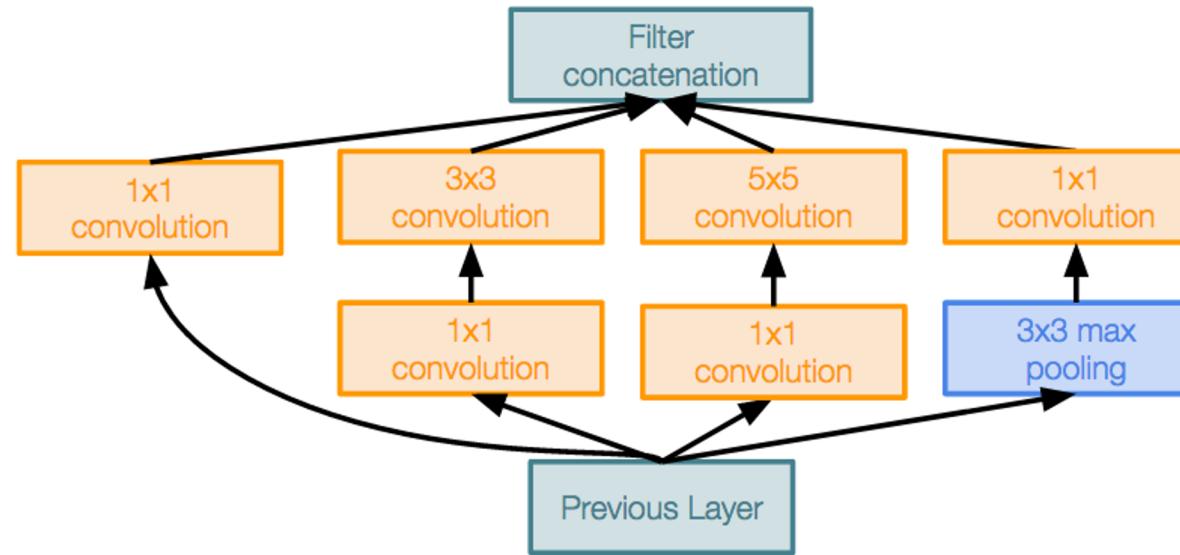


GoogLeNet: Inception Module

Inception module

Local unit with parallel branches

Local structure repeated many times throughout the network



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

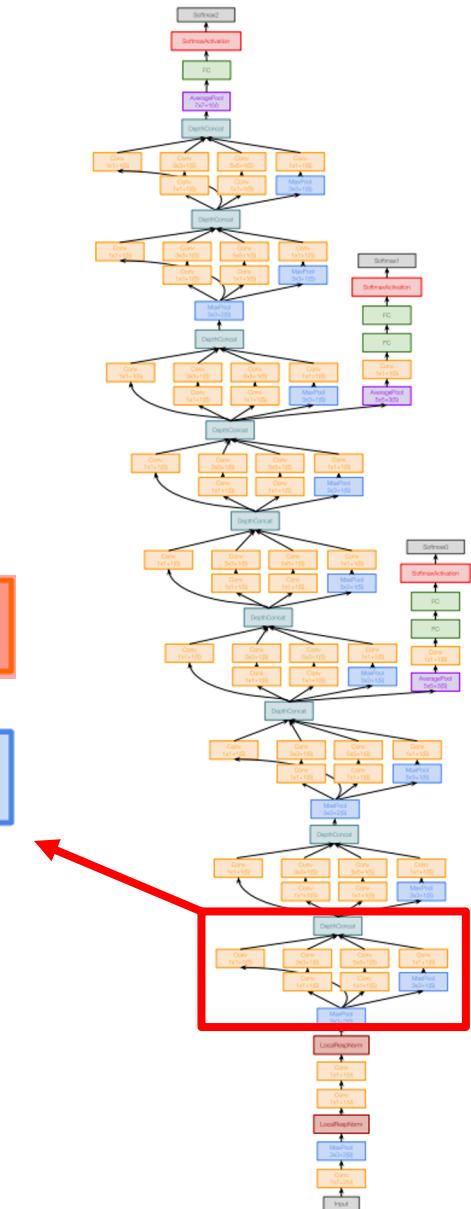
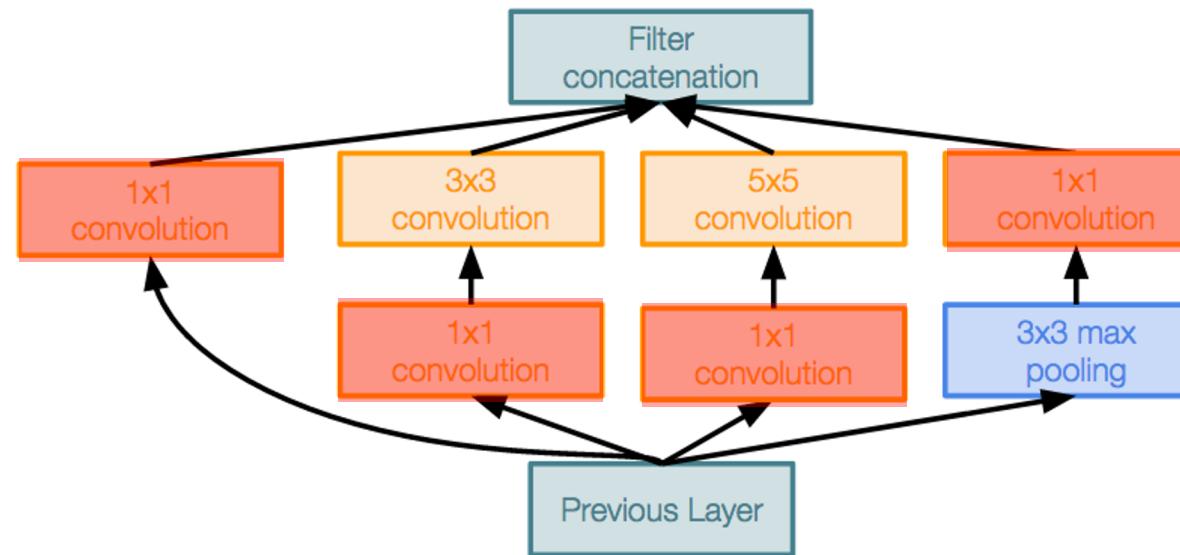
GoogLeNet: Inception Module

Inception module

Local unit with parallel branches

Local structure repeated many times throughout the network

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

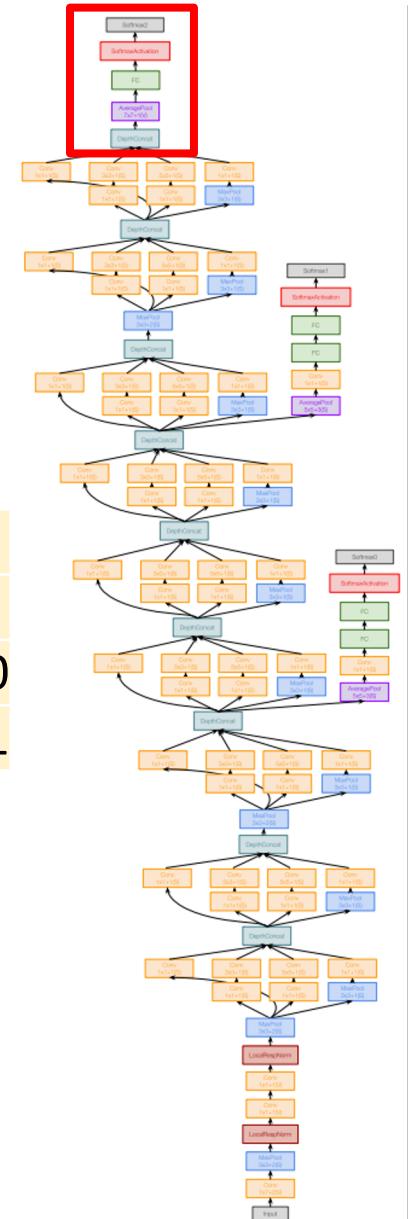


Szegedy et al, "Going deeper with convolutions", CVPR 2015

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



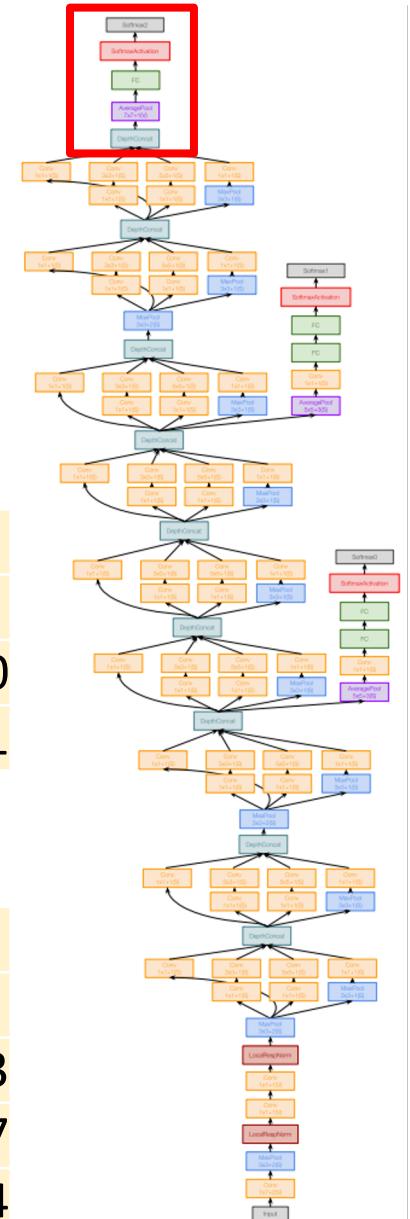
GoogLeNet: Global Average Pooling

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!)

	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

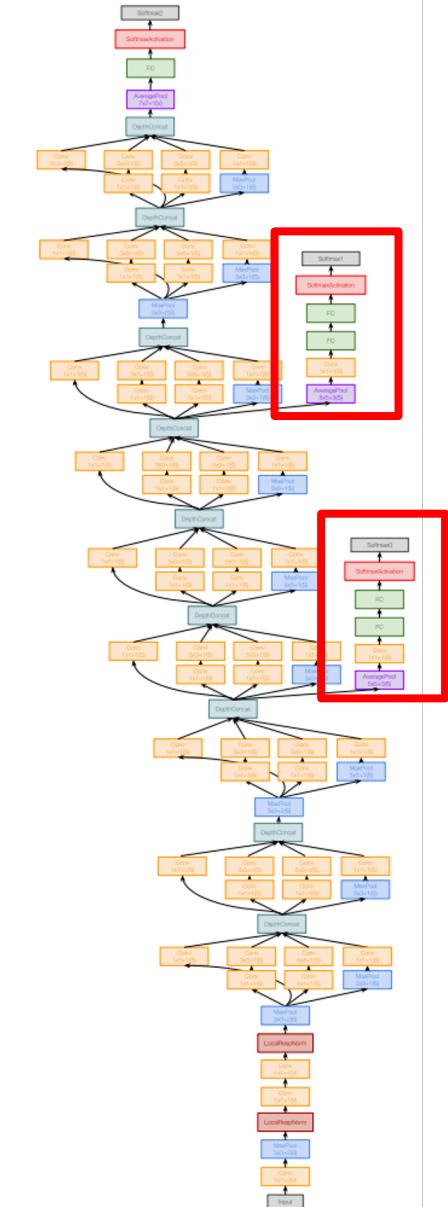


GoogLeNet: Auxiliary Classifiers

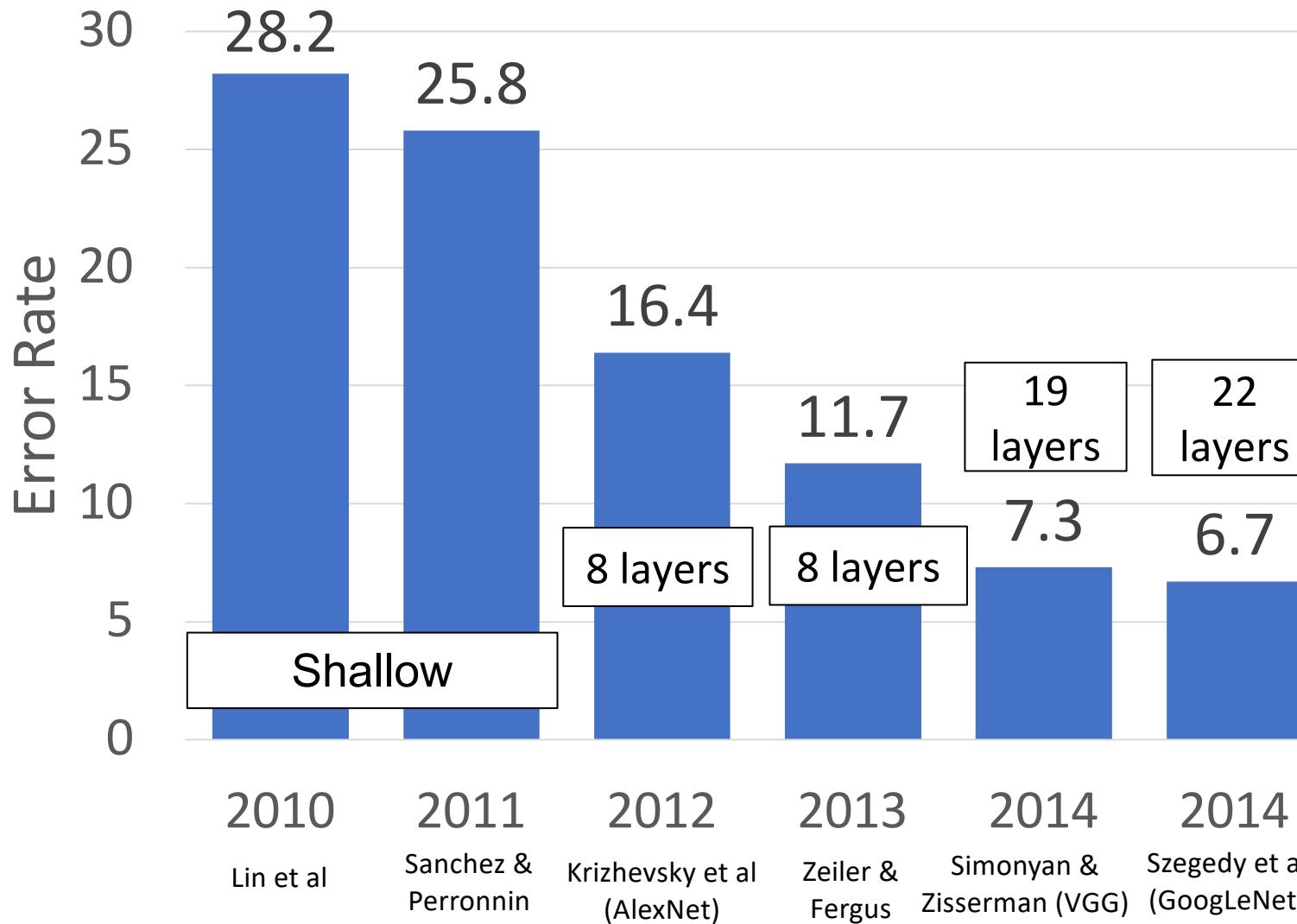
Training using loss at the end of the network didn't work well:
Network is too deep, gradients don't propagate cleanly

