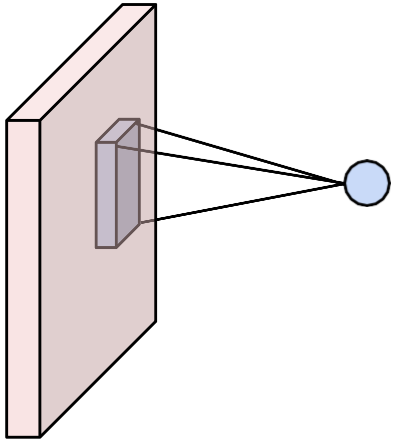
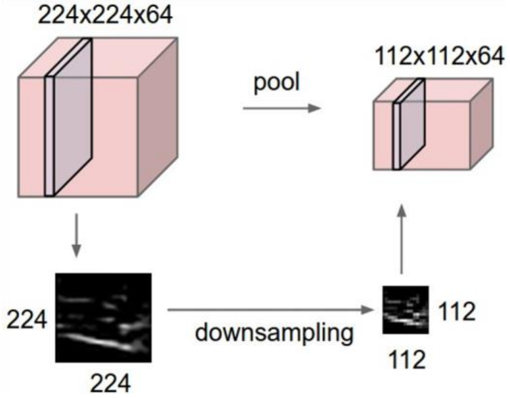
# Lecture 9:
# CNN Architectures (Part 1)

# Last Time: Components of Convolutional Networks

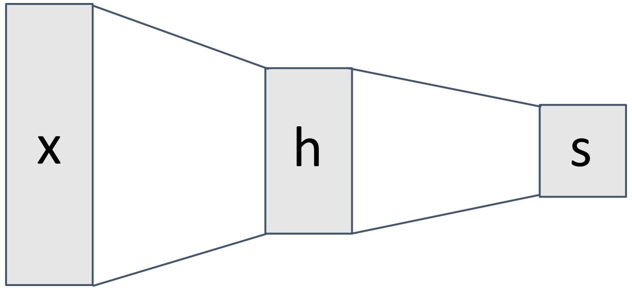
## Convolution Layers



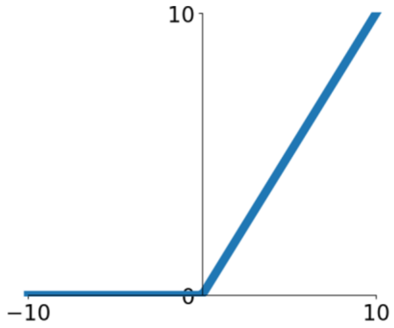
## Pooling Layers




## Fully-Connected Layers




## Activation Function




## Normalization

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

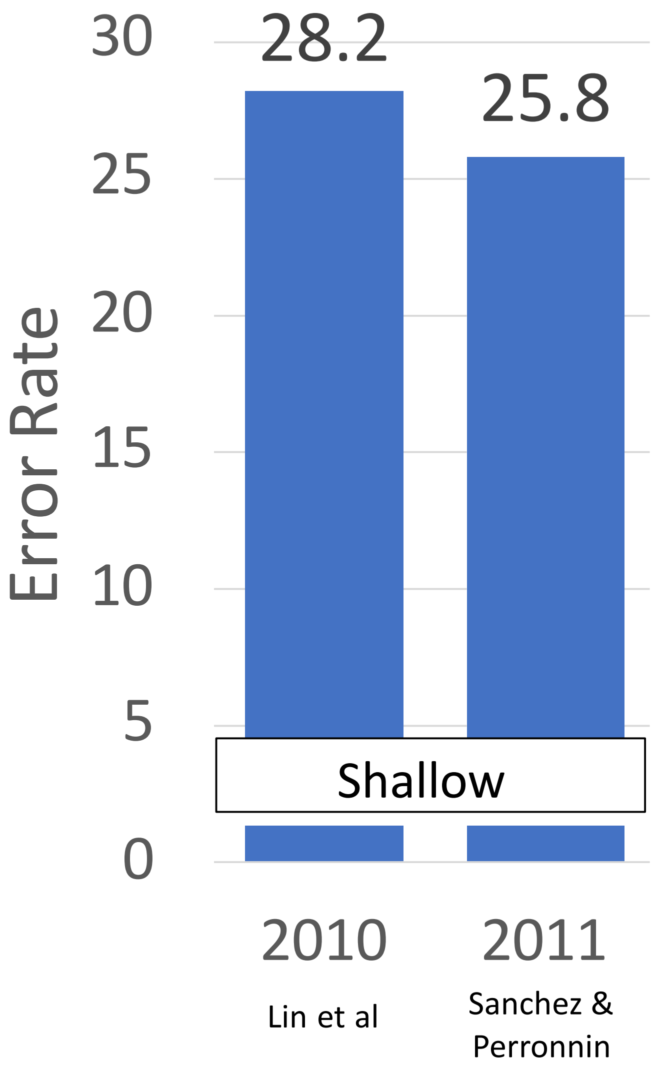
**Question**: How should we put them together?

# ILSVR

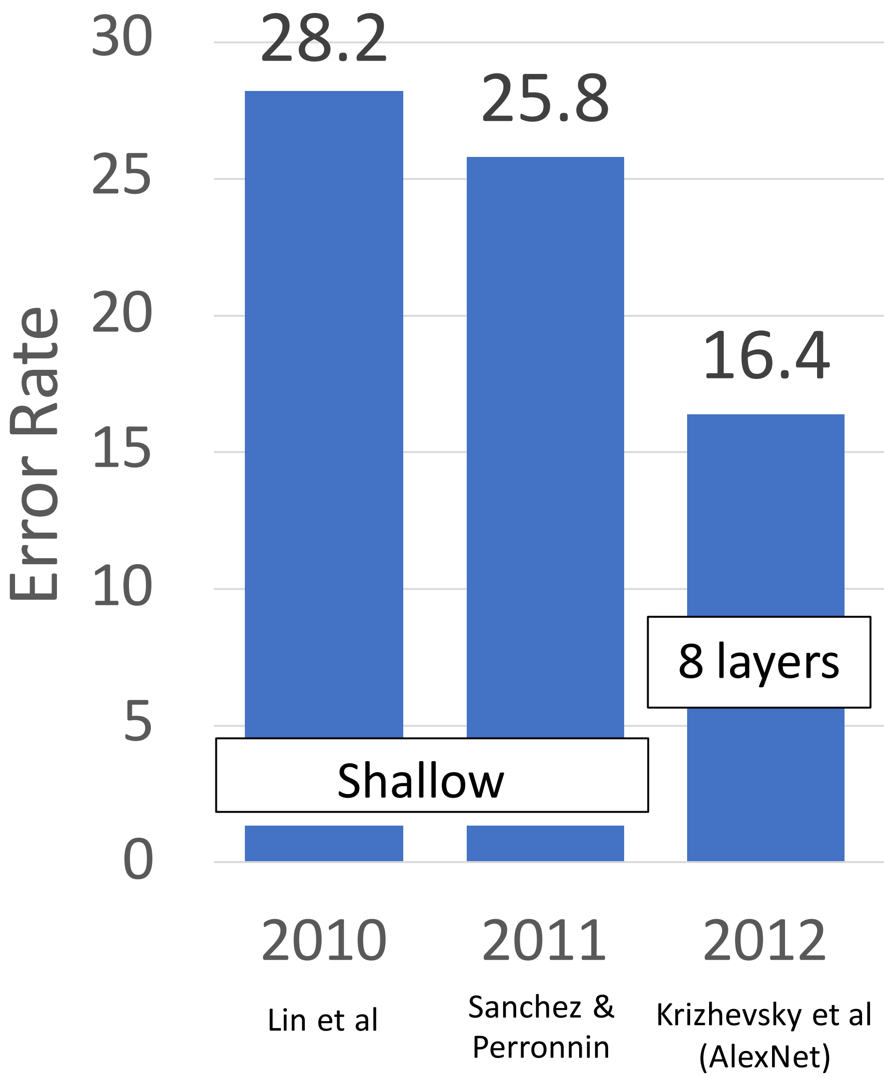
- ILSVRC（ImageNet Large Scale Visual Recognition Competition）是由ImageNet所舉辦的年度大規模視覺識別挑戰賽，自2010年開辦以來，全球各知名AI企業莫不以取得此項比賽最高名次為殊榮，以宣告其圖像辨識技術已達登峰之境。
- 剛開始是由ML及SVM等技術逐鹿，然而就在2012年，深度學習之父Hinton的高徒Alex Krizhevsky首次採用深度學習架構參與此競賽，並以極大的差距擊敗了使用SVM技術Xerox Research Centre Europe隊伍，自始以後，揭開了Deep learning吸引全球關注嶄露頭角的布幔。
- ILSVRC 競賽所使用的dataset來自於ImageNet。ILSVRC每年會從超過1400 萬張full-sized且標記的相片中取出部份樣本進行比賽。競賽中評比的Top-5 error rate分數，其計算方式是每位參賽者針對某張圖片進行預測，所給出的五個最有可能的預測中若有一個為正確就算答對，若沒有一個正確則算錯誤。

https://chtseng.wordpress.com/2017/11/20/ilsvrc-%E6%AD%B7%E5%B1%86%E7%9A%84%E6%B7%B1%E5%BA%A6%E5%AD%B8%E7%BF%92%E6%A8%A1%E5%9E%8B/

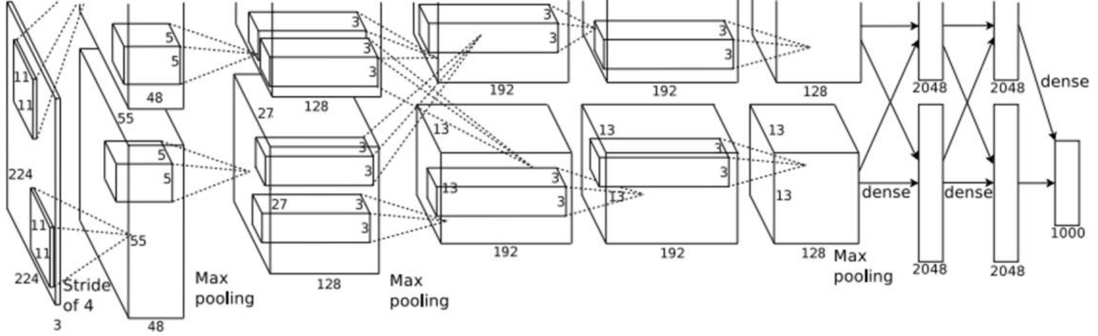
# ImageNet Classification Challenge



# ImageNet Classification Challenge

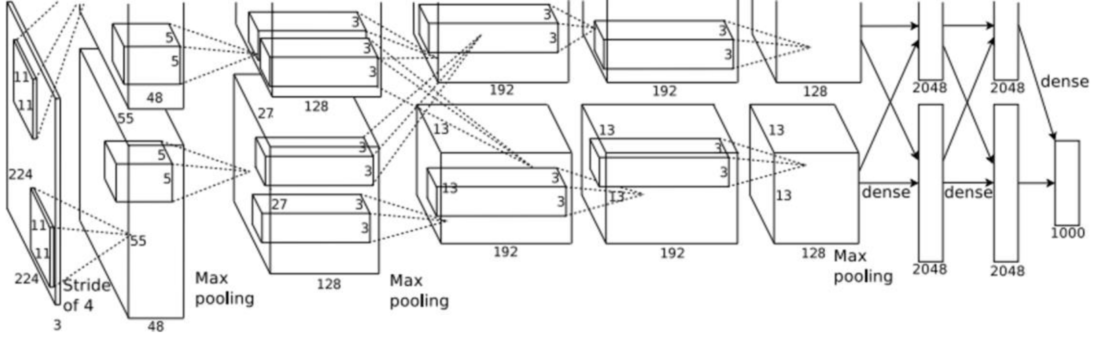


# AlexNet




224 x 224 inputs
5 Convolutional layers
Max pooling
3 fully-connected layers
ReLU nonlinearities

# AlexNet

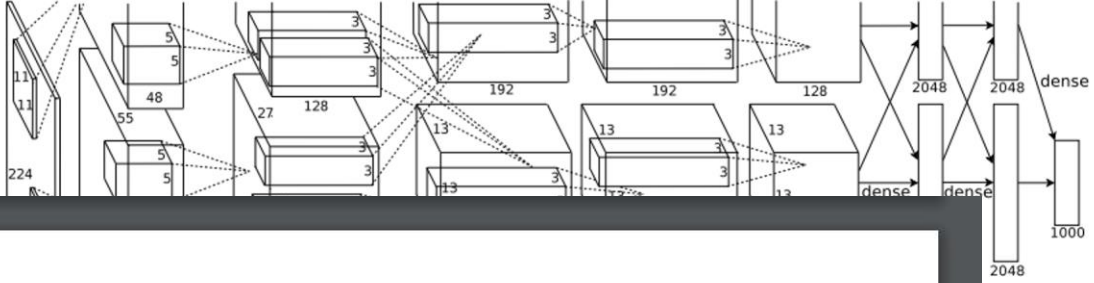



224 x 224 inputs
5 Convolutional layers
Max pooling
3 fully-connected layers
ReLU nonlinearities

Used “Local response normalization”; Not used anymore

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

# AlexNet



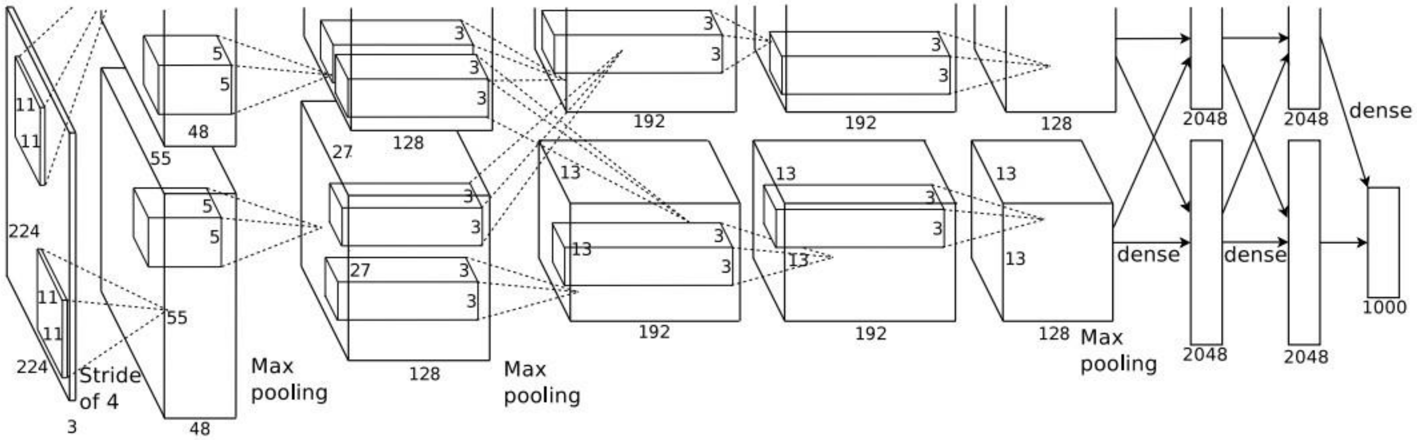
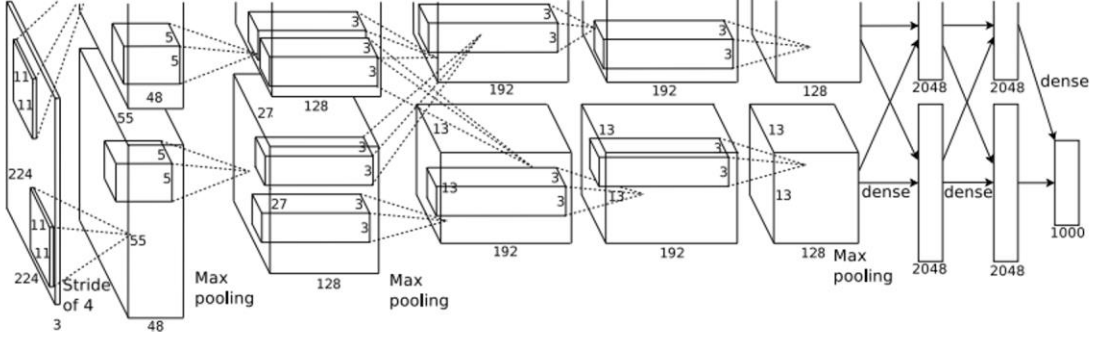
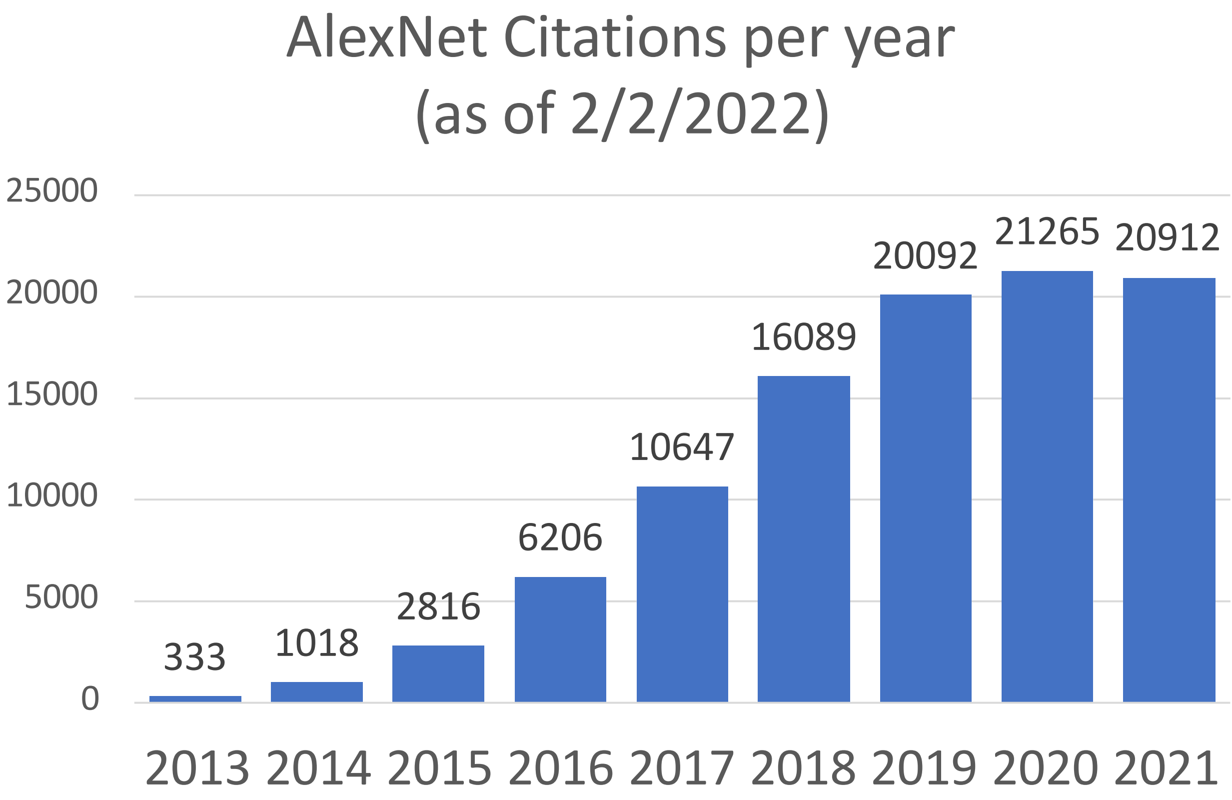




Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input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

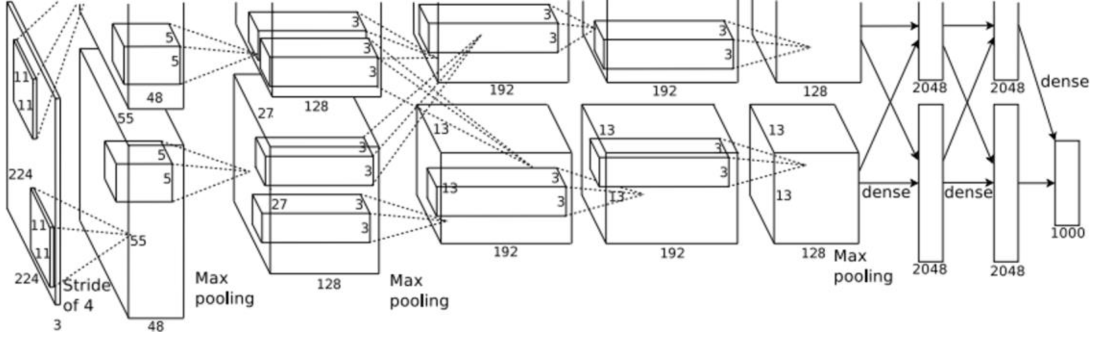
# AlexNet

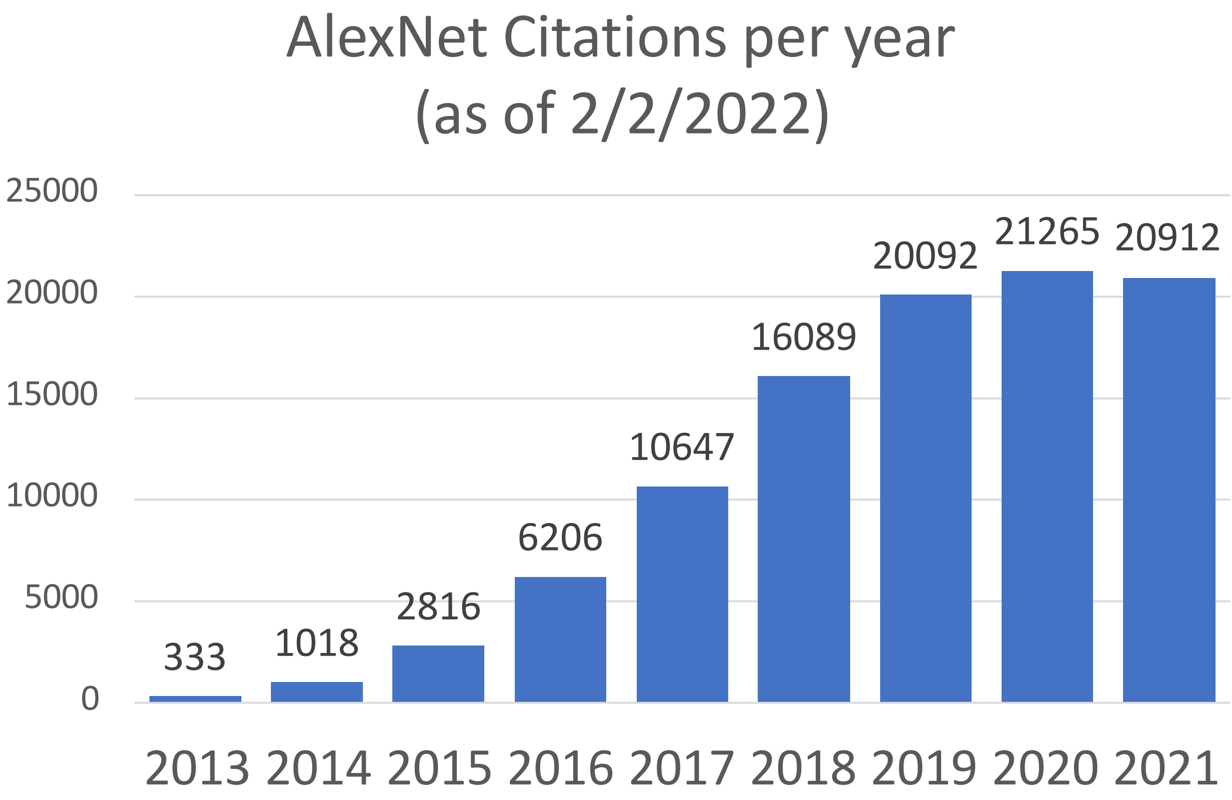



Total Citations: **102,486**

# AlexNet




Total Citations: **102,486**

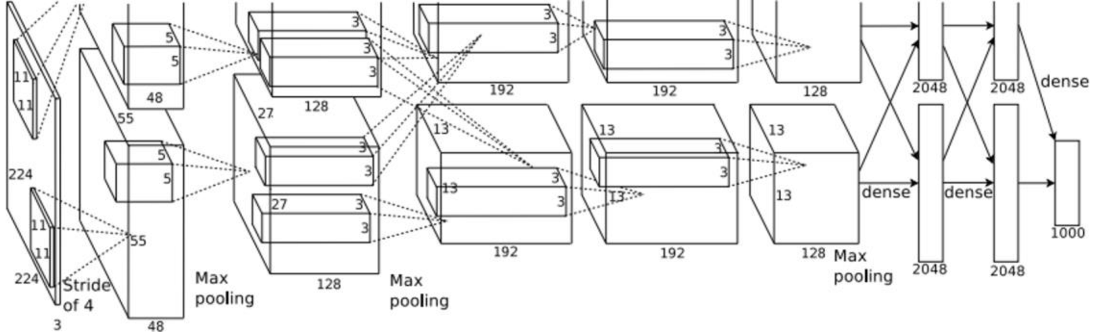
**Citation Counts**

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

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

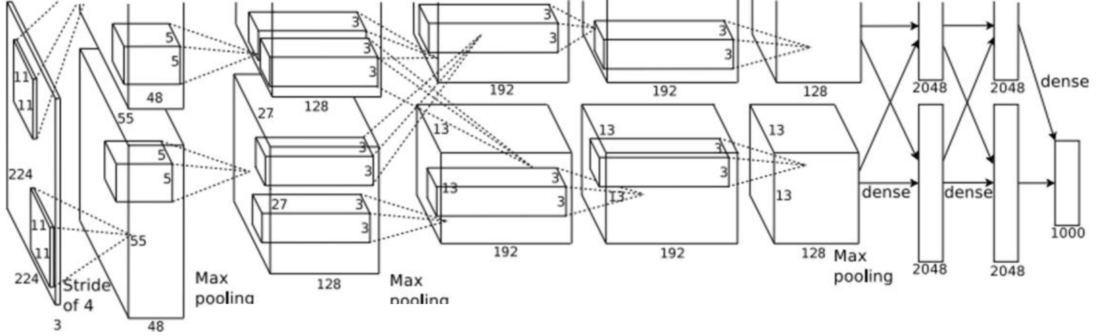
Watson and Crick, "Molecular Structure of Nucleic Acids", 1953: **16,298**

# AlexNet



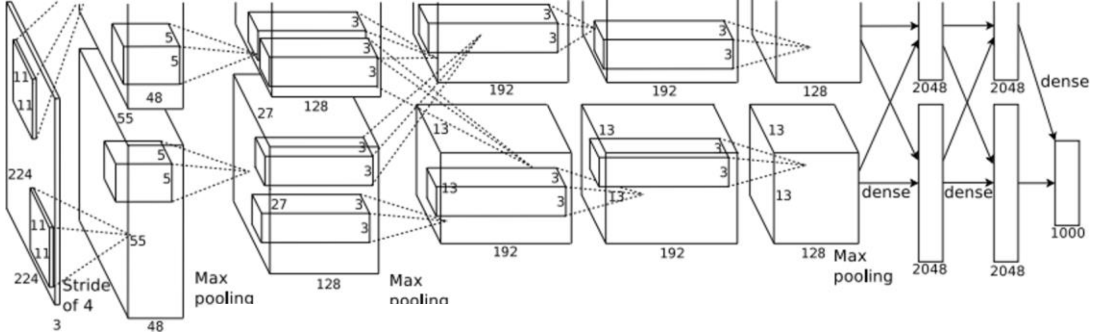

| | Input size | | Layer | | | | Output size | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Layer** | C | H / W | filters | kernel | stride | pad | C | H / W | memory (KB) | params (k) | flop (M) |
| conv1 | 3 | 224 | 96 | 11 | 4 | 2 | 96 | 55 | 1134 | 34 | 105 |
| pool1 | 96 | 55 | | 3 | 2 | 0 | 96 | 27 | 273 | 0 | 0 |
| conv2 | 96 | 27 | 256 | 5 | 1 | 2 | 256 | 27 | 729 | 614 | 447 |
| pool2 | 256 | 27 | | 3 | 2 | 0 | 256 | 13 | 169 | 0 | 0 |
| conv3 | 256 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 253 | 885 | 149 |
| conv4 | 384 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 253 | 1327 | 224 |
| conv5 | 256 | 13 | 256 | 3 | 1 | 1 | 256 | 13 | 169 | 590 | 99 |
| pool5 | 256 | 13 | | 3 | 2 | 0 | 256 | 6 | 36 | 0 | 0 |
| flatten | 256 | 6 | | | | | 9216 | | 36 | 0 | 0 |
| fc6 | 9216 | | 4096 | | | | 4096 | | 16 | 36,868 | 36 |
| fc7 | 4096 | | 4096 | | | | 4096 | | 16 | 16,388 | 16 |
| fc8 | 4096 | | 1000 | | | | 1000 | | 4 | 4,001 | 4 |

# AlexNet




| | Input size | | Layer | | | | Output size | |
|---|---|---|---|---|---|---|---|---|
| **Layer** | C | H / W | filters | kernel | stride | pad | C | H / W |
| conv1 | 3 | 224 | 96 | 11 | 4 | 2 | ? | |

# AlexNet

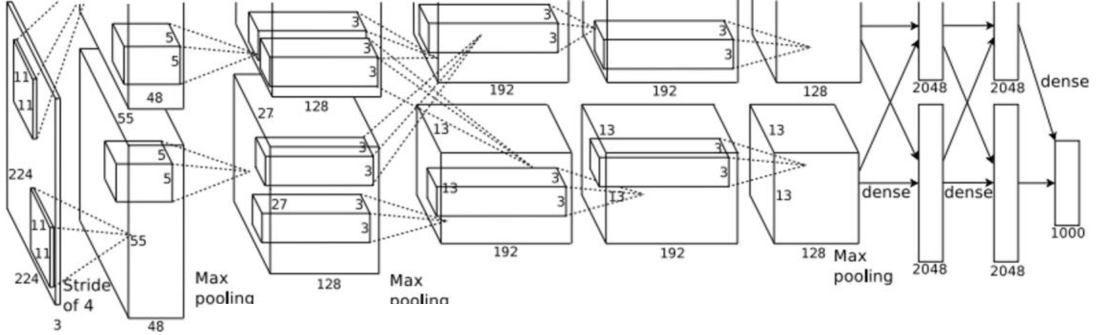



| | Input size | | Layer | | | | Output size | |
|---|---|---|---|---|---|---|---|---|
| **Layer** | C | H / W | filters | kernel | stride | pad | C | H / W |
| conv1 | 3 | 224 | 96 | 11 | 4 | 2 | 96 | ? |

Recall: Output channels = number of filters

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

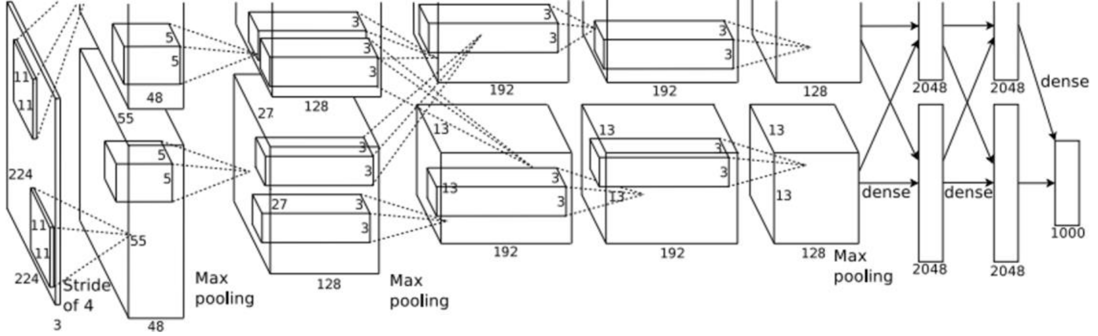
# AlexNet




| | Input size | | Layer | | | | Output size | |
|---|---|---|---|---|---|---|---|---|
| **Layer** | C | H / W | filters | kernel | stride | pad | C | H / W |
| conv1 | 3 | 224 | 96 | 11 | 4 | 2 | 96 | 55 |

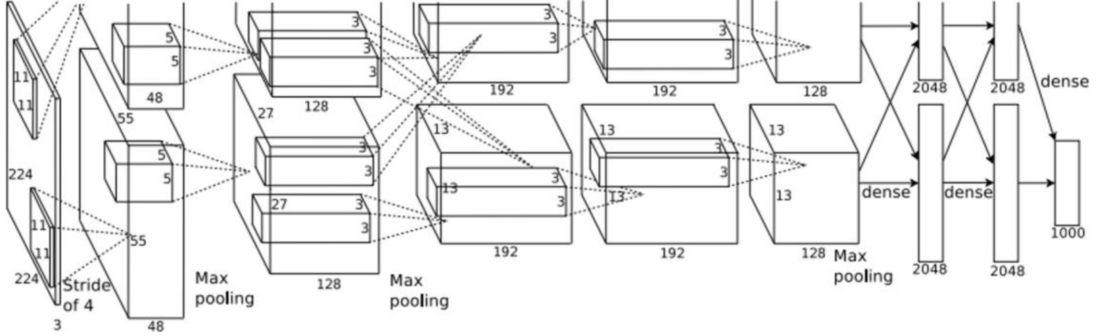
Recall: W' = (W – K + 2P) / S + 1
= (224 – 11 + 2*2) / 4 + 1
= 217/4 + 1 = 55

# AlexNet




| | Input size | | Layer | | | | Output size | | |
|---|---|---|---|---|---|---|---|---|---|
| **Layer** | C | H / W | filters | kernel | stride | pad | C | H / W | memory (KB) |
| conv1 | 3 | 227 | 96 | 11 | 4 | 2 | 96 | 55 | ? |

