

CNN Case Studies

M. Soleymani

Sharif University of Technology

Spring 2026

Slides are based on Fei Fei Li and colleagues lectures, cs231n, Stanford

AlexNet

[Krizhevsky, Sutskever, Hinton, 2012]

- ImageNet Classification with Deep Convolutional Neural Networks

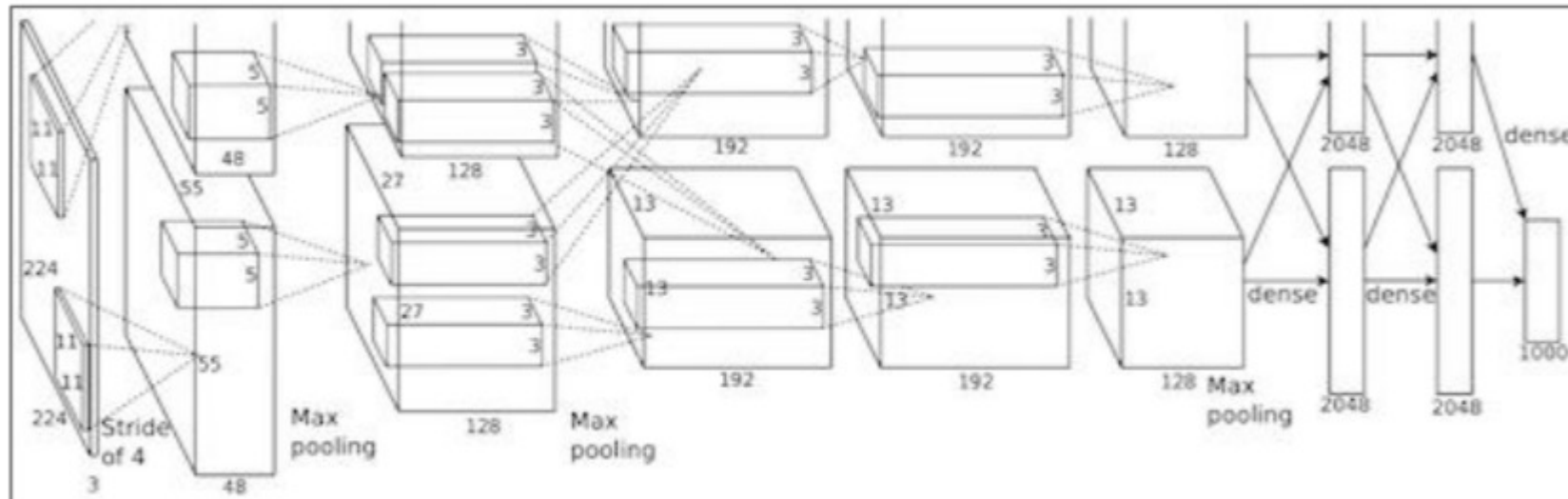
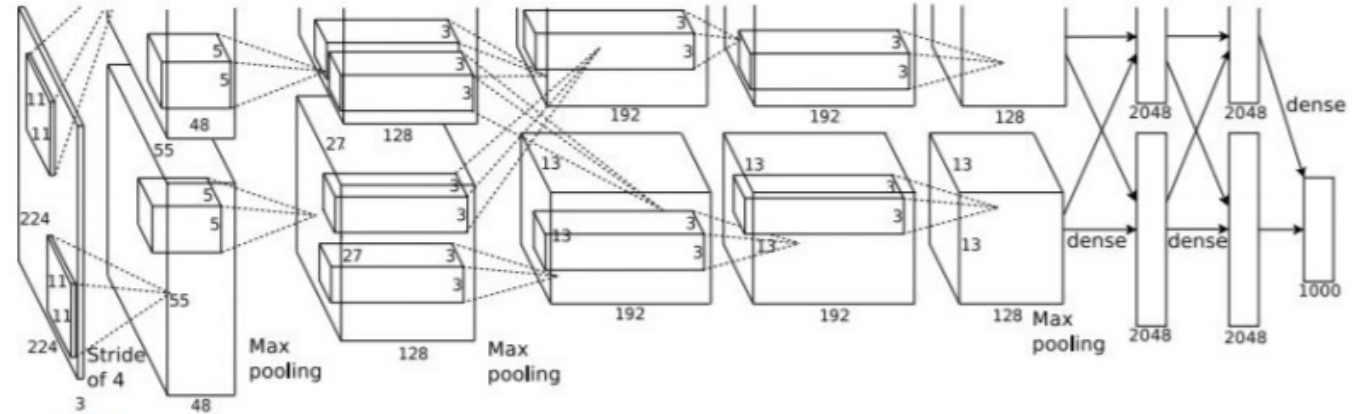