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

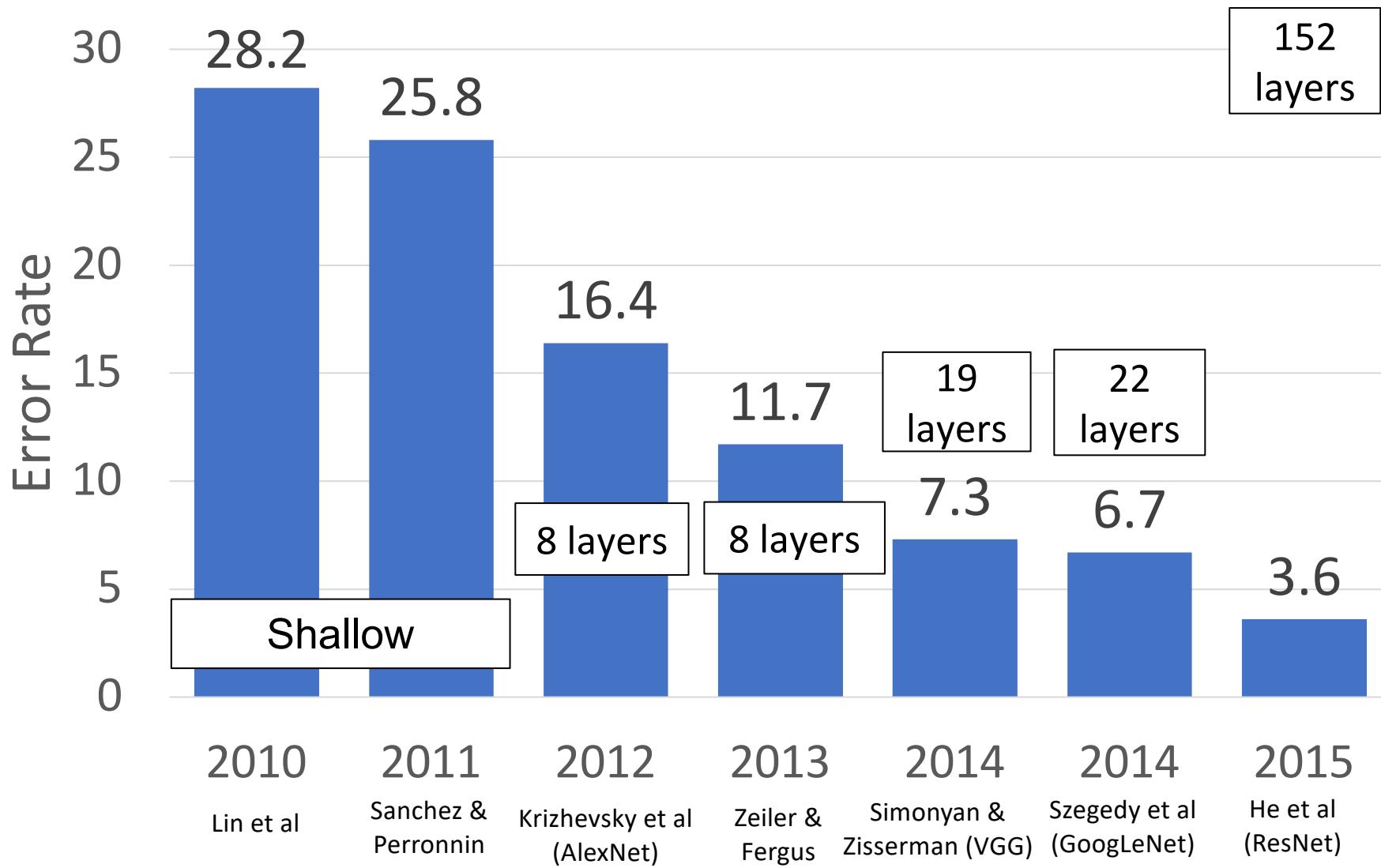
GoogLeNet was before batch normalization! With BatchNorm no longer need to use this trick



ImageNet Classification Challenge



ImageNet Classification Challenge



Residual Networks

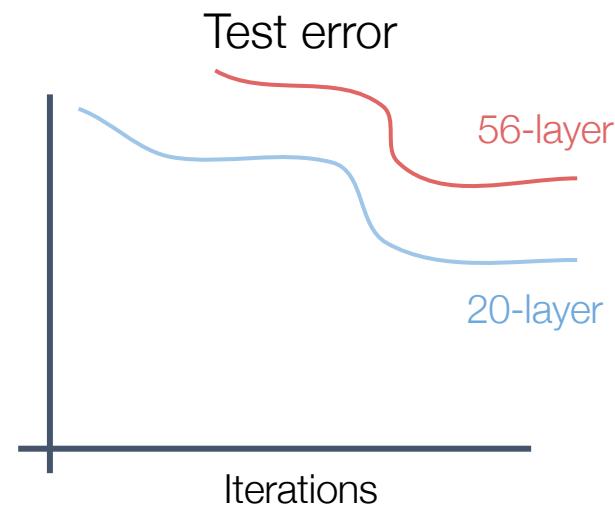
Once we have Batch Normalization, we can train networks with 10+ layers.
What happens as we go deeper?

Residual Networks

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

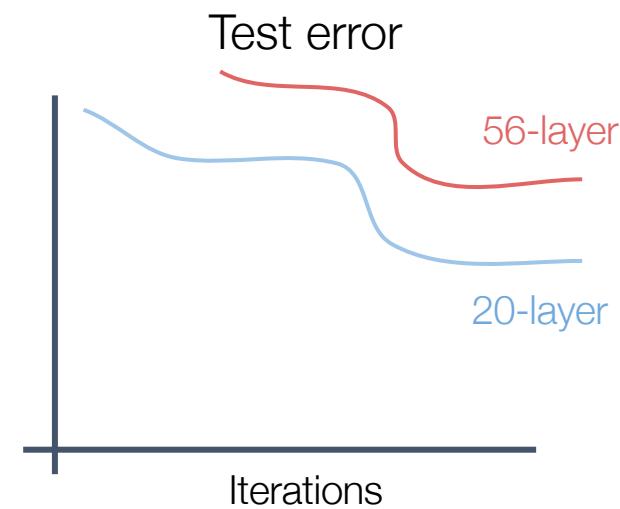
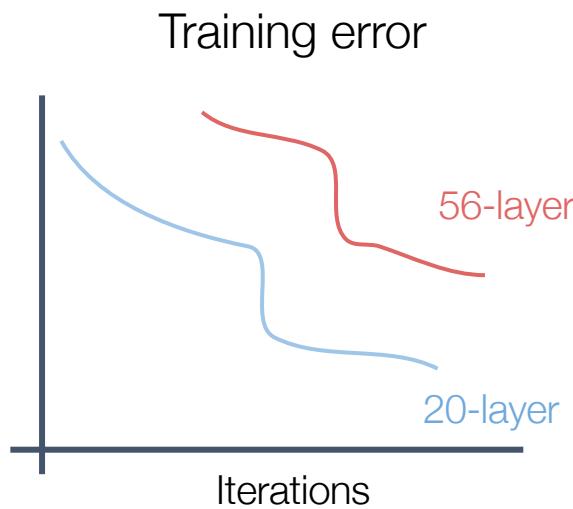
Deeper model does worse than shallow model!

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?



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

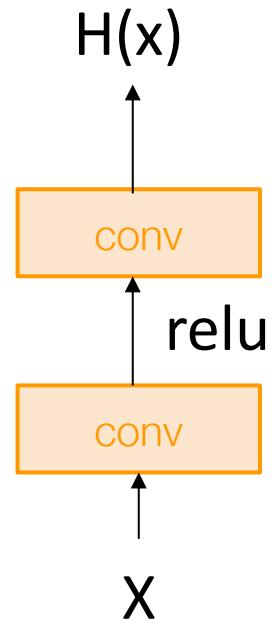
Thus deeper models should do at least as good as shallow models

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

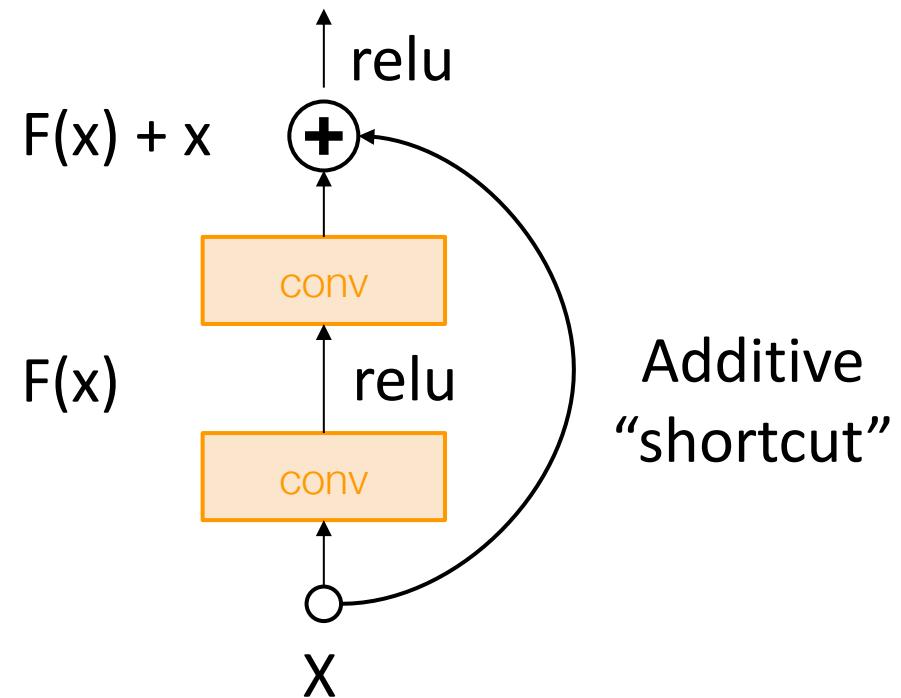
Solution: Change the network so learning identity functions with extra layers is easy!