# AlexNet



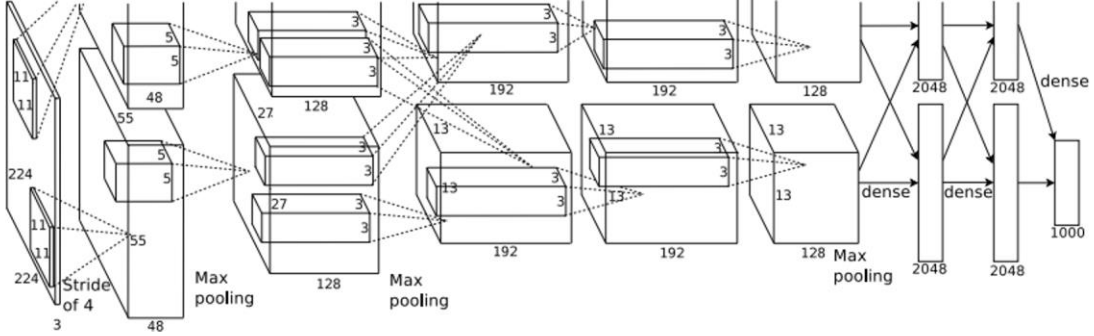

| | Input size | | Layer | | | | Output size | | |
|---|---|---|---|---|---|---|---|---|---|
| **Layer** | C | H / W | filters | kernel | stride | pad | C | H / W | memory (KB) |
| conv1 | 3 | 227 | 96 | 11 | 4 | 2 | 96 | 55 | 1134 |

Number of output elements = C * H' * W'
= 96*55*55 = 290,400

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

KB = (number of elements) * (bytes per elem) / 1024
= 290400 * 4 / 1024
= **1134**

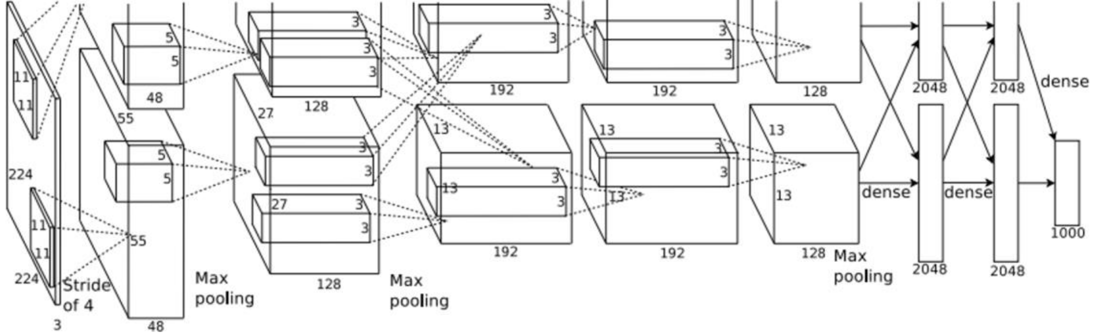
# AlexNet




| | Input size | | Layer | | | | Output size | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Layer** | C | H / W | filters | kernel | stride | pad | C | H / W | memory (KB) | params (k) |
| conv1 | 3 | 227 | 96 | 11 | 4 | 2 | 96 | 56 | 1176 | ? |

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

# AlexNet



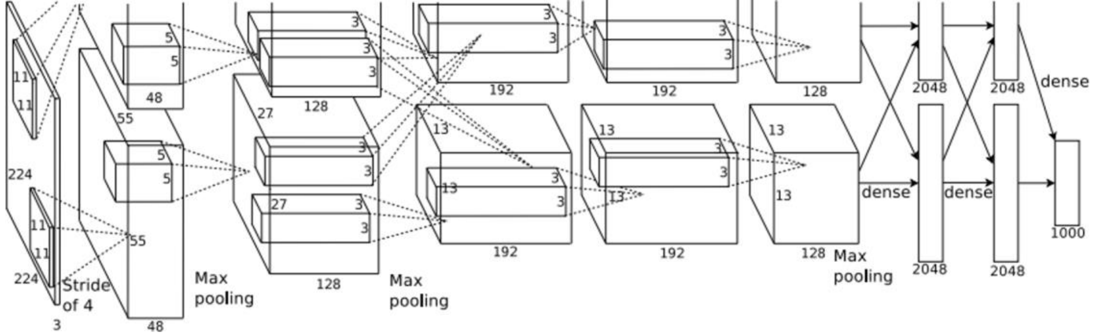

| | Input size | | Layer | | | | Output size | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Layer** | C | H / W | filters | kernel | stride | pad | C | H / W | memory (KB) | params (k) |
| conv1 | 3 | 224 | 96 | 11 | 4 | 2 | 96 | 55 | 1134 | 34 |

Weight shape = $C_{out}$ x $C_{in}$ x K x K
= 96 x 3 x 11 x 11

Bias shape = $C_{out}$ = 96

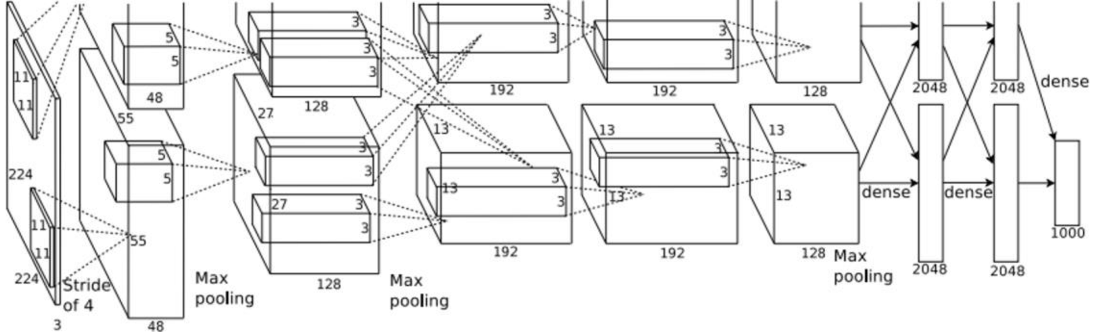
Number of weights = 96*3*11*11 + 96
= **34,942**

# 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 | 224 | 96 | 11 | 4 | 2 | 96 | 55 | 1134 | 34 | ? |

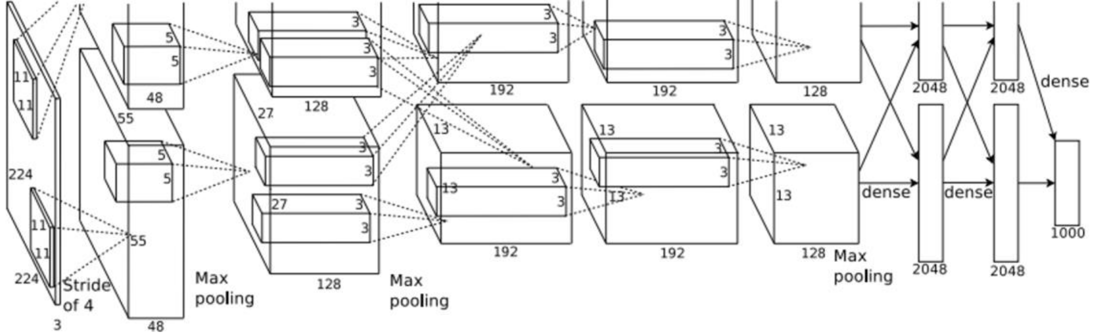
# AlexNet



| | Input size | | Layer | | | | Output size | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Layer** | C | H / W | filters | kernel | stride | pad | C | H / W | memory (KB) | params (k) | flop (M) |
| conv1 | 3 | 224 | 96 | 11 | 4 | 2 | 96 | 55 | 1134 | 34 | 105 |

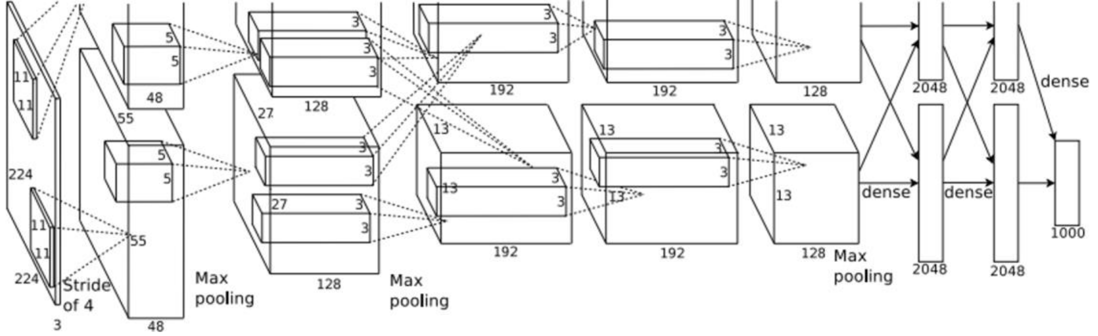
Number of floating point operations (multiply+add)
= (number of output elements) * (ops per output elem)
= ($C_{out}$ x H' x W') * ($C_{in}$ x K x K)
= (96 * 55 * 55) * (3 * 11 * 11)
= 290,400 * 363
= **105,415,200**

# 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 | 224 | 96 | 11 | 4 | 2 | 96 | 55 | 1134 | 34 | 105 |
| pool1 | 96 | 55 | | 3 | 2 | 0 | | | | | |

# AlexNet



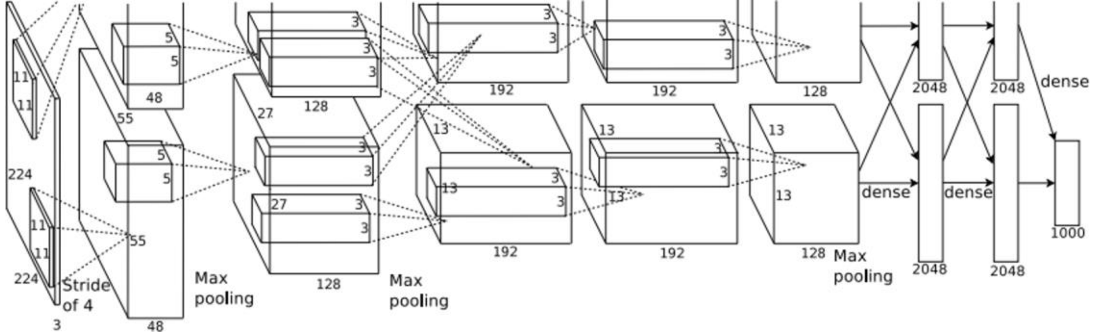

| | Input size | | Layer | | | | Output size | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Layer** | C | H / W | filters | kernel | stride | pad | C | H / W | memory (KB) | params (k) | flop (M) |
| conv1 | 3 | 224 | 96 | 11 | 4 | 2 | 96 | 55 | 1134 | 34 | 105 |
| pool1 | 96 | 55 | | 3 | 2 | 0 | 96 | 27 | | | |

For pooling layer:

#output channels = #input channels = 64

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

# 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 | 224 | 96 | 11 | 4 | 2 | 96 | 55 | 1134 | 34 | 105 |
| pool1 | 96 | 55 | | 3 | 2 | 0 | 96 | 27 | ? | | |

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

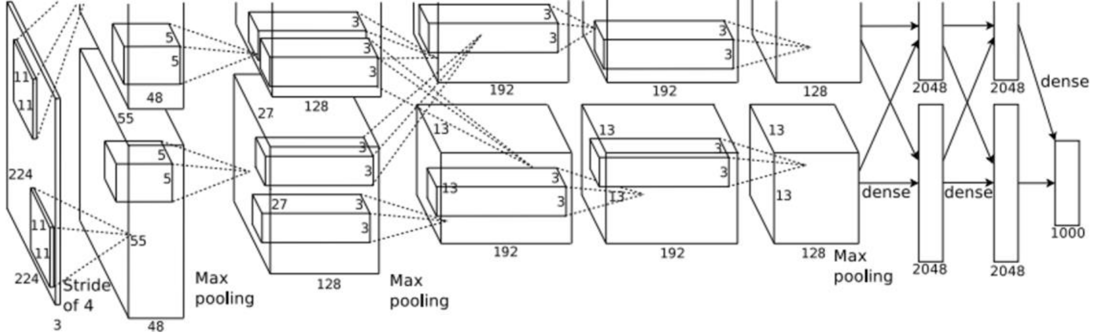
Bytes per elem = 4

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

=96 * 27 * 27 * 4 / 1024

= **273.375**

# 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 | 224 | 96 | 11 | 4 | 2 | 96 | 55 | 1134 | 34 | 105 |
| pool1 | 96 | 55 | | 3 | 2 | 0 | 96 | 27 | 273 | ? | |

# AlexNet

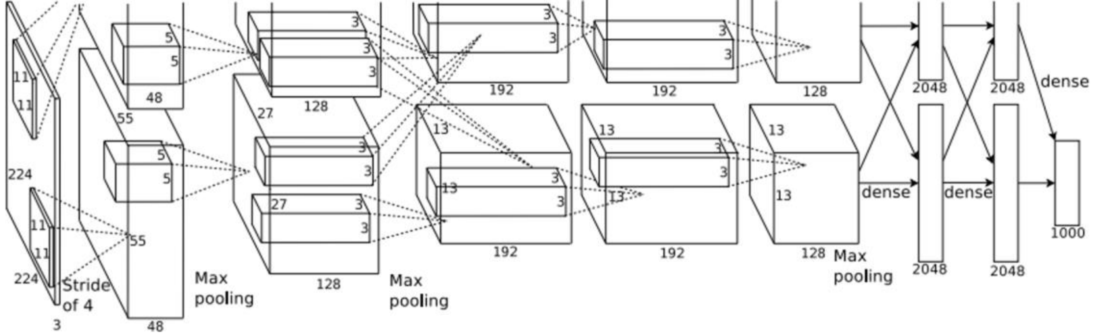
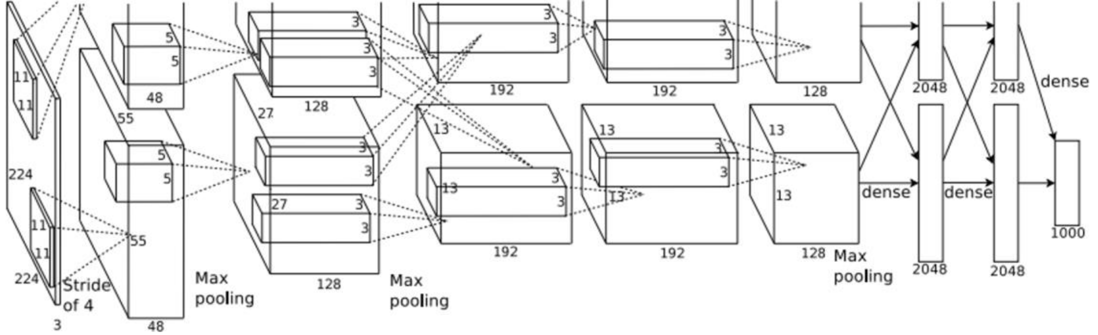


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

| | Input size | | Layer | | | | Output size | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Layer** | C | H / W | filters | kernel | stride | pad | C | H / W | memory (KB) | params (k) | flop (M) |
| conv1 | 3 | 224 | 96 | 11 | 4 | 2 | 96 | 55 | 1134 | 34 | 105 |
| pool1 | 96 | 55 | | 3 | 2 | 0 | 96 | 27 | 273 | 0 | ? |

Pooling layers have no learnable parameters!

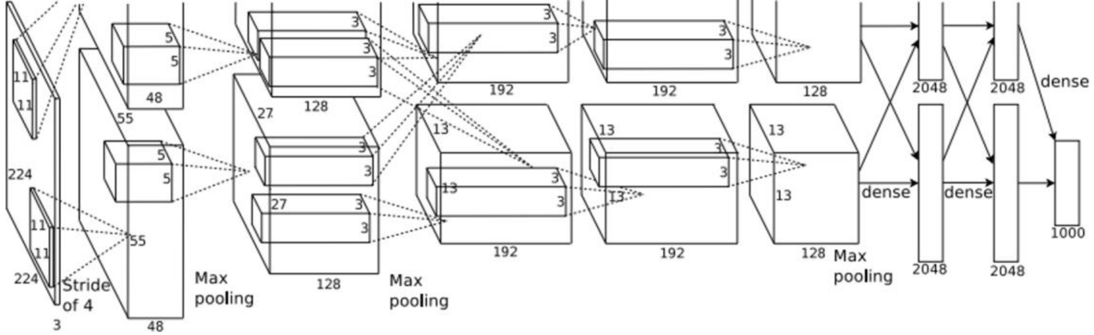
# AlexNet




| | Input size | | Layer | | | | Output size | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Layer** | C | H / W | filters | kernel | stride | pad | C | H / W | memory (KB) | params (k) | flop (M) |
| conv1 | 3 | 224 | 96 | 11 | 4 | 2 | 96 | 55 | 1134 | 34 | 105 |
| pool1 | 96 | 55 | | 3 | 2 | 0 | 96 | 27 | 273 | 0 | 0 |

No floating-point ops for pooling layer!

# AlexNet



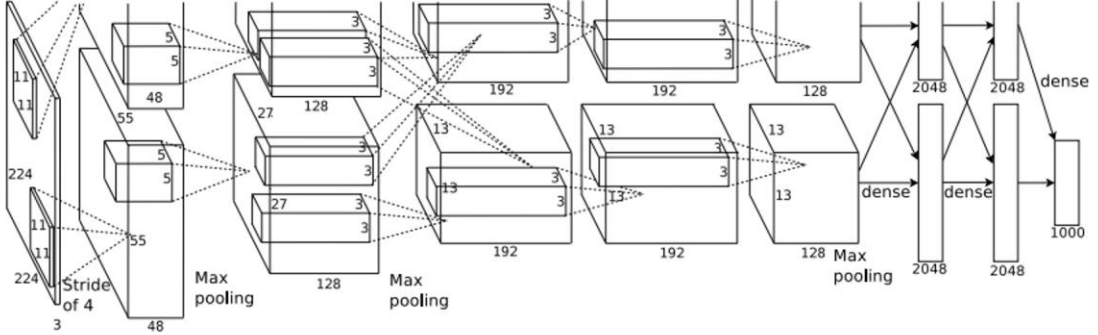

| | Input size | | Layer | | | | Output size | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Layer** | C | H / W | filters | kernel | stride | pad | C | H / W | memory (KB) | params (k) | flop (M) |
| conv1 | 3 | 224 | 96 | 11 | 4 | 2 | 96 | 55 | 1134 | 34 | 105 |
| pool1 | 96 | 55 | | 3 | 2 | 0 | 96 | 27 | 273 | 0 | 0 |
| conv2 | 96 | 27 | 256 | 5 | 1 | 2 | 256 | 27 | 729 | 614 | 447 |
| pool2 | 256 | 27 | | 3 | 2 | 0 | 256 | 13 | 169 | 0 | 0 |
| conv3 | 256 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 253 | 885 | 149 |
| conv4 | 384 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 253 | 1327 | 224 |
| conv5 | 256 | 13 | 256 | 3 | 1 | 1 | 256 | 13 | 169 | 590 | 99 |
| 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

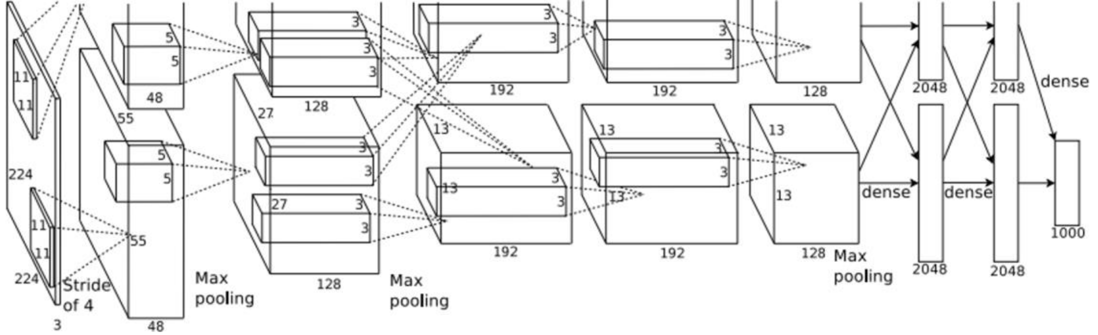



| | Input size | | Layer | | | | Output size | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Layer** | C | H / W | filters | kernel | stride | pad | C | H / W | memory (KB) | params (k) | flop (M) |
| conv1 | 3 | 224 | 96 | 11 | 4 | 2 | 96 | 55 | 1134 | 34 | 105 |
| pool1 | 96 | 55 | | 3 | 2 | 0 | 96 | 27 | 273 | 0 | 0 |
| conv2 | 96 | 27 | 256 | 5 | 1 | 2 | 256 | 27 | 729 | 614 | 447 |
| pool2 | 256 | 27 | | 3 | 2 | 0 | 256 | 13 | 169 | 0 | 0 |
| conv3 | 256 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 253 | 885 | 149 |
| conv4 | 384 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 253 | 1327 | 224 |
| conv5 | 256 | 13 | 256 | 3 | 1 | 1 | 256 | 13 | 169 | 590 | 99 |
| pool5 | 256 | 13 | | 3 | 2 | 0 | 256 | 6 | 36 | 0 | 0 |
| flatten | 256 | 6 | | | | | 9216 | | 36 | 0 | 0 |
| fc6 | 9216 | | 4096 | | | | 4096 | | 16 | 36,868 | 36 |

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

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

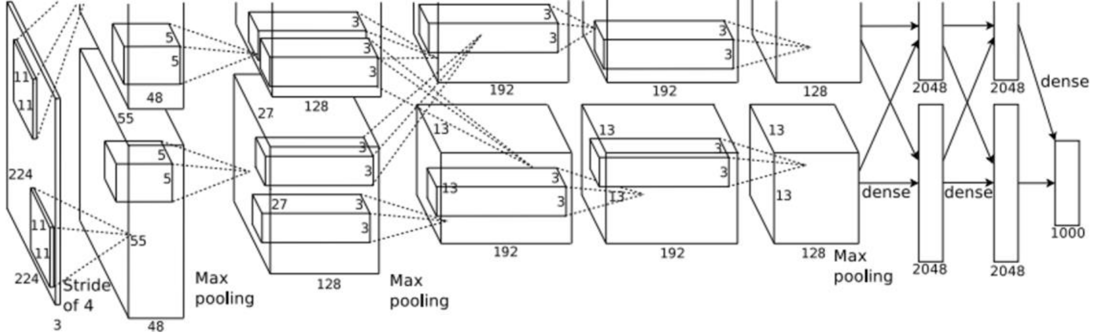
# AlexNet




| | Input size | | Layer | | | | Output size | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Layer** | C | H / W | filters | kernel | stride | pad | C | H / W | memory (KB) | params (k) | flop (M) |
| conv1 | 3 | 224 | 96 | 11 | 4 | 2 | 96 | 55 | 1134 | 34 | 105 |
| pool1 | 96 | 55 | | 3 | 2 | 0 | 96 | 27 | 273 | 0 | 0 |
| conv2 | 96 | 27 | 256 | 5 | 1 | 2 | 256 | 27 | 729 | 614 | 447 |
| pool2 | 256 | 27 | | 3 | 2 | 0 | 256 | 13 | 169 | 0 | 0 |
| conv3 | 256 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 253 | 885 | 149 |
| conv4 | 384 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 253 | 1327 | 224 |
| conv5 | 256 | 13 | 256 | 3 | 1 | 1 | 256 | 13 | 169 | 590 | 99 |
| pool5 | 256 | 13 | | 3 | 2 | 0 | 256 | 6 | 36 | 0 | 0 |
| flatten | 256 | 6 | | | | | 9216 | | 36 | 0 | 0 |
| fc6 | 9216 | | 4096 | | | | 4096 | | 16 | 36,868 | 36 |
| fc7 | 4096 | | 4096 | | | | 4096 | | 16 | 16,388 | 16 |
| fc8 | 4096 | | 1000 | | | | 1000 | | 4 | 4,001 | 4 |

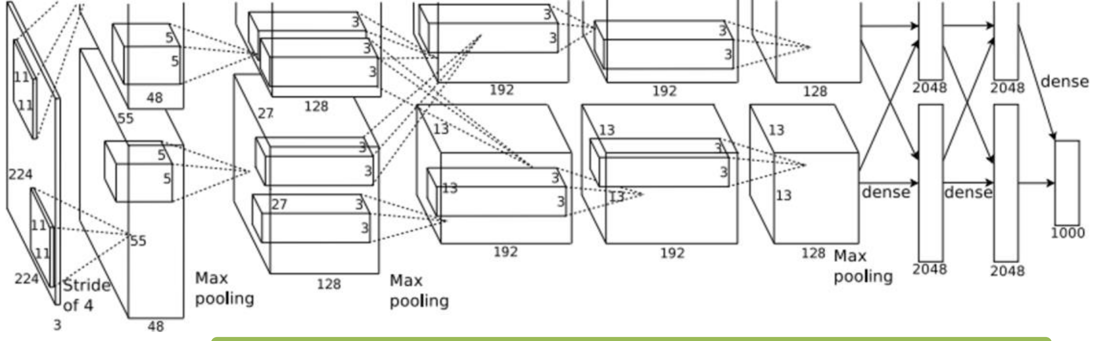
# AlexNet

How to choose this?
Trial and error =(



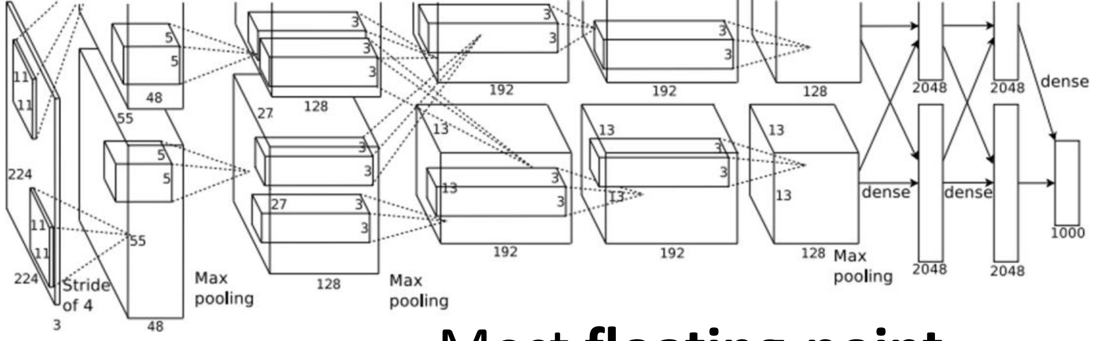

| | Input size | | Layer | | | | Output size | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Layer** | C | H / W | filters | kernel | stride | pad | C | H / W | memory (KB) | params (k) | flop (M) |
| conv1 | 3 | 224 | 96 | 11 | 4 | 2 | 96 | 55 | 1134 | 34 | 105 |
| pool1 | 96 | 55 | | 3 | 2 | 0 | 96 | 27 | 273 | 0 | 0 |
| conv2 | 96 | 27 | 256 | 5 | 1 | 2 | 256 | 27 | 729 | 614 | 447 |
| pool2 | 256 | 27 | | 3 | 2 | 0 | 256 | 13 | 169 | 0 | 0 |
| conv3 | 256 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 253 | 885 | 149 |
| conv4 | 384 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 253 | 1327 | 224 |
| conv5 | 256 | 13 | 256 | 3 | 1 | 1 | 256 | 13 | 169 | 590 | 99 |
| pool5 | 256 | 13 | | 3 | 2 | 0 | 256 | 6 | 36 | 0 | 0 |
| flatten | 256 | 6 | | | | | 9216 | | 36 | 0 | 0 |
| fc6 | 9216 | | 4096 | | | | 4096 | | 16 | 36,868 | 36 |
| fc7 | 4096 | | 4096 | | | | 4096 | | 16 | 16,388 | 16 |
| fc8 | 4096 | | 1000 | | | | 1000 | | 4 | 4,001 | 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 | 96 | 11 | 4 | 2 | 96 | 56 | 1176 | 34 | 73 |
| pool1 | 96 | 56 | | 3 | 2 | 0 | 96 | 27 | 273 | 0 | 0 |
| conv2 | 96 | 27 | 256 | 5 | 1 | 2 | 256 | 27 | 729 | 409 | 224 |
| pool2 | 192 | 27 | | 3 | 2 | 0 | 256 | 13 | 169 | 0 | 0 |
| conv3 | 192 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 253 | 663 | 112 |
| conv4 | 384 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 253 | 1327 | 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 |



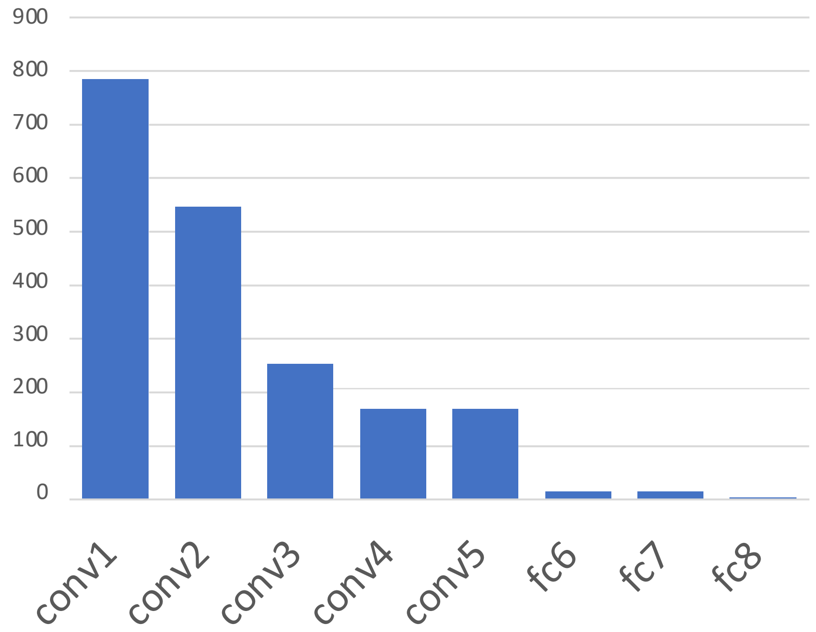

# AlexNet

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

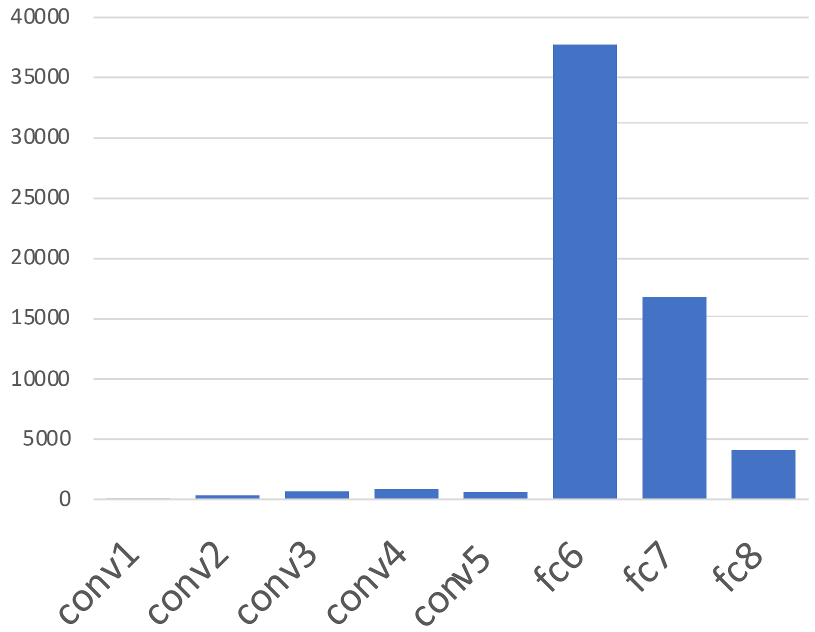
Nearly all **parameters** are in the fully-connected layers

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

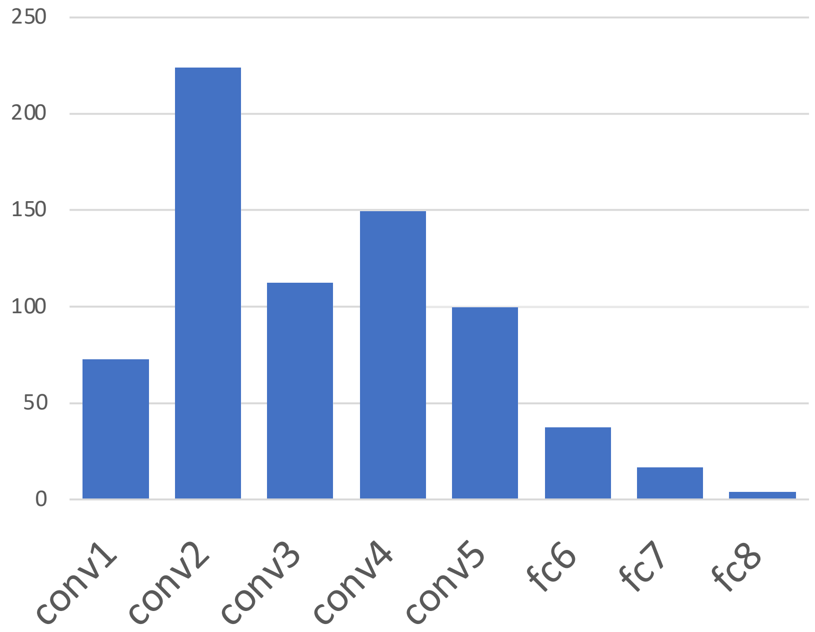
Memory (KB)



