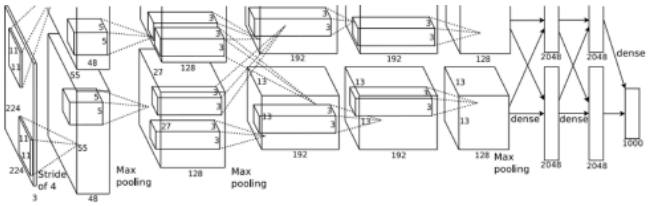
Lecture 8 - CNN Architectures

1. AlexNet

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

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

	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

- **Convolutional Layer**
  - Output channel number = number of filters --> $C = 64$
  - Output height (H') / width (W'):

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

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**

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton,

- Number of parameters / weights:

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**

- Number of flops (floating point operations --> multiply + add):

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

- **Pooling Layer**
  - Output channel number = number of filters --> $C = 64$
  - Output height (H') / width (W'):

    W' = floor((W – K) / S + 1) = floor(53 / 2 + 1) = floor(27.5) = 27
  - Memory (KB):

#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**

  - Number of parameters / weights: Pooling layers have no learnable parameters!
  - Flops:

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**

- **Fully-connected Layer**

	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

  - Output channel = flatten output size (number of all elements)

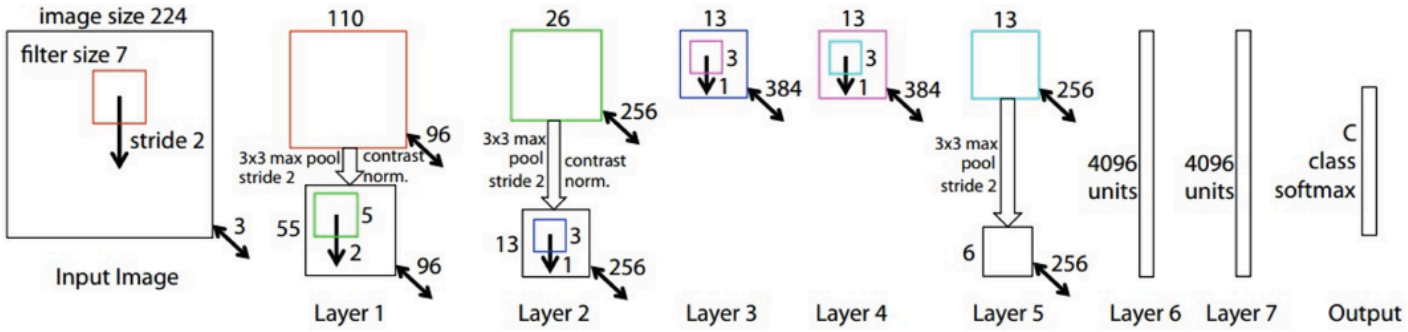
    Flatten output size = $C_{in} \times H \times W$ = 256 * 6 * 6 = 9216
  - Number of parameters / weights:

    FC params = $C_{in} \times C_{out} + C_{out}$= 9216 $\times$ 4096 + 4096 = 37,725,832
  - Number of flops:

    FC flops = $C_{in} \times C_{out}$ = 9216 $\times$ 4096 = 37,748,736

- **Summary**
  - 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

2. ZFNet: A Bigger AlexNet



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 =(

3. VGG: Deeper Networks, Regular Design

--> a much bigger network than AlexNet

- **Design Rules:**
  - **All conv are 3x3 stride 1 pad 1**

Option 1:	Option 2:
Conv(5x5, C -> C)	Conv(3x3, C -> C) Conv(3x3, C -> C)
Params: $25C^2$	Params: $18C^2$
FLOPs: $25C^2HW$	FLOPs: $18C^2HW$

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

  - **All max pool are 2x2 stride 2**
  - **After pool, double #channels**

VGG: Deeper Networks, Regular Design

VGG Design rules:

All conv are 3x3 stride 1 pad 1

All max pool are 2x2 stride 2

After pool, double #channels

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

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

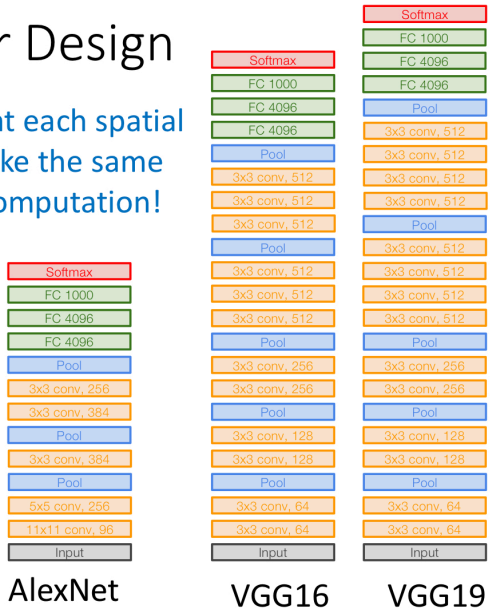
Memory: 4HWC
Params: $9C^2$
FLOPs: $36HWC^2$

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

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

4HWC x 3 x 3 x C

2HWC x 3 x 3 x 2 x 2 C



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

4. GoogLeNet: Focus on Efficiency

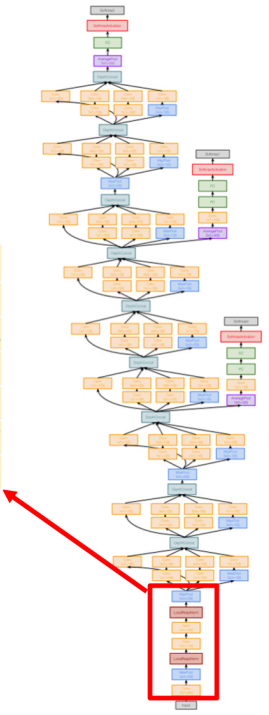
--> reduce parameter count, memory usage, and computation

GoogLeNet: Aggressive Stem

reduce spatial size

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

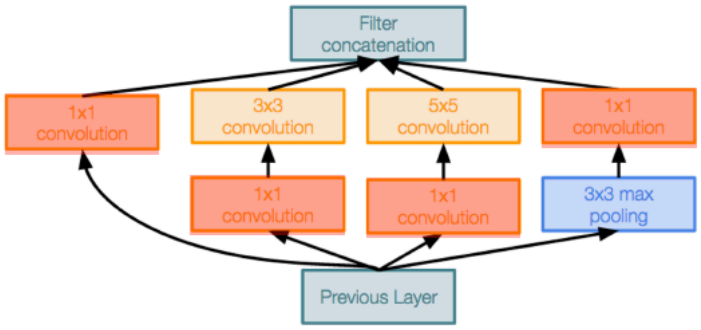
Compare VGG-16:
Memory: 42.9 MB (5.7x)
Params: 1.1M (8.9x)
MFLOP: 7485 (17.8x)