Residual Networks

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



“Plain” block

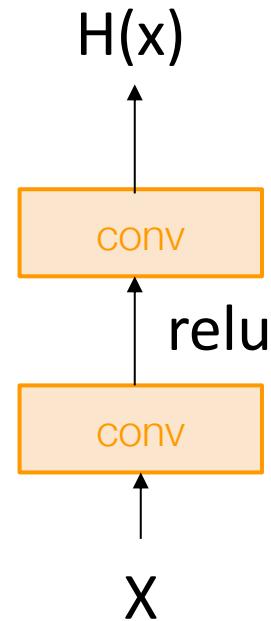


Residual Block

Additive
“shortcut”

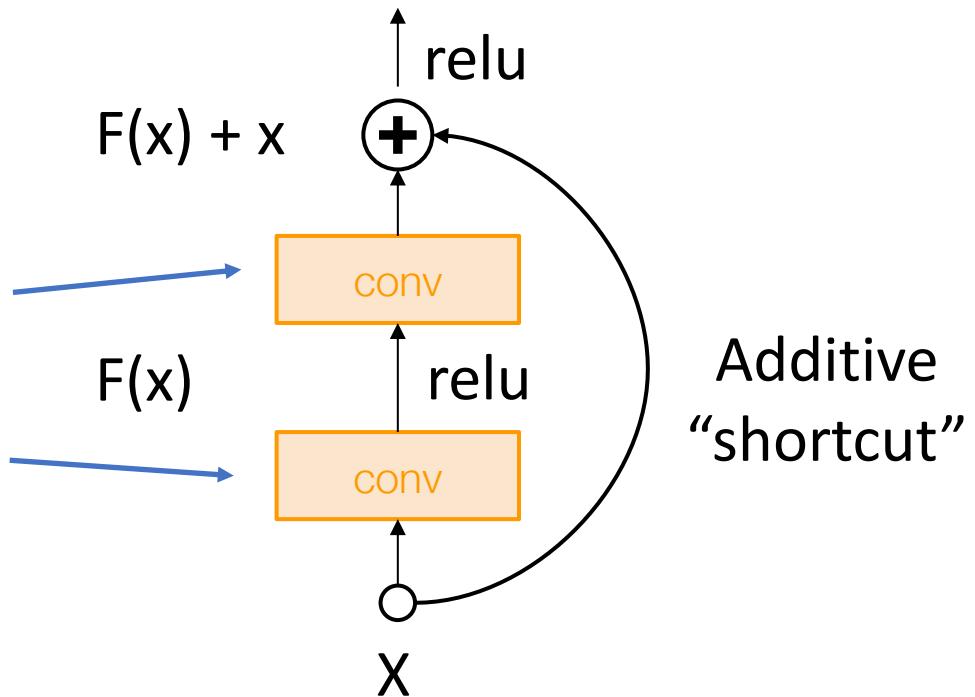
Residual Networks

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

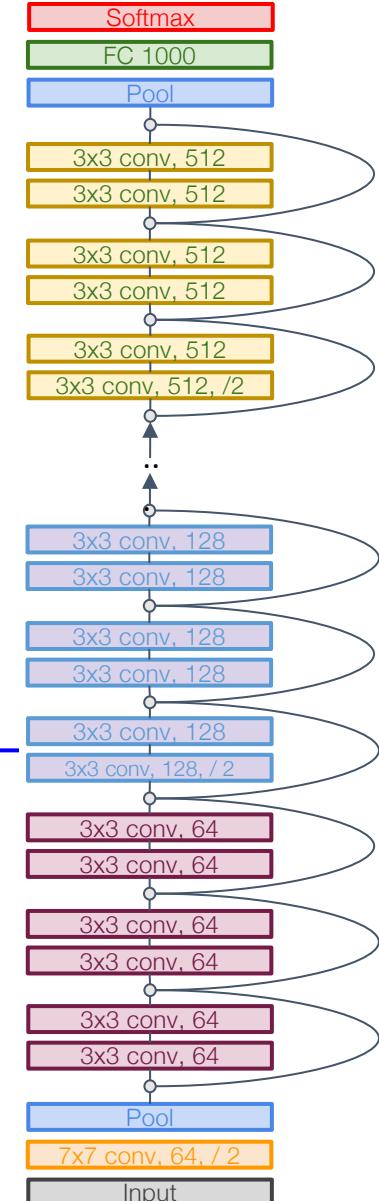
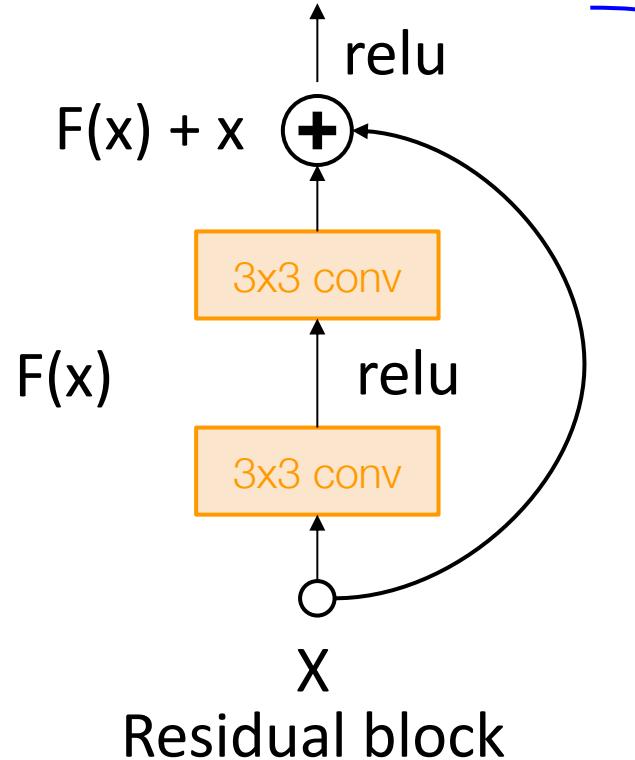
Additive
“shortcut”

Residual Networks

A residual network is a stack of many residual blocks

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

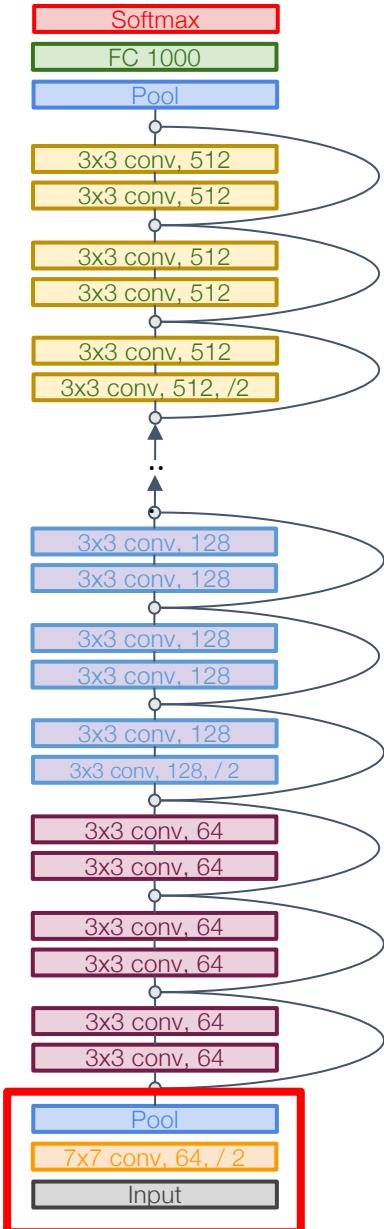
Network is divided into **stages**: the first block of each stage halves the resolution (with stride-2 conv) and doubles the number of channels



Residual Networks

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

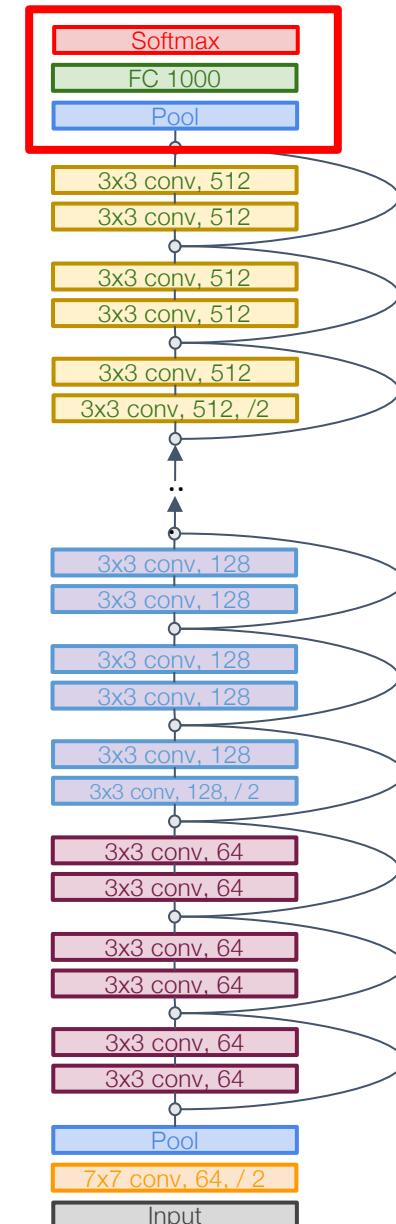
Layer	Input size		Layer				Output size		params (k)	flop (M)
	C	H/W	filters	kernel	stride	pad	C	H/W		
conv	3	224	64	7	2	3	64	112	3136	9 118
max-pool	64	112		3	2	1	64	56	784	0 2



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

Residual Networks

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



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

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

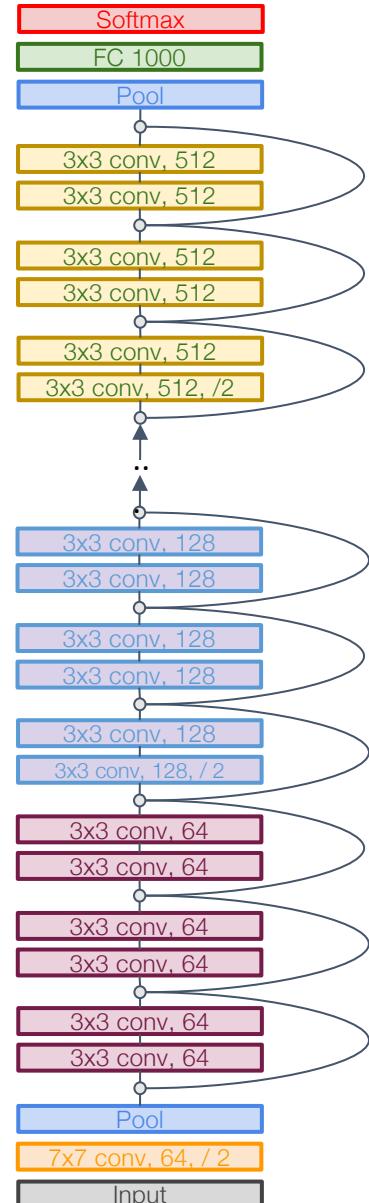
Stage 3 (C=256): 2 res. block = 4 conv

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

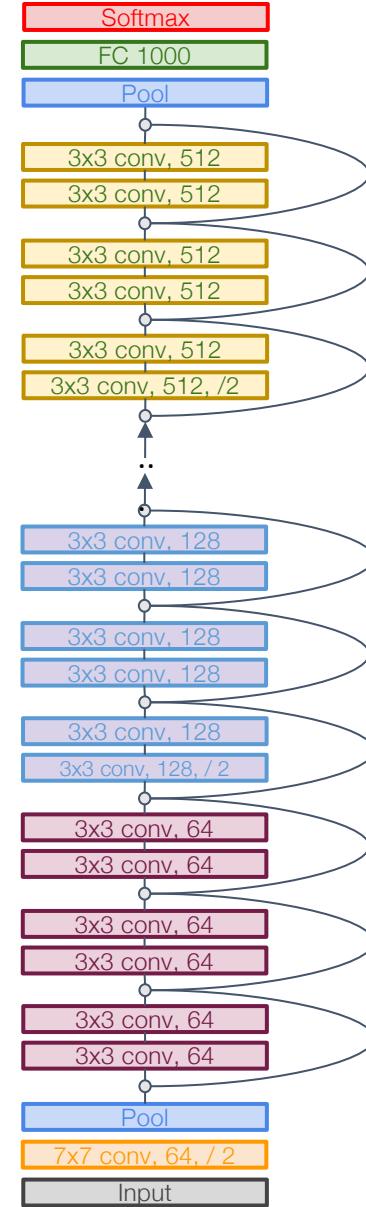
Stage 3: 6 res. block = 12 conv

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

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

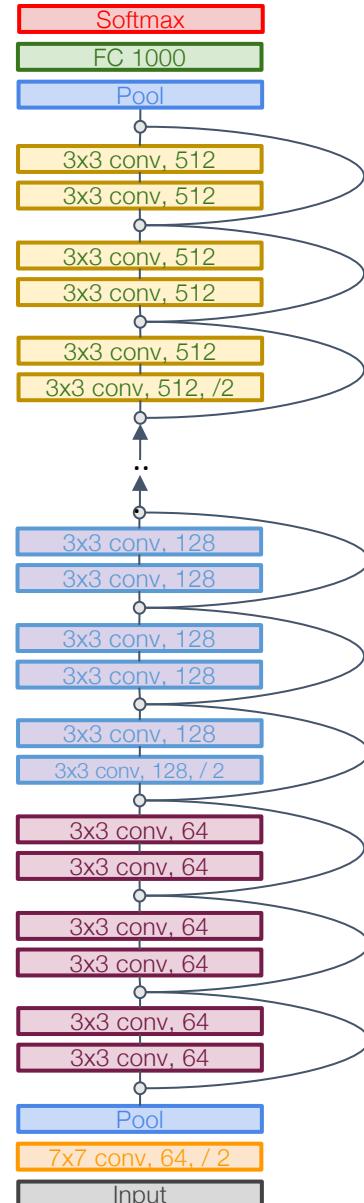
ImageNet top-5 error: 8.58

GFLOP: 3.6

VGG-16:

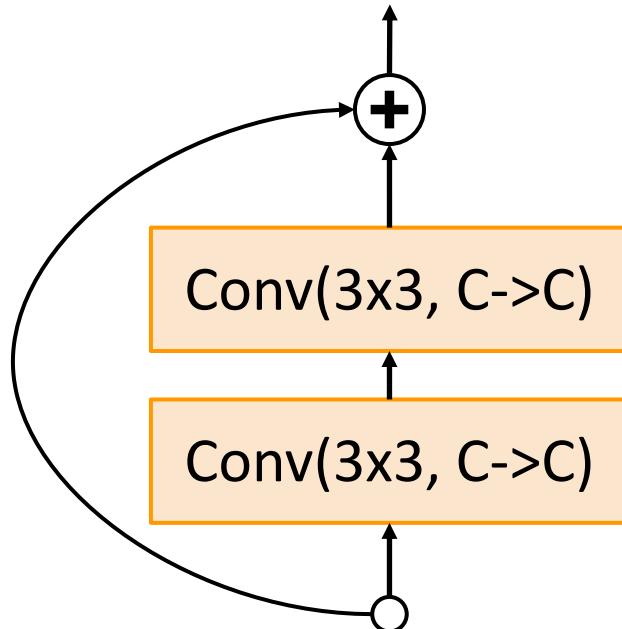
ImageNet top-5 error: 9.62

GFLOP: 13.6



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

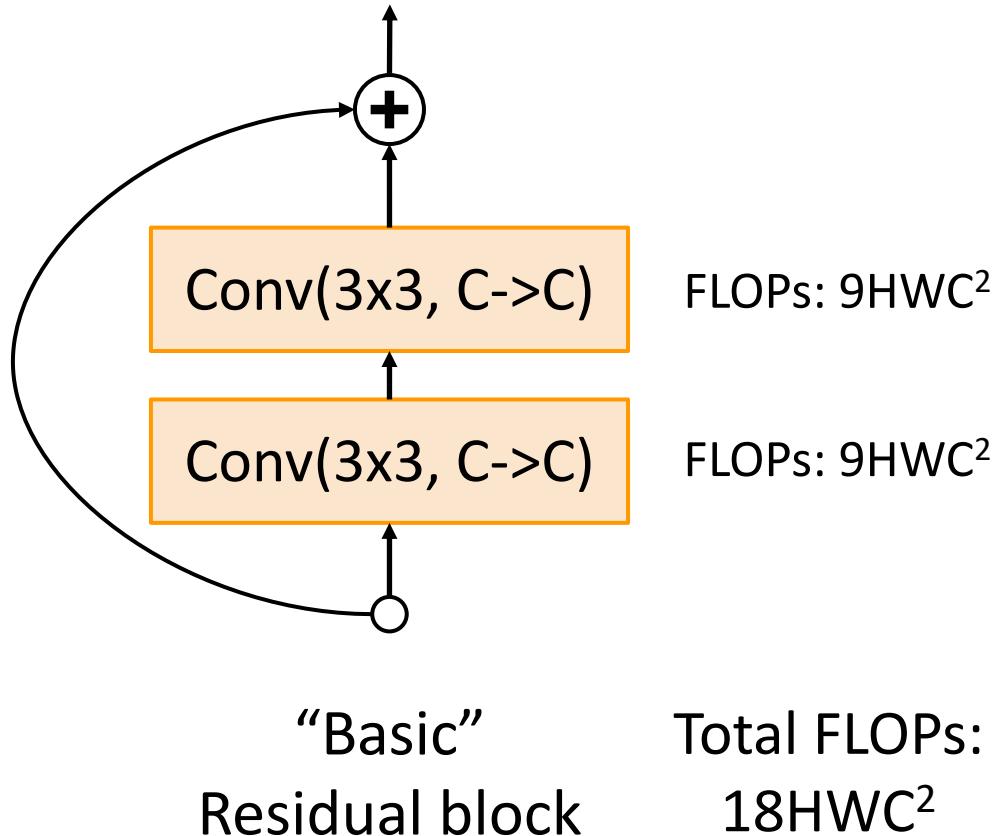
Residual Networks: Basic Block



“Basic”
Residual block

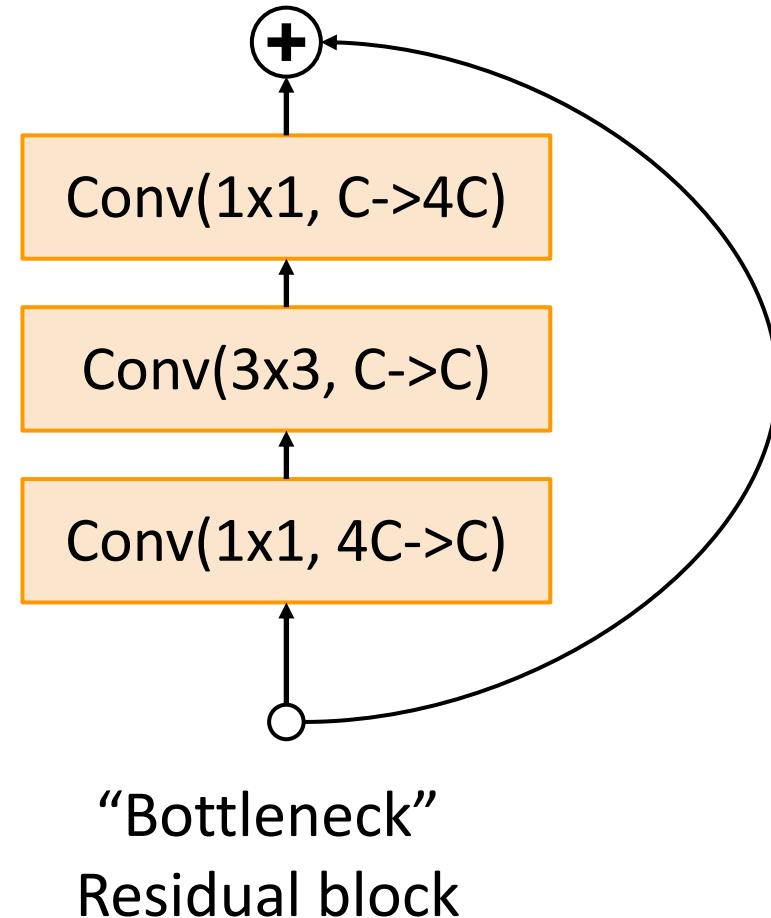
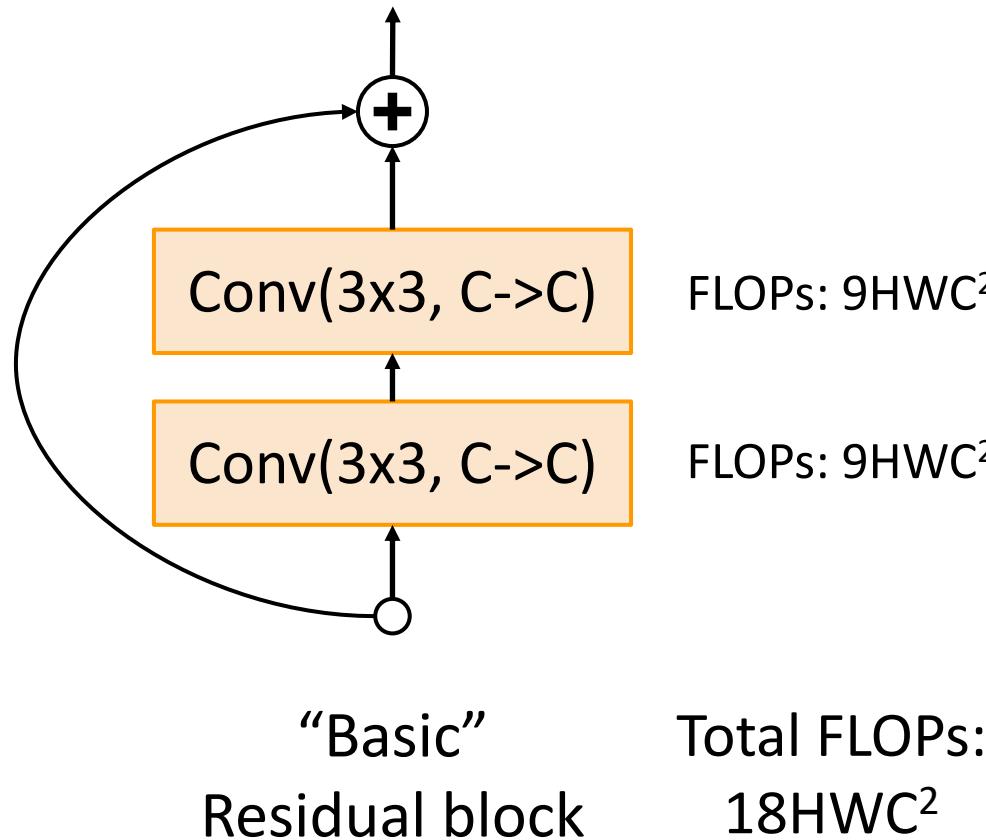
He et al, “Deep Residual Learning for Image Recognition”, CVPR 2016

Residual Networks: Basic Block



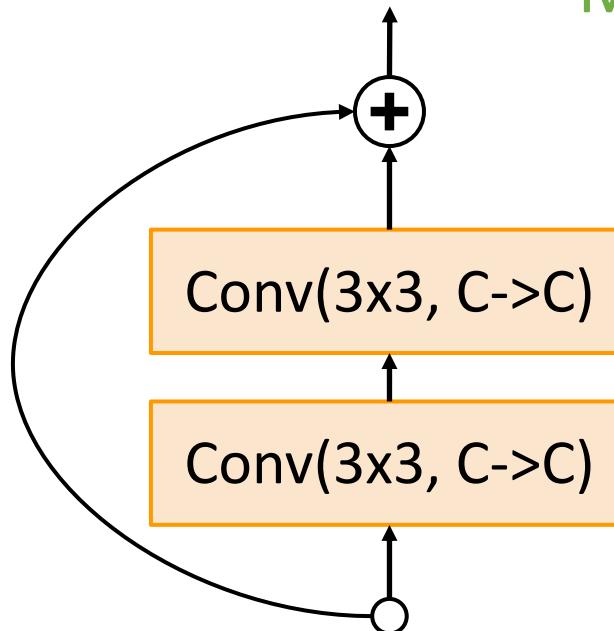
He et al, “Deep Residual Learning for Image Recognition”, CVPR 2016

Residual Networks: Bottleneck Block



Residual Networks: Bottleneck Block

More layers, less computational cost!



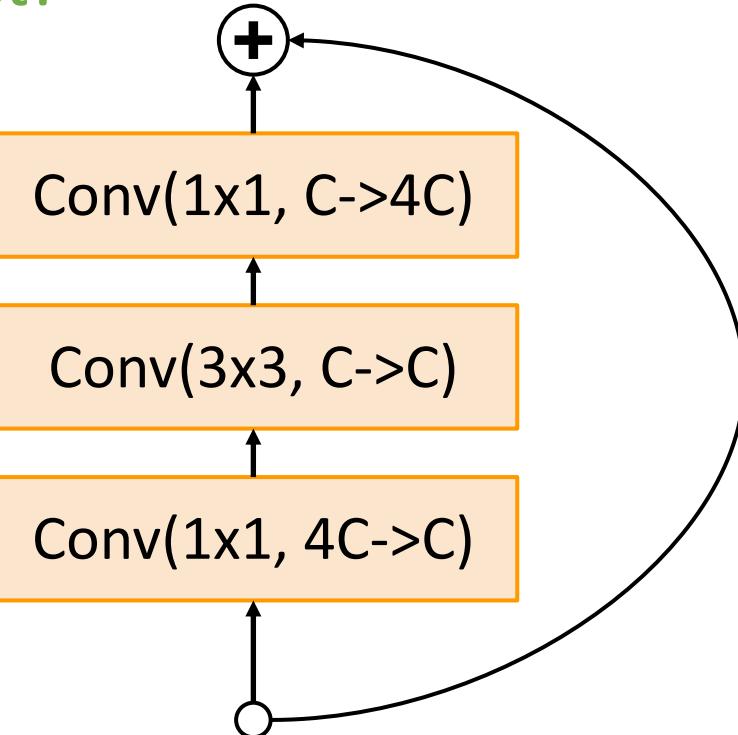
"Basic"
Residual block

Total FLOPs:
 $18HWC^2$

FLOPs: $4HWC^2$

FLOPs: $9HWC^2$

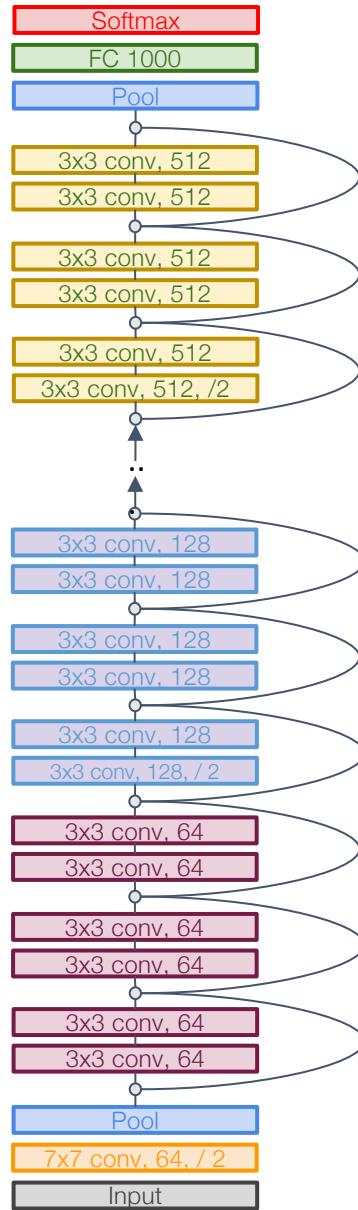
FLOPs: $4HWC^2$



"Bottleneck"
Residual block

Total FLOPs:
 $17HWC^2$

Residual Networks



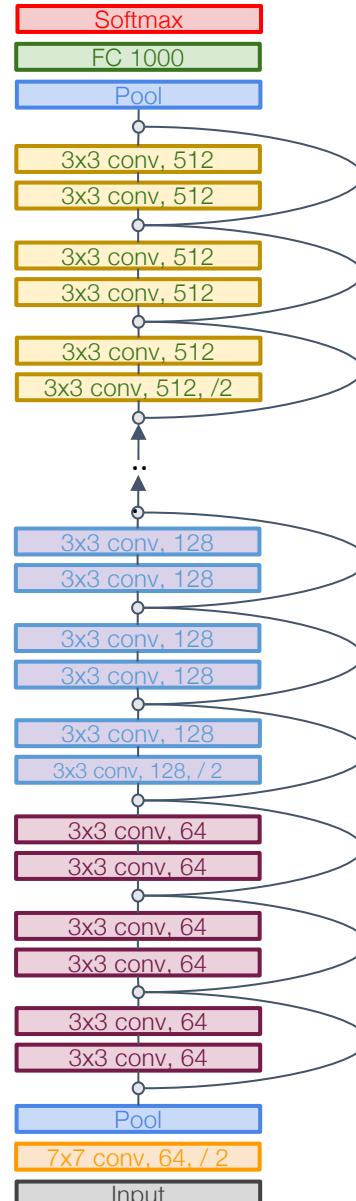
			Stage 1		Stage 2		Stage 3		Stage 4		FC	ImageNet
	Block type	Stem layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	GFLOP	top-5 error
ResNet-18	Basic	1	2	4	2	4	2	4	2	4	1	1.8
ResNet-34	Basic	1	3	6	4	8	6	12	3	6	1	3.6

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

Residual Networks

ResNet-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!