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

CNN Architectures

- Case Studies
 - AlexNet
 - VGG
 - GoogLeNet
 - ResNet
- Also....
 - Wide ResNet
 - ResNeXT
 - DenseNet
 - SqueezeNet
 - MobileNet
 - EfficientNet

Case Study: AlexNet



Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] **CONV1**: 96 11x11 filters at stride 4, pad 0

[27x27x96] **MAX POOL1**: 3x3 filters at stride 2

[27x27x96] **NORM1**: Normalization layer

[27x27x256] **CONV2**: 256 5x5 filters at stride 1, pad 2

[13x13x256] **MAX POOL2**: 3x3 filters at stride 2

[13x13x256] **NORM2**: Normalization layer

[13x13x384] **CONV3**: 384 3x3 filters at stride 1, pad 1

[13x13x384] **CONV4**: 384 3x3 filters at stride 1, pad 1

[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

[6x6x256] **MAX POOL3**: 3x3 filters at stride 2

[4096] **FC6**: 4096 neurons

[4096] **FC7**: 4096 neurons

[1000] **FC8**: 1000 neurons (class scores)

Input: 227x227x3

First layer (CONV1): 11x11x3 stride 4

=>

Output: $(227-11)/4+1 = 55$

Parameters: $(11*11*3)*96 = 35K$

Case Study: AlexNet

Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] **CONV1**: 96 11x11 filters at stride 4, pad 0

[27x27x96] **MAX POOL1**: 3x3 filters at stride 2

[27x27x96] **NORM1**: Normalization layer

[27x27x256] **CONV2**: 256 5x5 filters at stride 1, pad 2

[13x13x256] **MAX POOL2**: 3x3 filters at stride 2

[13x13x256] **NORM2**: Normalization layer

[13x13x384] **CONV3**: 384 3x3 filters at stride 1, pad 1

[13x13x384] **CONV4**: 384 3x3 filters at stride 1, pad 1

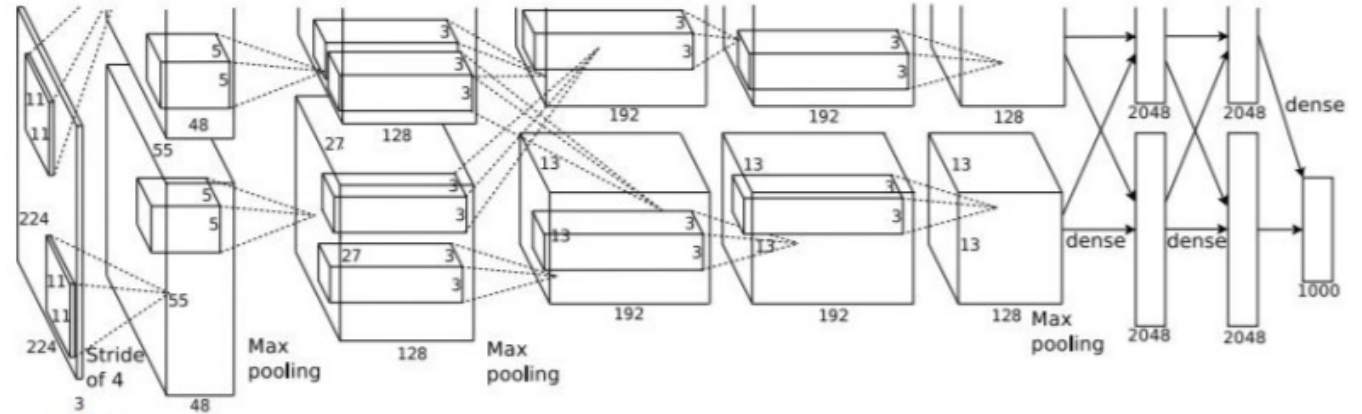
[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

[6x6x256] **MAX POOL3**: 3x3 filters at stride 2

[4096] **FC6**: 4096 neurons

[4096] **FC7**: 4096 neurons

[1000] **FC8**: 1000 neurons (class scores)



Second layer (POOL1): 3x3 filters stride 2

Output volume: 27x27x96

#Parameters: 0!

Case Study: AlexNet

Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] **CONV1**: 96 11x11 filters at stride 4, pad 0