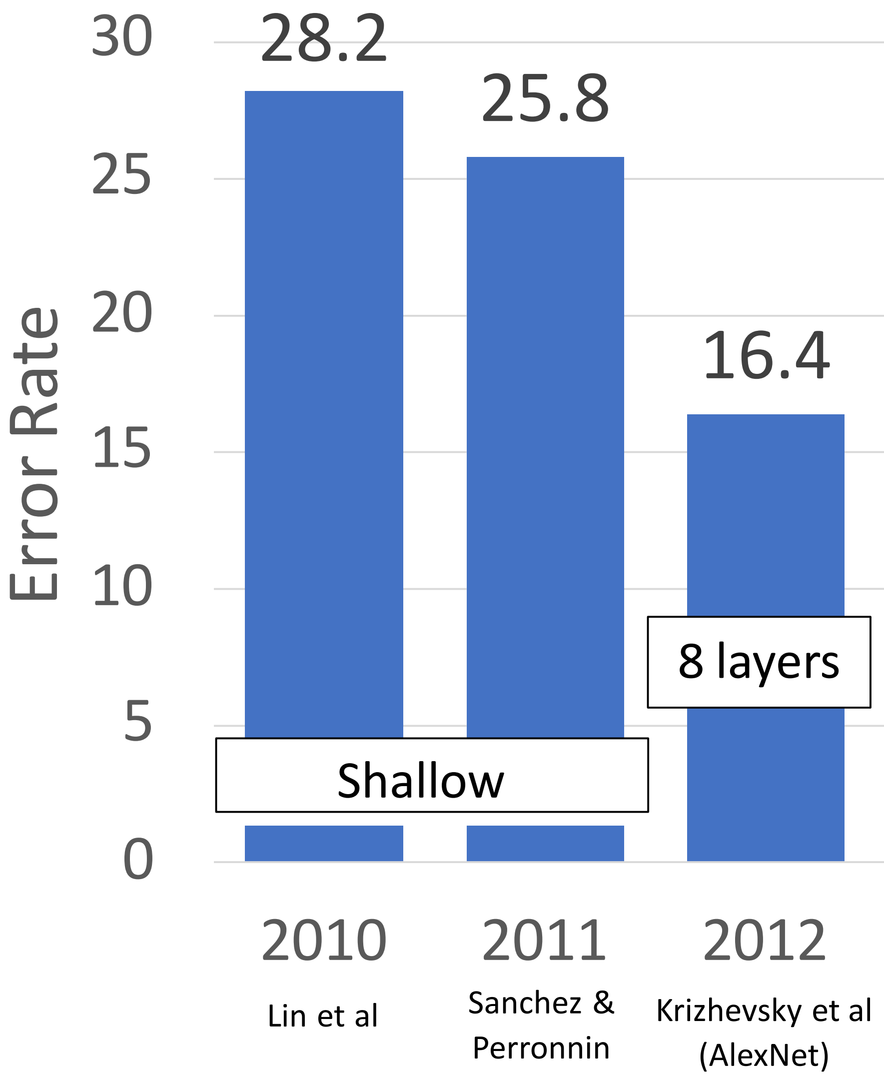

Params (K)



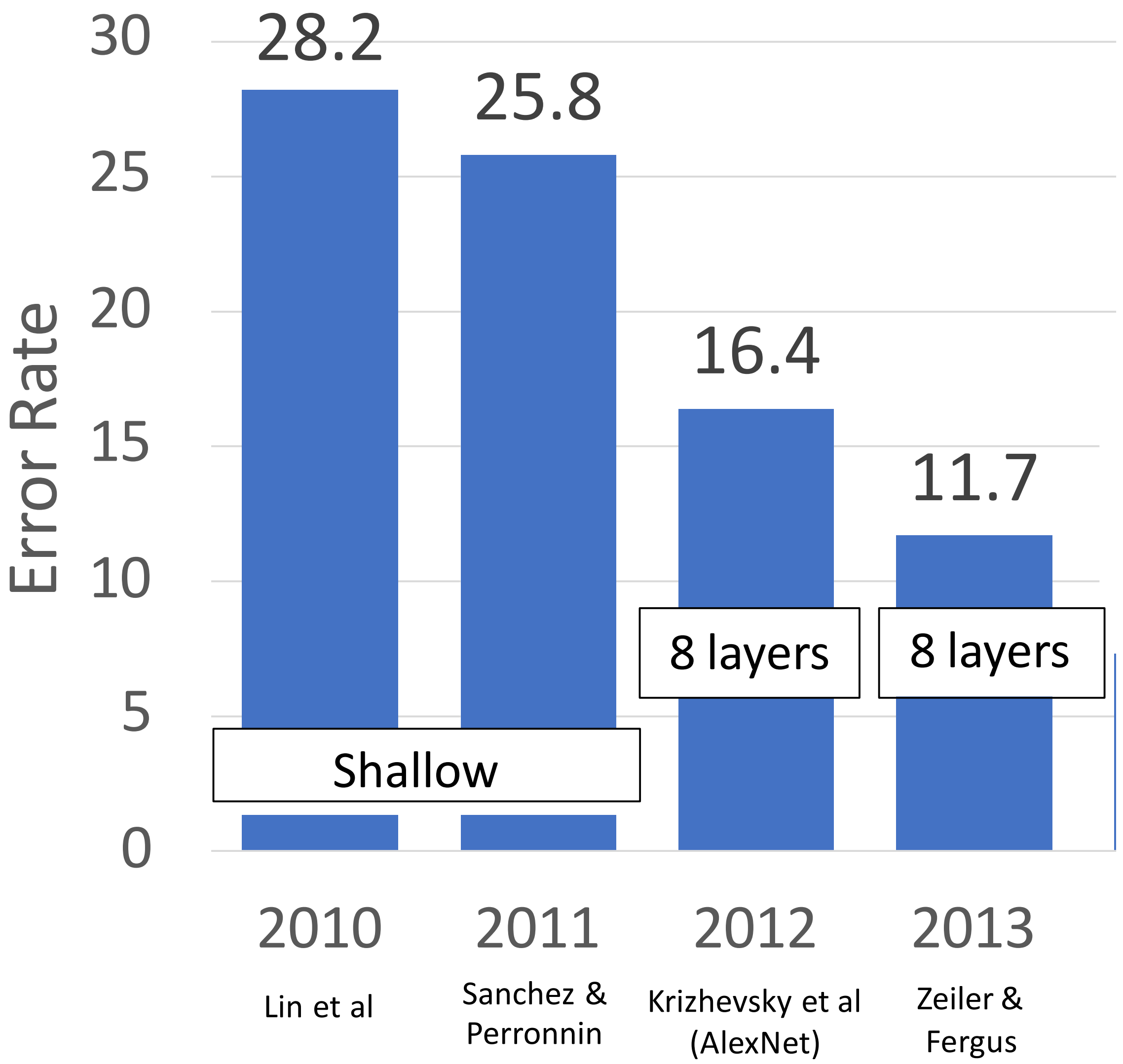

MFLOP



# ImageNet Classification Challenge

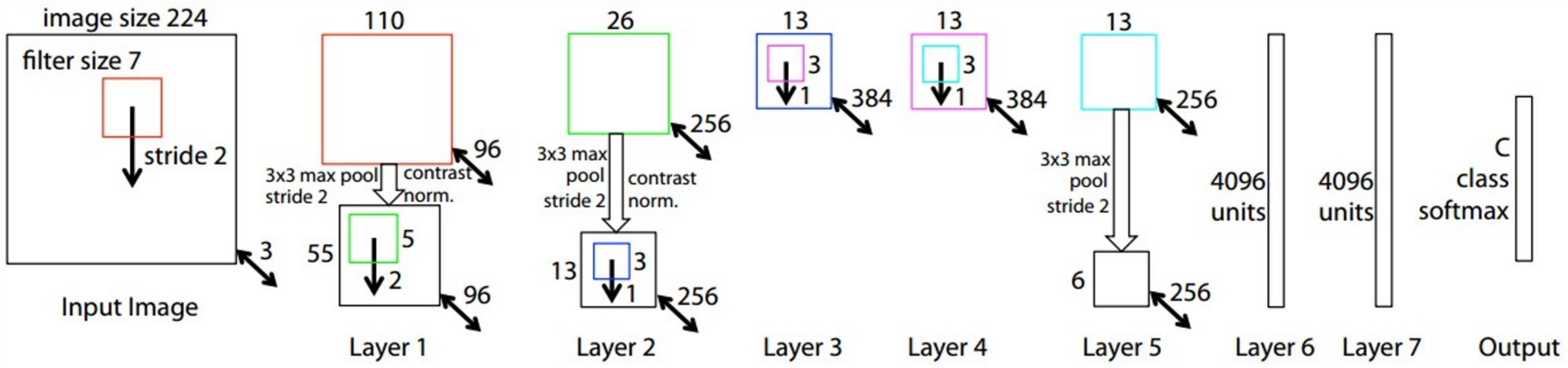


# ImageNet Classification Challenge



# ZFNet: A Bigger AlexNet

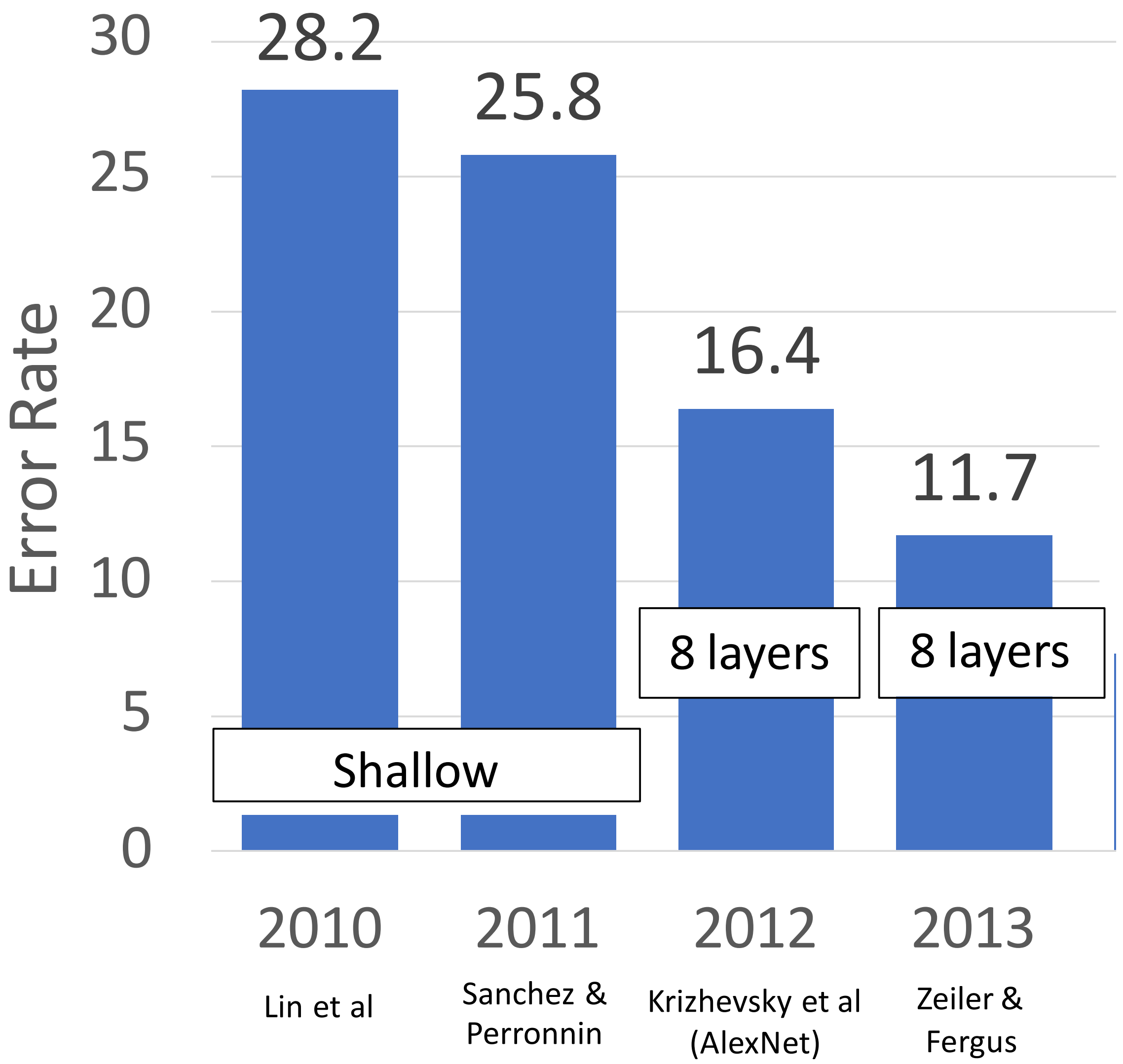
ImageNet top 5 error: 16.4% -> 11.7%



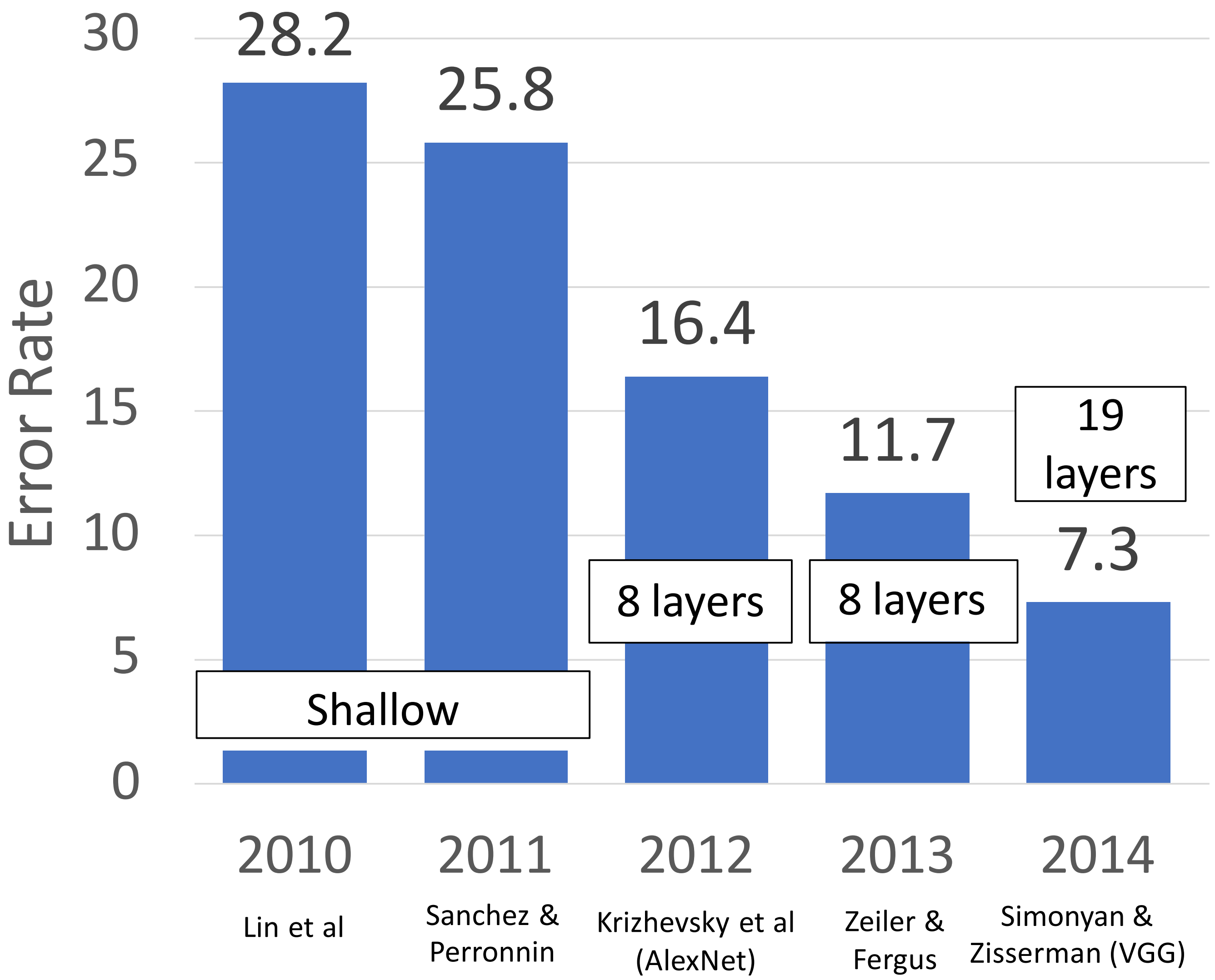

AlexNet but:
CONV1: change from (11x11 stride 4) to (7x7 stride 2)
CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512
More trial and error =(

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

# ImageNet Classification Challenge



# ImageNet Classification Challenge



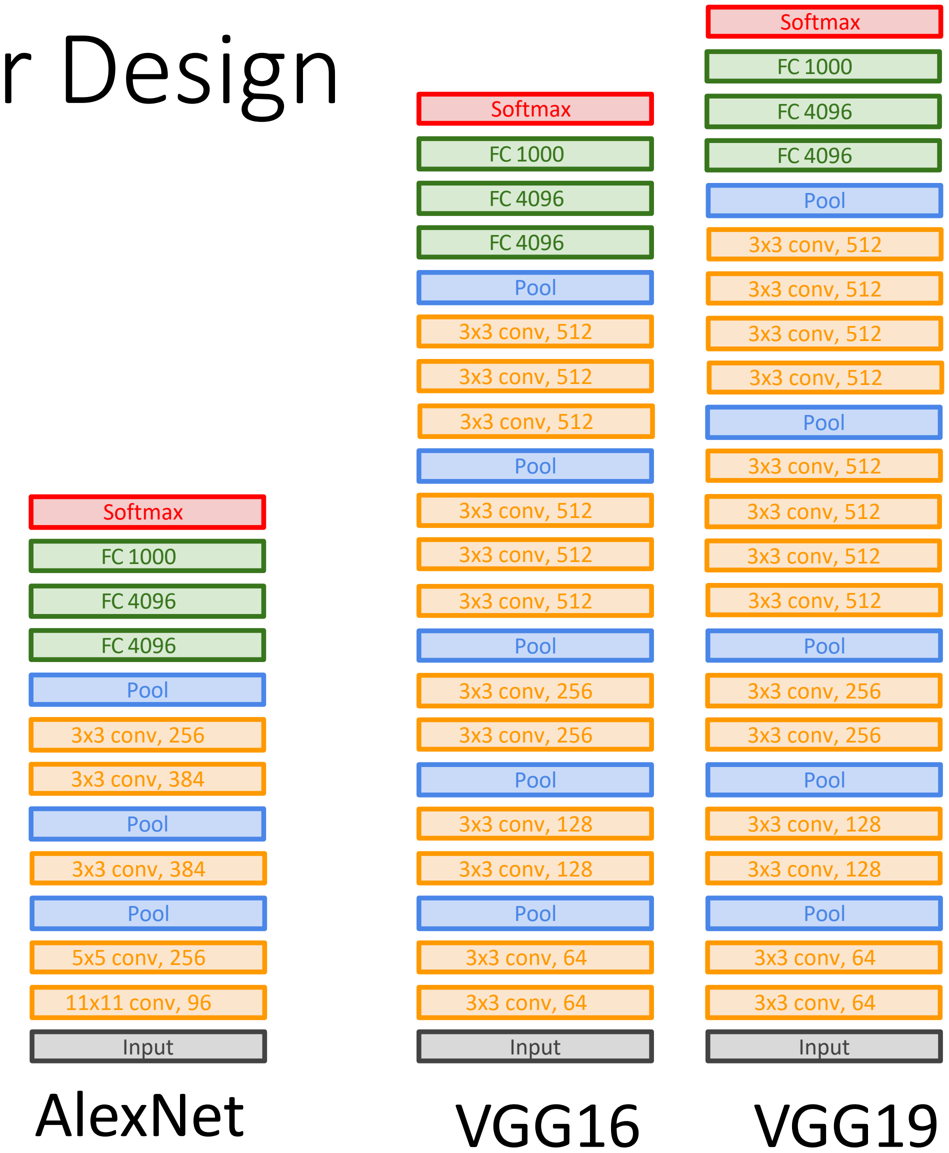
# VGG: Deeper Networks, Regular Design

<u>**VGG Design rules:**</u>

All conv are 3x3 stride 1 pad 1

All max pool are 2x2 stride 2

After pool, double #channels




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

# 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)


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

# 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>

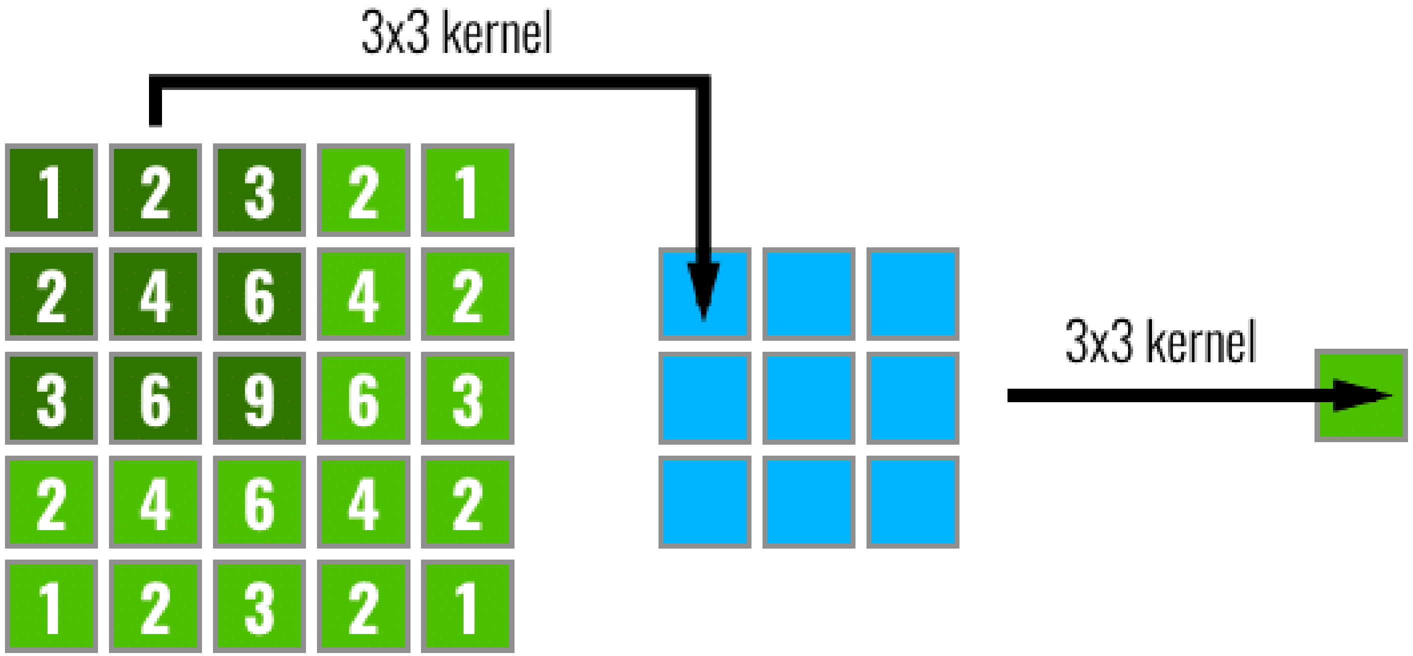
Conv(5x5, C -> C)

Params: $25C^2$

FLOPs: $25C^2HW$


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

Q: What is the effective receptive field of two 3x3 conv (stride 1) layers?




Receptive Field across 3 different layers using 3×3 filters

# 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: $25C^2HW$

<u>Option 2:</u>

Conv(3x3, C -> C)

Conv(3x3, C -> C)

Params: $18C^2$

FLOPs: $18C^2HW$


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

# 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

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

<u>Option 1:</u>

Conv(5x5, C -> C)

Params: $25C^2$

FLOPs: $25C^2HW$

<u>Option 2:</u>

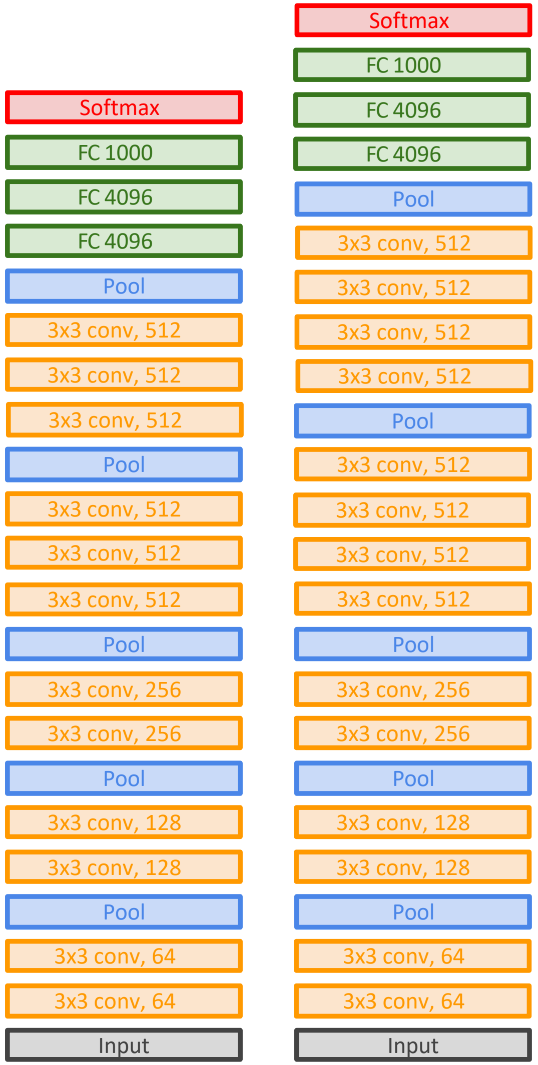
Conv(3x3, C -> C)

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

<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: $36HWC^2$


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

# 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 | Input: 2C x H x W |
| Layer: Conv(3x3, C->C) | Conv(3x3, 2C -> 2C) |
| | |
| Memory: 4HWC | Memory: 2HWC |
| Params: $9C^2$ | Params: $36C^2$ |
| FLOPs: $36HWC^2$ | FLOPs: $36HWC^2$ |


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

# VGG: Deeper Networks, Regular Design

<u>VGG Design rules:</u>

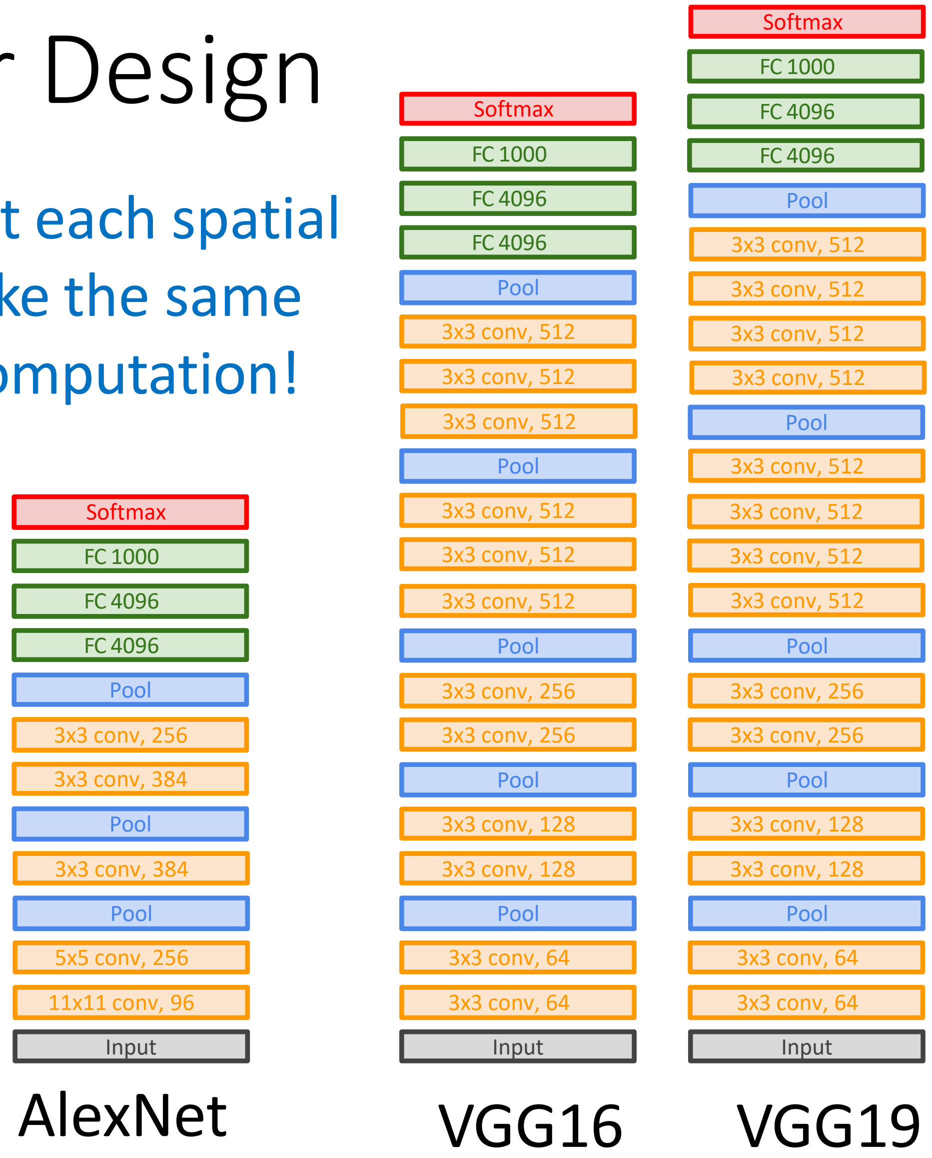
All conv are 3x3 stride 1 pad 1

**All max pool are 2x2 stride 2**

**After pool, double #channels**

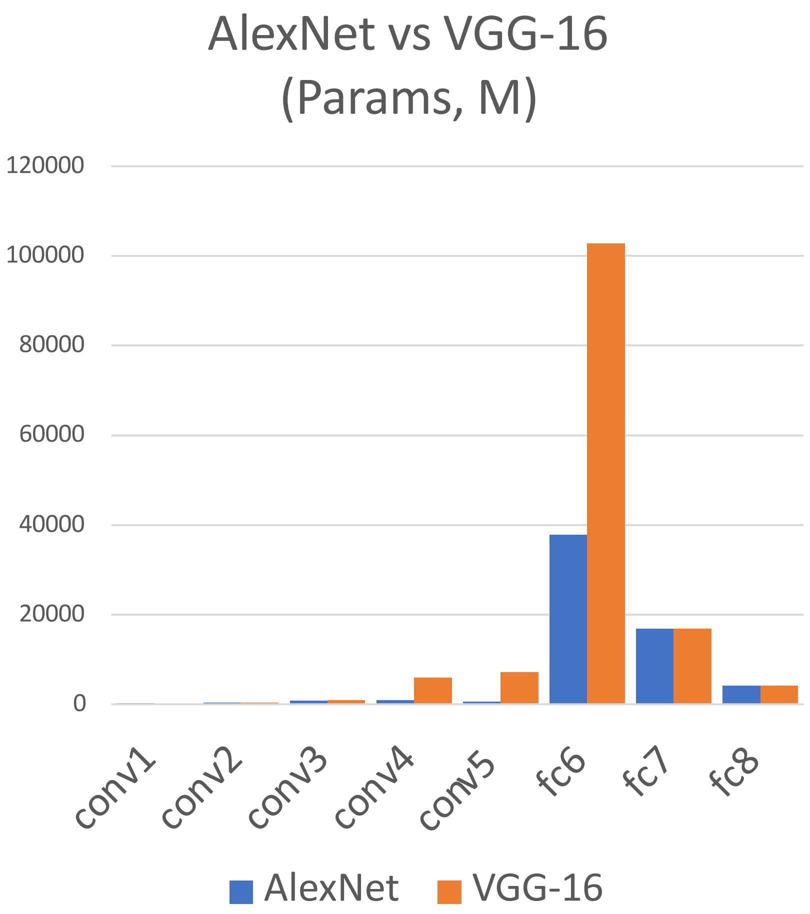
Conv layers at each spatial resolution take the same amount of computation!