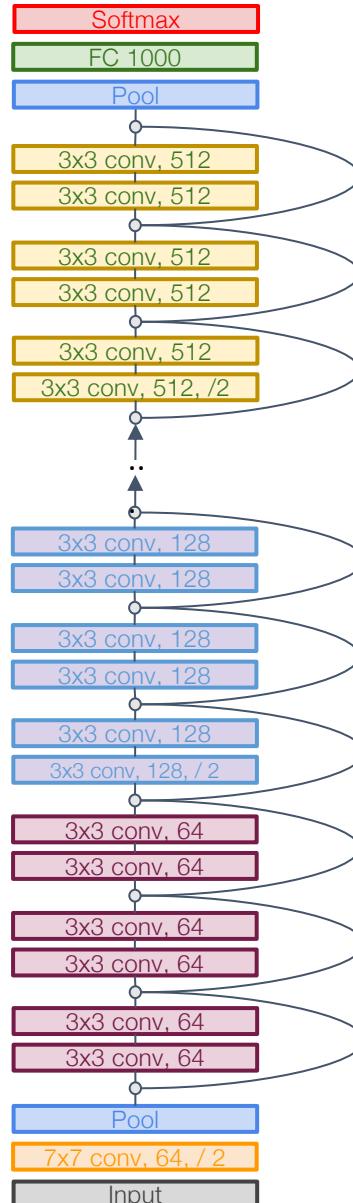
			Stage 1		Stage 2		Stage 3		Stage 4		FC	ImageNet
	Block type	Stem layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	layers	GFLOP top-5 error
ResNet-18	Basic	1	2	4	2	4	2	4	2	4	1	1.8 10.92
ResNet-34	Basic	1	3	6	4	8	6	12	3	6	1	3.6 8.58
ResNet-50	Bottle	1	3	9	4	12	6	18	3	9	1	3.8 7.13



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
ResNet-34	Basic	1	3	6	4	8	6	12	3	6	1	3.6
ResNet-50	Bottle	1	3	9	4	12	6	18	3	9	1	3.8
ResNet-101	Bottle	1	3	9	4	12	23	69	3	9	1	7.6
ResNet-152	Bottle	1	3	9	8	24	36	108	3	9	1	11.3

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

Residual Networks

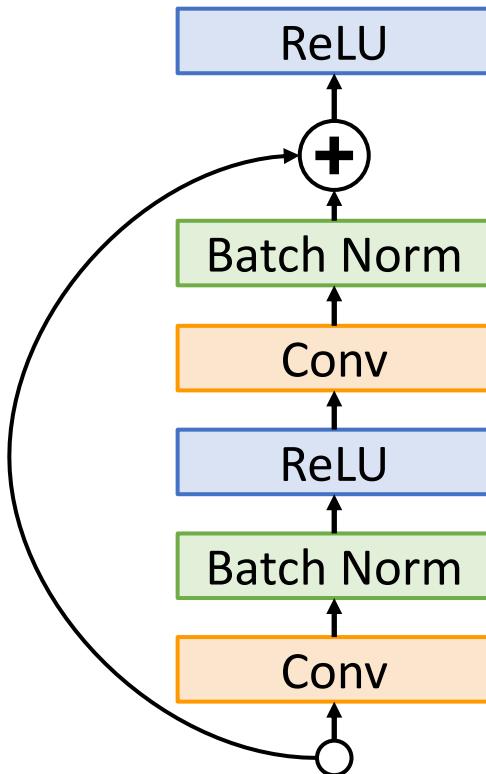
- Able to train very deep networks
- Deeper networks do better than shallow networks (as expected)
- Swept 1st place in all ILSVRC and COCO 2015 competitions
- Still widely used today!

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

Improving Residual Networks: Block Design

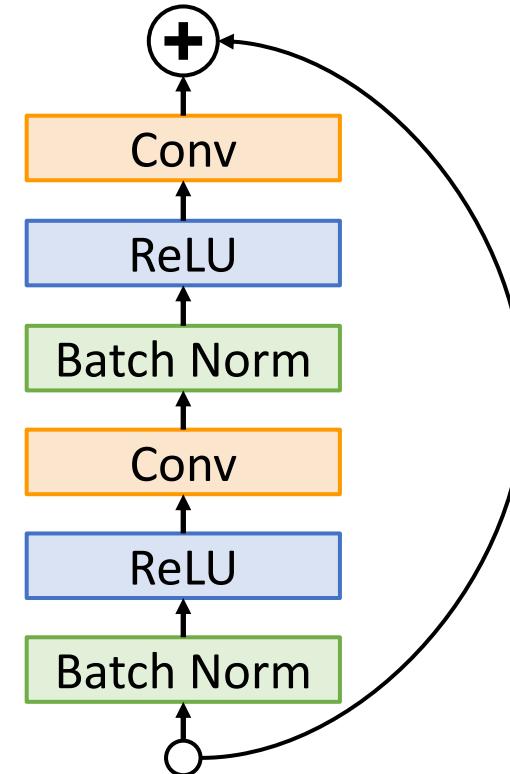
Original ResNet block



Note ReLU **after** residual:

Cannot actually learn
identity function since
outputs are nonnegative!

“Pre-Activation” ResNet Block

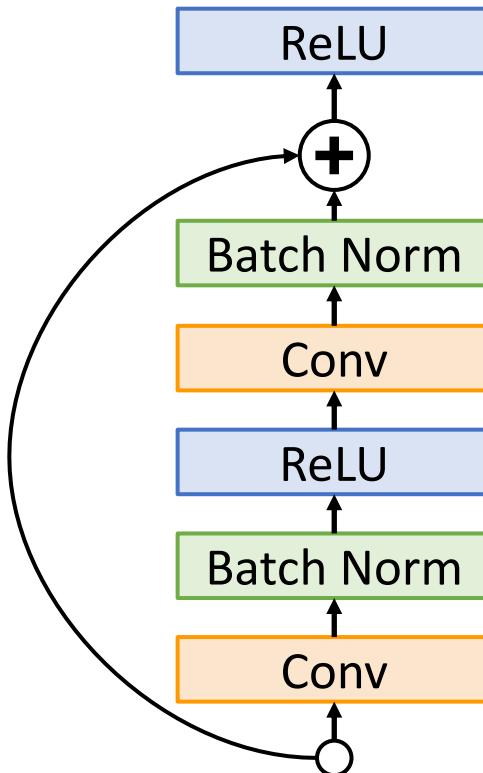


Note ReLU **inside** residual:

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

Improving Residual Networks: Block Design

Original ResNet block

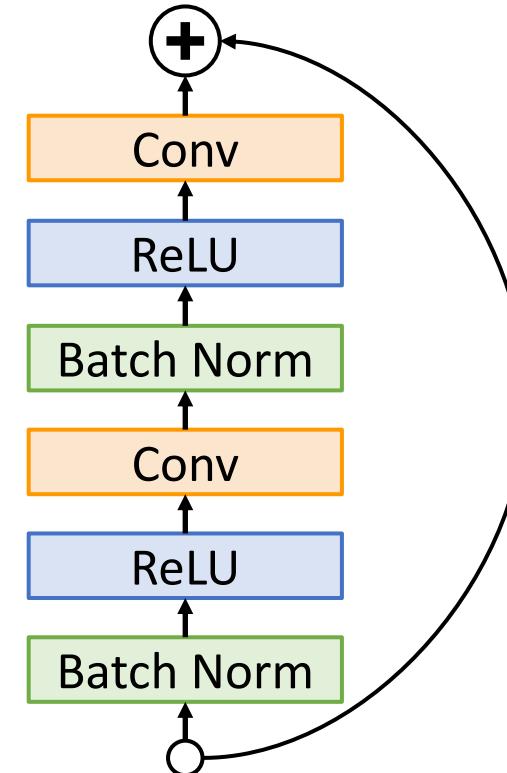


Slight improvement in accuracy
(ImageNet top-1 error)

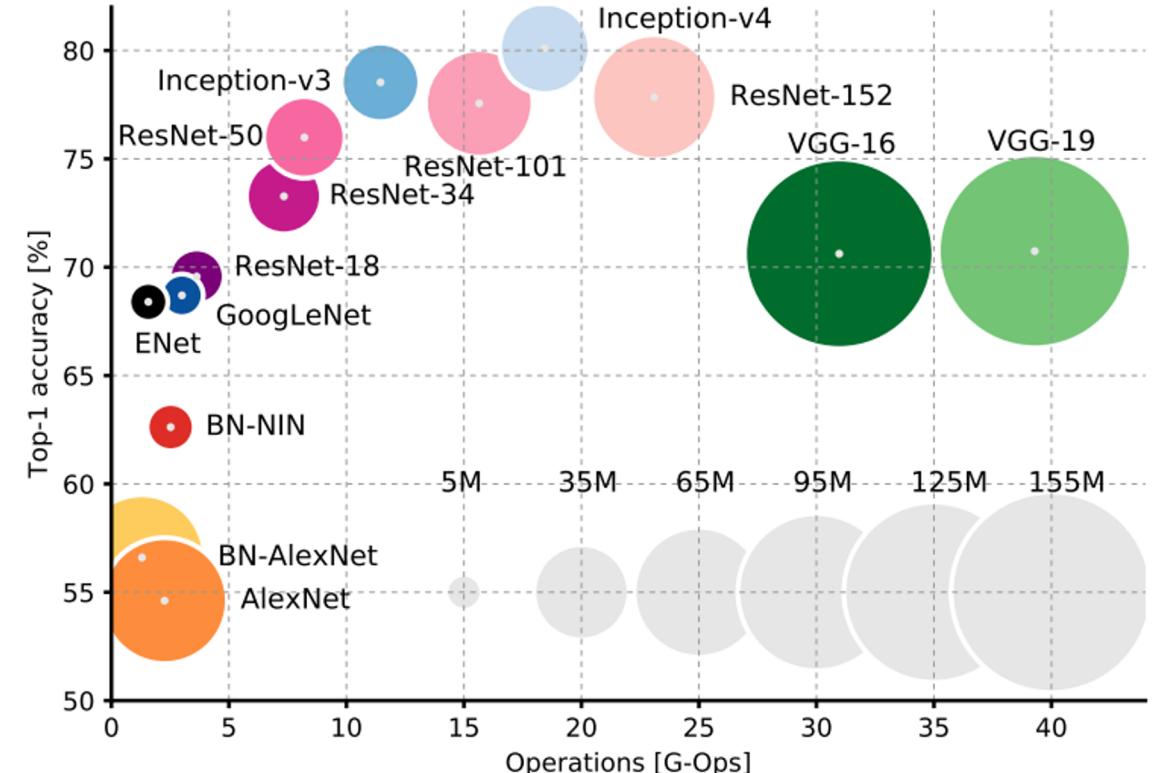
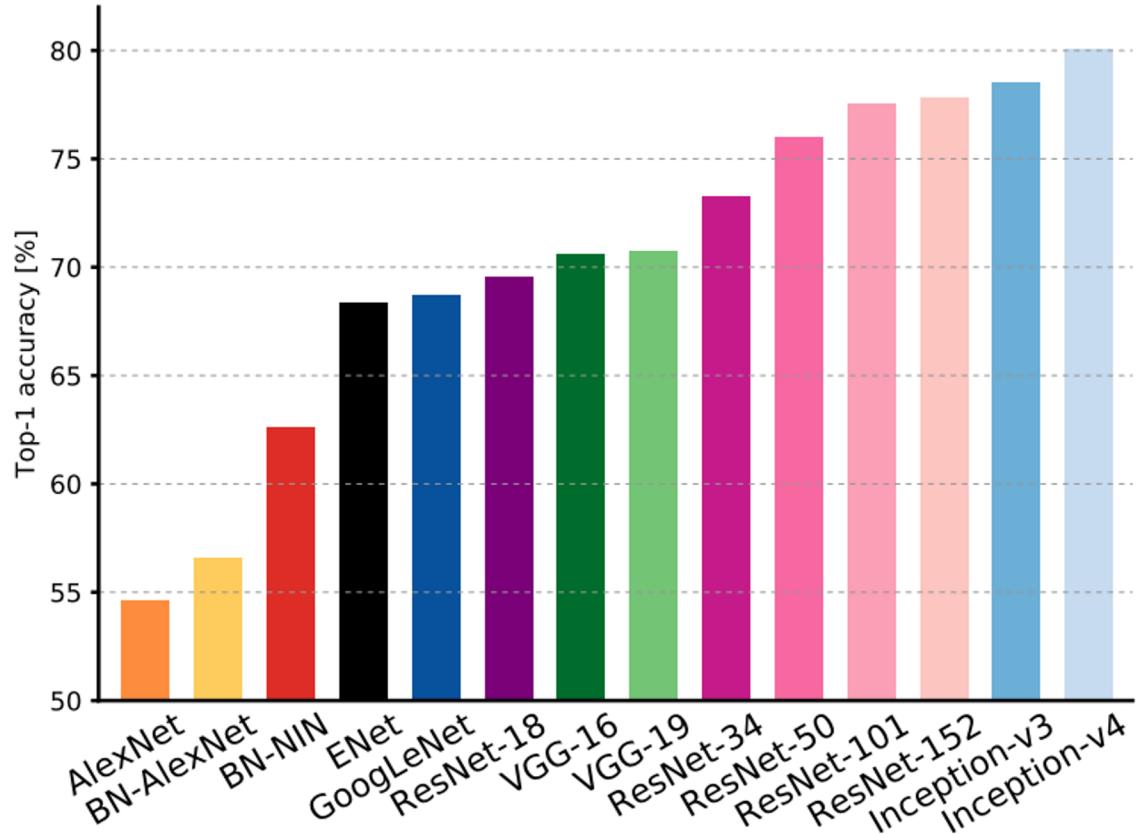
ResNet-152: 21.3 vs **21.1**
ResNet-200: 21.8 vs **20.7**