[27x27x96] **MAX POOL1**: 3x3 filters at stride 2

[27x27x96] **NORM1**: Normalization layer

[27x27x256] **CONV2**: 256 5x5 filters at stride 1, pad 2

[13x13x256] **MAX POOL2**: 3x3 filters at stride 2

[13x13x256] **NORM2**: Normalization layer

[13x13x384] **CONV3**: 384 3x3 filters at stride 1, pad 1

[13x13x384] **CONV4**: 384 3x3 filters at stride 1, pad 1

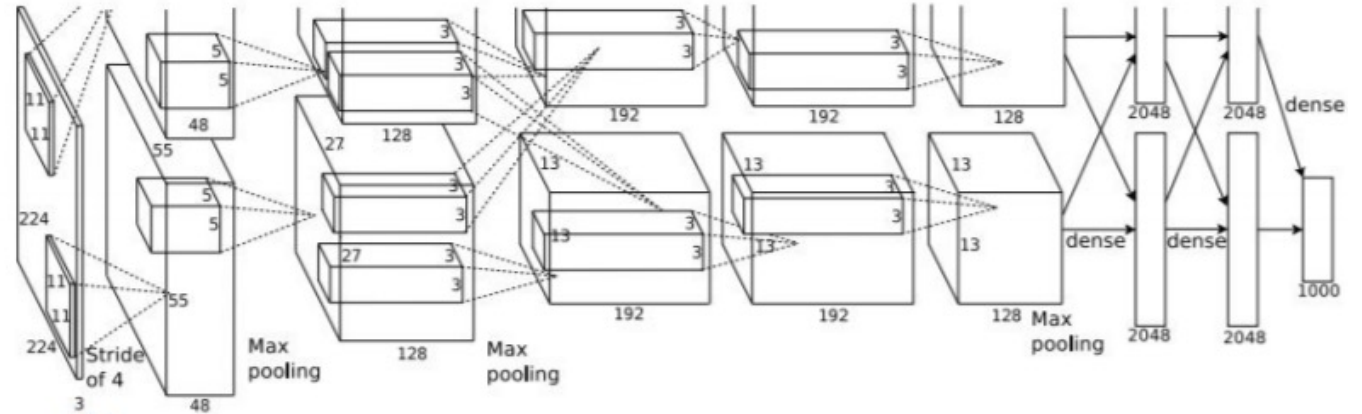
[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

[6x6x256] **MAX POOL3**: 3x3 filters at stride 2

[4096] **FC6**: 4096 neurons

[4096] **FC7**: 4096 neurons

[1000] **FC8**: 1000 neurons (class scores)

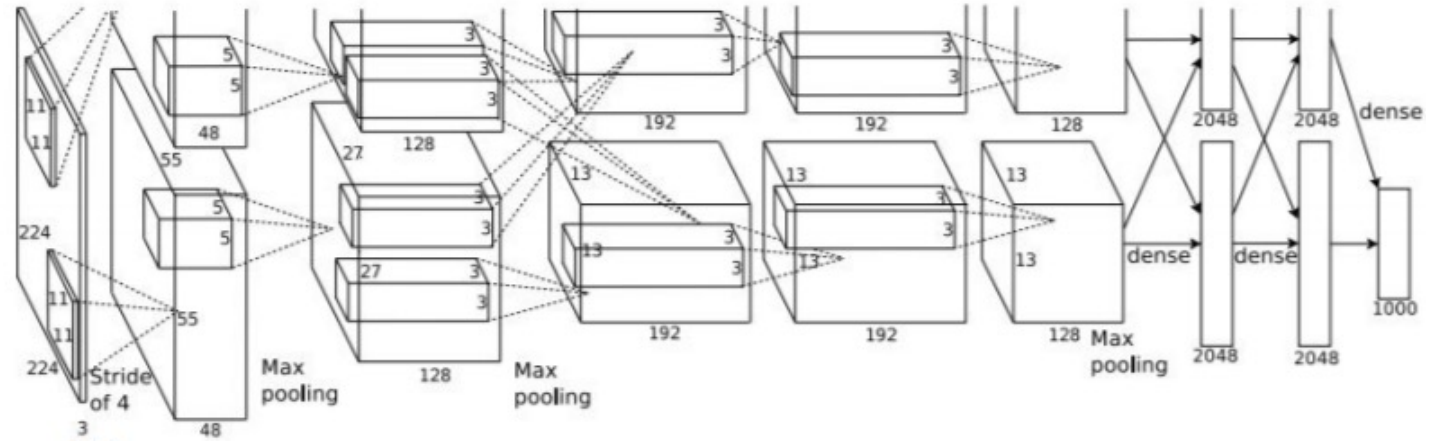


Input: 227x227x3

After CONV1: 55x55x96

After POOL1: 27x27x96

Case Study: AlexNet



Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] **CONV1**: 96 11x11 filters at stride 4, pad 0

[27x27x96] **MAX POOL1**: 3x3 filters at stride 2

[27x27x96] **NORM1**: Normalization layer

[27x27x256] **CONV2**: 256 5x5 filters at stride 1, pad 2

[13x13x256] **MAX POOL2**: 3x3 filters at stride 2

[13x13x256] **NORM2**: Normalization layer

[13x13x384] **CONV3**: 384 3x3 filters at stride 1, pad 1

[13x13x384] **CONV4**: 384 3x3 filters at stride 1, pad 1

[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

[6x6x256] **MAX POOL3**: 3x3 filters at stride 2

[4096] **FC6**: 4096 neurons

[4096] **FC7**: 4096 neurons

[1000] **FC8**: 1000 neurons (class scores)