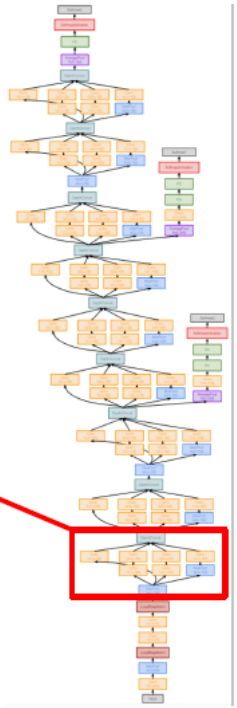
GoogLeNet: Inception Module

Inception module
Local unit with parallel branches

Local structure repeated many times throughout the network

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

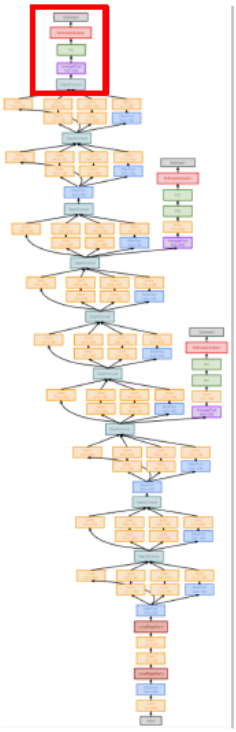




Szegedy et al, "Going deeper with convolutions", CVPR 2015

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

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

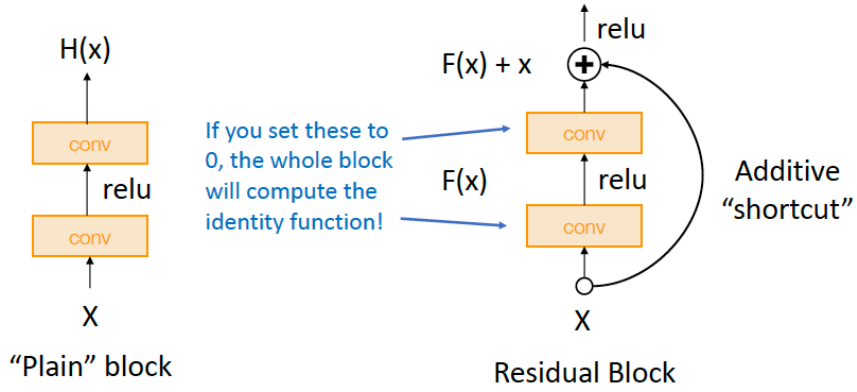
5. Residual Networks

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

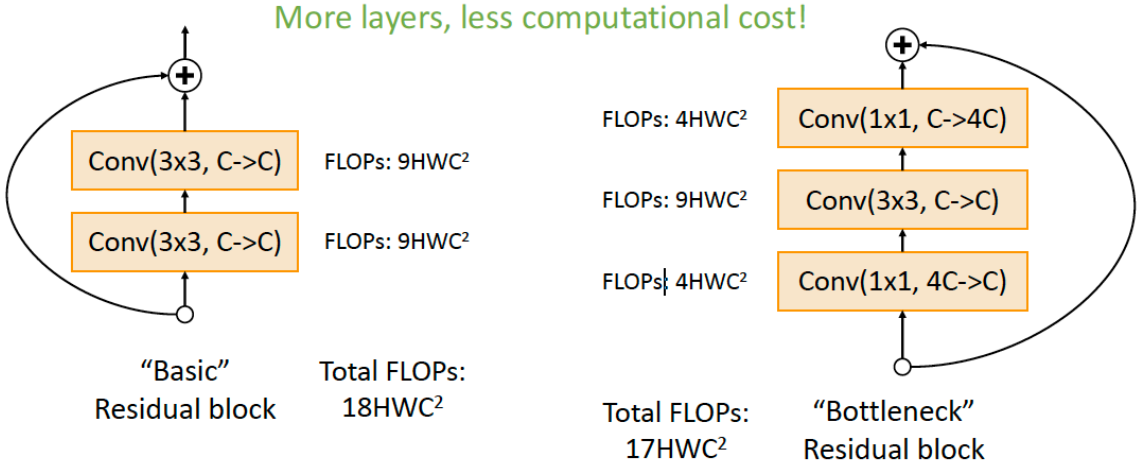
Thus deeper models should do at least as good as shallow models

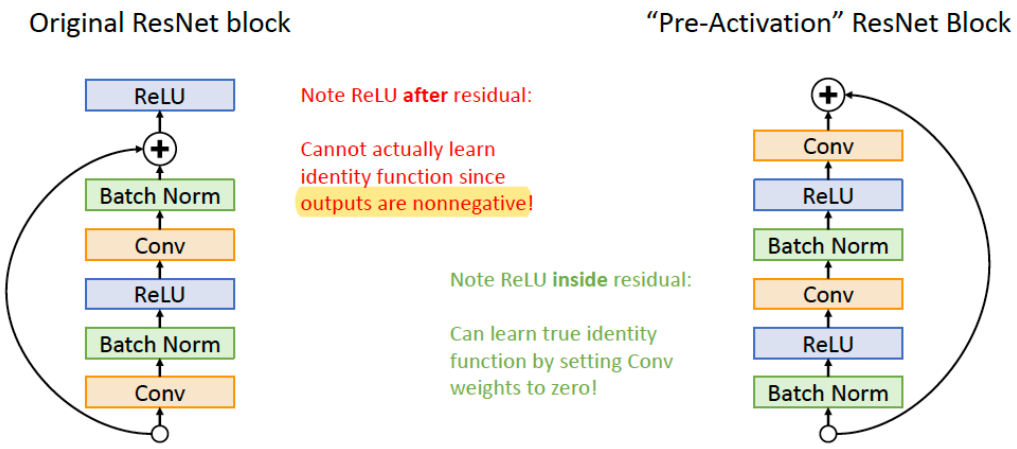
Hypothesis: This is an optimization problem. Deeper models are harder to optimize, and in particular don't learn identity functions to emulate shallow models

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



More layers, less computational cost!