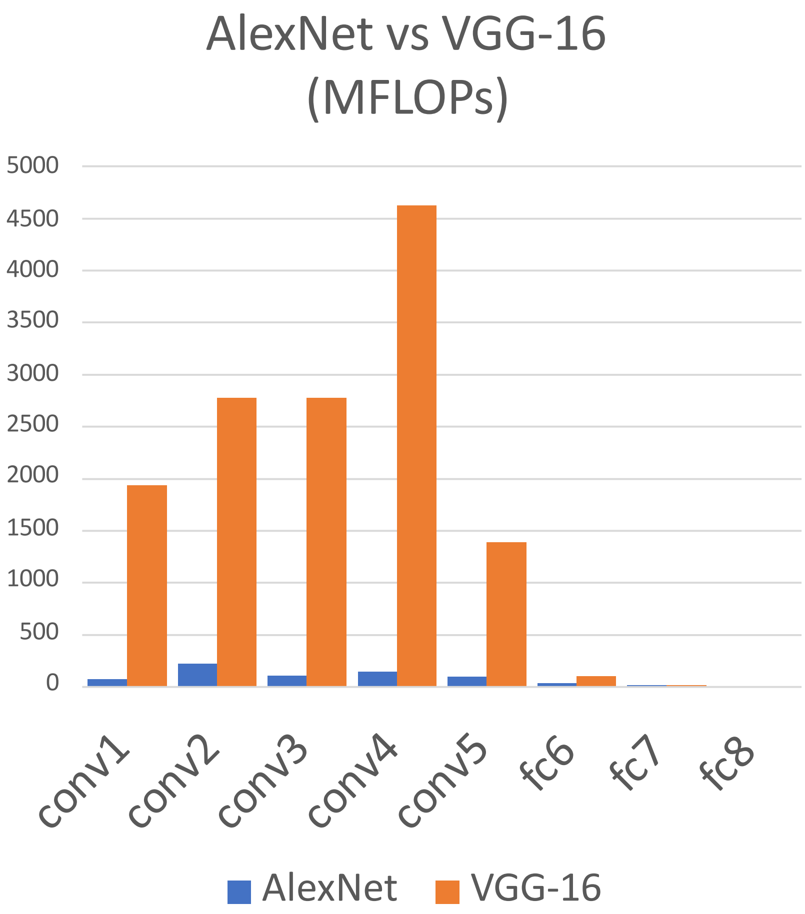
| | |
|---|---|
| Input: C x 2H x 2W | Input: 2C x H x W |
| Layer: Conv(3x3, C->C) | Conv(3x3, 2C -> 2C) |
| Memory: 4HWC | Memory: 2HWC |
| Params: $9C^2$ | Params: $36C^2$ |
| FLOPs: $36HWC^2$ | FLOPs: $36HWC^2$ |




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

# AlexNet vs VGG-16: Much bigger network!

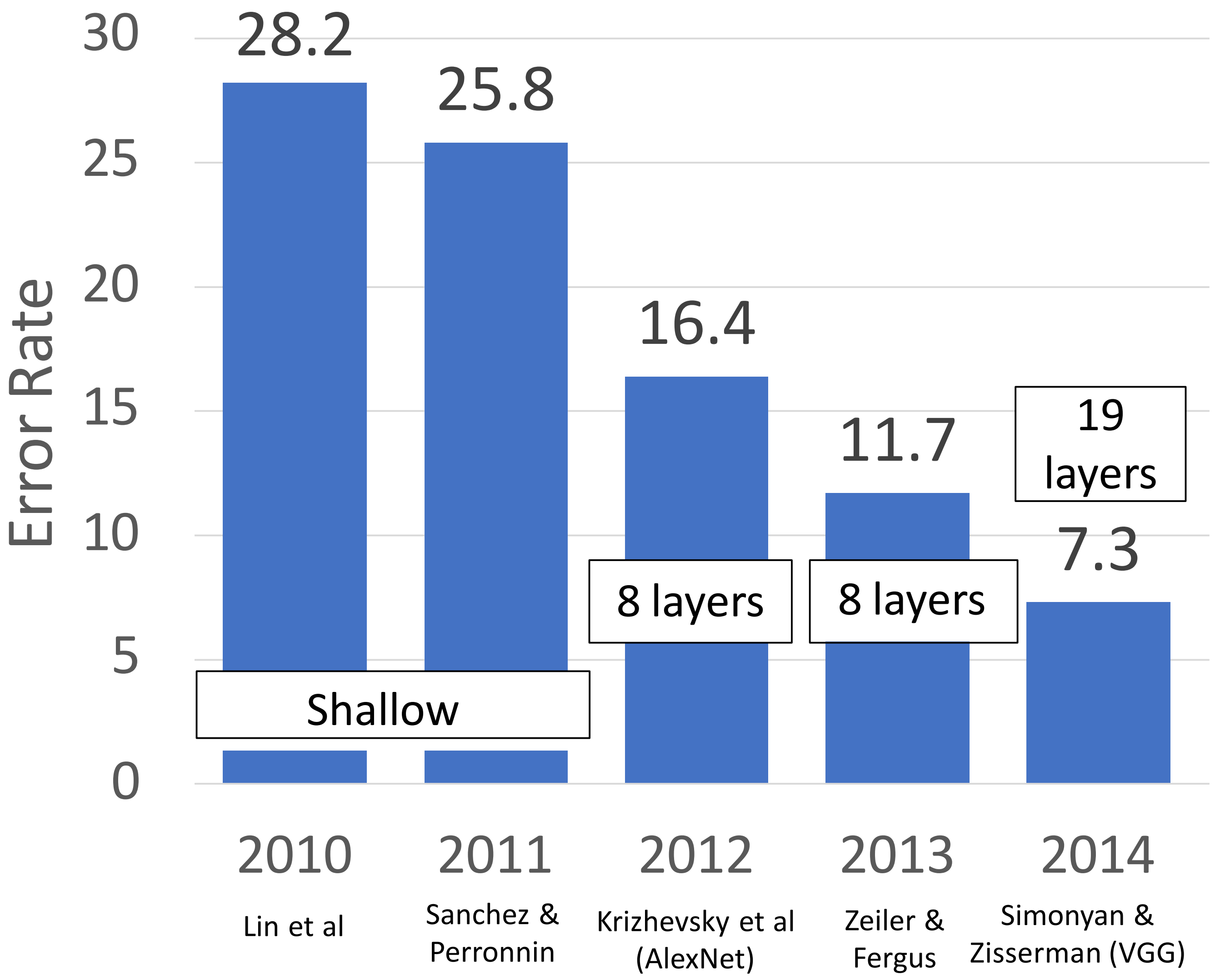




AlexNet total: 1.9 MB
VGG-16 total: 48.6 MB (25x)

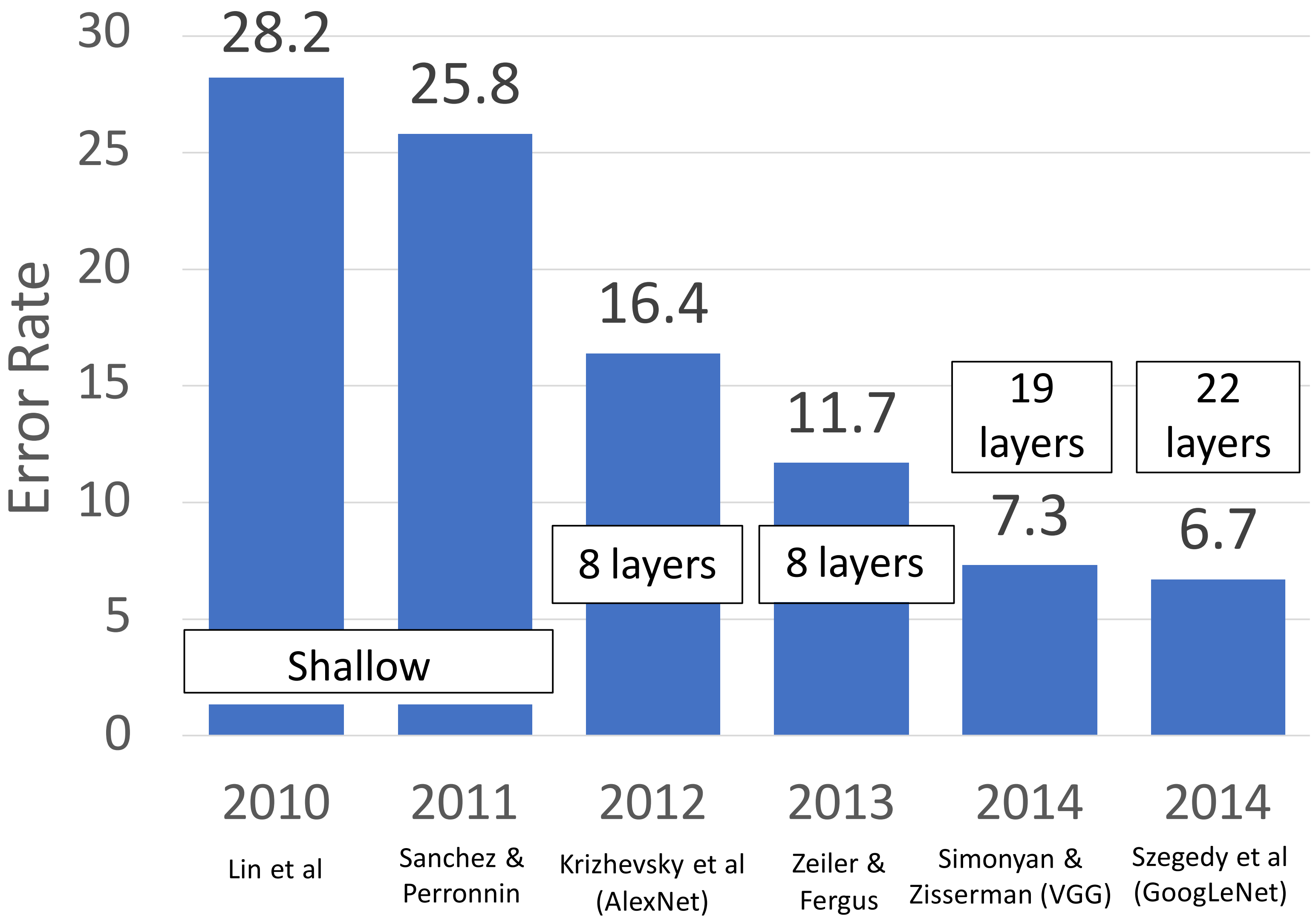
AlexNet total: 61M
VGG-16 total: 138M (2.3x)

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

# ImageNet Classification Challenge

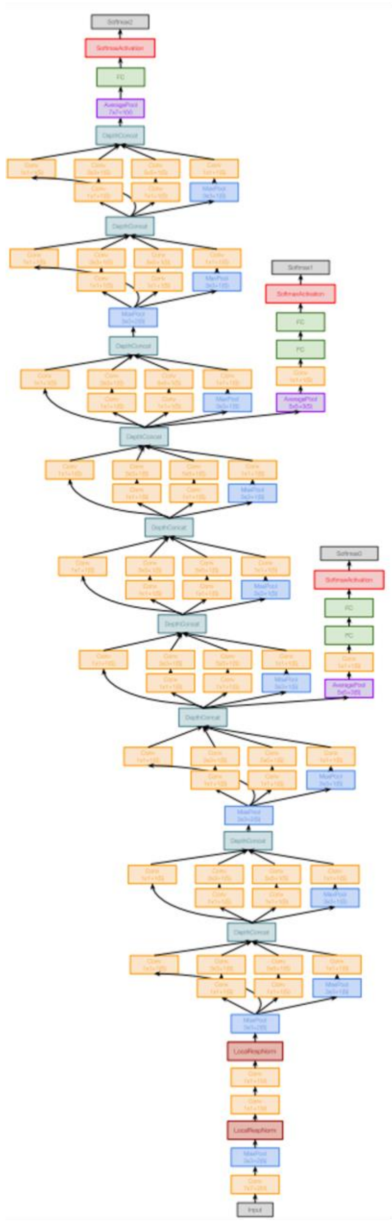


# ImageNet Classification Challenge



# GoogLeNet: Focus on Efficiency

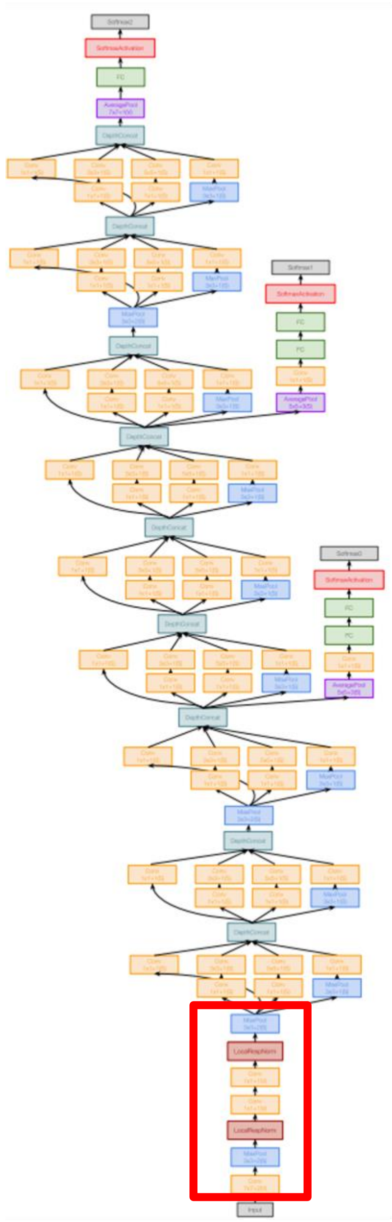
Many innovations for efficiency: reduce parameter count, memory usage, and computation



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)

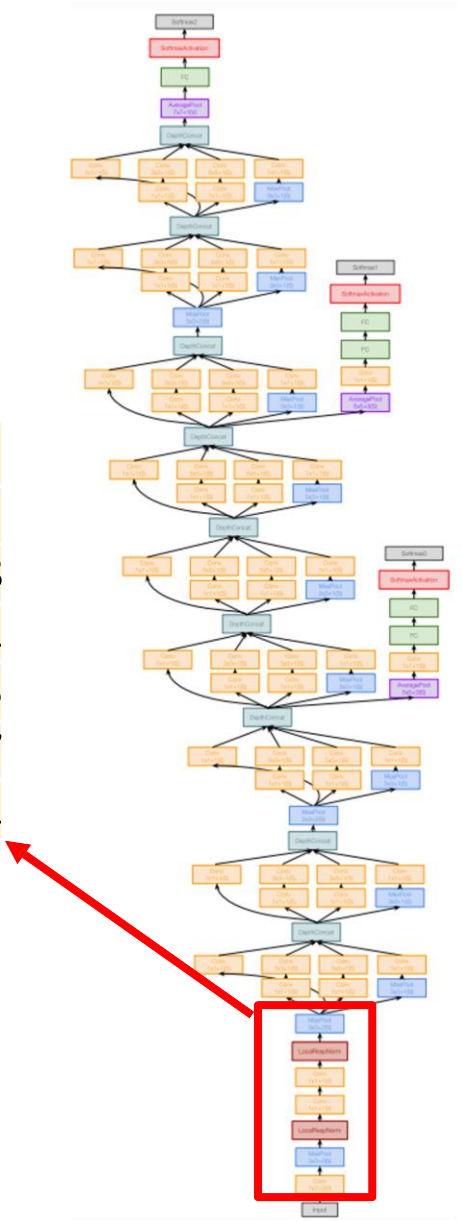


Szegedy et al, “Going deeper with convolutions”, CVPR 2015

# GoogLeNet: Aggressive Stem

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

| | 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 |



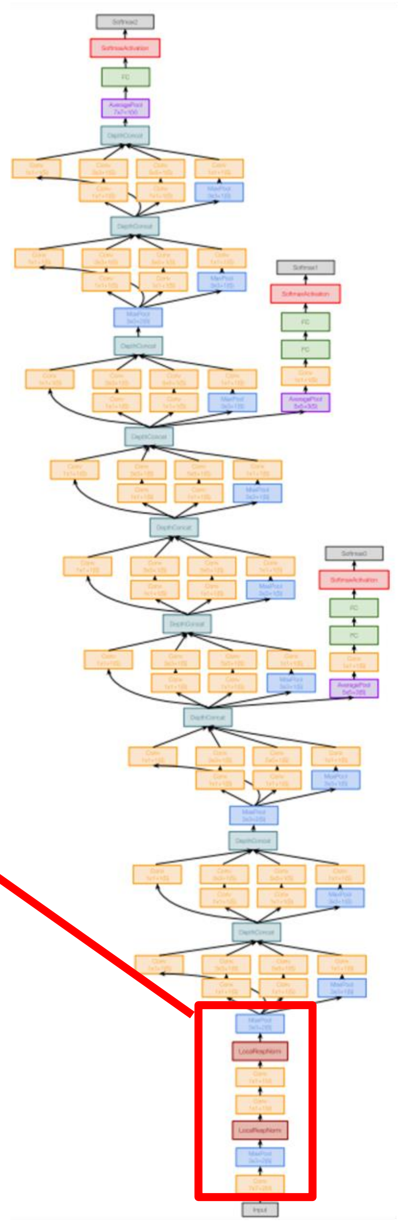
Total from 224 to 28 spatial resolution:
Memory: 7.5 MB
Params: 124K
MFLOP: 418

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)

| | 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

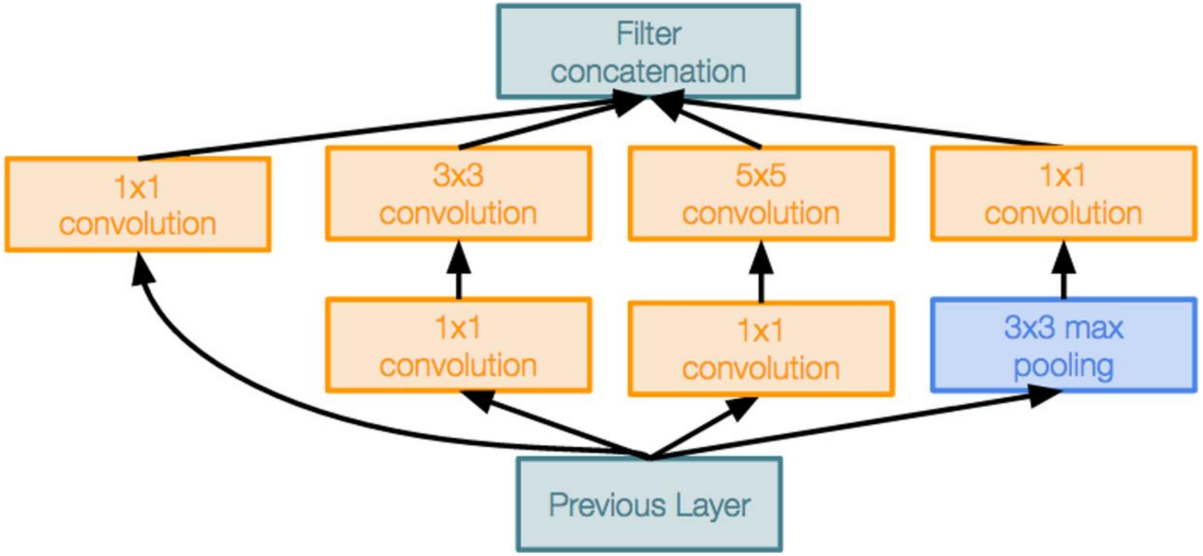
Compare VGG-16:
Memory: 42.9 MB (5.7x)
Params: 1.1M (8.9x)
MFLOP: 7485 (17.8x)

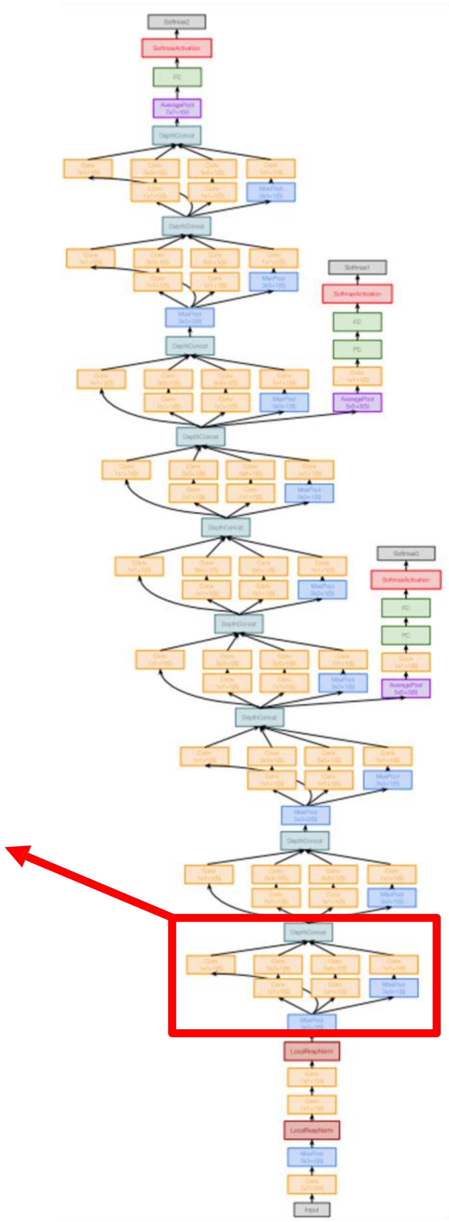
Szegedy et al, “Going deeper with convolutions”, CVPR 2015

# GoogLeNet: Inception Module

**Inception module**
Local unit with
parallel branches

Local structure repeated
many times throughout the
network



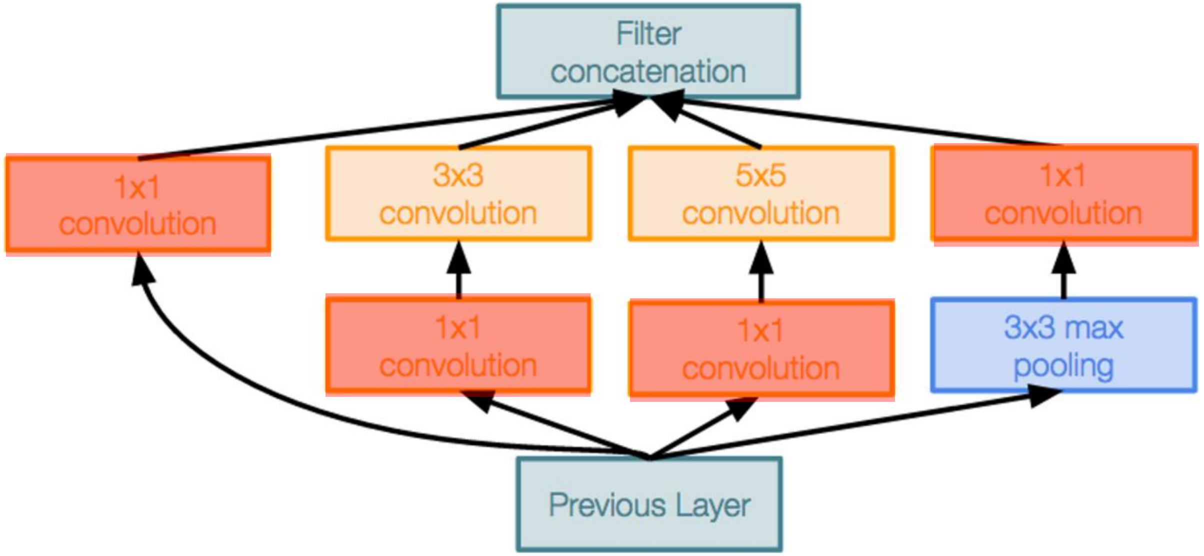
Szegedy et al, “Going deeper with convolutions”, CVPR 2015

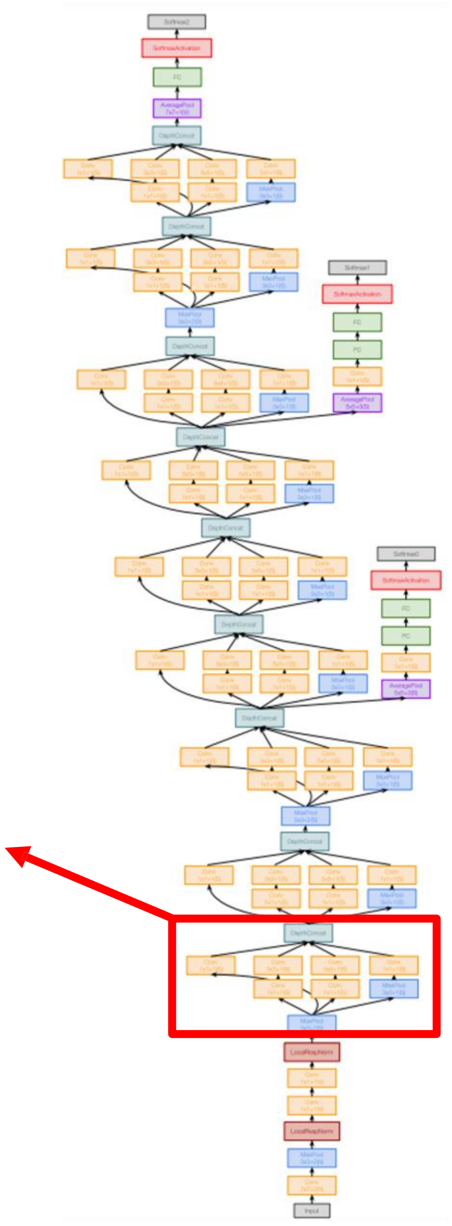
# GoogLeNet: Inception Module

**Inception module**
Local unit with
parallel branches

Local structure repeated
many times throughout the
network

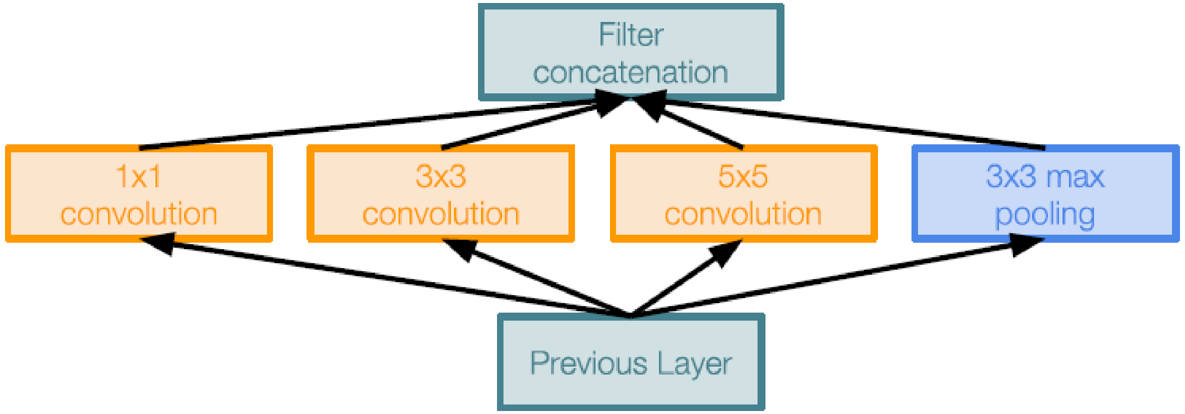
Uses 1x1 “Bottleneck”
layers to reduce channel
dimension before
expensive conv



Szegedy et al, “Going deeper with convolutions”, CVPR 2015

# Why Bottleneck




Naive Inception module

Apply parallel filter operations on the input from previous layer:
- Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
- Pooling operation (3x3)

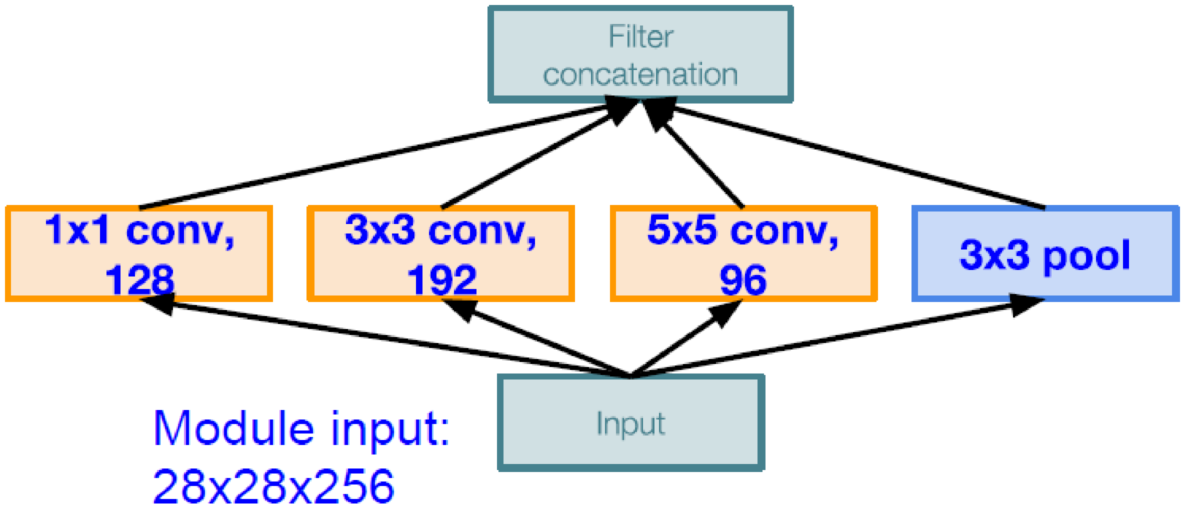
Concatenate all filter outputs together depth-wise

Q: What is the problem with this?
[Hint: Computational complexity]

# Why Bottleneck

Q: What is the problem with this?
[Hint: Computational complexity]

Example:



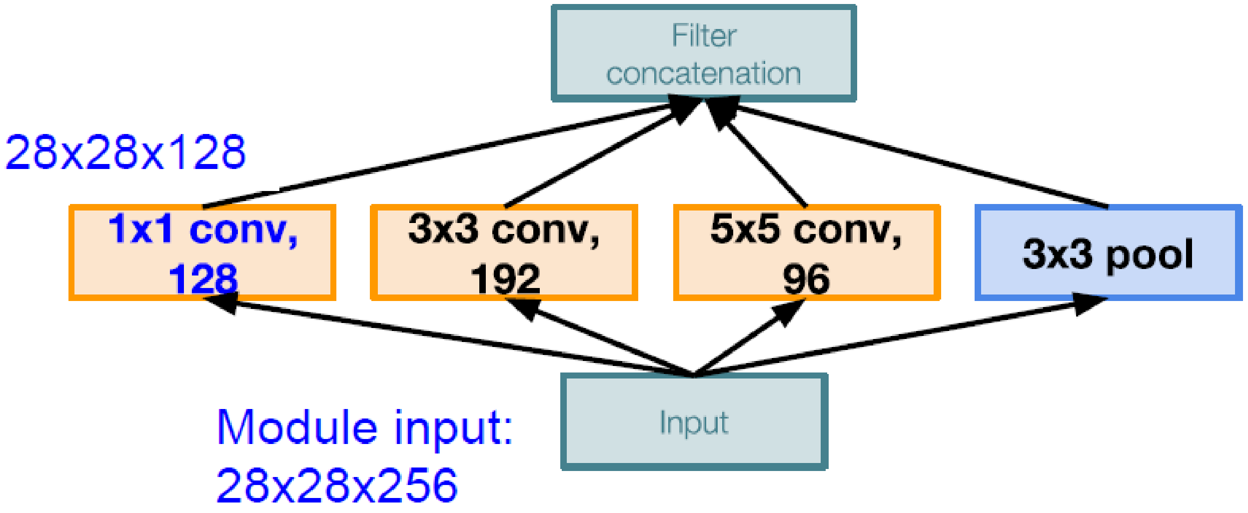

Naive Inception module

# Why Bottleneck

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q1: What is the output size of the 1x1 conv, with 128 filters?



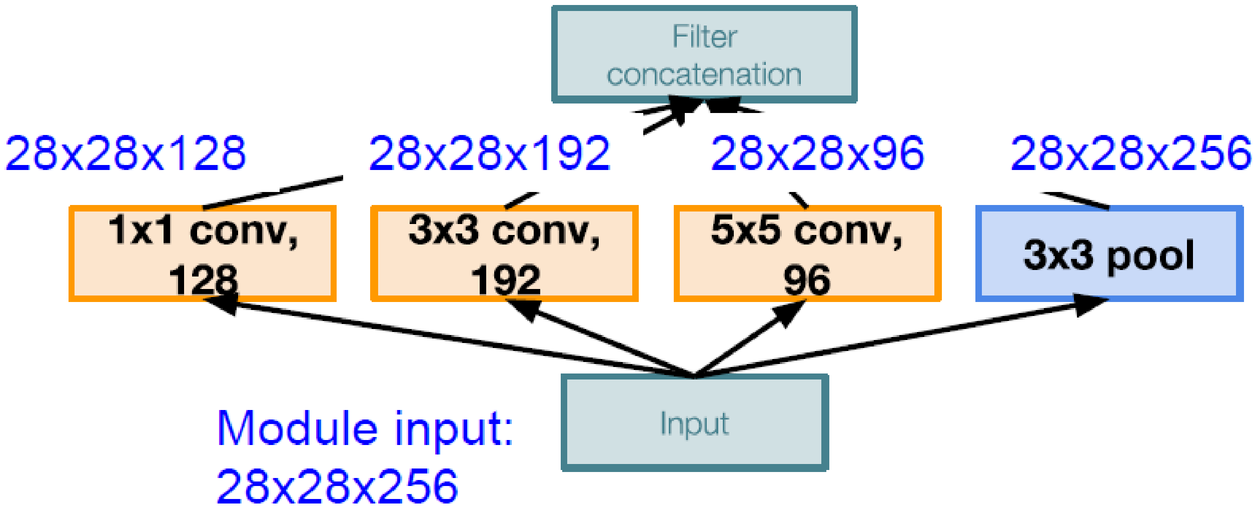

Naive Inception module

# Why Bottleneck

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q2: What are the output sizes of all different filter operations?