Details/Retrospectives:

- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

Case Study: AlexNet

Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] **CONV1**: 96 11x11 filters at stride 4, pad 0

[27x27x96] **MAX POOL1**: 3x3 filters at stride 2

[27x27x96] **NORM1**: Normalization layer

[27x27x256] **CONV2**: 256 5x5 filters at stride 1, pad 2

[13x13x256] **MAX POOL2**: 3x3 filters at stride 2

[13x13x256] **NORM2**: Normalization layer

[13x13x384] **CONV3**: 384 3x3 filters at stride 1, pad 1

[13x13x384] **CONV4**: 384 3x3 filters at stride 1, pad 1

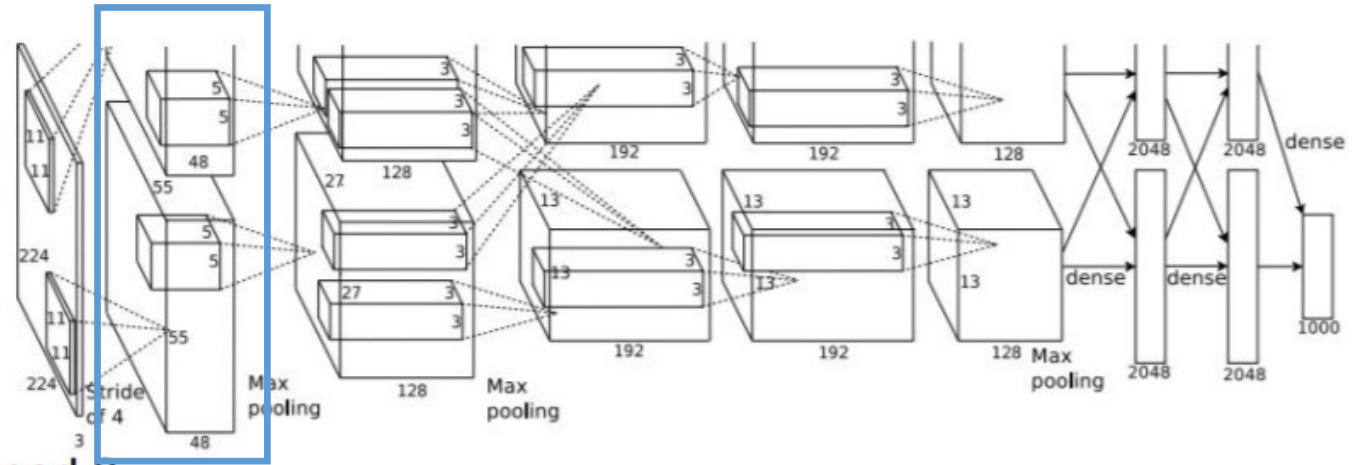
[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

[6x6x256] **MAX POOL3**: 3x3 filters at stride 2

[4096] **FC6**: 4096 neurons

[4096] **FC7**: 4096 neurons

[1000] **FC8**: 1000 neurons (class scores)



Historical note: Trained on GTX 580 GPU with only 3 GB of memory. Network spread across 2 GPUs, half the neurons (feature maps) on each GPU.

Case Study: AlexNet

Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] **CONV1**: 96 11x11 filters at stride 4, pad 0

[27x27x96] **MAX POOL1**: 3x3 filters at stride 2

[27x27x96] **NORM1**: Normalization layer

[27x27x256] **CONV2**: 256 5x5 filters at stride 1, pad 2

[13x13x256] **MAX POOL2**: 3x3 filters at stride 2

[13x13x256] **NORM2**: Normalization layer

[13x13x384] **CONV3**: 384 3x3 filters at stride 1, pad 1

[13x13x384] **CONV4**: 384 3x3 filters at stride 1, pad 1

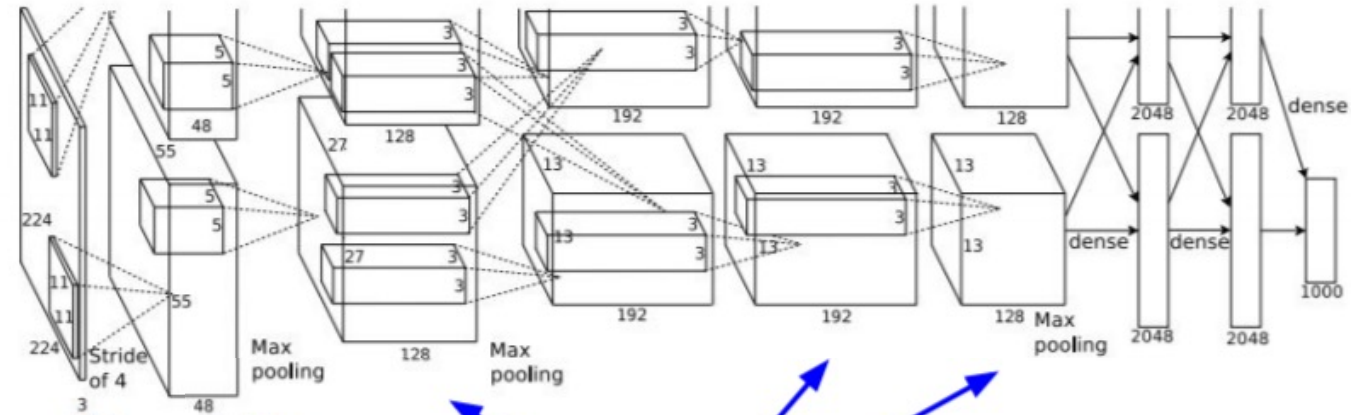
[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