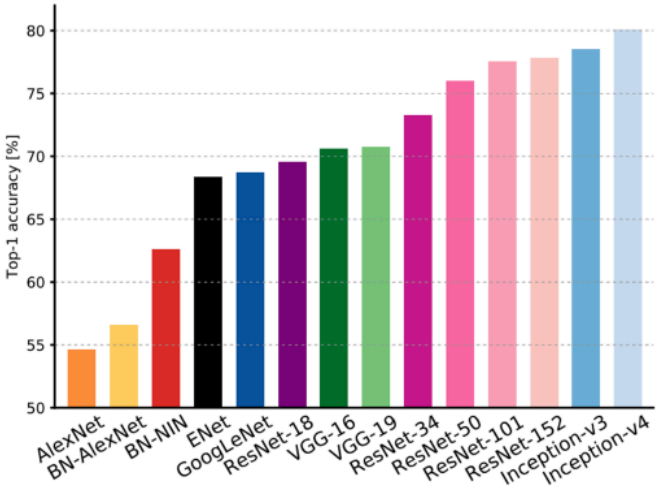


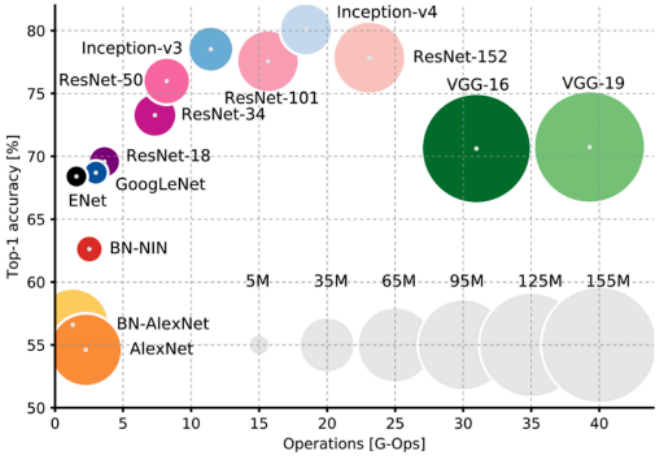


- **Properties:**
  - A residual network is a stack of many residual blocks

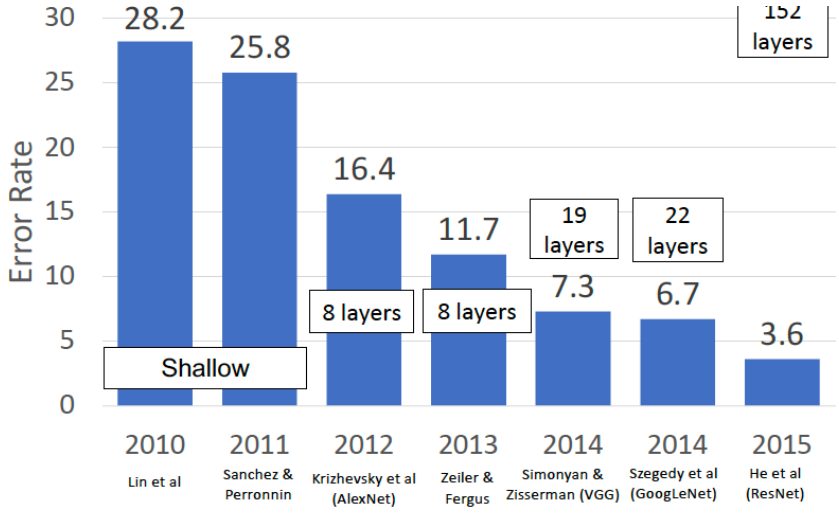
    Network is divided into stages: the first block of each stage halves the resolution (with stride-2 conv) and doubles the number of channels
  - Use the same aggressive stem as GoogleNet to **downsample** the input 4x before applying residual blocks
  - Use **global average pooling** and a single linear layer at the end (no fully-connected-layer)

6. Comparing Complexity





- Inception-v4: Resnet + Inception!
- VGG: Highest memory, most operations
- GoogLeNet: Very efficient!
- AlexNet: Low compute, lots of parameters
- ResNet: Simple design, moderate efficiency, high accuracy


Error Rate
30
25
20
15
10
5
0
28.2
25.8
16.4
11.7
7.3
6.7
3.6
Shallow
8 layers
8 layers
19 layers
22 layers
152 layers
2010
Lin et al
2011
Sanchez & Perronnin
2012
Krizhevsky et al (AlexNet)
2013
Zeiler & Fergus
2014
Simonyan & Zisserman (VGG)
2014
Szegedy et al (GoogLeNet)
2015
He et al (ResNet)