Naive Inception module

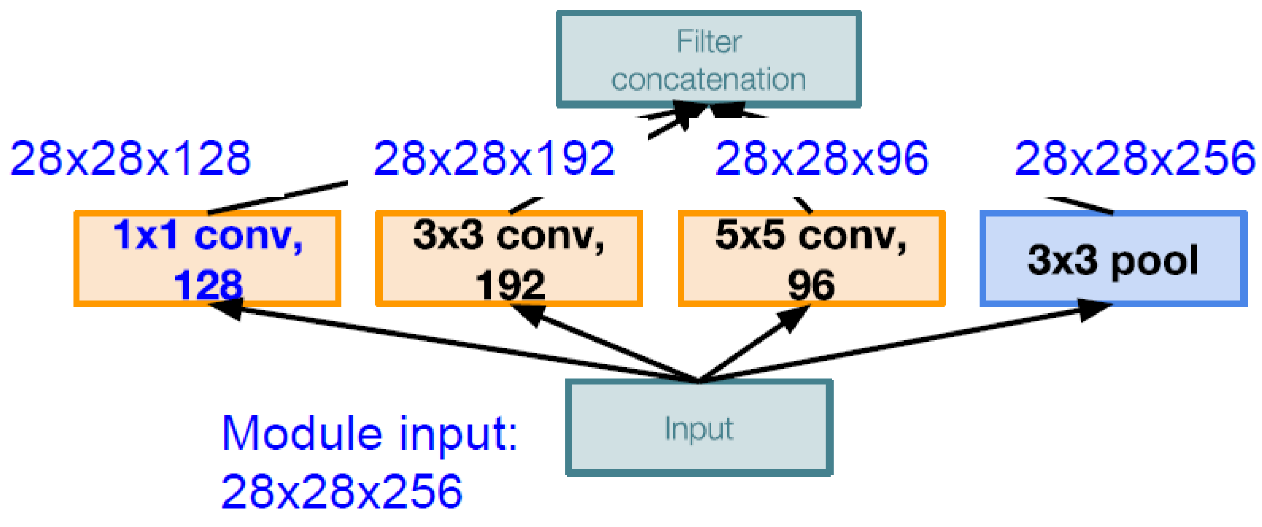
# Why Bottleneck

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q3:What is output size after filter concatenation?

28x28x(128+192+96+256) = 28x28x672




Naive Inception module

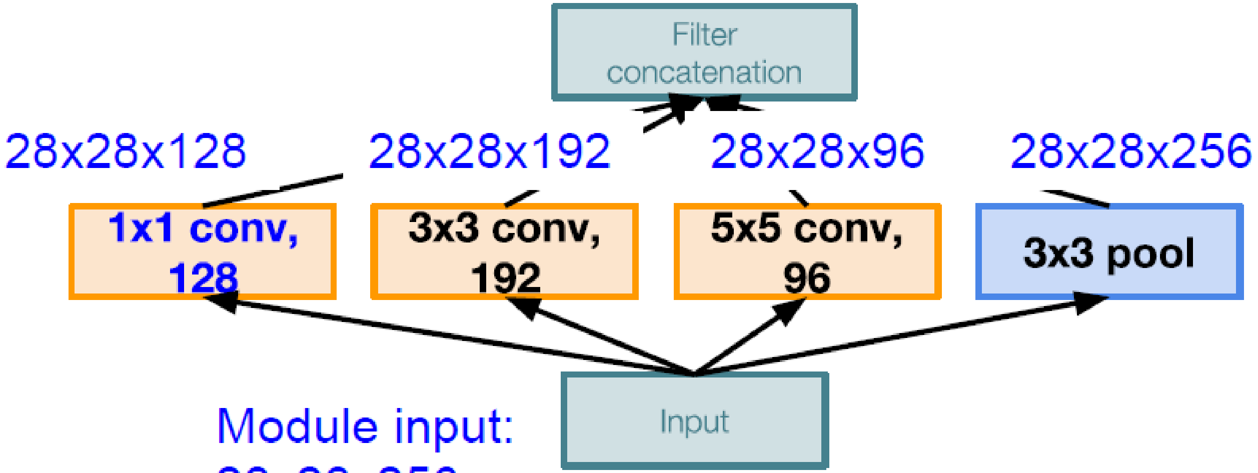
# Why Bottleneck

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q3:What is output size after filter concatenation?

28x28x(128+192+96+256) = 28x28x672




Module input:
28x28x256

Naive Inception module

**Conv Ops:**

[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x256
[5x5 conv, 96] 28x28x96x5x5x256
**Total: 854M ops**

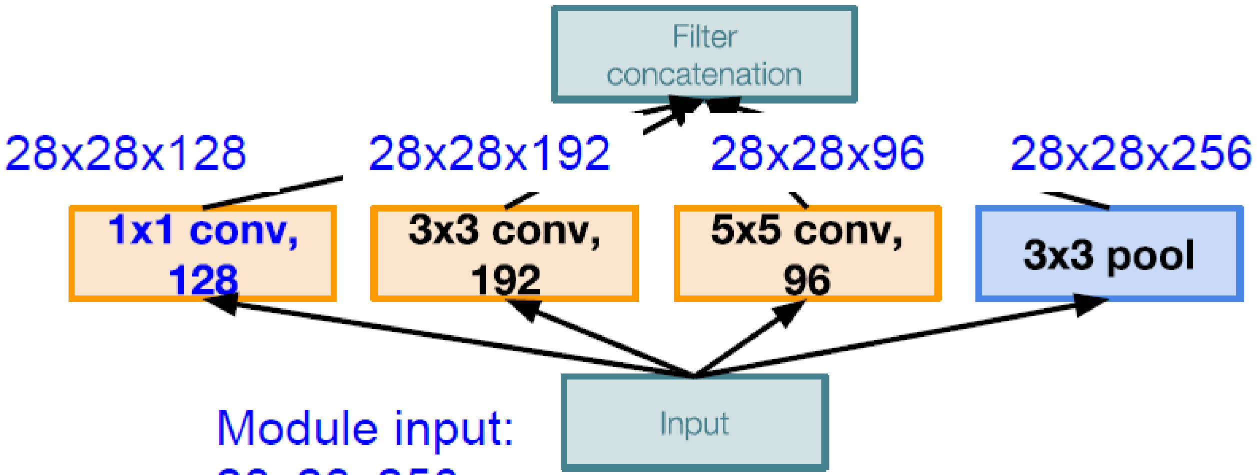
# Why Bottleneck

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q3:What is output size after filter concatenation?

28x28x(128+192+96+256) = 28x28x672




Module input:
28x28x256

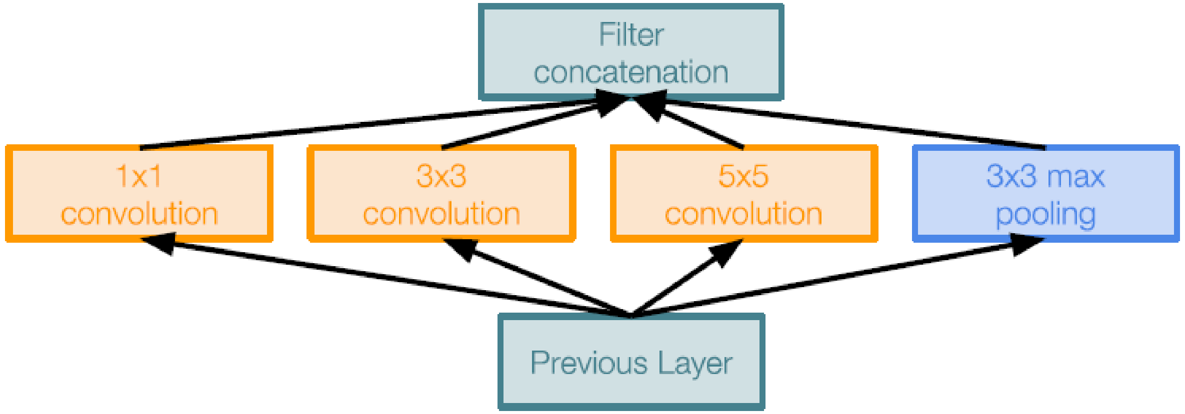
Naive Inception module

**Conv Ops:**
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x256
[5x5 conv, 96] 28x28x96x5x5x256
**Total: 854M ops**

Very expensive compute

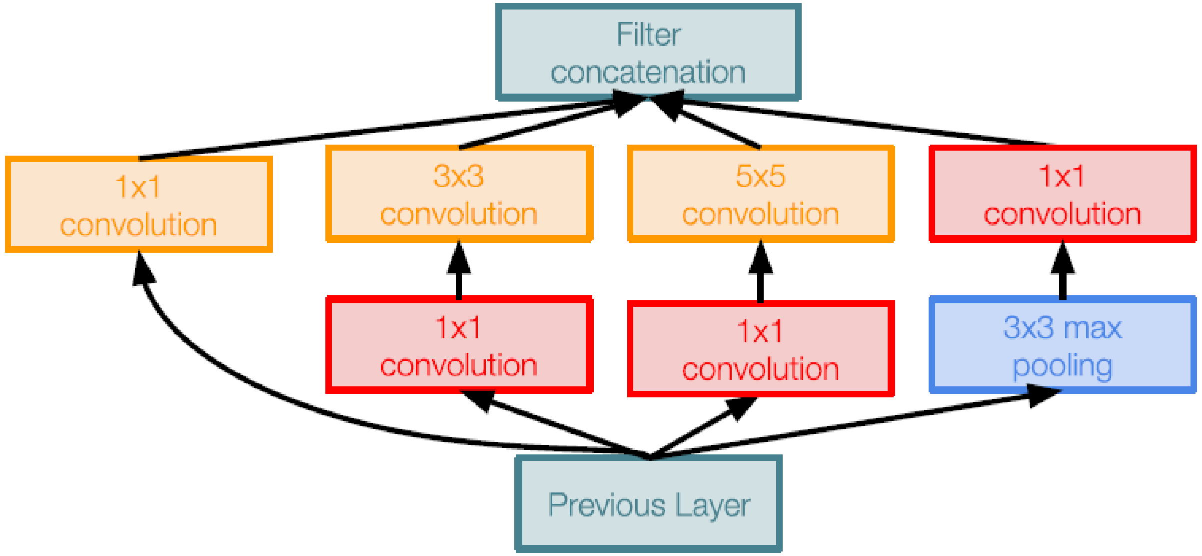
Pooling layer also preserves feature depth, which means total depth after concatenation can only grow at every layer!

# Solution: Bottleneck




Naive Inception module

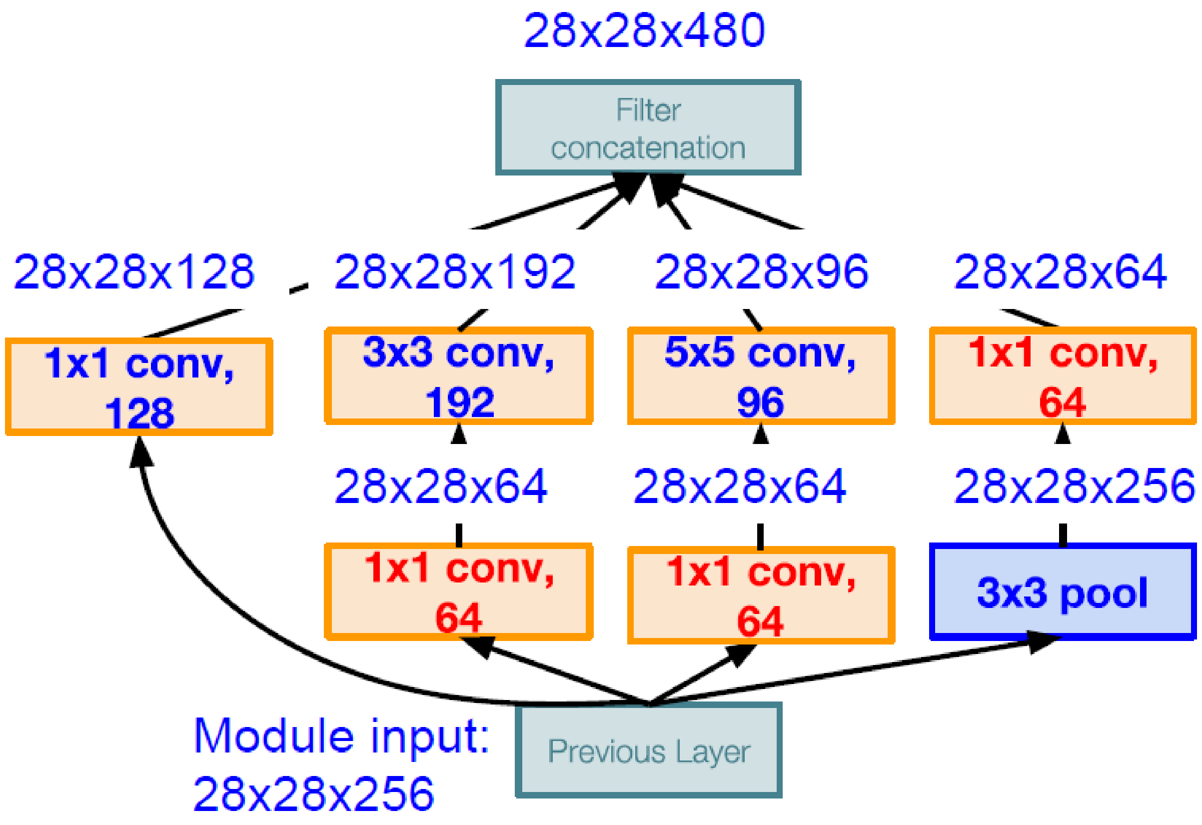
1x1 conv "bottleneck" layers




Inception module with dimension reduction

# Solution: Bottleneck

Using same parallel layers as naive example, and adding "1x1 conv, 64 filter" bottlenecks:




Inception module with dimension reduction

**Conv Ops:**

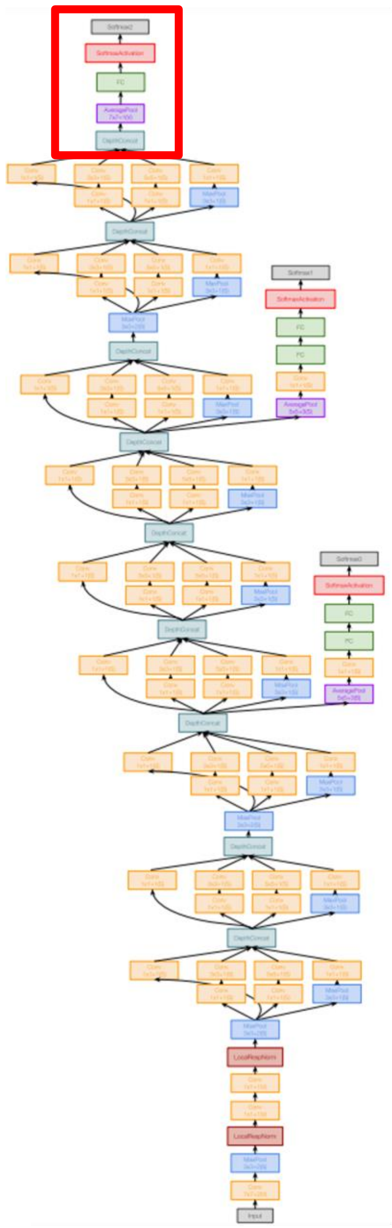
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x64
[5x5 conv, 96] 28x28x96x5x5x64
[1x1 conv, 64] 28x28x64x1x1x256

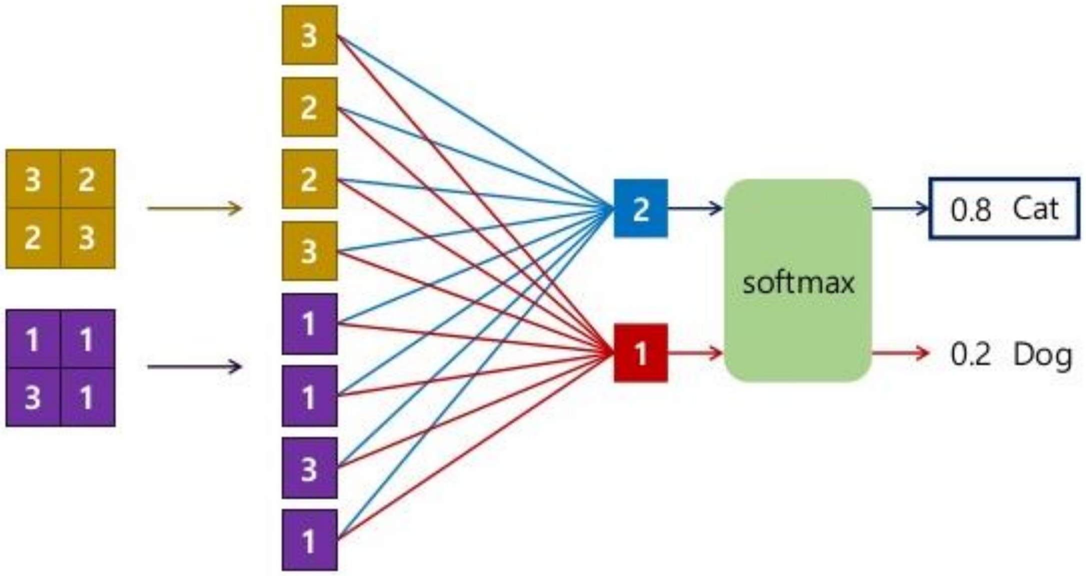
**Total: 358M ops**

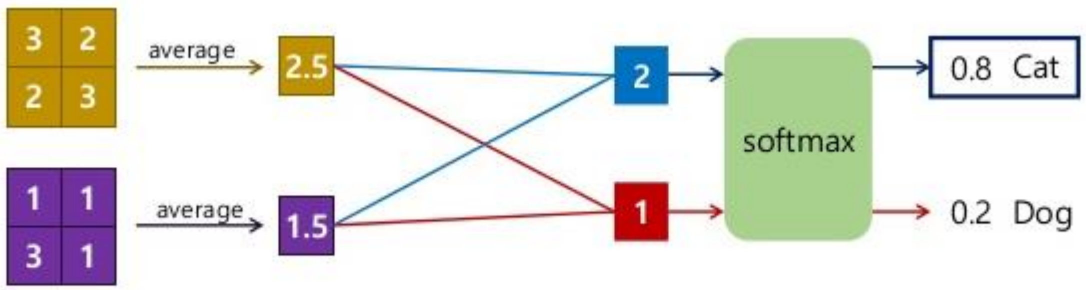
Compared to 854M ops for naive version
Bottleneck can also reduce depth after pooling layer

# 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!)





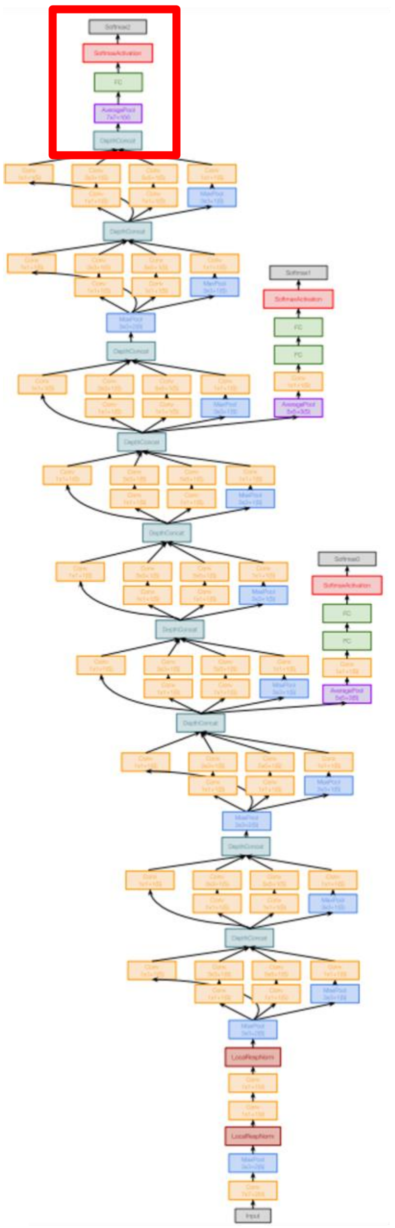

FC layer uses Softmax at the last layer to output the classification results

GoogLeNet uses GAP architecture instead of FC layer to directly output classification, which can improve model efficiency and reduce resource usage.

# GoogLeNet: Global Average Pooling

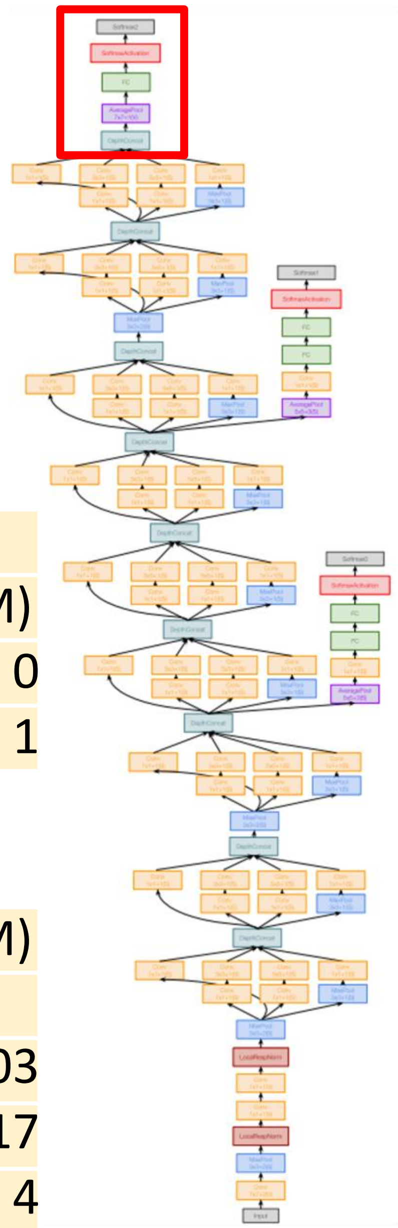
No large FC layers at the end! Instead uses **global average pooling** to collapse spatial dimensions, and one linear layer to produce class scores
(Recall VGG-16: Most parameters were in the FC layers!)

| | 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:

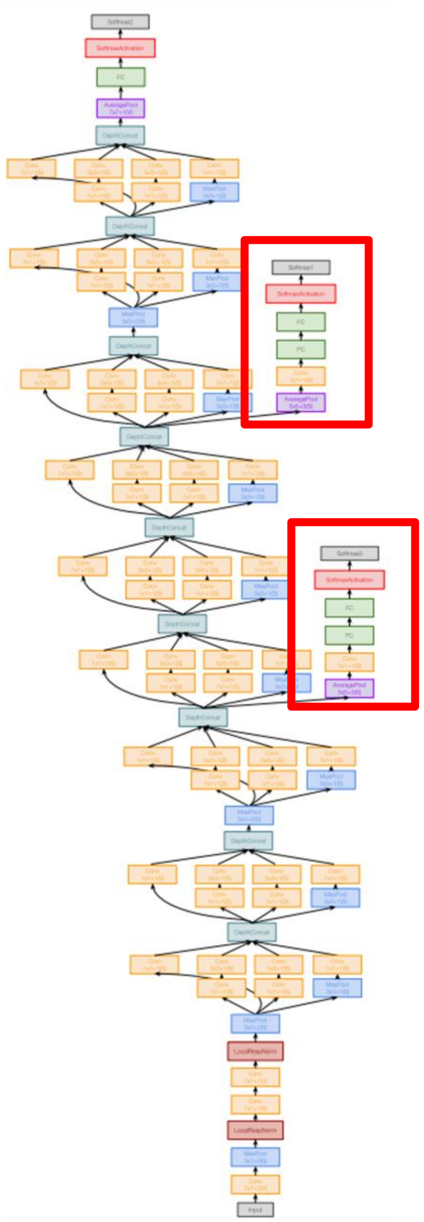
| Layer | C | H/W | filters | kernel | stride | pad | C | H/W | memory (KB) | params (K) | flop (M) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| flatten | 512 | 7 | | | | | 25088 | | 98 | | |
| fc6 | 25088 | | | 4096 | | | 4096 | | 16 | 102760 | 103 |
| fc7 | 4096 | | | 4096 | | | 4096 | | 16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

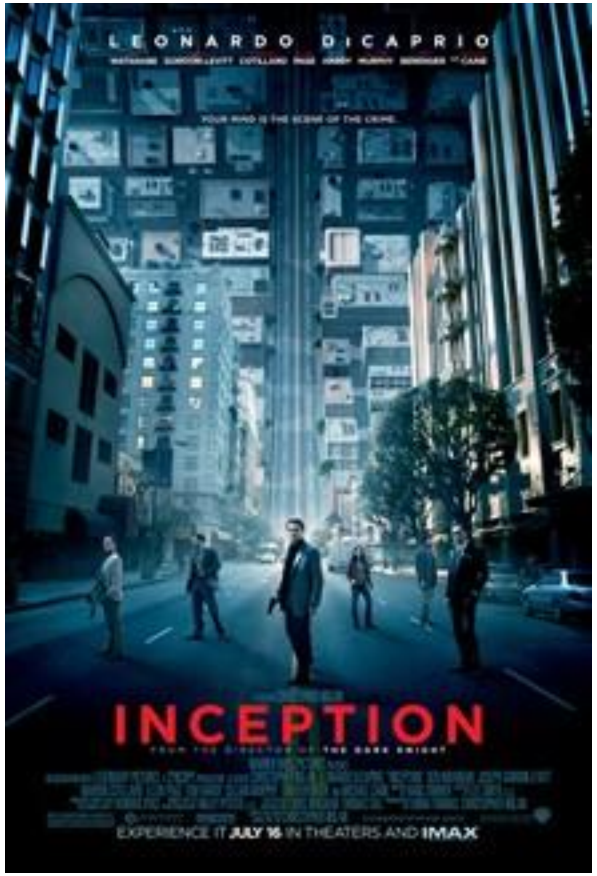
# GoogLeNet: Auxiliary Classifiers

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

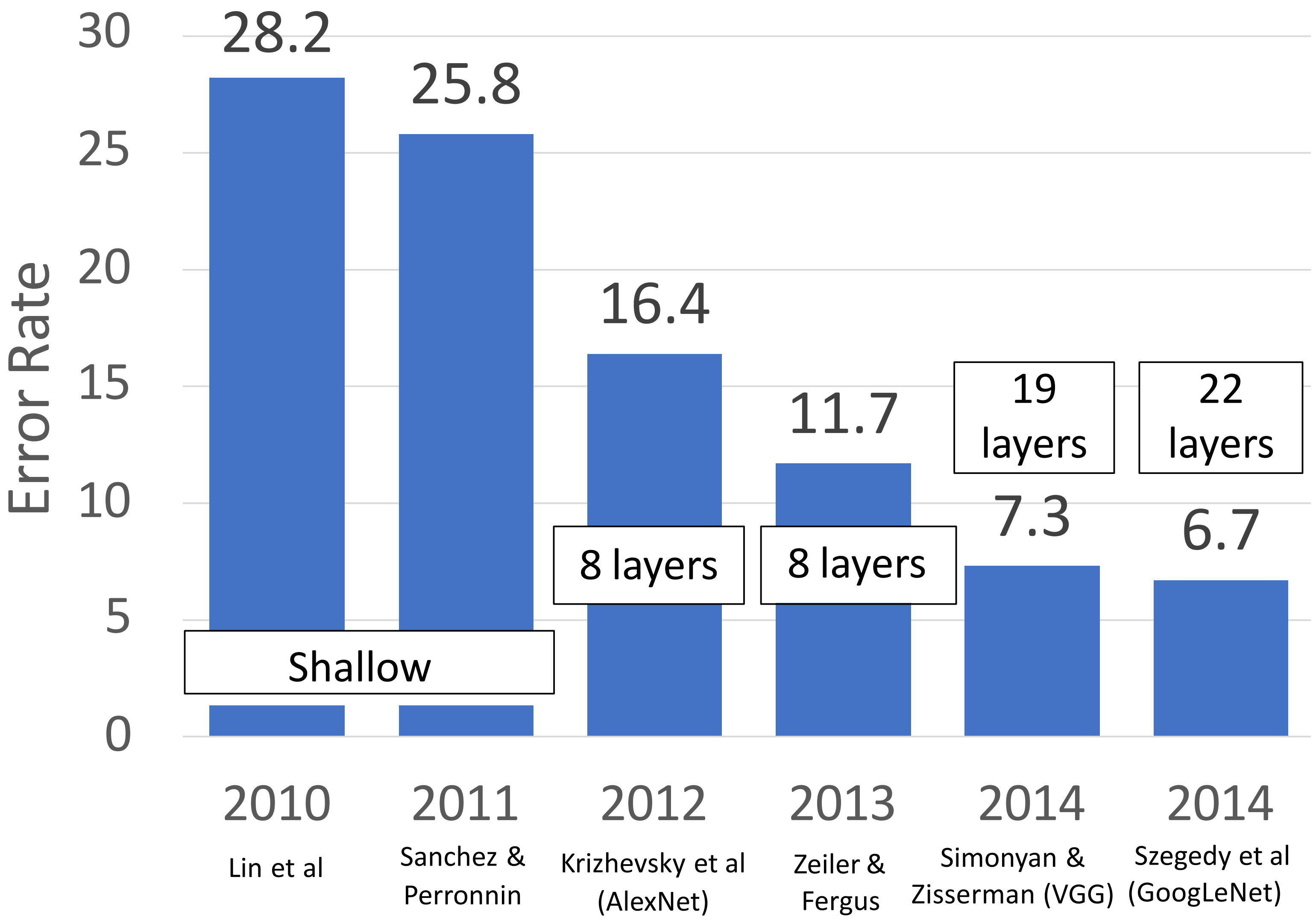



LEONARDO DiCAPRIO
INCEPTION
EXPERIENCE IT JULY 16 IN THEATERS AND IMAX

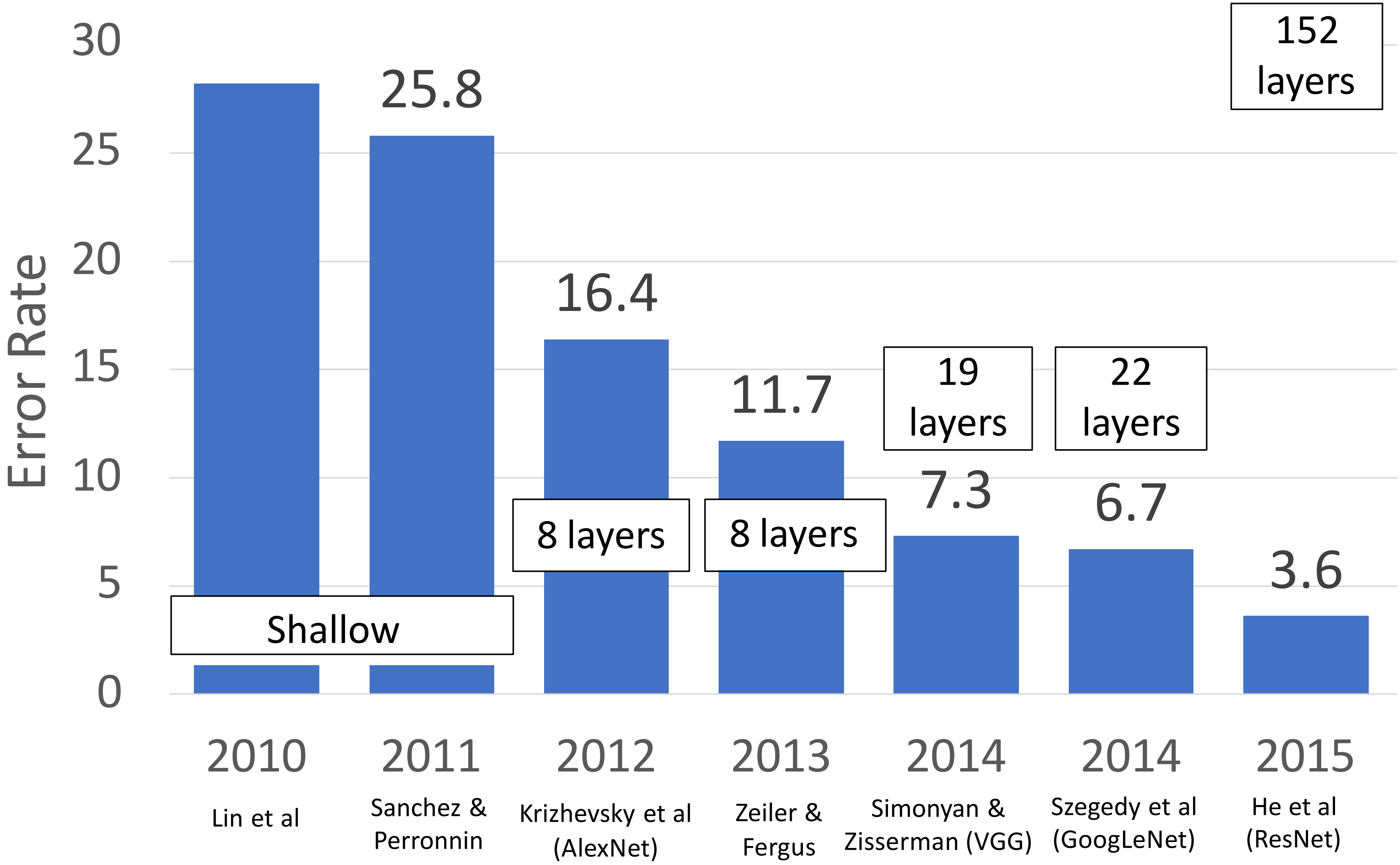
全面啟動


WE NEED TO GO
DEEPER

# 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?

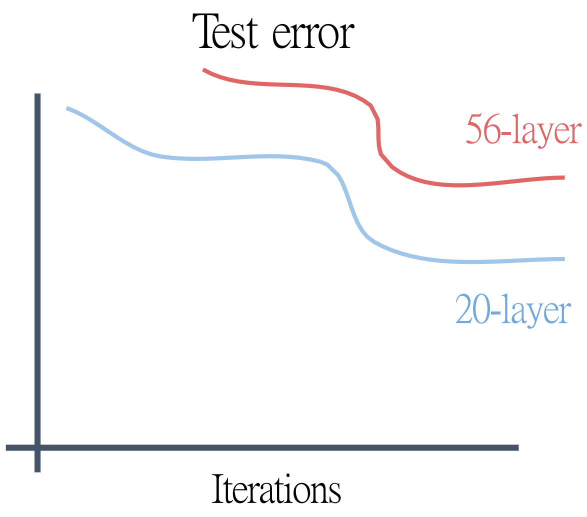
He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