[6x6x256] **MAX POOL3**: 3x3 filters at stride 2

[4096] **FC6**: 4096 neurons

[4096] **FC7**: 4096 neurons

[1000] **FC8**: 1000 neurons (class scores)



CONV1, CONV2, CONV4, CONV5:
Connections only with feature maps
on same GPU

Case Study: AlexNet

Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] **CONV1**: 96 11x11 filters at stride 4, pad 0

[27x27x96] **MAX POOL1**: 3x3 filters at stride 2

[27x27x96] **NORM1**: Normalization layer

[27x27x256] **CONV2**: 256 5x5 filters at stride 1, pad 2

[13x13x256] **MAX POOL2**: 3x3 filters at stride 2

[13x13x256] **NORM2**: Normalization layer

[13x13x384] **CONV3**: 384 3x3 filters at stride 1, pad 1

[13x13x384] **CONV4**: 384 3x3 filters at stride 1, pad 1

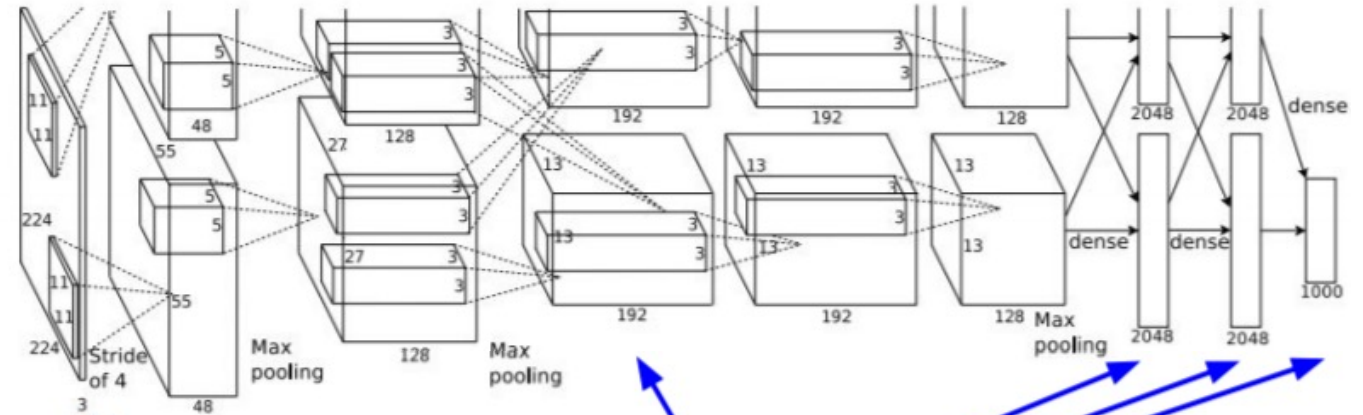
[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