Not actually used that much in practice

“Pre-Activation” ResNet Block



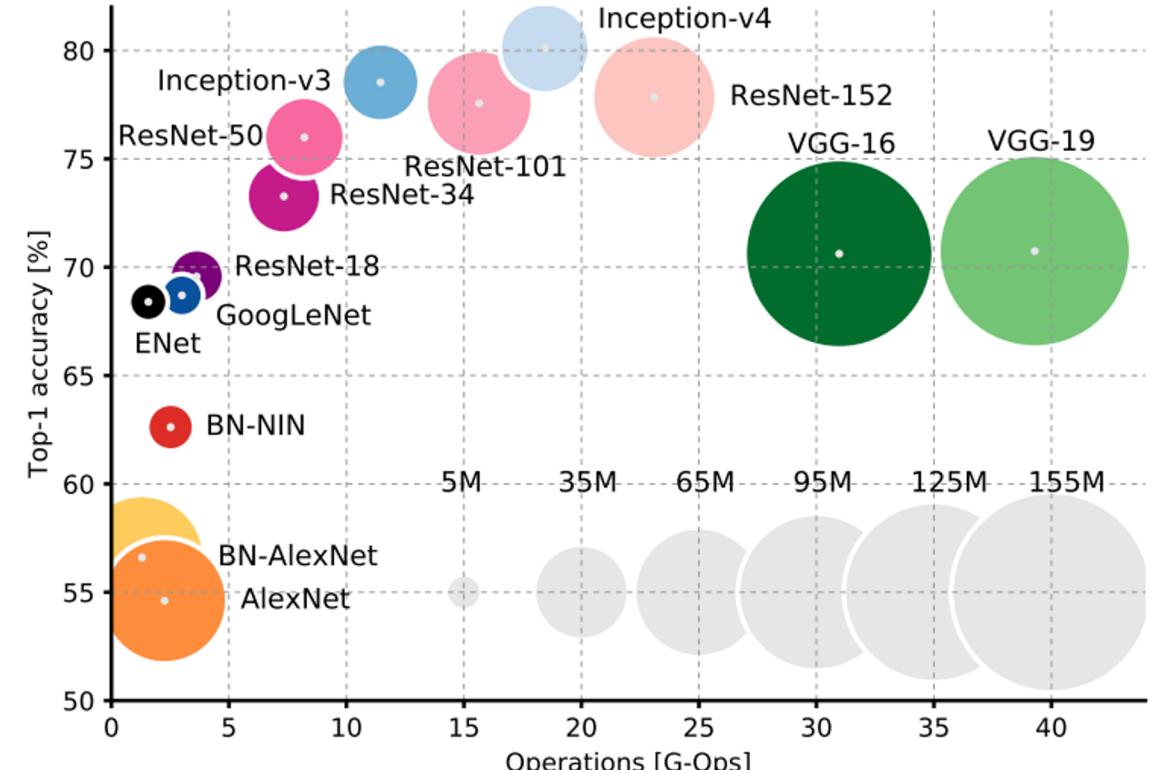
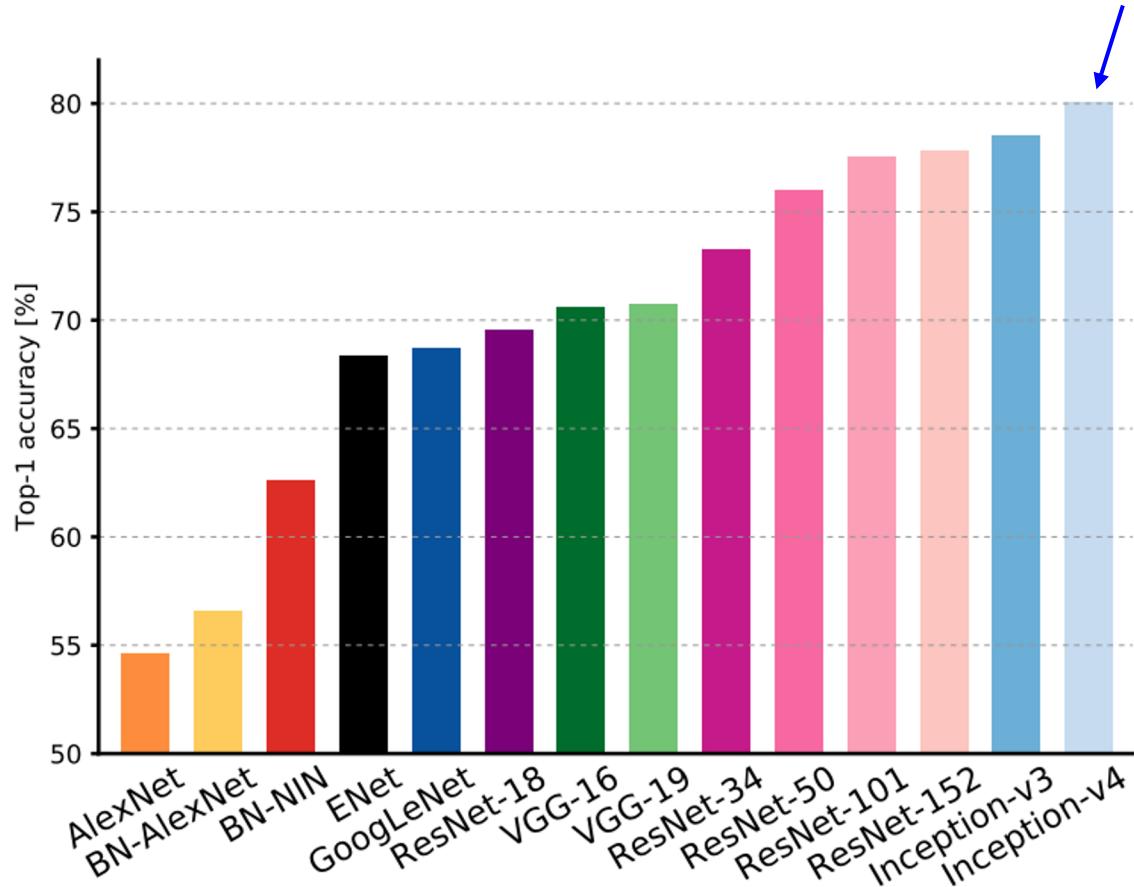
Comparing Complexity



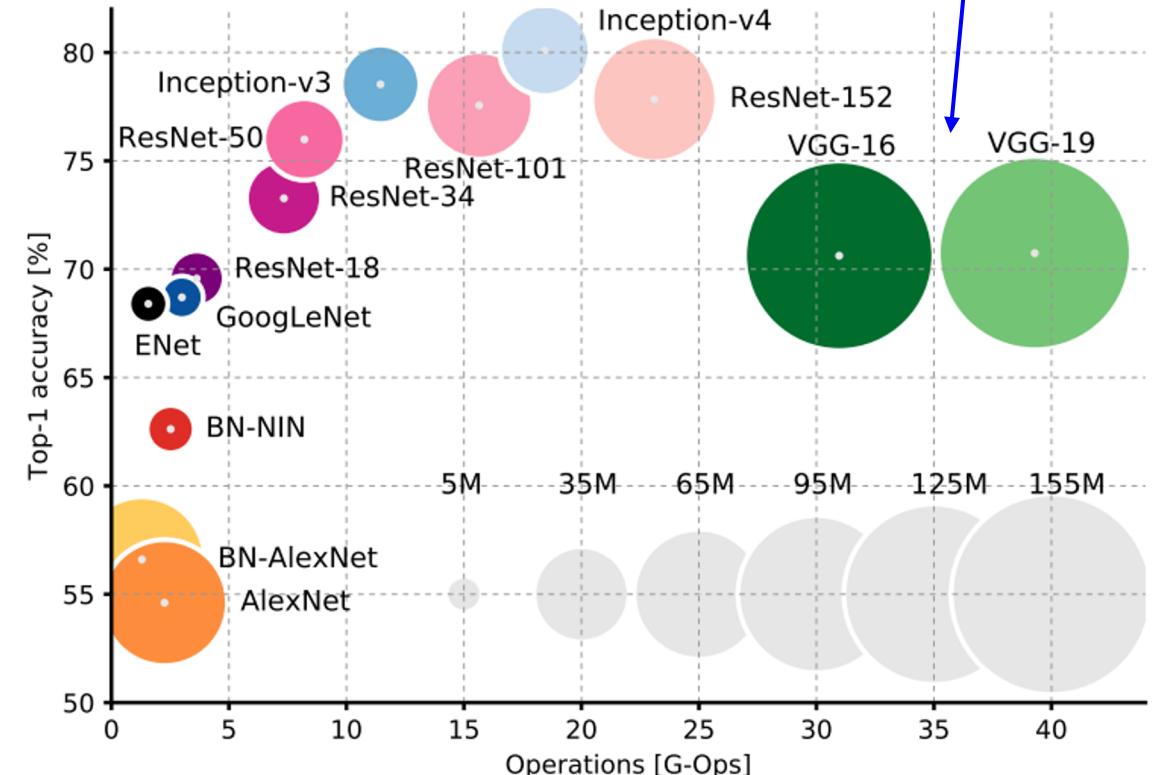
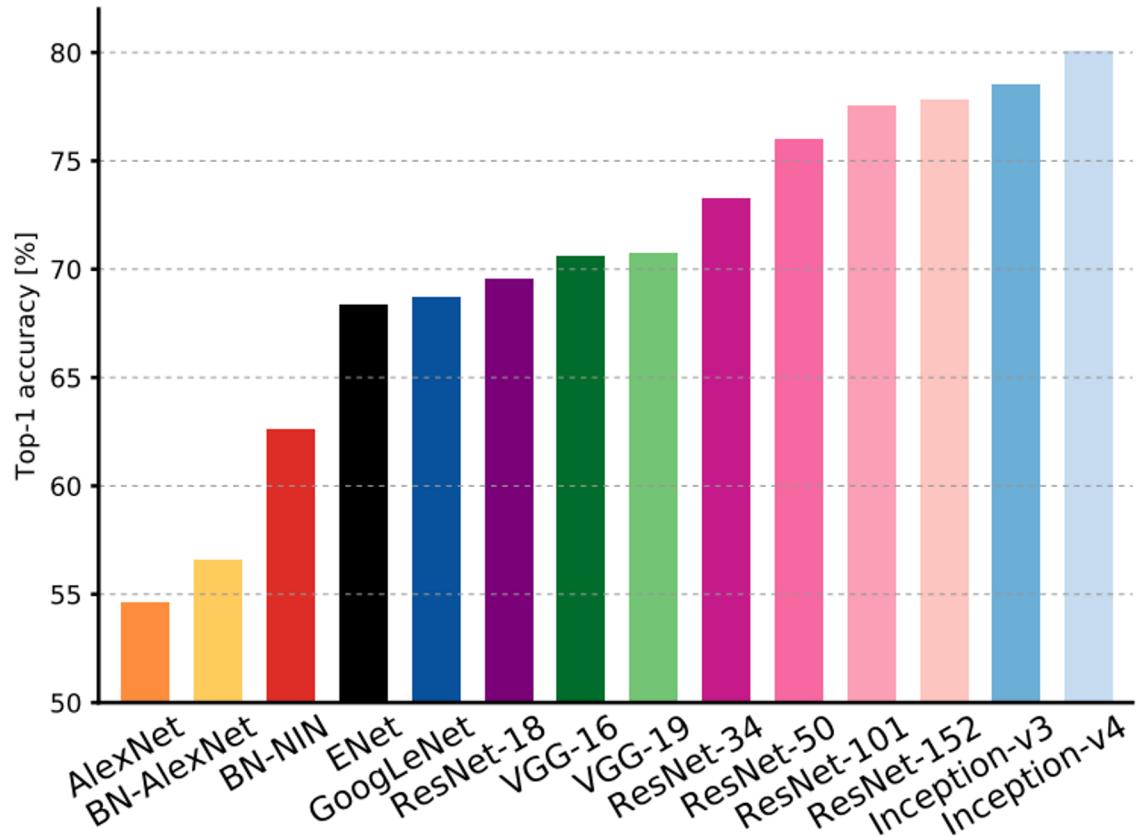
Canziani et al, "An analysis of deep neural network models for practical applications", 2017

Comparing Complexity

Inception-v4: Resnet + Inception!