# 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!

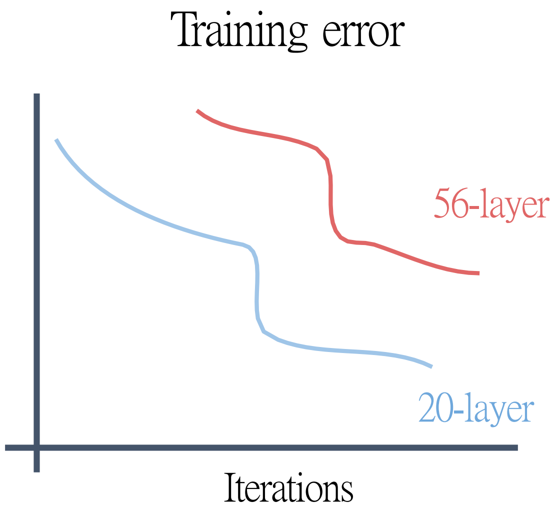
Initial guess: Deep model is **overfitting** since it is much bigger than the other model

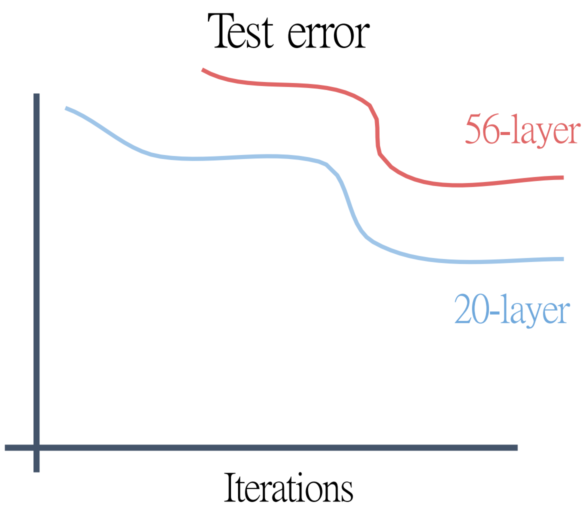



He et al, “Deep Residual Learning for Image Recognition”, CVPR 2016

# 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**

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

# Residual Networks

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

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

# Residual Networks

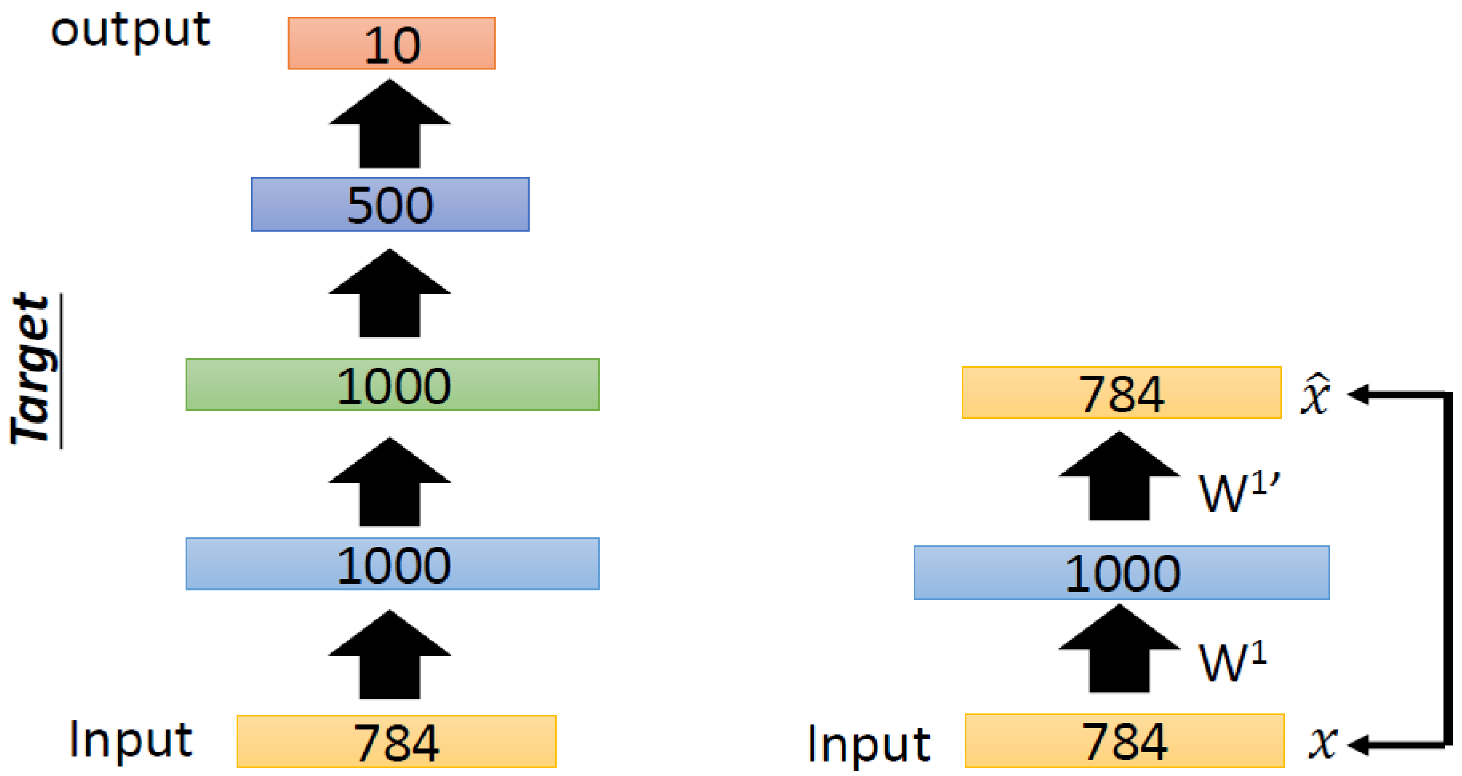
A deeper model can emulate a shallower model: copy layers from shallower model, set extra layers to identity

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

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

# Auto-encoder – Pre-training DNN

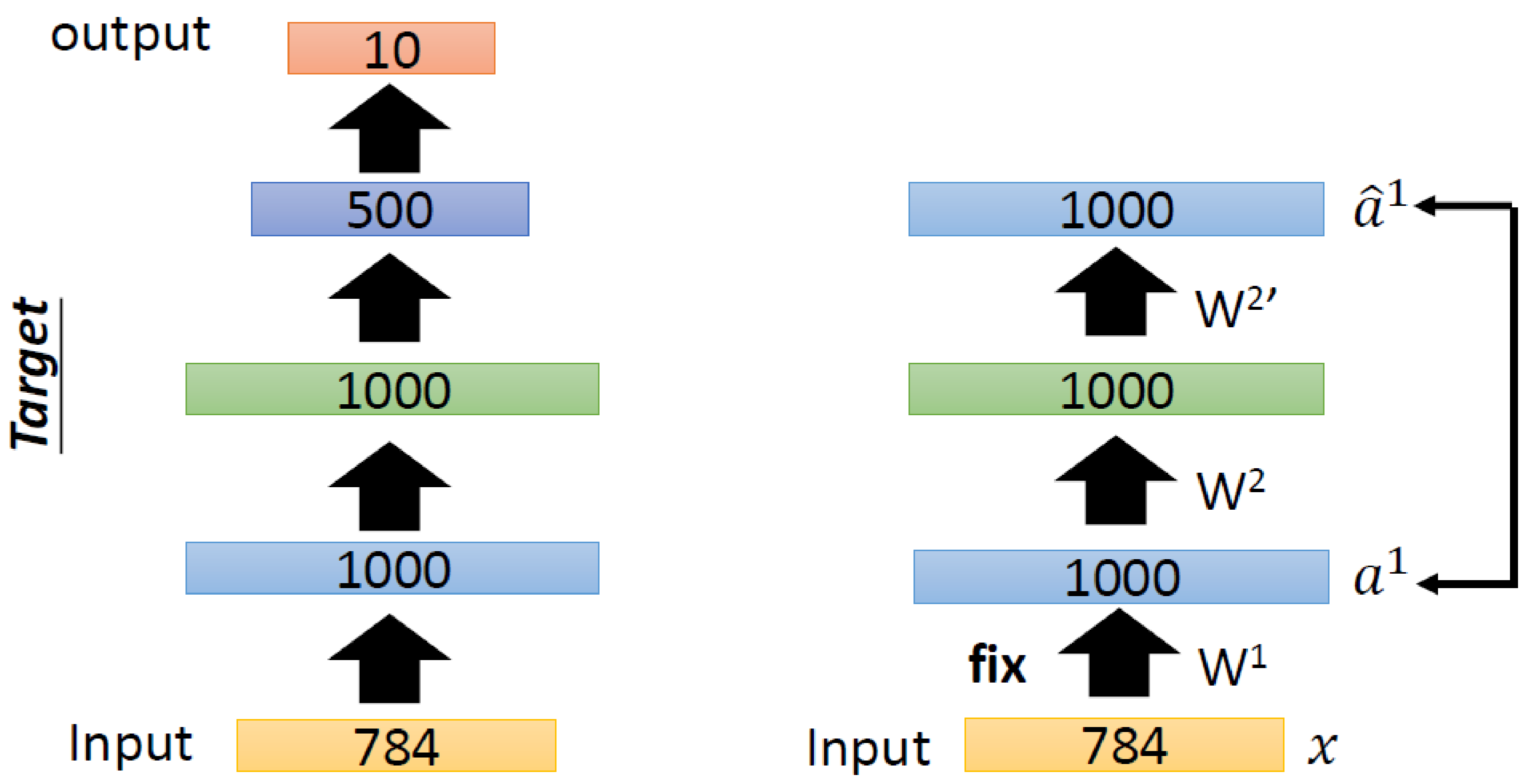
- Greedy Layer-wise Pre-training *again*




原圖: 台大 李宏毅 教授

# Auto-encoder – Pre-training DNN

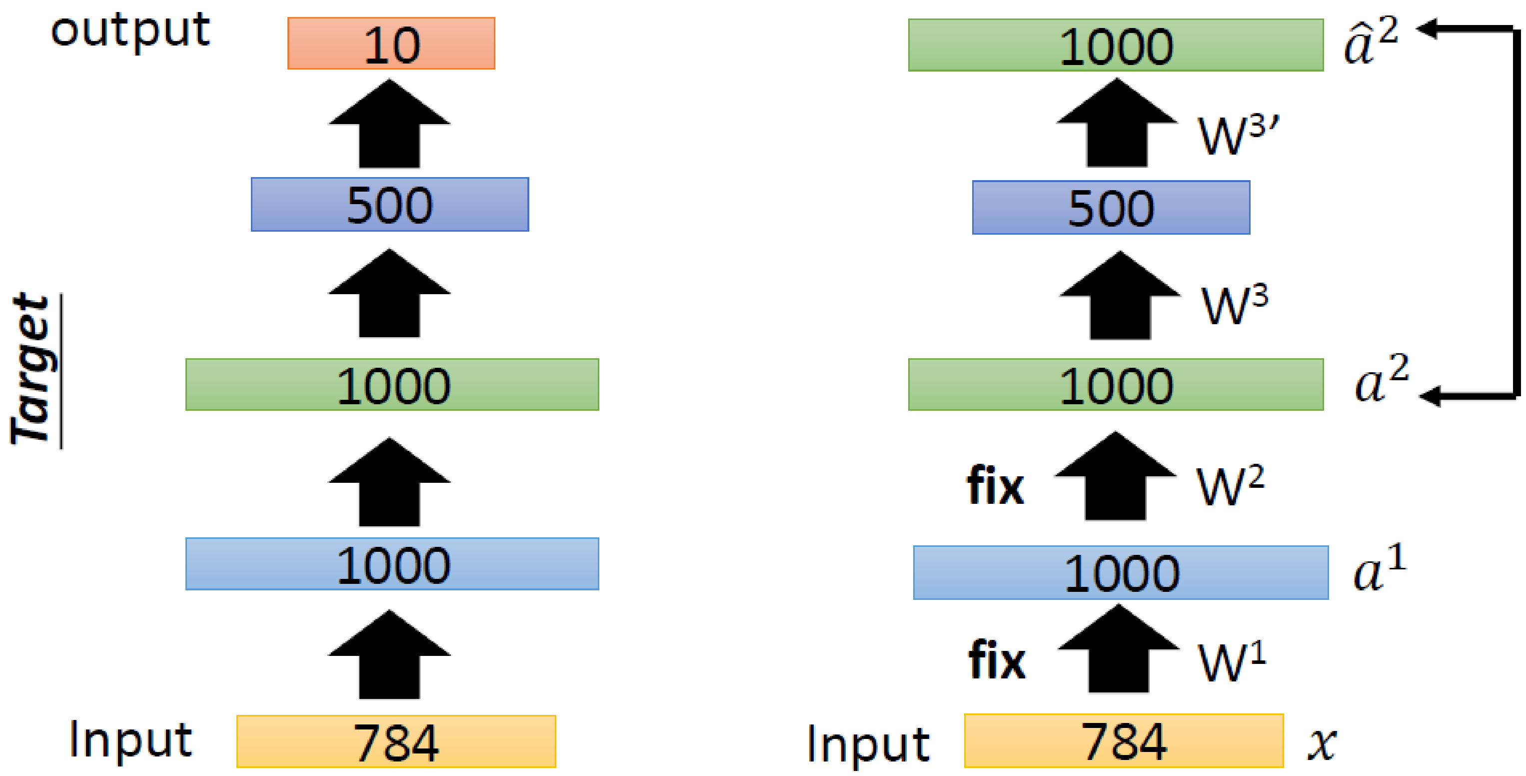
- Greedy Layer-wise Pre-training *again*




原圖: 台大 李宏毅 教授

# Auto-encoder – Pre-training DNN

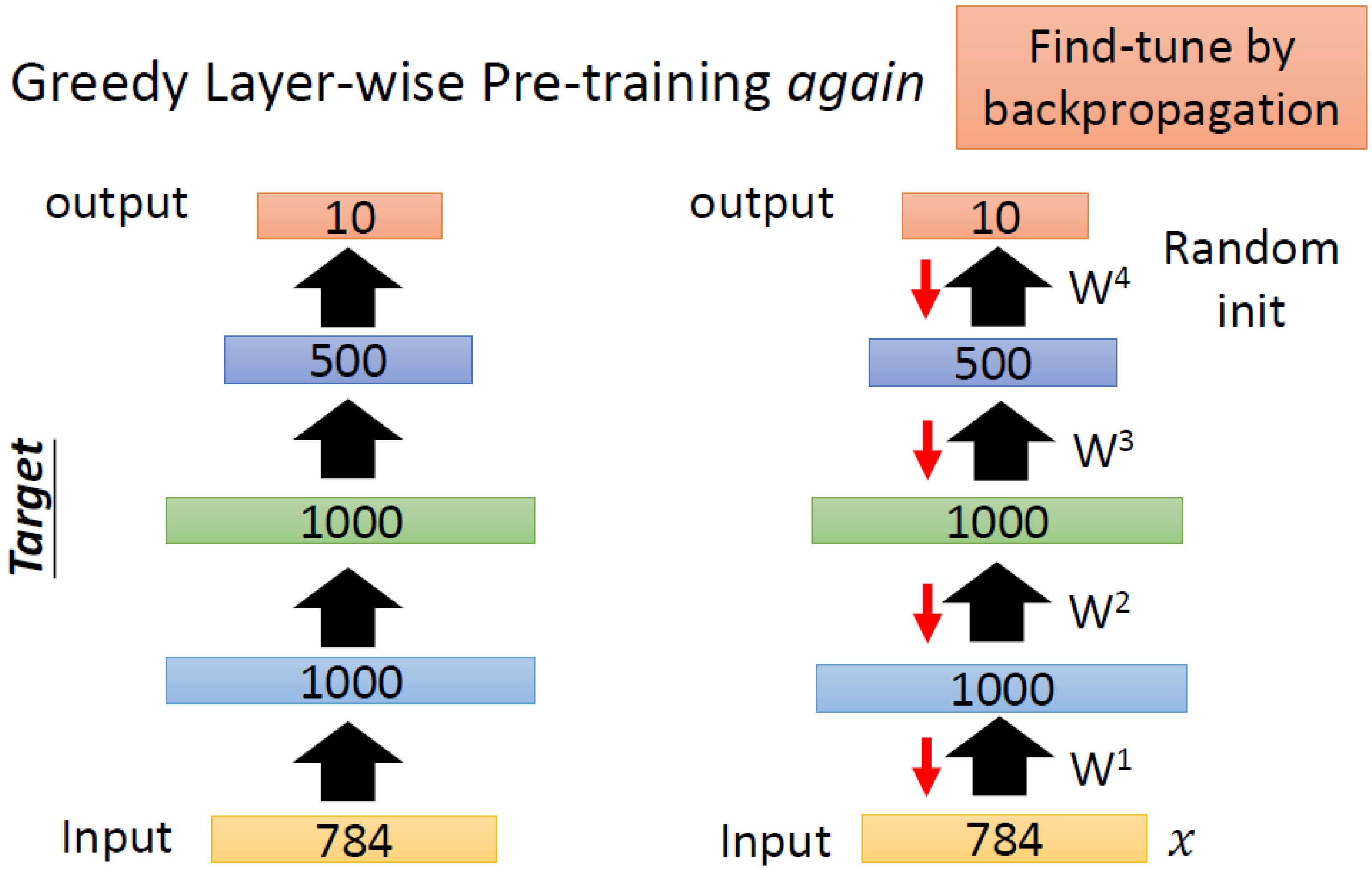
- Greedy Layer-wise Pre-training *again*




原圖: 台大 李宏毅 教授

# Auto-encoder – Pre-training DNN

- Greedy Layer-wise Pre-training *again*




原圖: 台大 李宏毅 教授

# 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

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

# 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

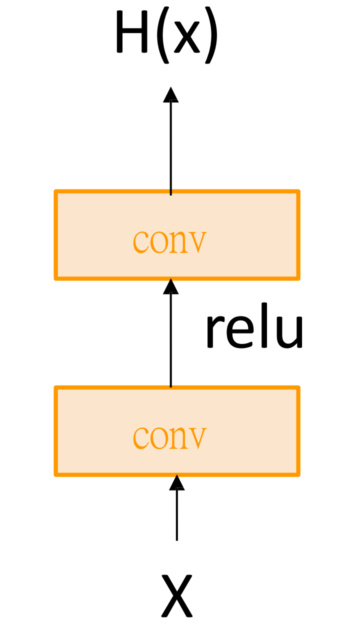
**Hypothesis**: This is an optimization problem. Deeper models are harder to optimize, and in particular don't learn identity functions to emulate shallow models

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

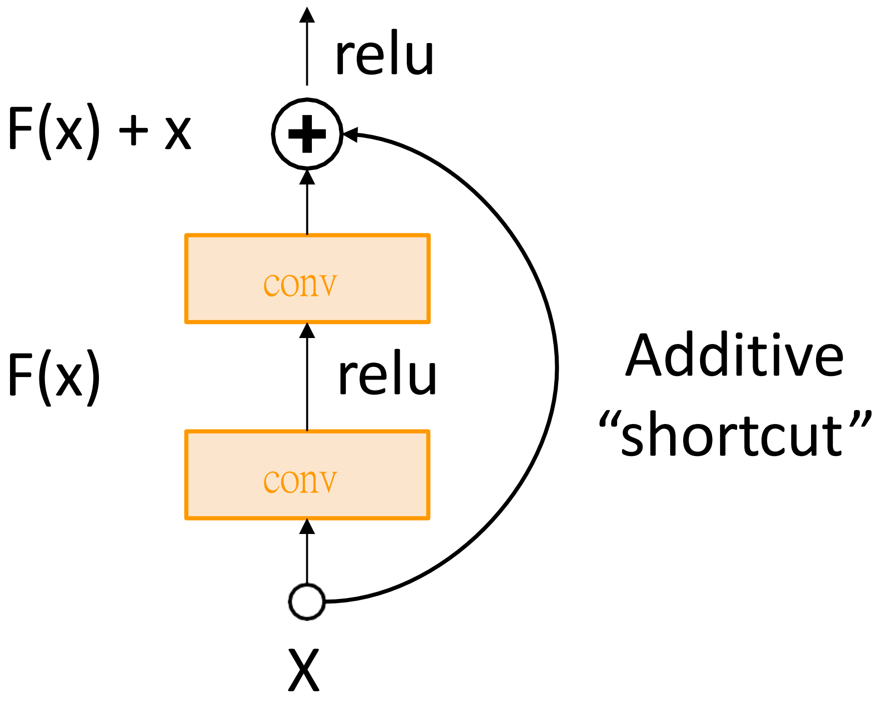
He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

# Residual Networks

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




“Plain” block

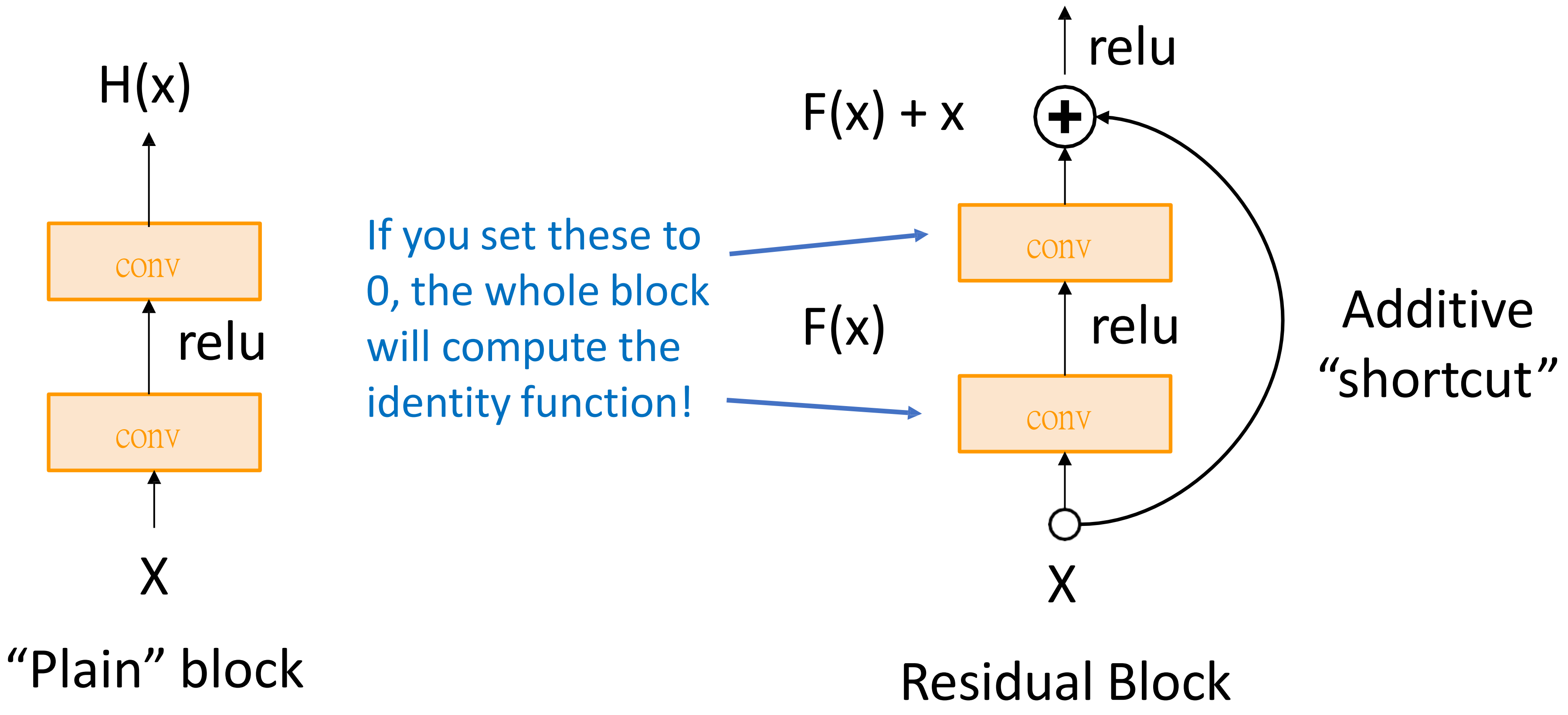



Residual Block

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

# Residual Networks

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



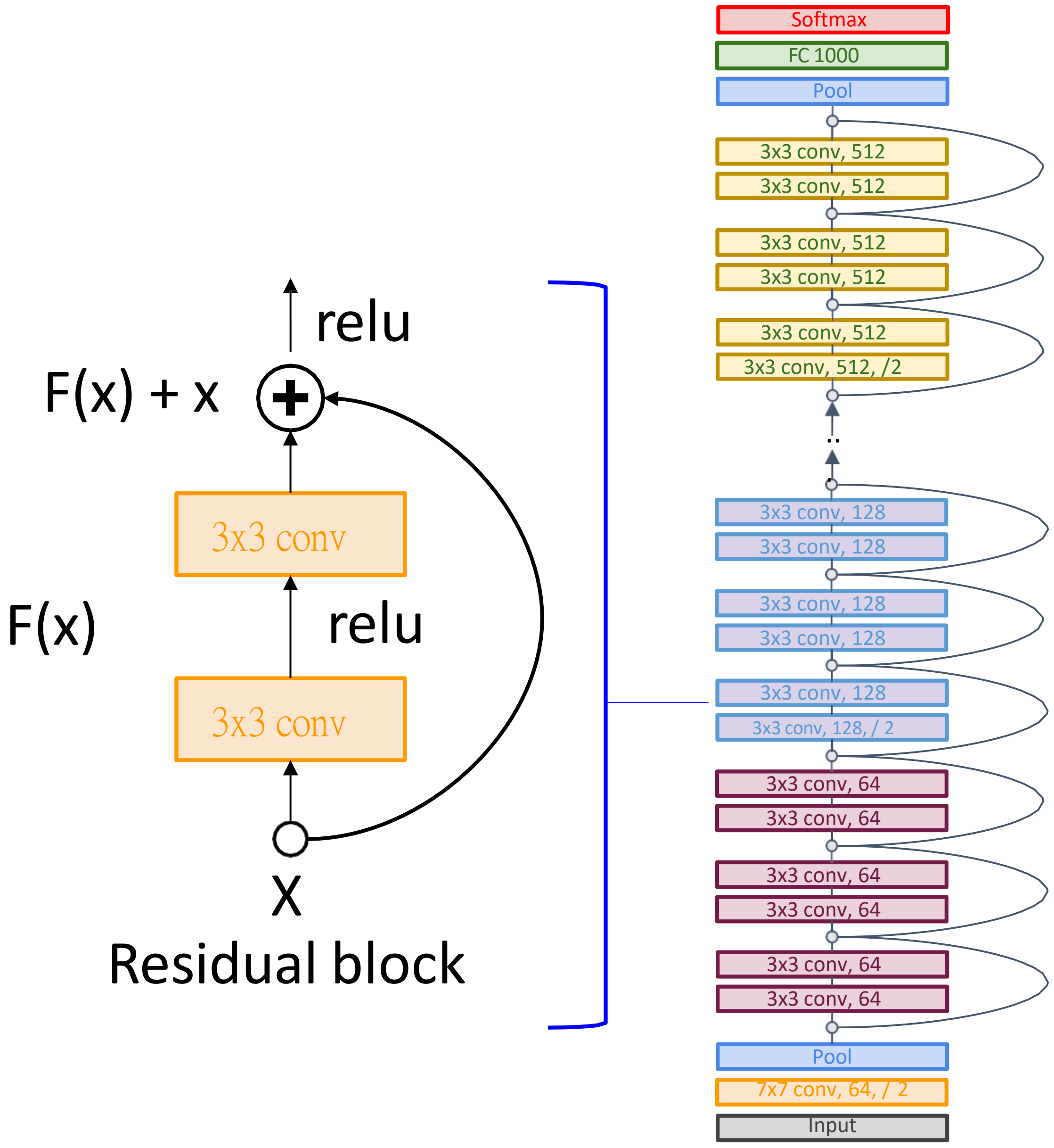

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

# 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

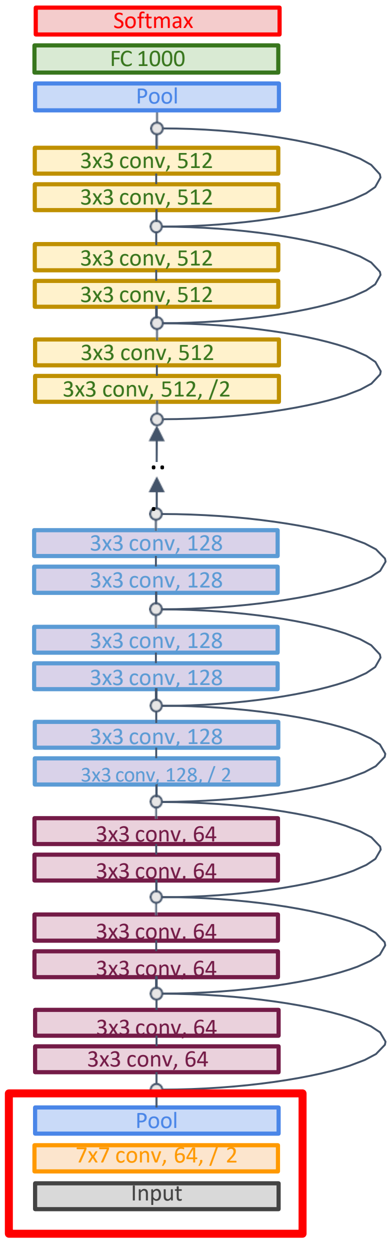



He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

# Residual Networks

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

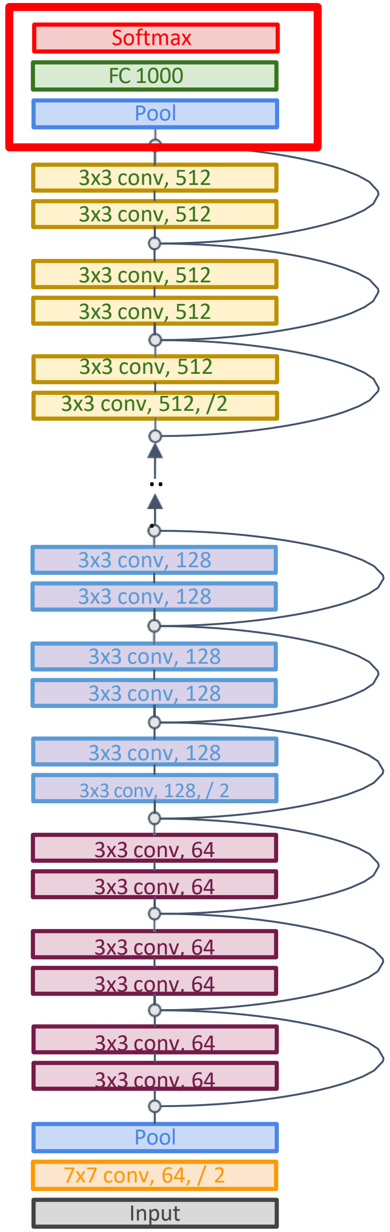
| | Input size | | Layer | | | | Output size | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Layer | C | H/W | filters | kernel | stride | pad | C | H/W | memory (KB) | params (k) | flop (M) |
| conv | 3 | 224 | 64 | 7 | 2 | 3 | 64 | 112 | 3136 | 9 | 118 |
| max-pool | 64 | 112 | | 3 |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