[6x6x256] **MAX POOL3**: 3x3 filters at stride 2

[4096] **FC6**: 4096 neurons

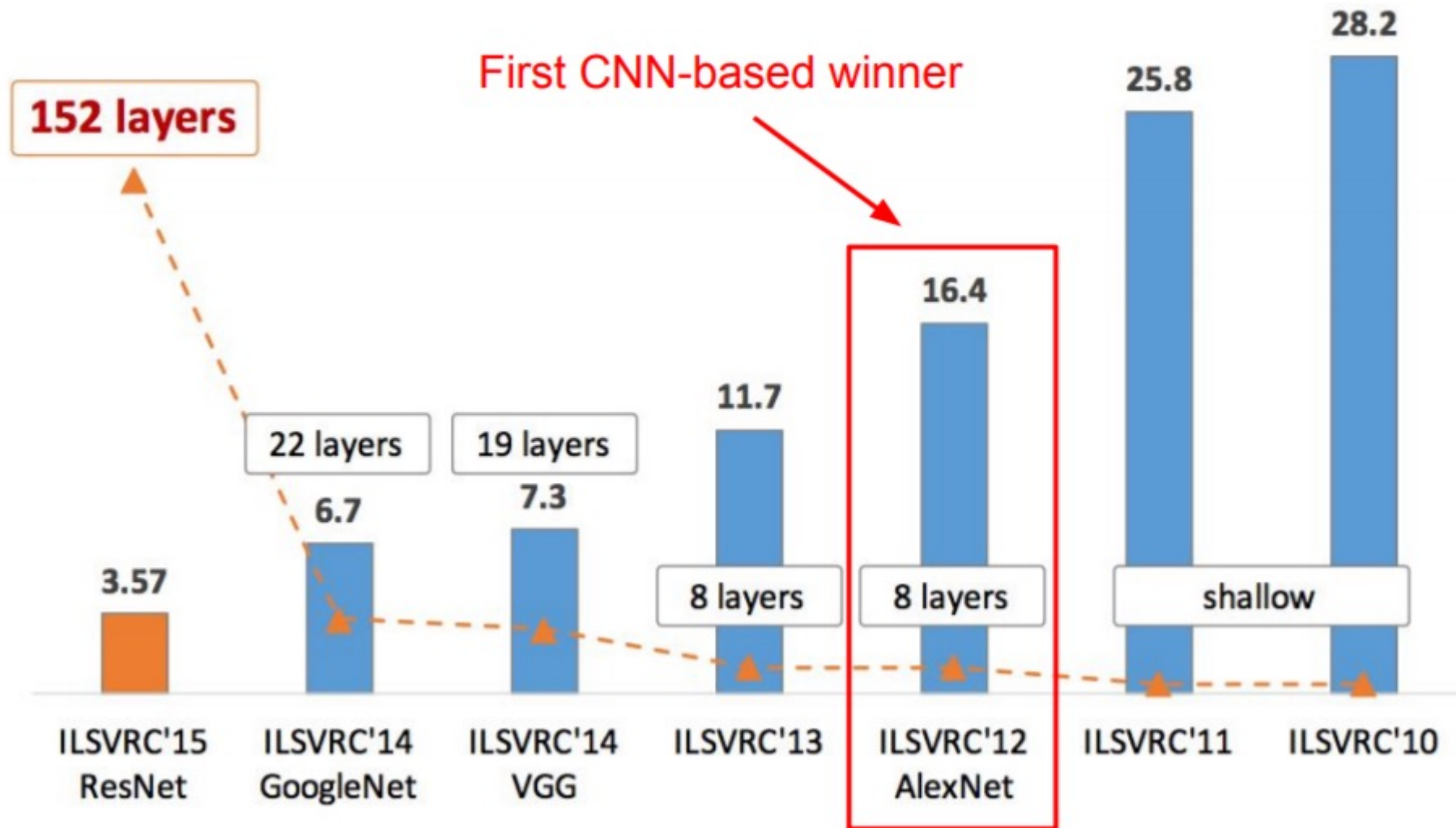
[4096] **FC7**: 4096 neurons

[1000] **FC8**: 1000 neurons (class scores)