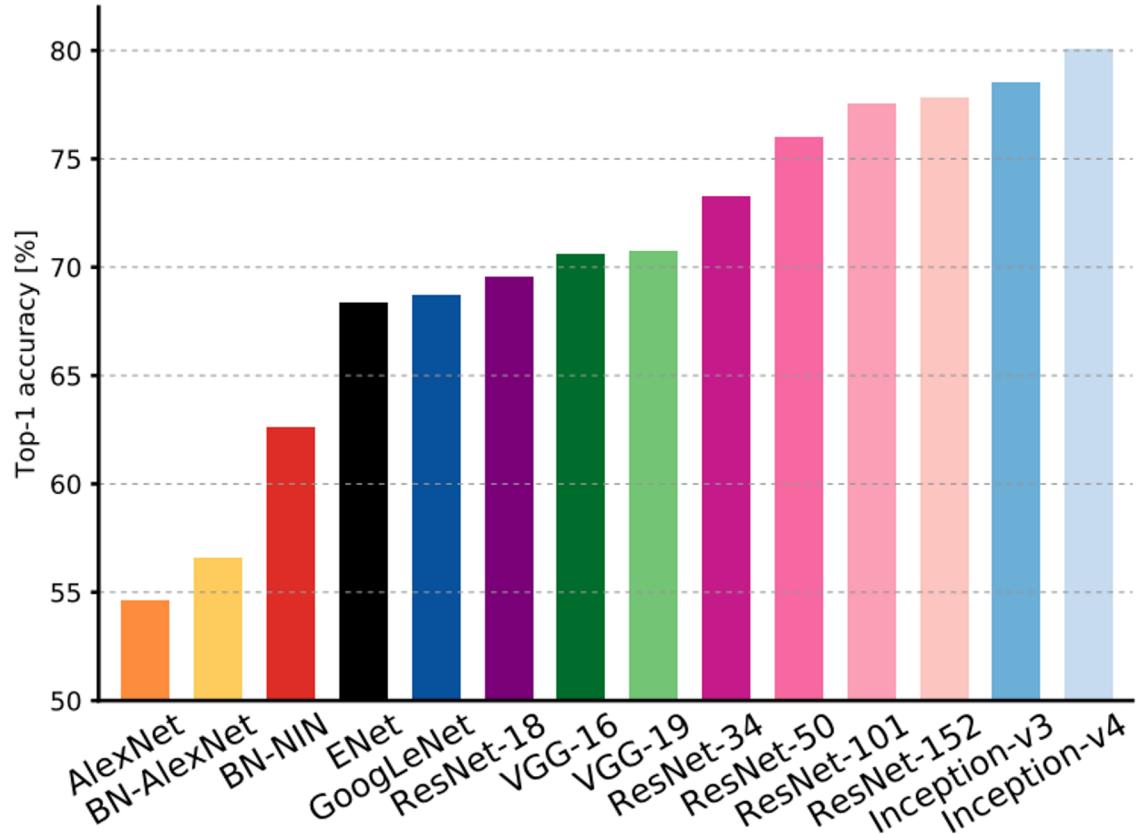


Comparing Complexity

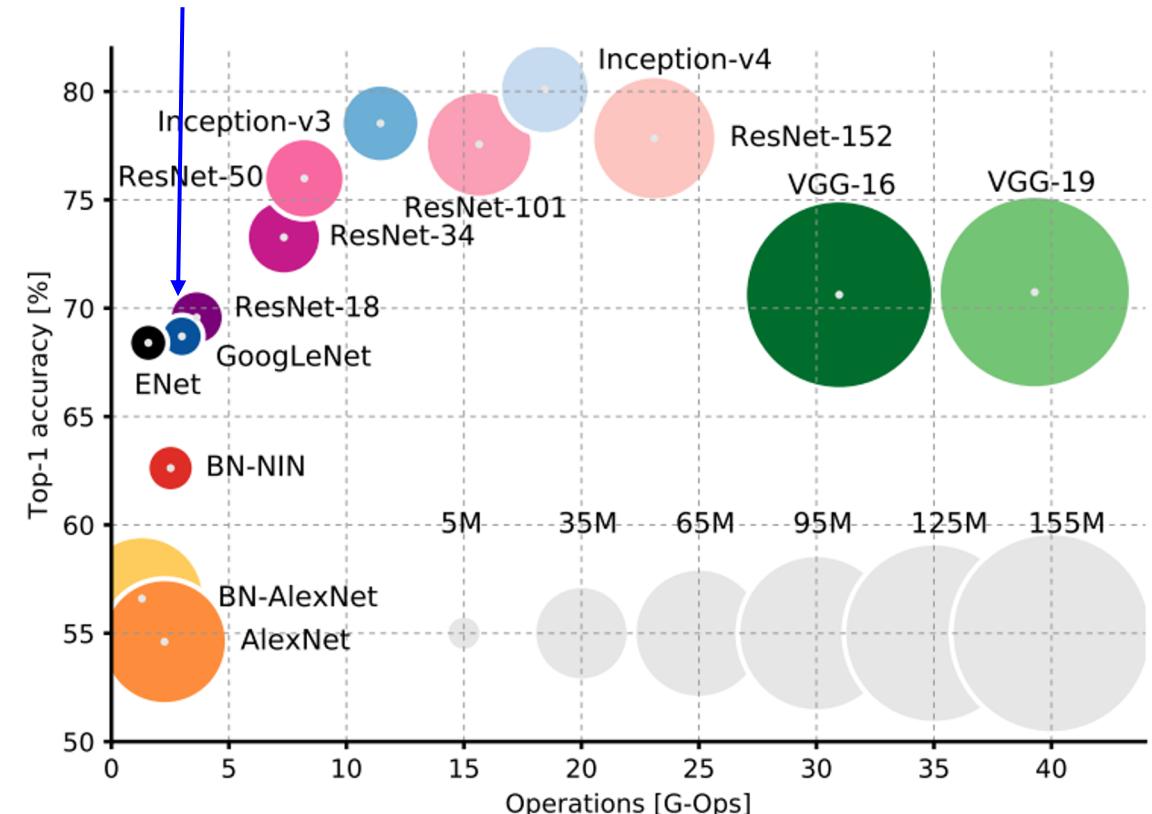


VGG: Highest memory, most operations

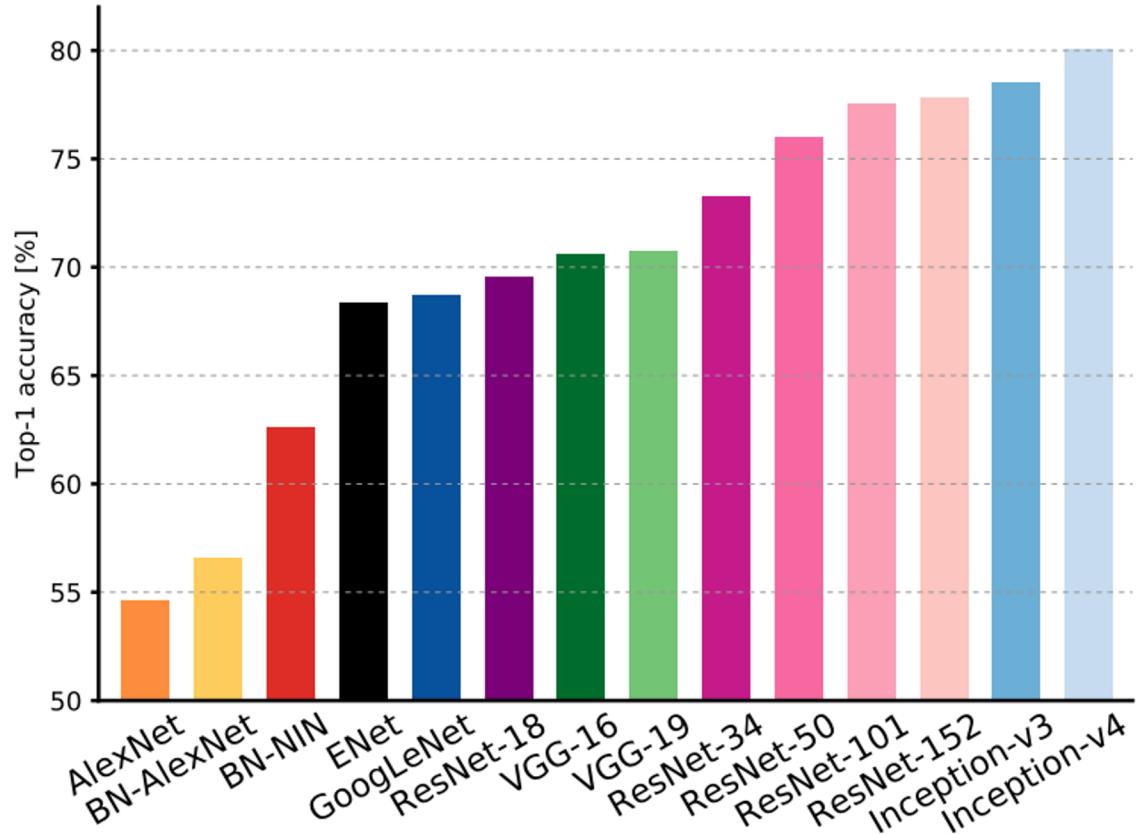
Comparing Complexity



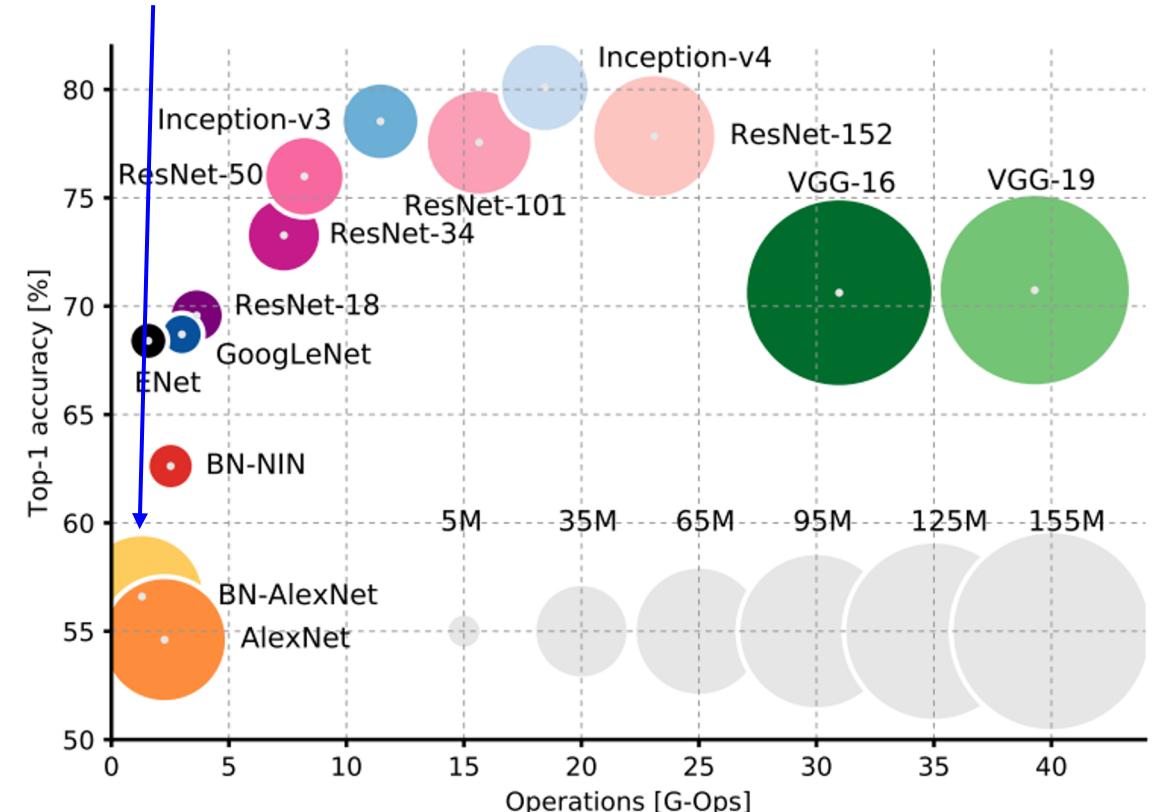
GoogLeNet:
Very efficient!