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

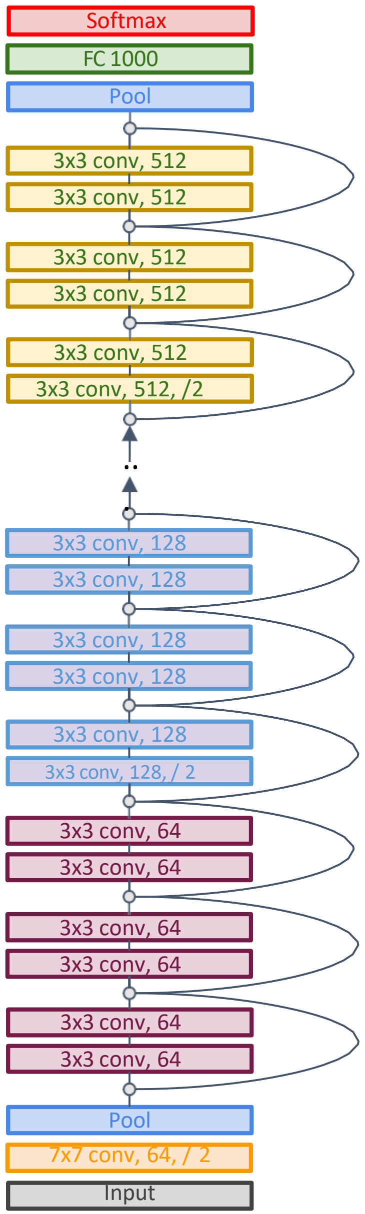



He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

# 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

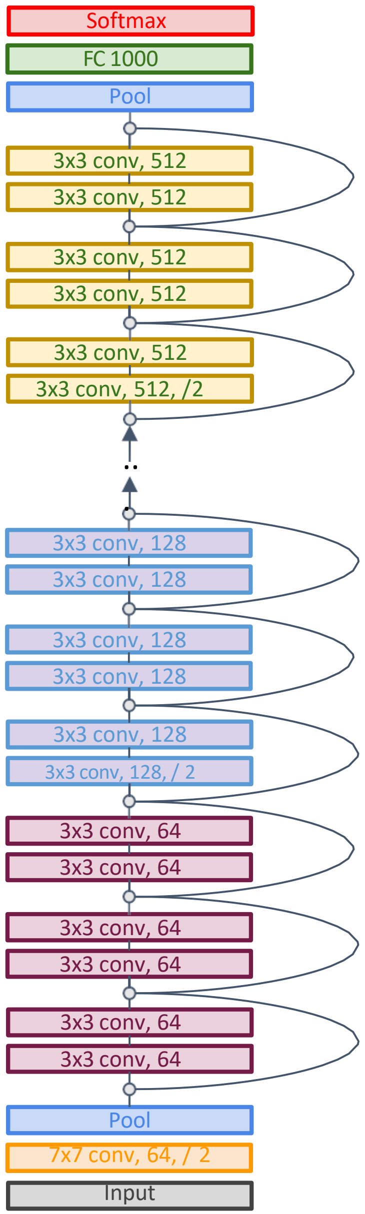
# Residual Networks

**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](torchvision)

# Residual Networks

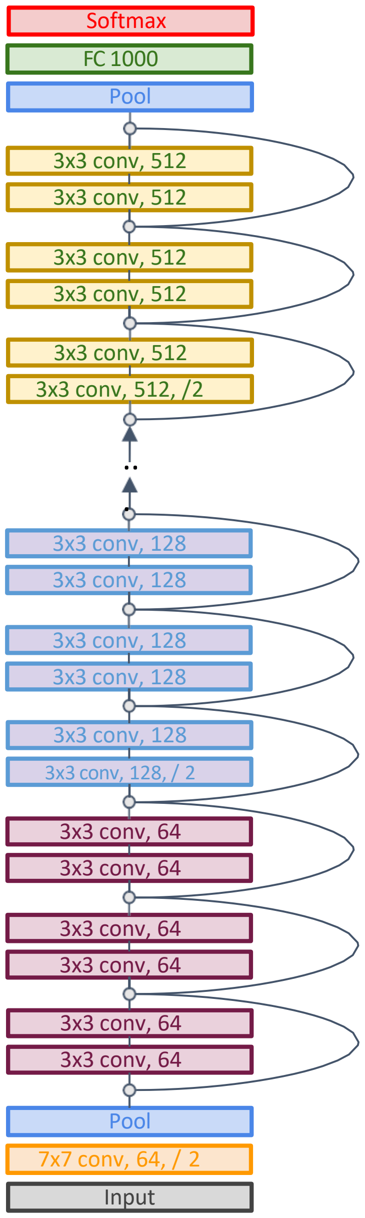
**ResNet-18**:
Stem: 1 conv layer
Stage 1 (C=64): 2 res. block = 4 conv
Stage 2 (C=128): 2 res. block = 4 conv
Stage 3 (C=256): 2 res. block = 4 conv
Stage 4 (C=512): 2 res. block = 4 conv
Linear

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

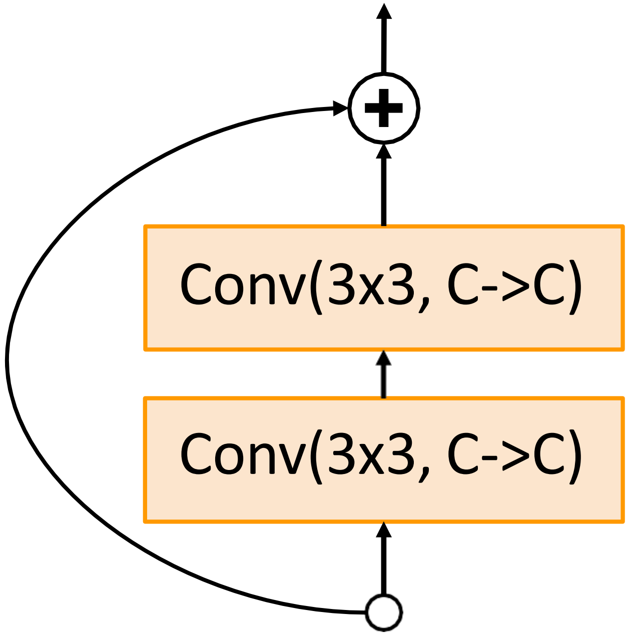
ImageNet top-5 error: 8.58
GFLOP: 3.6

**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]

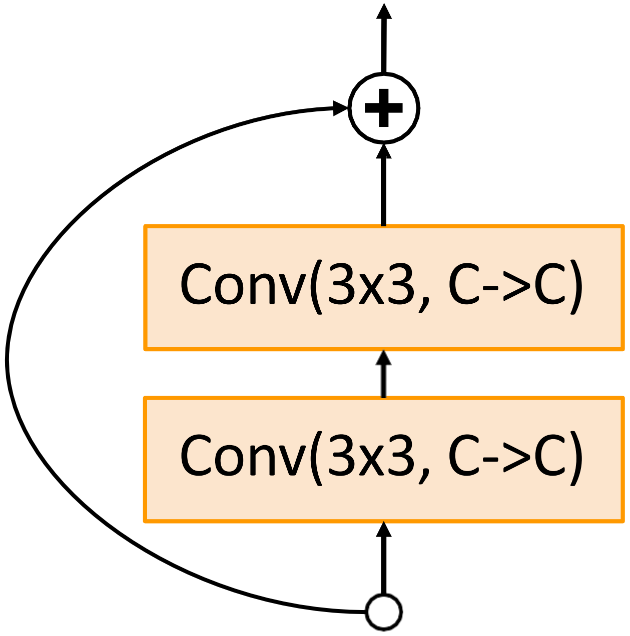
# Residual Networks: Basic Block




"Basic"
Residual block

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

# Residual Networks: Basic Block




FLOPs: $9HWC^2$

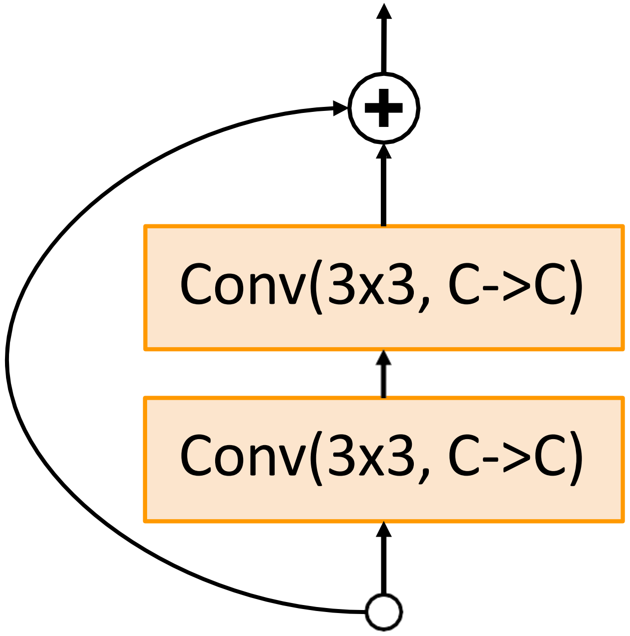
FLOPs: $9HWC^2$

"Basic"
Residual block

Total FLOPs:
$18HWC^2$

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

# Residual Networks: Bottleneck Block

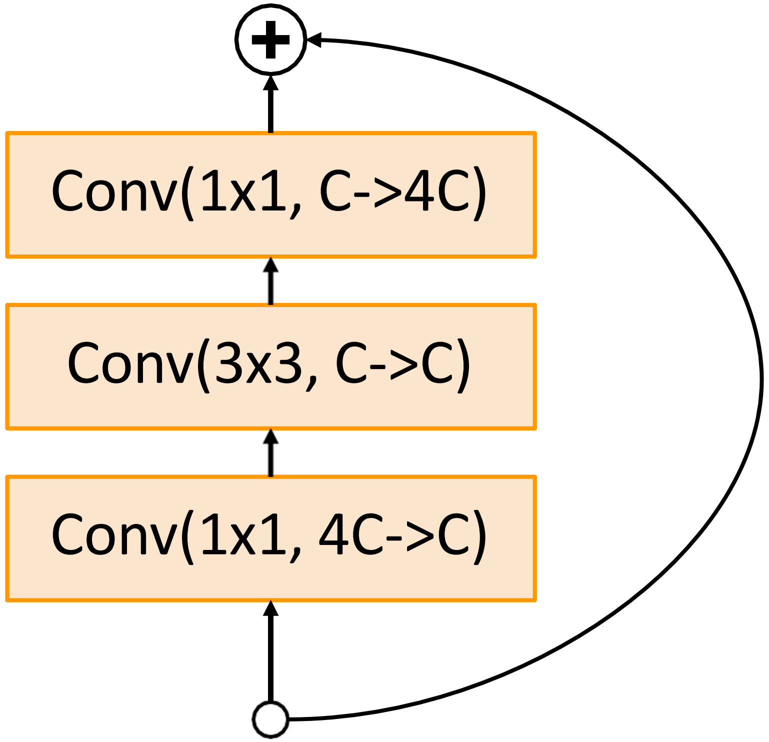



FLOPs: $9HWC^2$

FLOPs: $9HWC^2$

"Basic"
Residual block

Total FLOPs:
$18HWC^2$

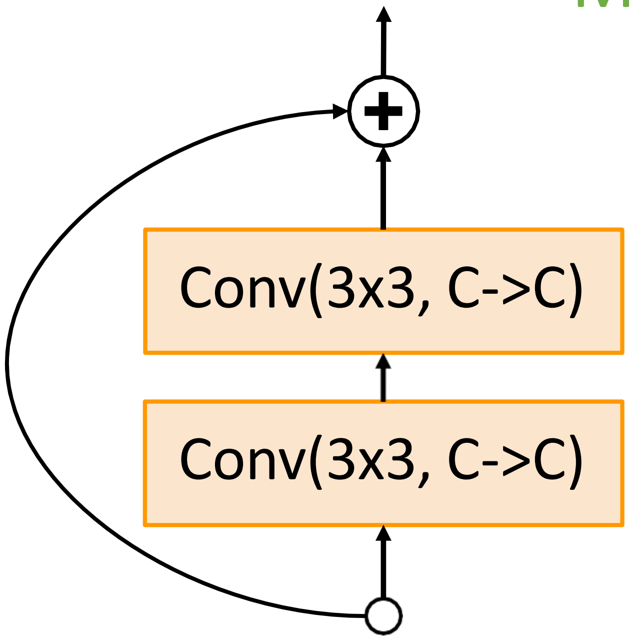



"Bottleneck"
Residual block

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

# Residual Networks: Bottleneck Block

More layers, less computational cost!




FLOPs: $9HWC^2$

FLOPs: $9HWC^2$

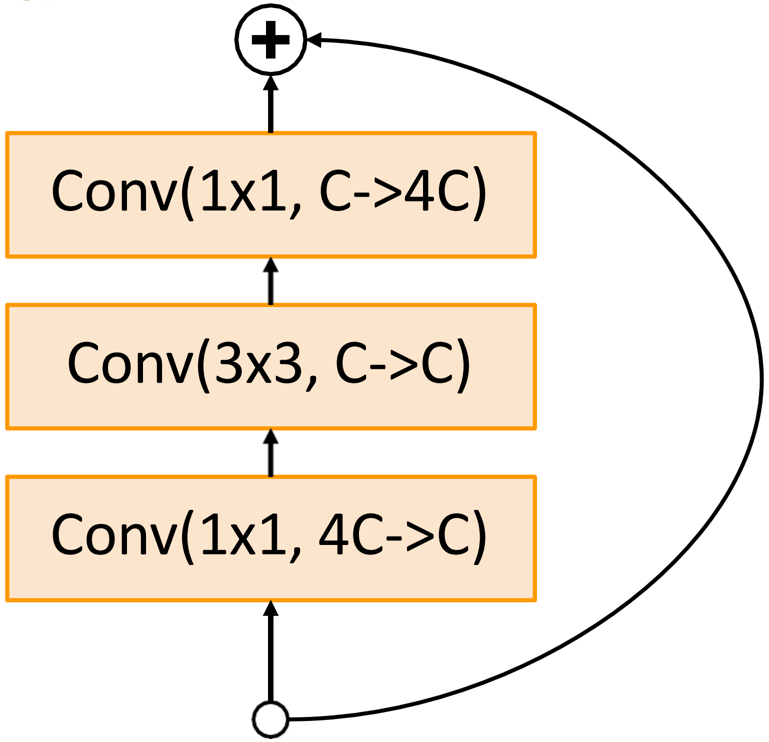
"Basic"
Residual block

Total FLOPs:
$18HWC^2$

FLOPs: $4HWC^2$

FLOPs: $9HWC^2$

FLOPs: $4HWC^2$



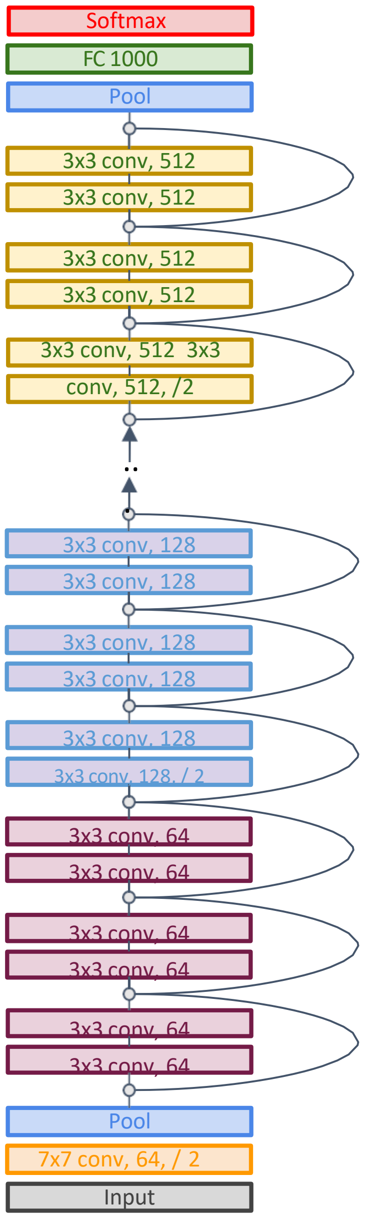

Total FLOPs:
$17HWC^2$

"Bottleneck"
Residual block

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

# 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 |

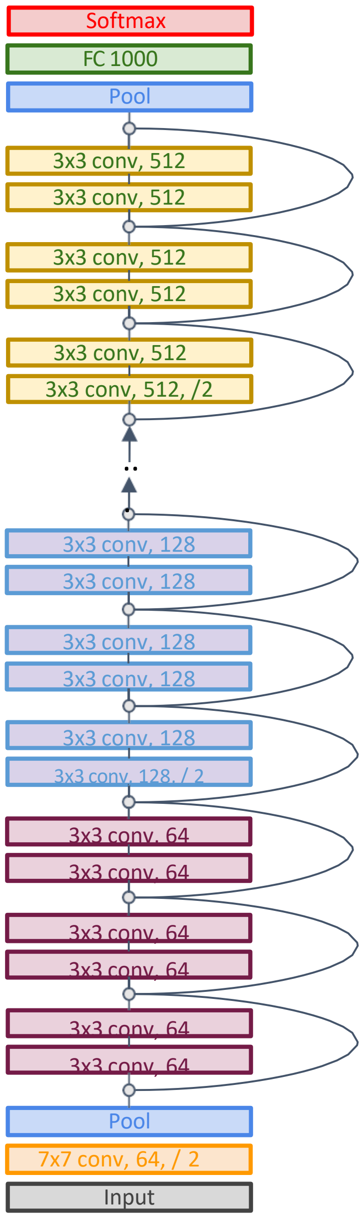



He et al, “Deep Residual Learning for Image Recognition”, CVPR 2016
Error rates are 224x224 single-crop testing, reported by torchvision

# 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 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 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 |

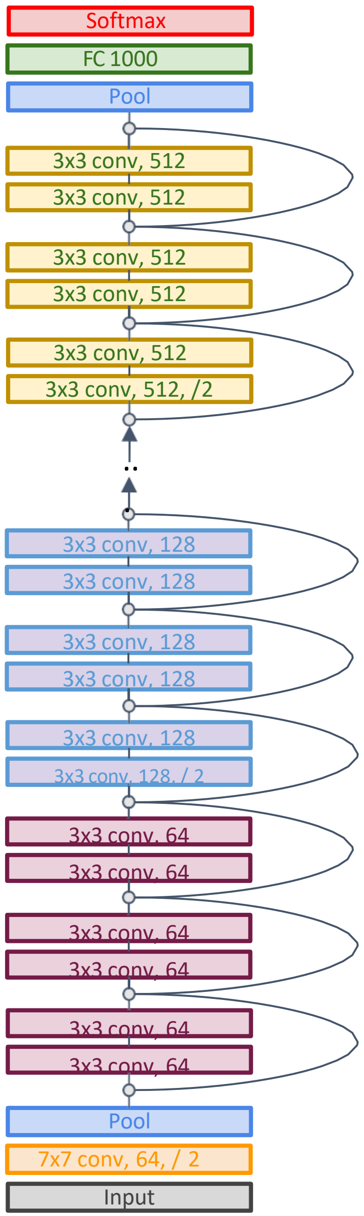



He et al, "Deep Residual Learning for Image Recognition", CVPR 2016
Error rates are 224x224 single-crop testing, reported by torchvision

# 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 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 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 |




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!

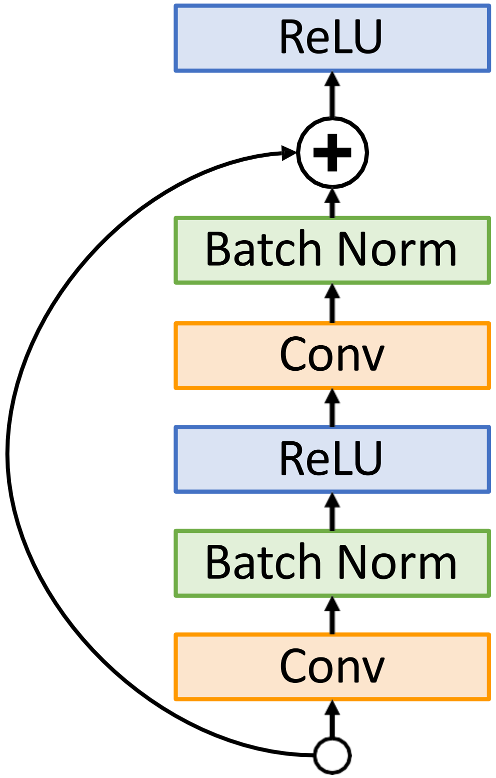
MSRA @ ILSVRC & COCO 2015 Competitions

- **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

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

# Improving Residual Networks: Block Design

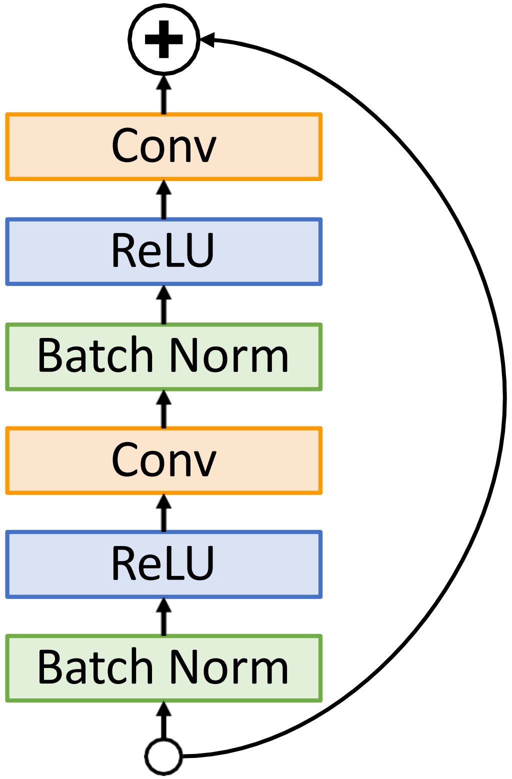
## Original ResNet block




Note ReLU **after** residual:

Cannot actually learn identity function since outputs are nonnegative!

## "Pre-Activation" ResNet Block



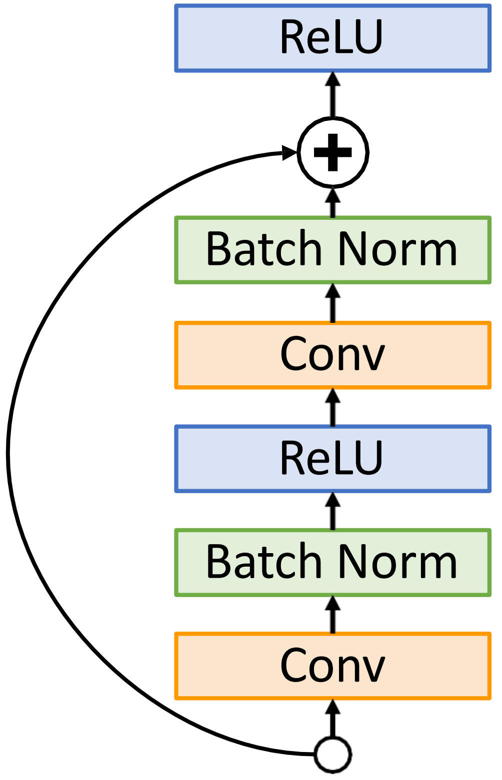

Note ReLU **inside** residual:

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

He et al, "Identity mappings in deep residual networks", ECCV 2016

# Improving Residual Networks: Block Design

Original ResNet block

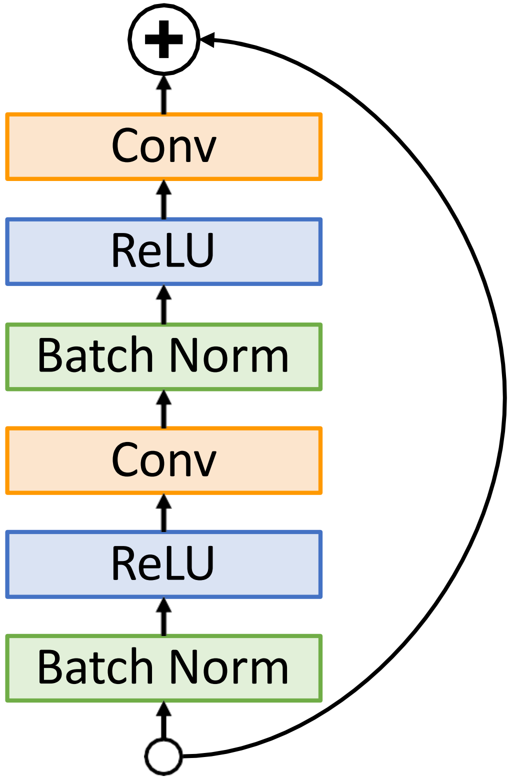



Slight improvement in accuracy (ImageNet top-1 error)

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

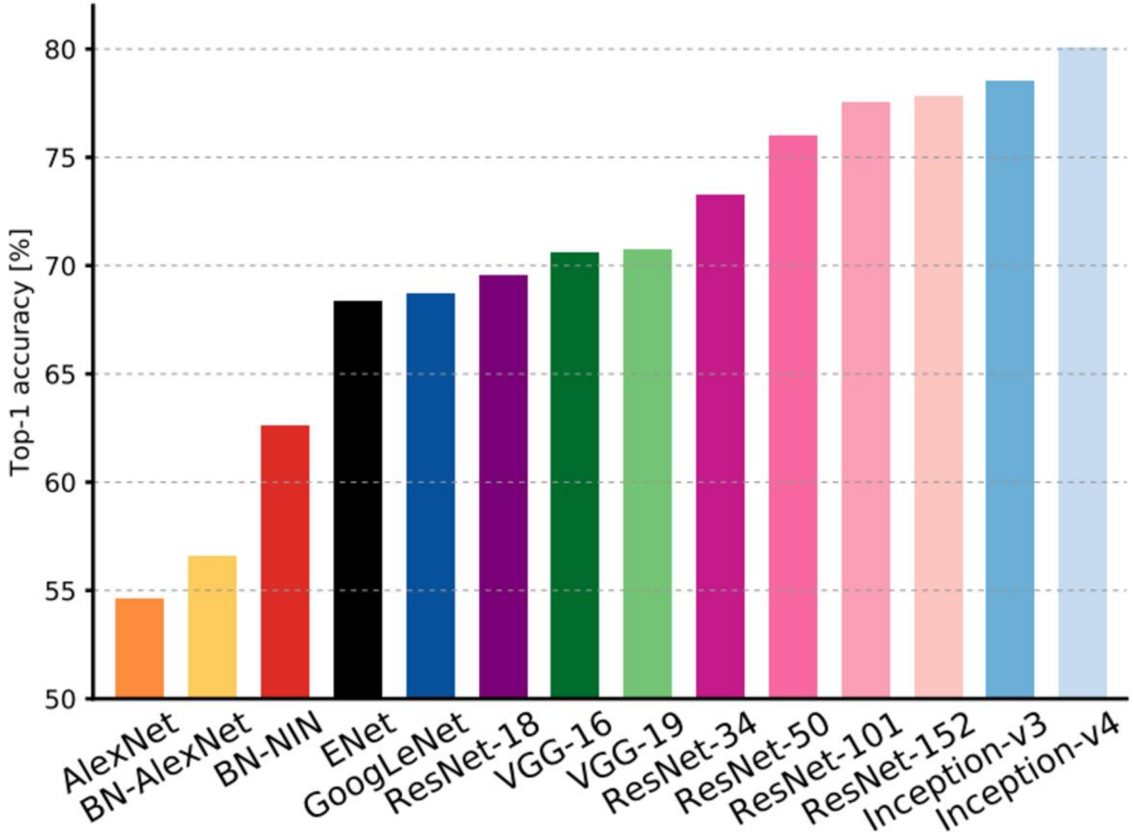
Not actually used that much in practice

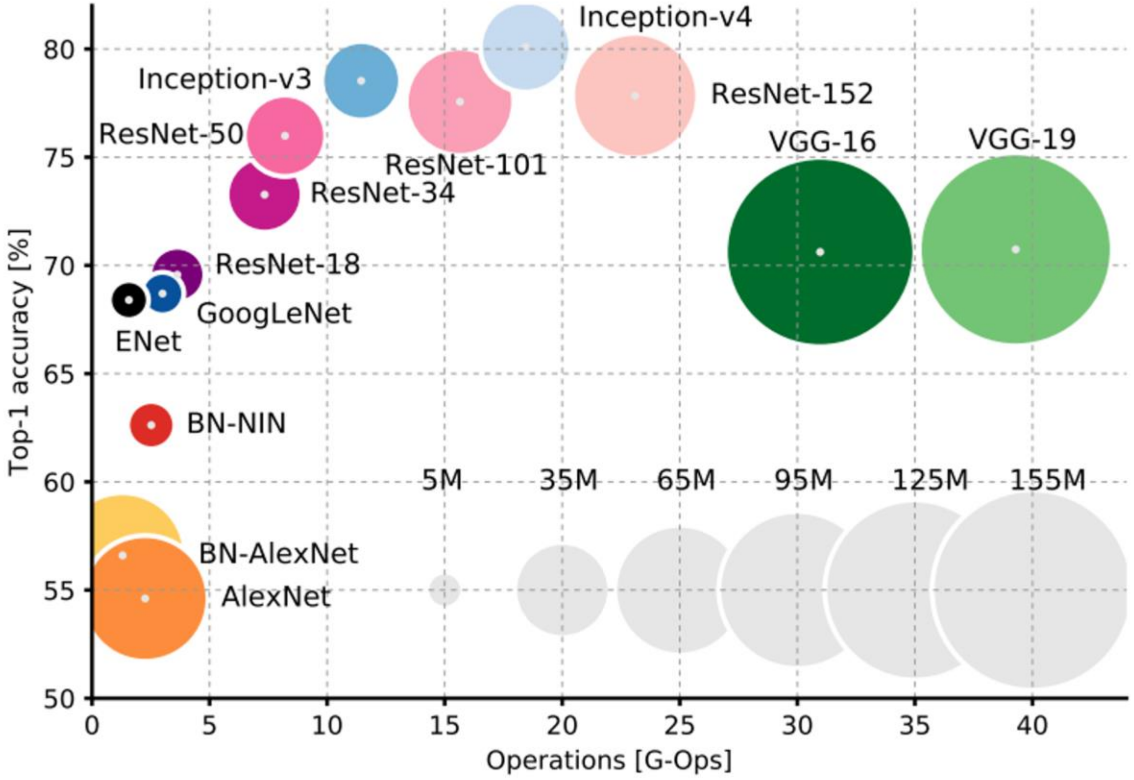
"Pre-Activation" ResNet Block




He et al, "Identity mappings in deep residual networks", ECCV 2016

# Comparing Complexity

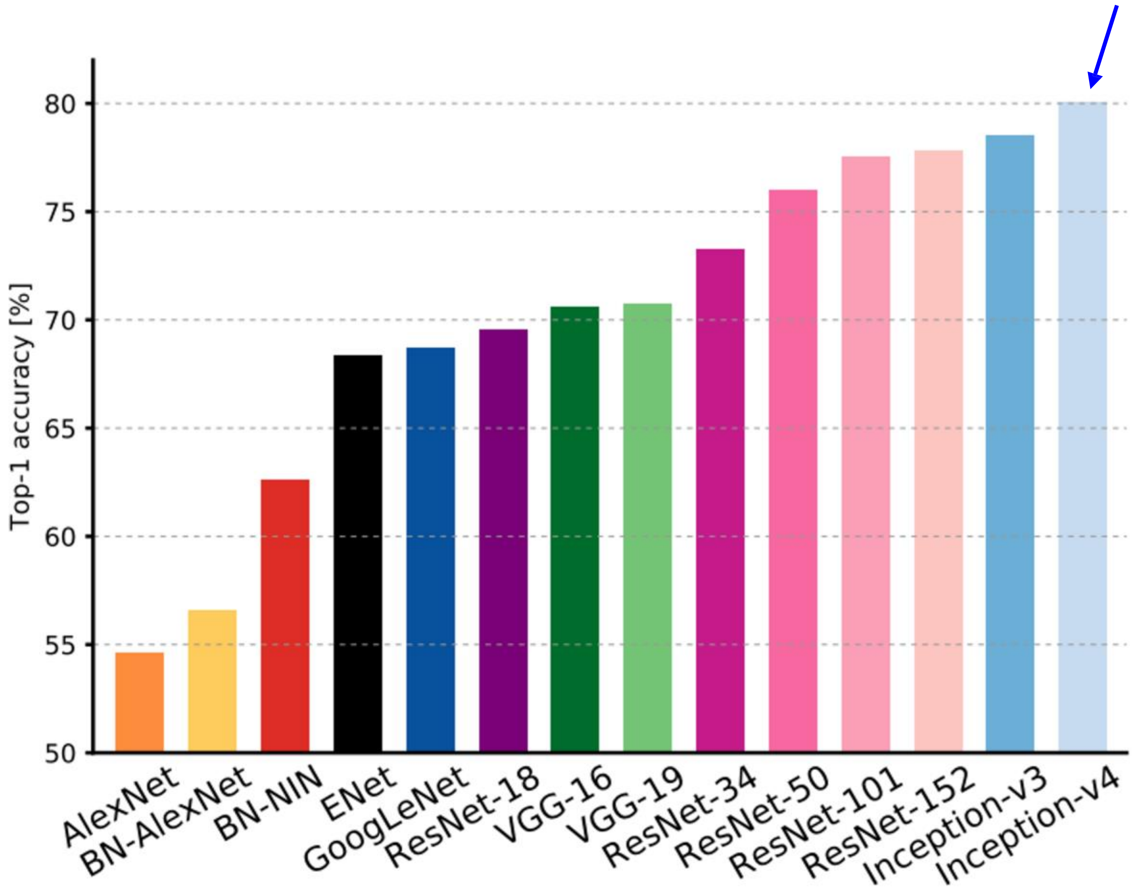


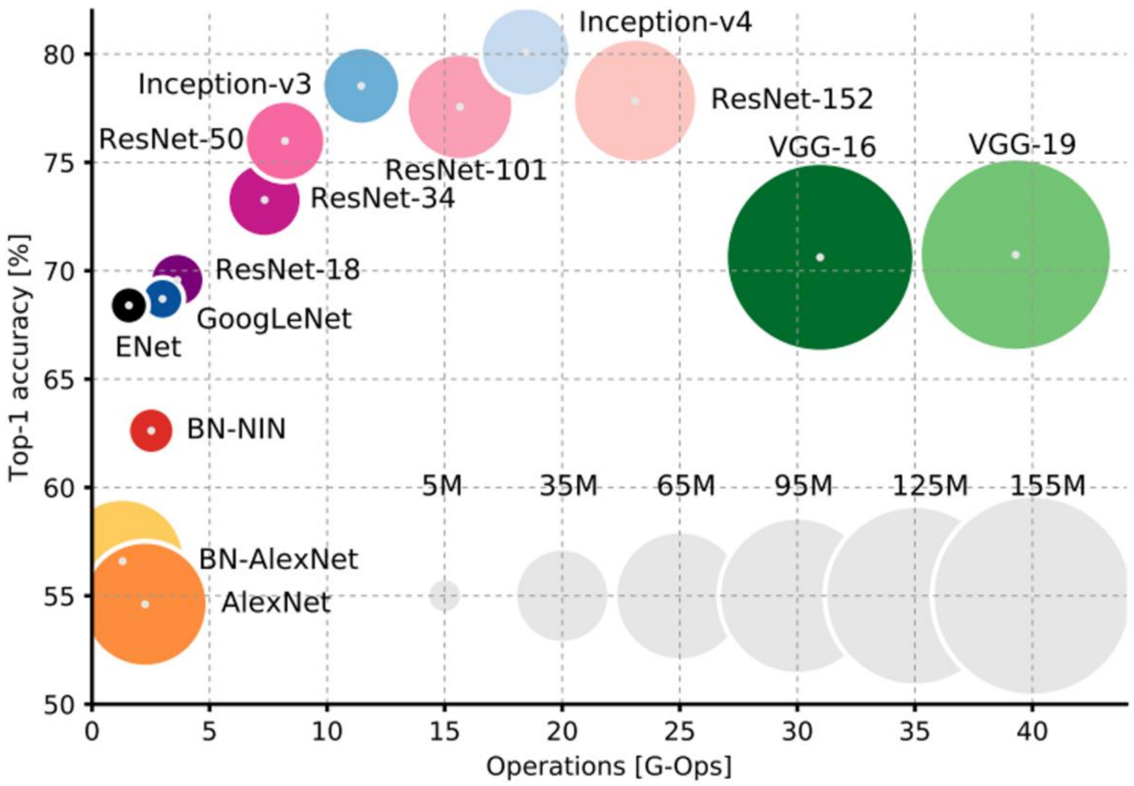




Canziani et al, "An analysis of deep neural network models for practical applications", 2017

# Comparing Complexity

Inception-v4: Resnet + Inception!