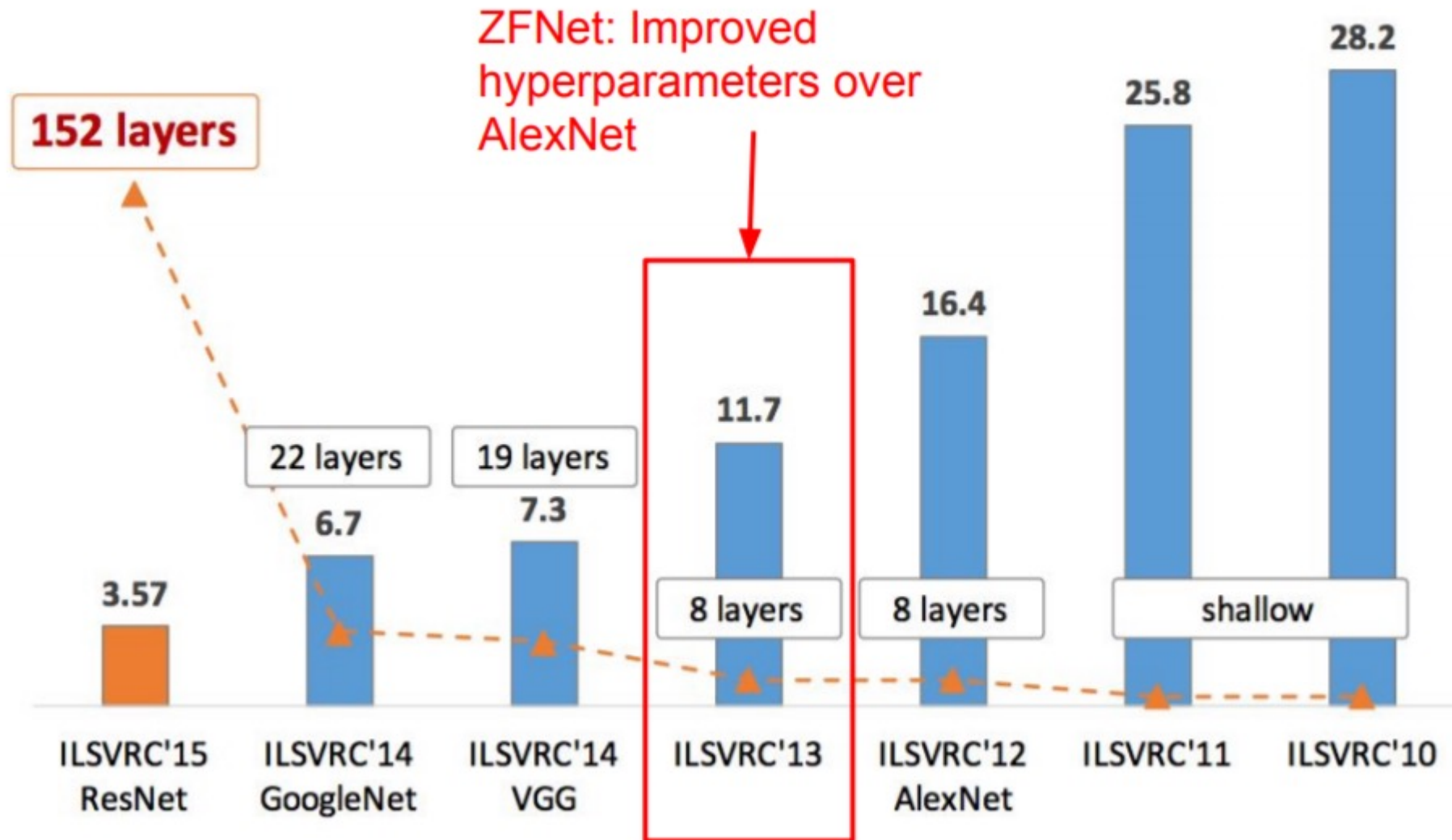


CONV3, FC6, FC7, FC8:
Connections with all feature maps in preceding layer, communication across GPUs

ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

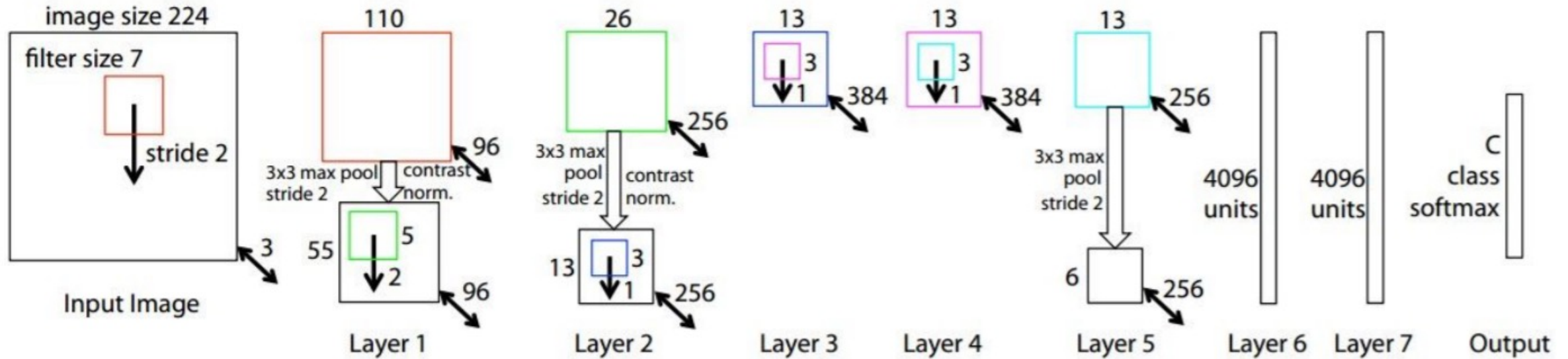


ImageNet Large Scale Visual Recognition Challenge (ILSVRC)



ZFNet

[Zeiler and Fergus, 2013]