Comparing Complexity

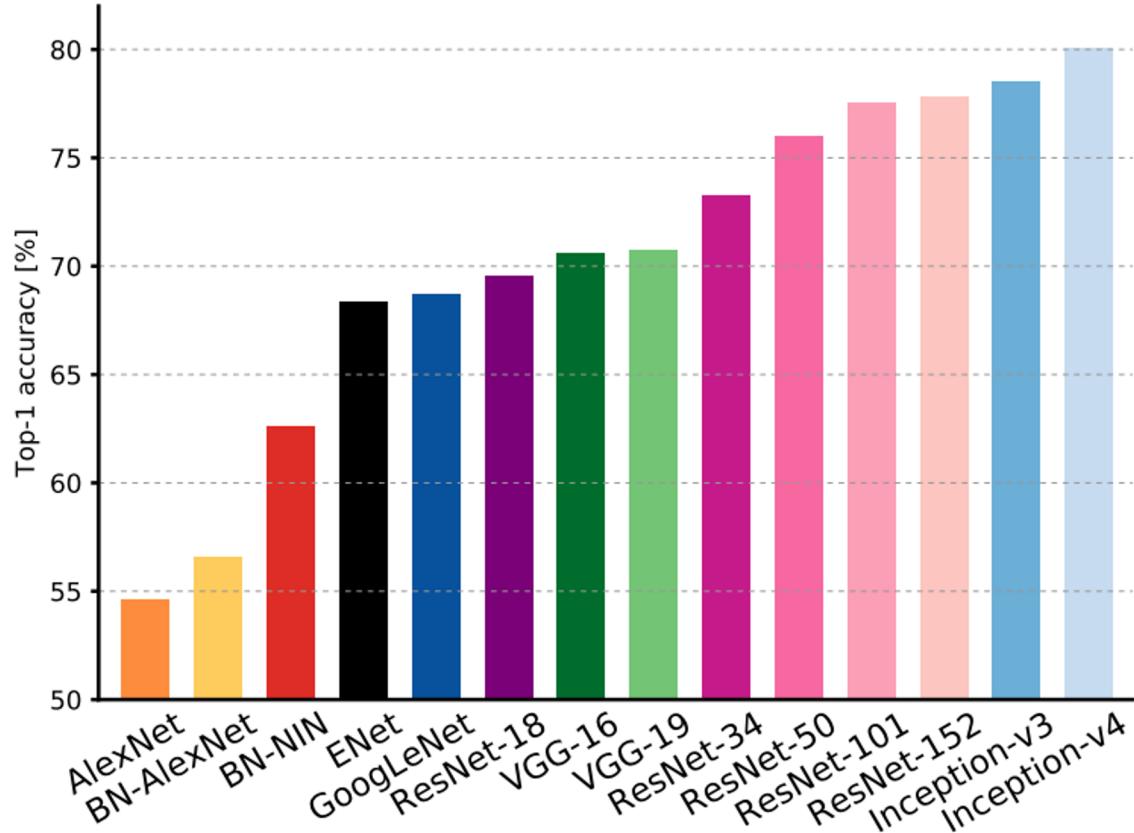


AlexNet: Low compute, lots of parameters

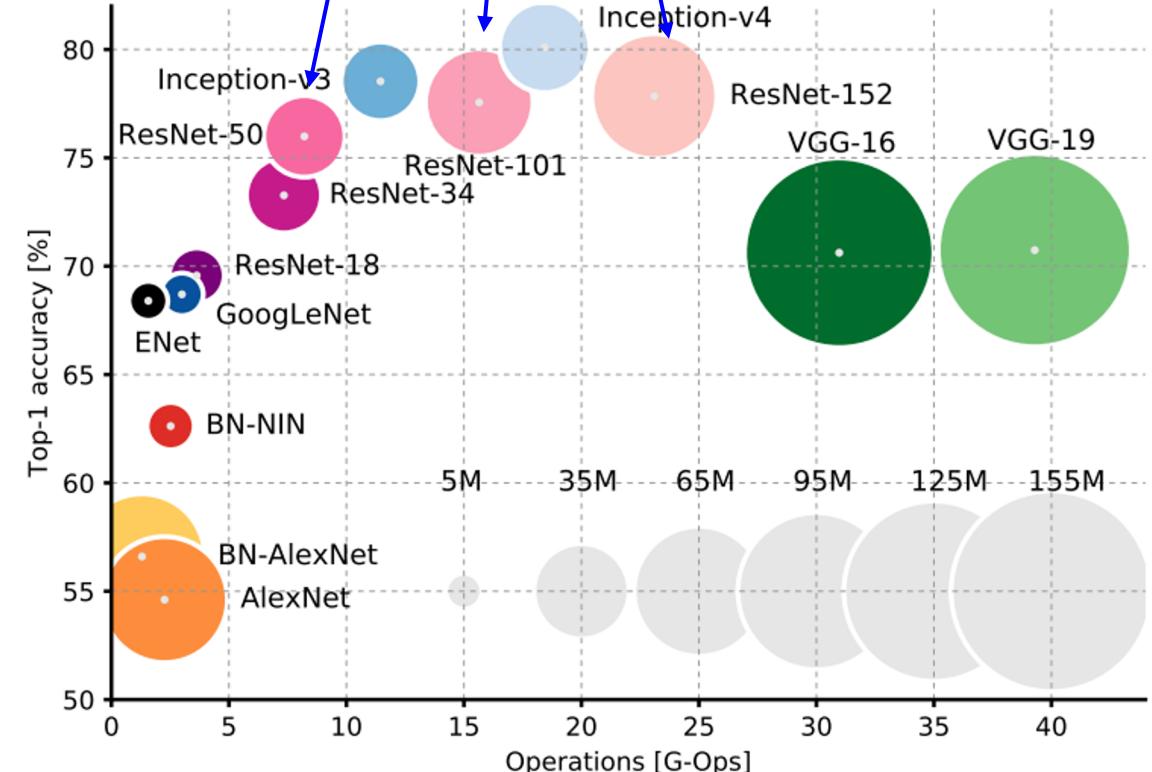


Canziani et al, "An analysis of deep neural network models for practical applications", 2017

Comparing Complexity

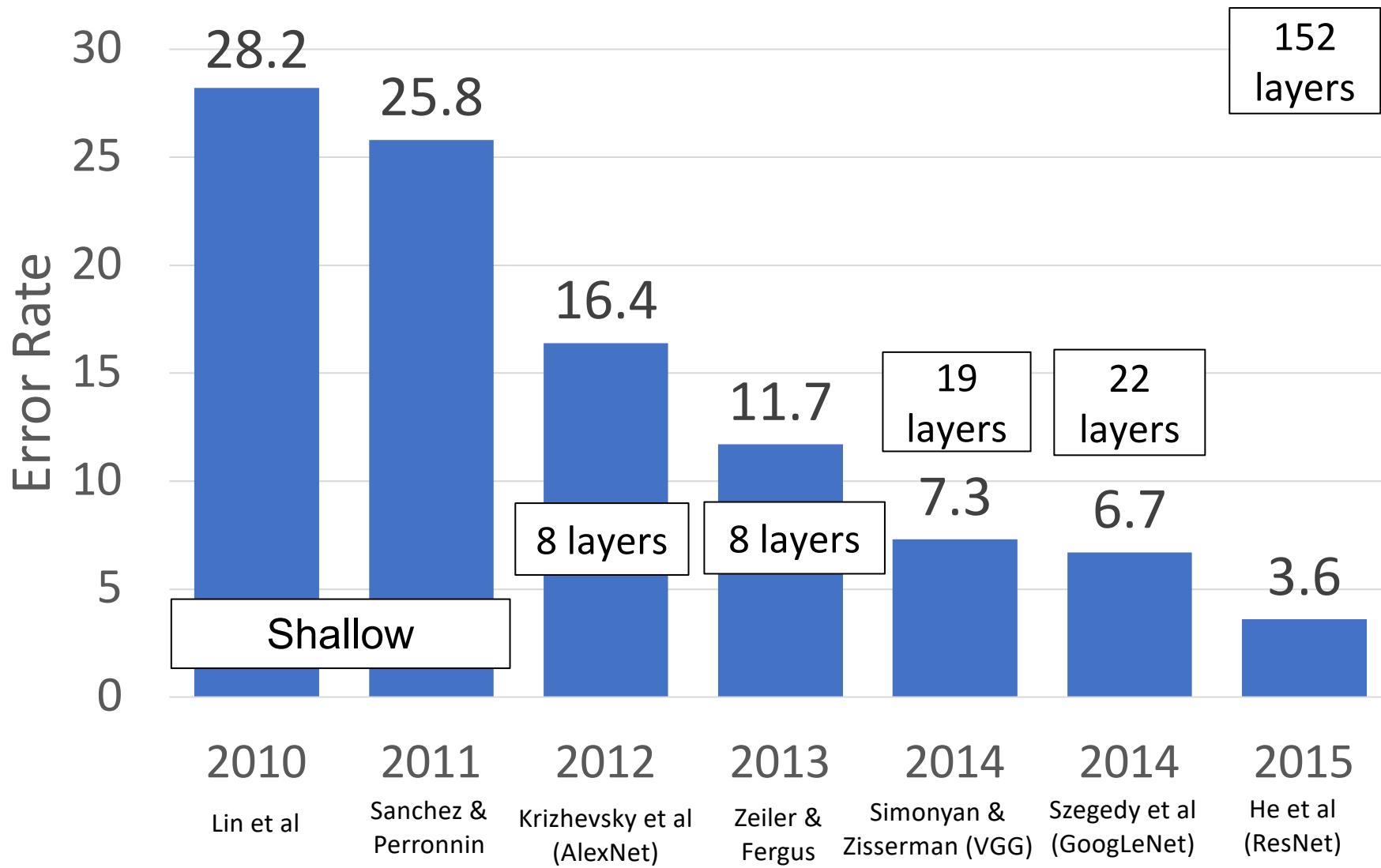


ResNet: Simple design,
moderate efficiency,
high accuracy

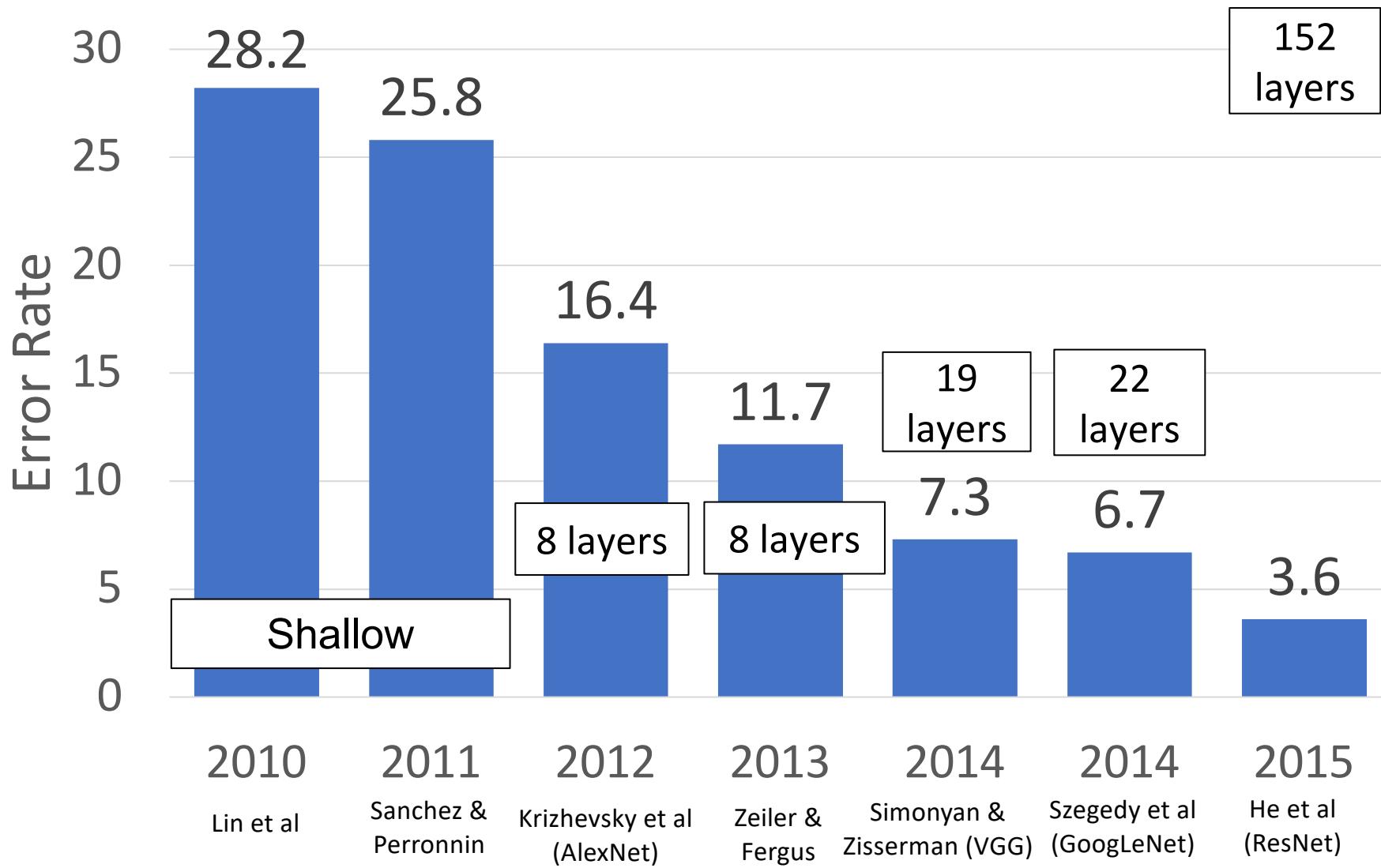


Canziani et al, "An analysis of deep neural network models for practical applications", 2017

ImageNet Classification Challenge



ImageNet Classification Challenge



CNN architectures have continued to evolve!

We will see more in Lecture 11



Next Time:

How to Train your CNN

- Activation functions
- Initialization
- Data preprocessing
- Data Augmentation
- Regularization