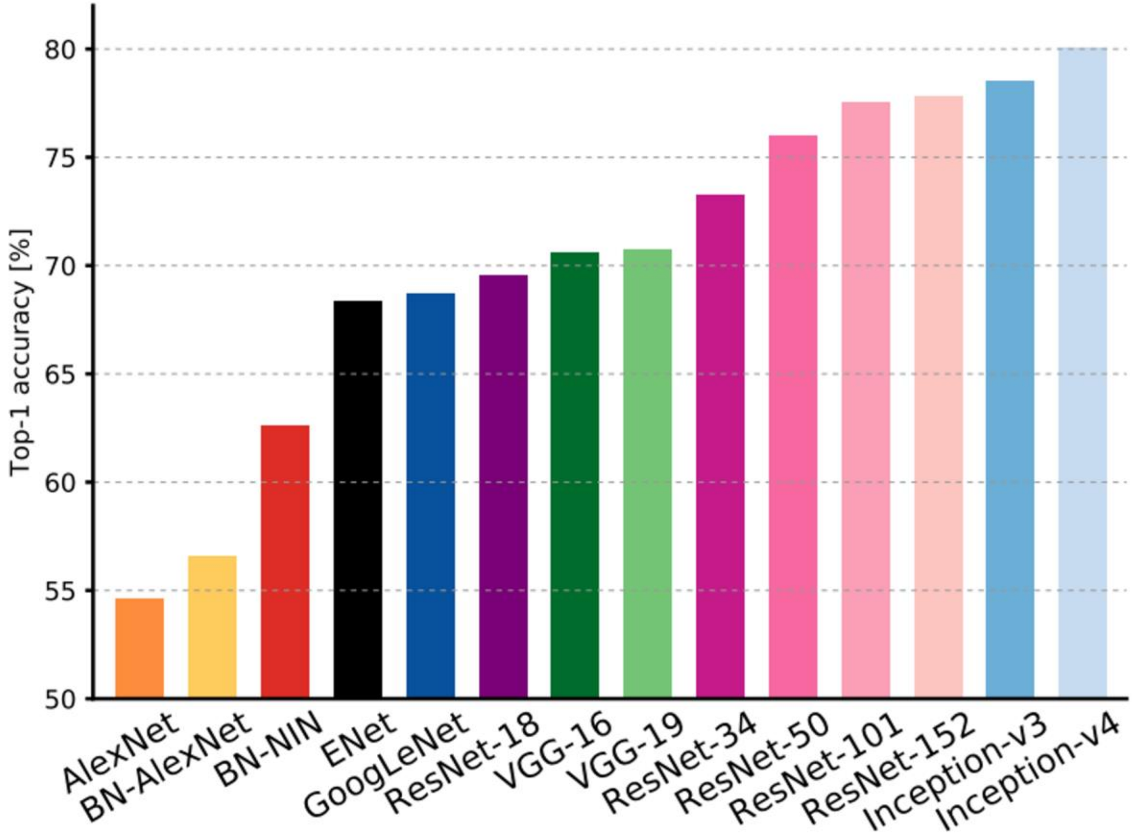


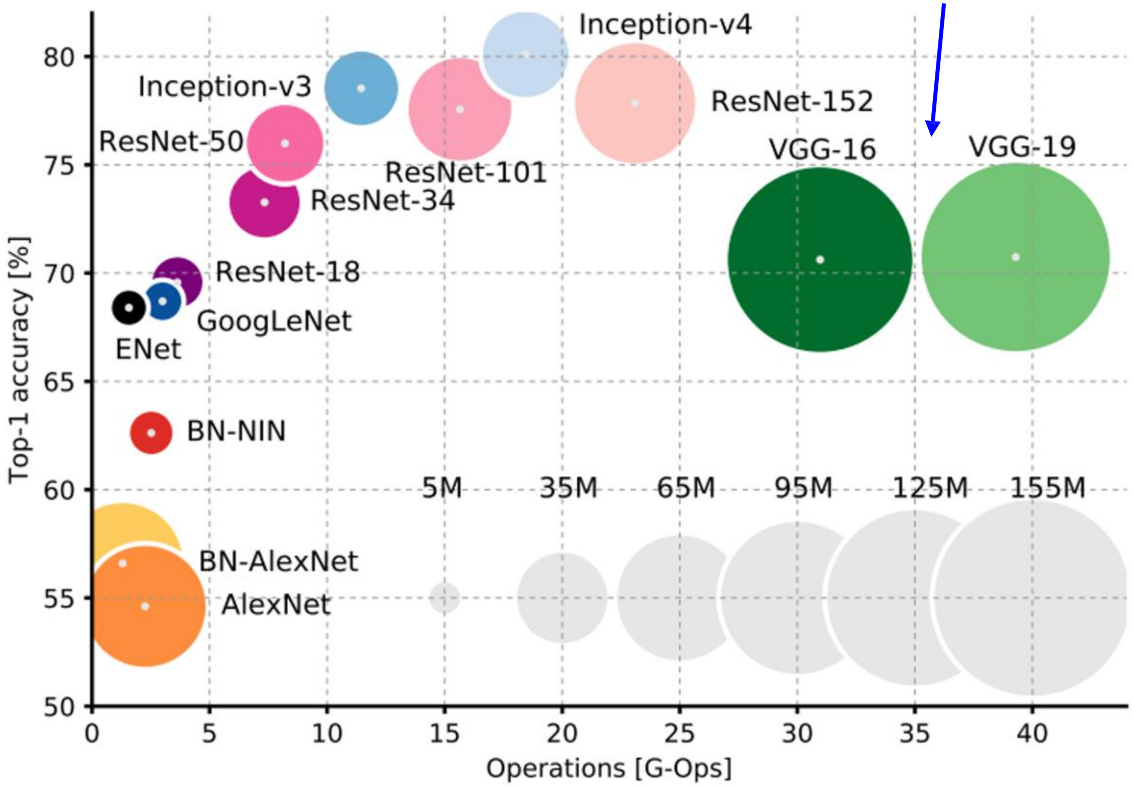




Canziani et al, "An analysis of deep neural network models for practical applications", 2017

# Comparing Complexity

VGG: Highest memory, most operations

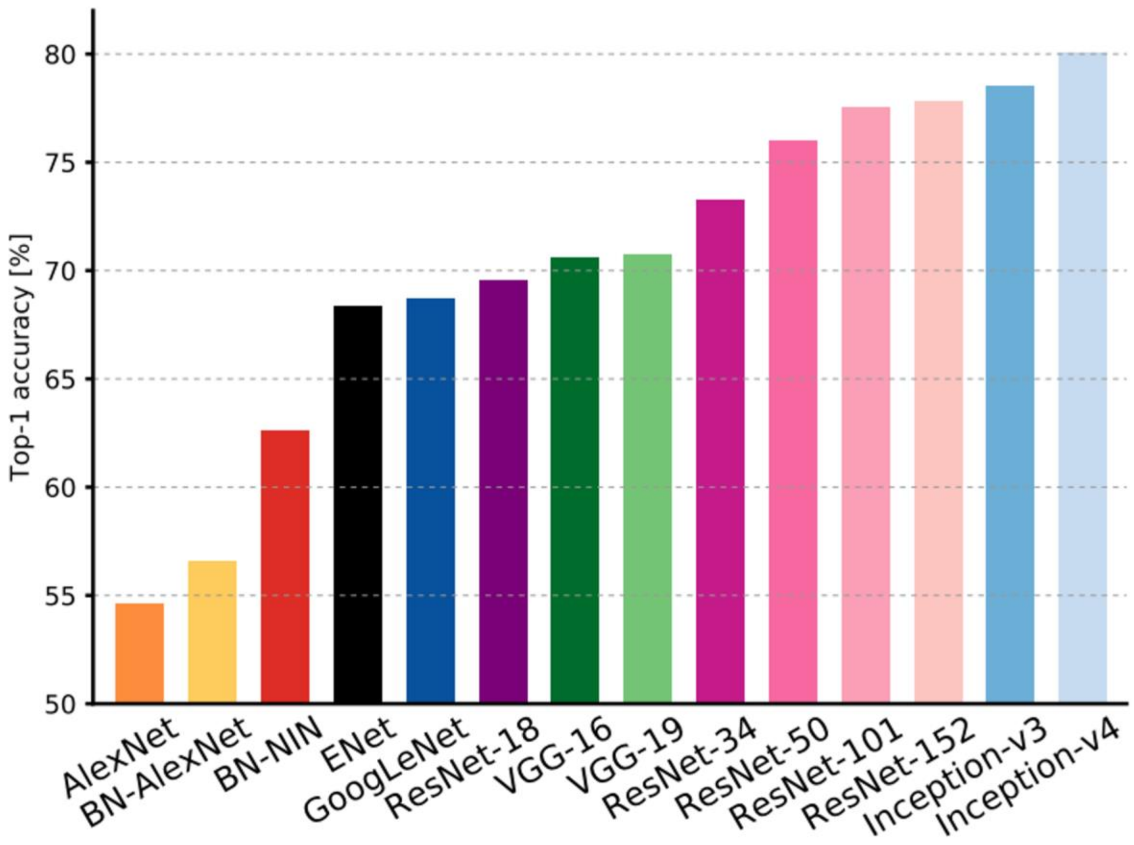


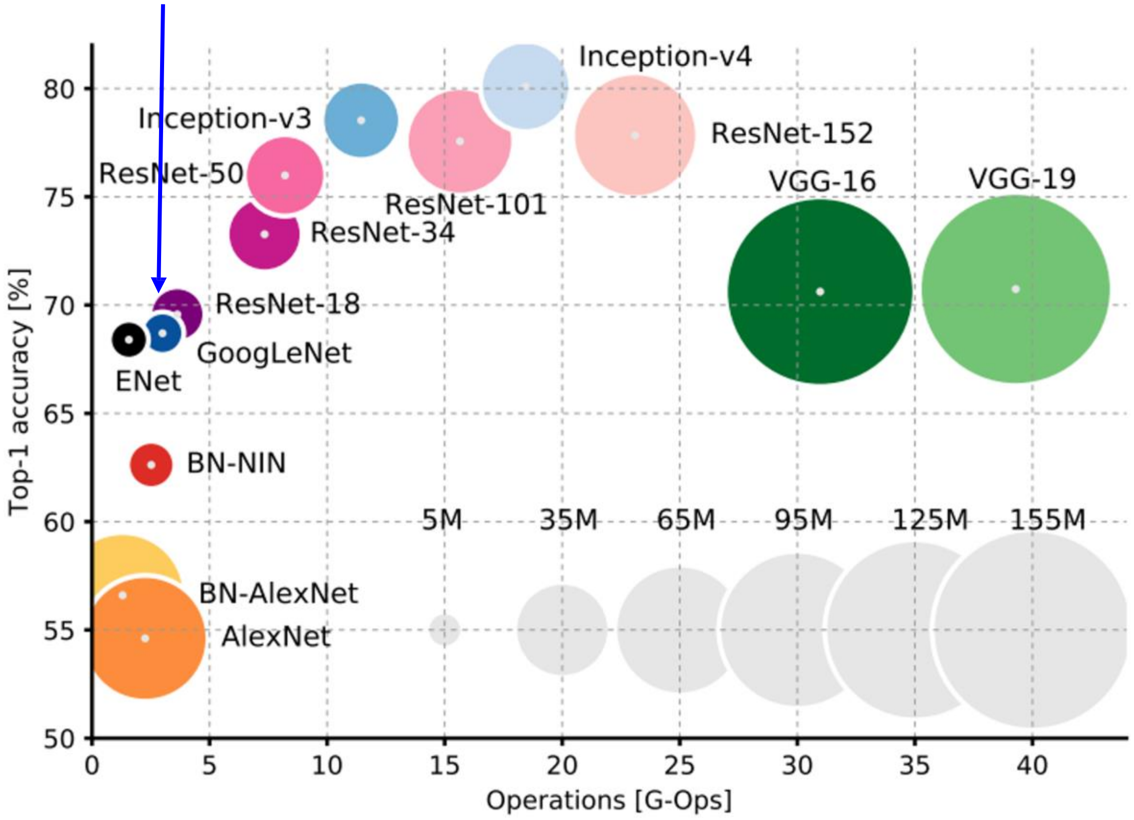




Canziani et al, “An analysis of deep neural network models for practical applications”, 2017

# Comparing Complexity

GoogLeNet:
Very efficient!

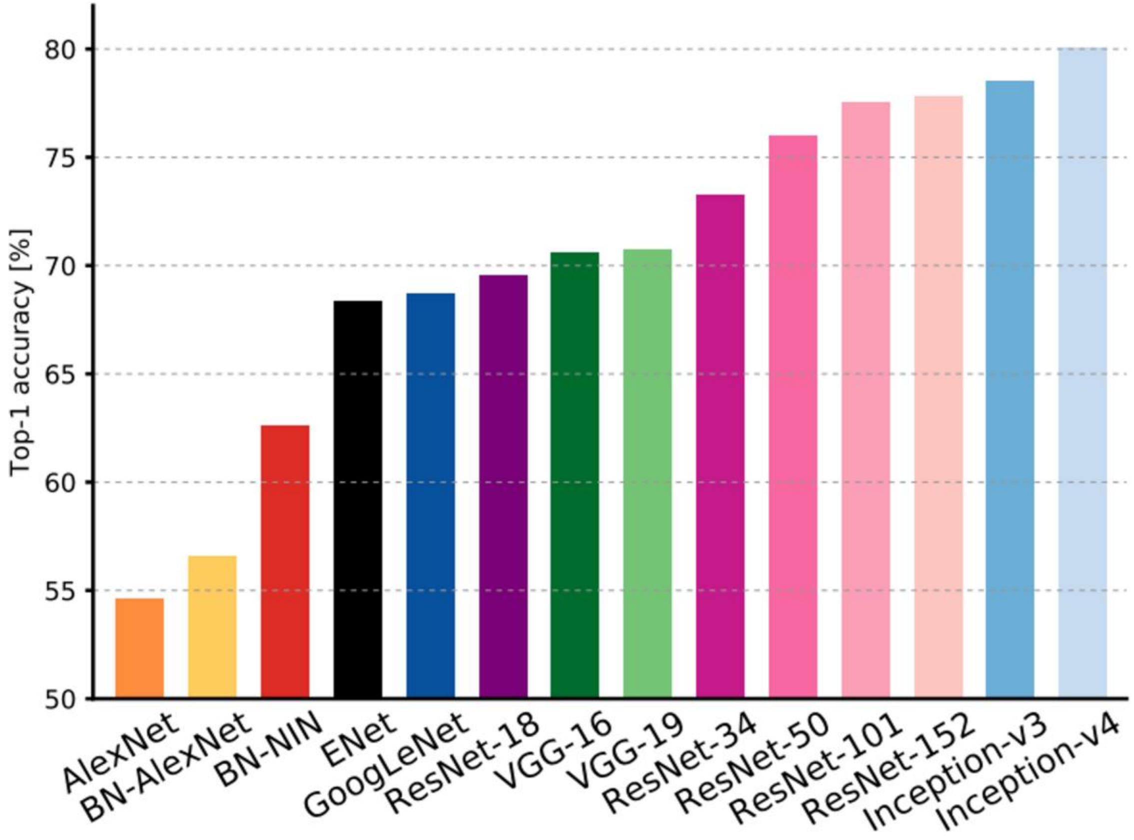


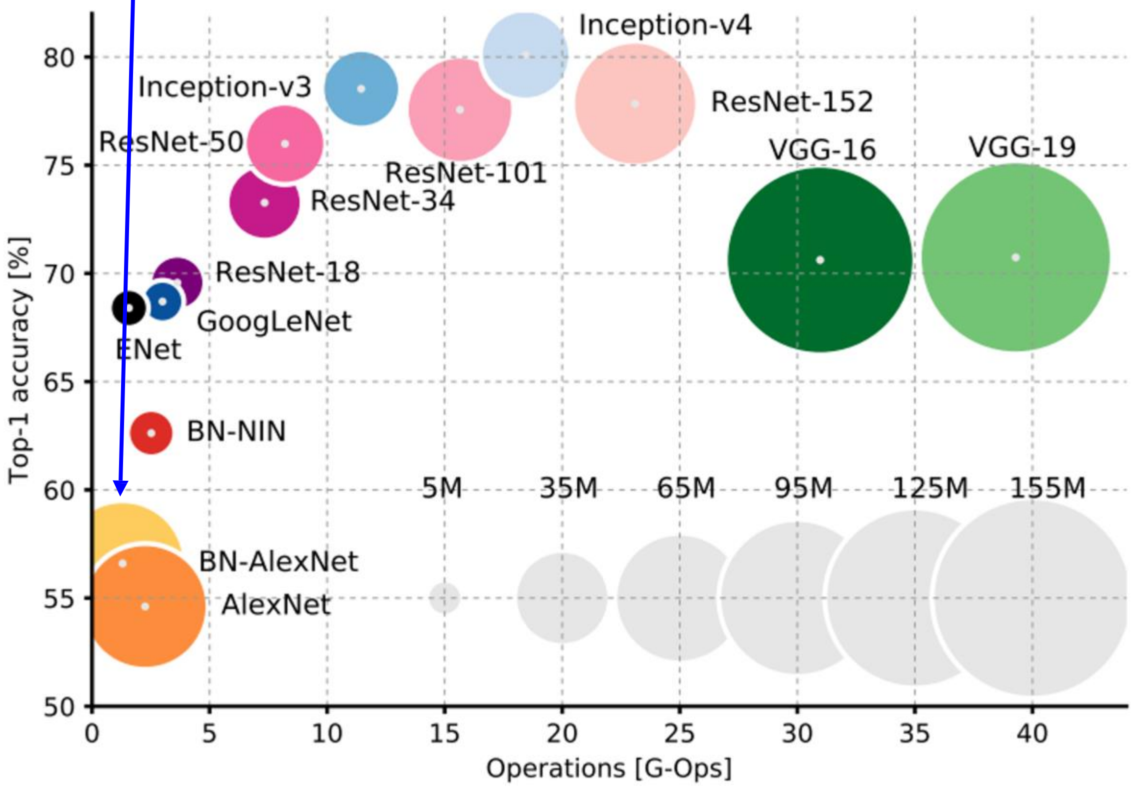




Canziani et al, "An analysis of deep neural network models for practical applications", 2017

# Comparing Complexity

AlexNet: Low compute, lots of parameters

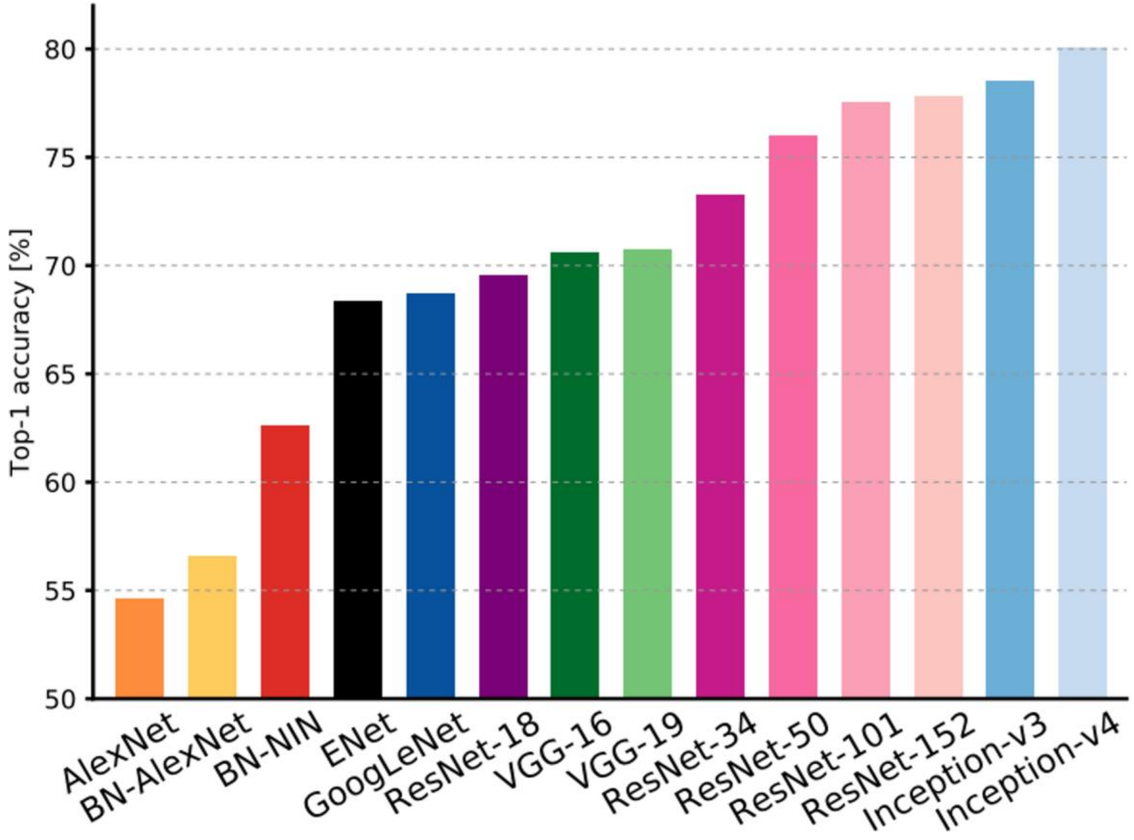


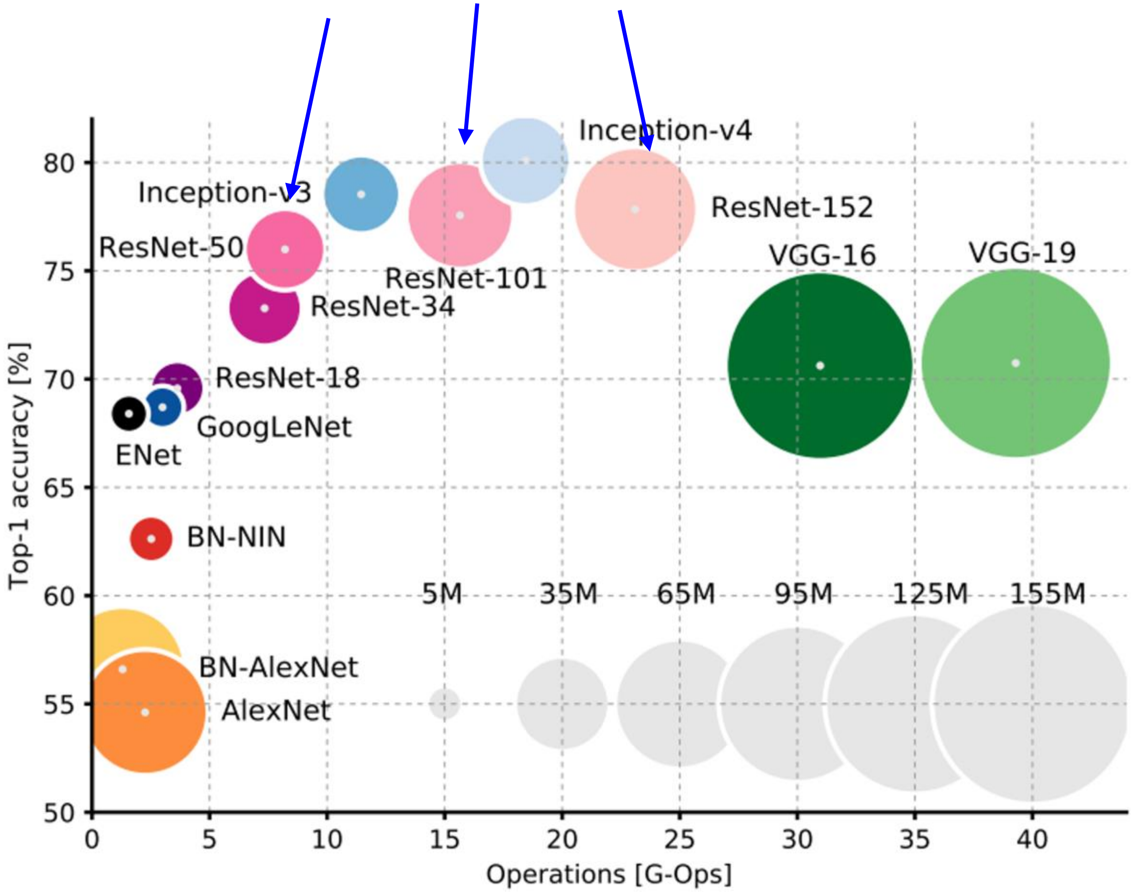




Canziani et al, "An analysis of deep neural network models for practical applications", 2017

# Comparing Complexity

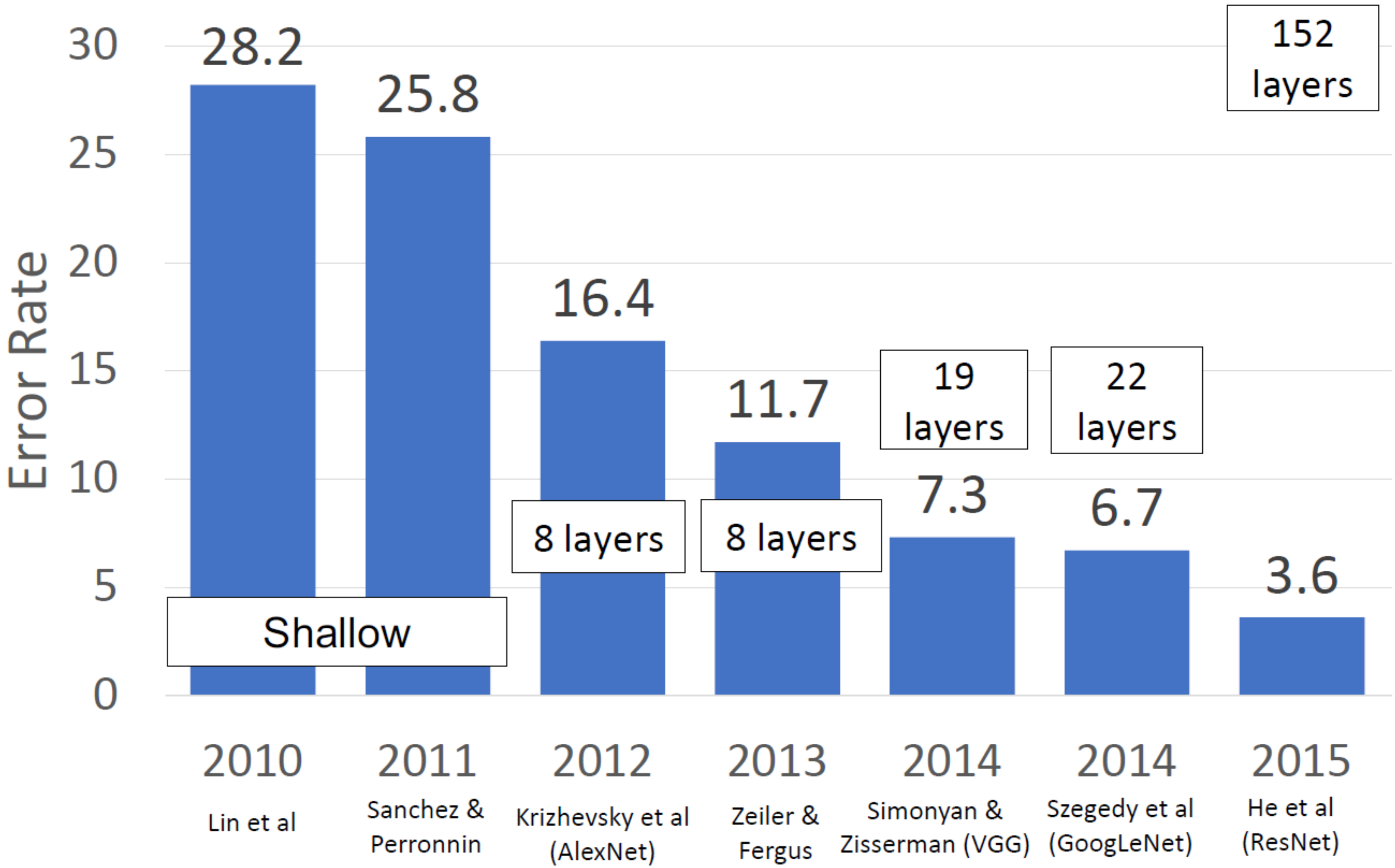
ResNet: Simple design,
moderate efficiency,
high accuracy



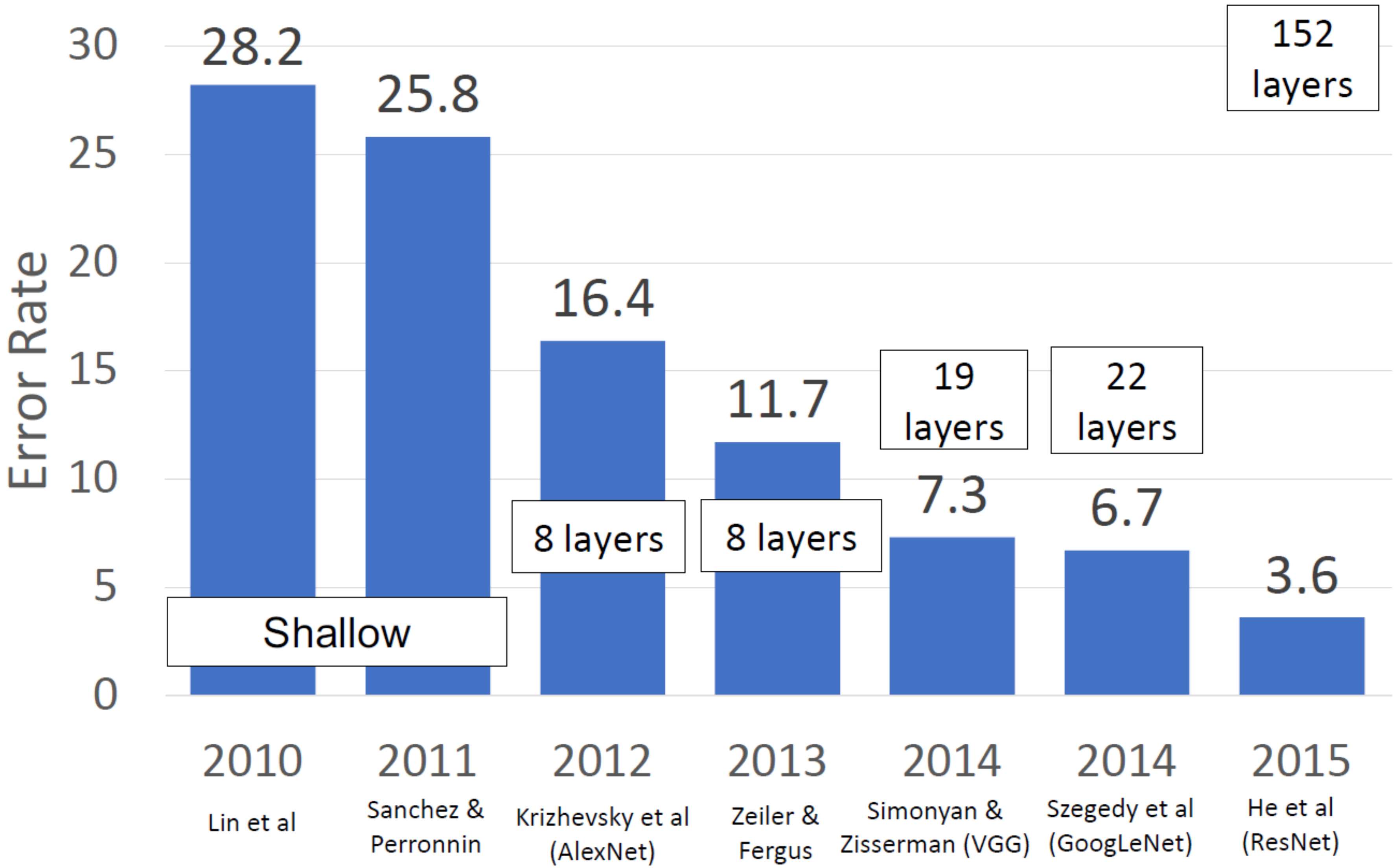




Canziani et al, “An analysis of deep neural network models for practical applications”, 2017

# ImageNet Classification Challenge



# ImageNet Classification Challenge




CNN architectures have continued to evolve!

We will see more in Lecture 9

# CNN Architectures Summary

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

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

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

After ResNet: **Efficient networks** became central: how can we improve the accuracy without increasing the complexity? (Lecture 9)

# Which Architecture should I use?

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

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

Next Time:

How to Train your CNN

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