TODO: remake figure

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

ImageNet top 5 error: 16.4% -> 11.7%

ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

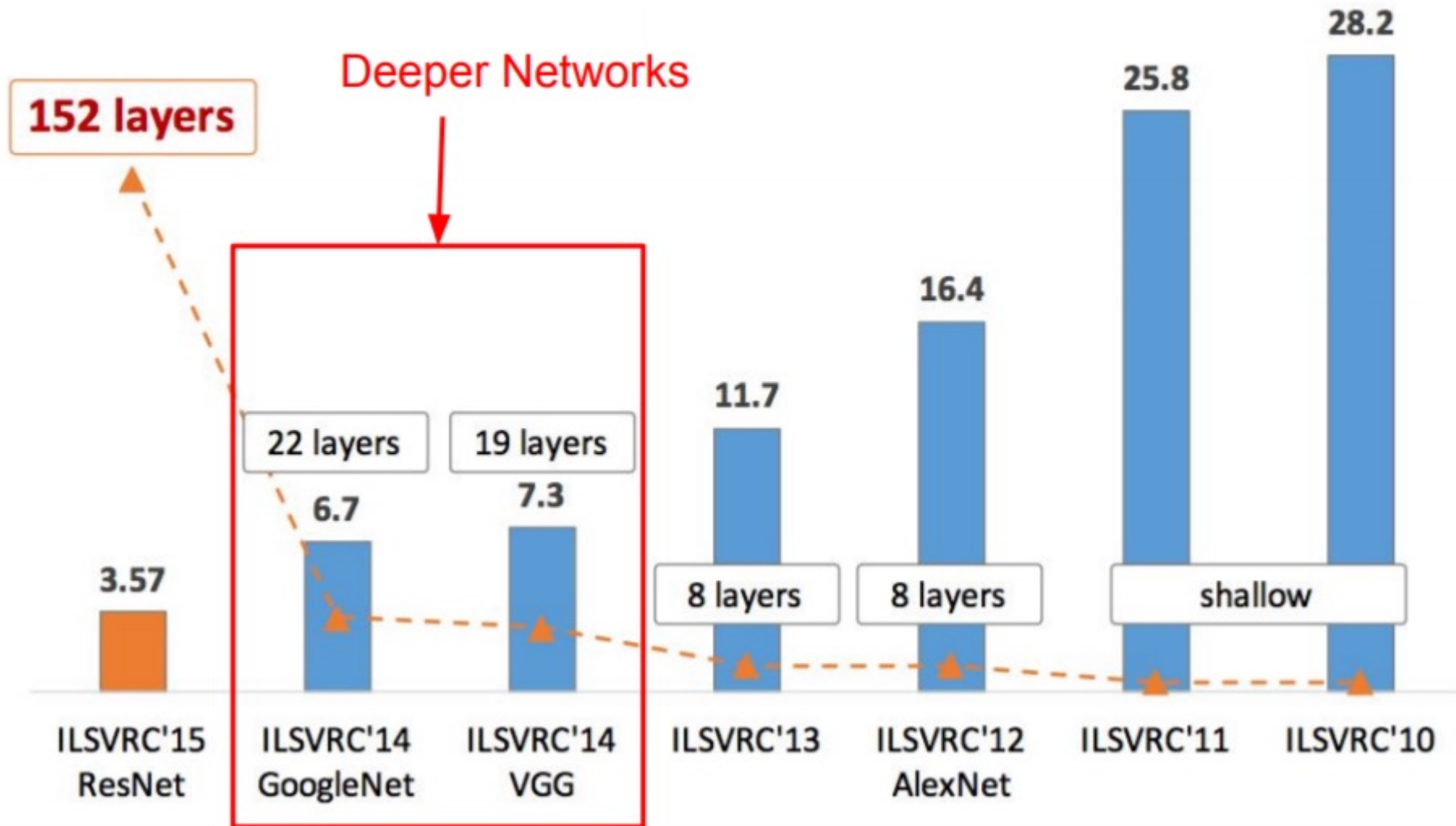
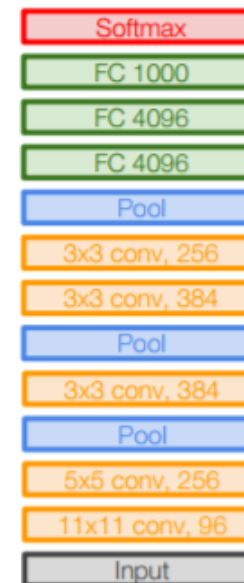


Figure copyright Kaiming He, 2016. Reproduced with permission.

Case Study: VGGNet

- Small filters
- Deeper networks
 - 8 layers (AlexNet) -> 16 - 19 layers (VGG16Net)
- Only 3x3 CONV
 - stride 1, pad 1
- 2x2 MAX POOL stride 2
- **11.7%** top 5 error in ILSVRC'13 (ZFNet)
 - > **7.3%** top 5 error in ILSVRC'14

[Simonyan and Zisserman, 2014]



AlexNet



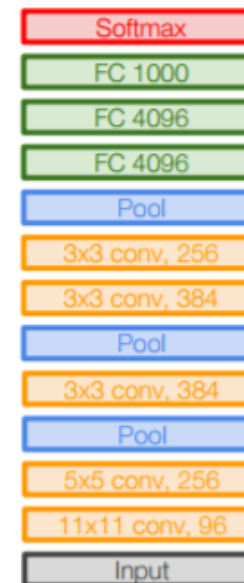
VGG16

VGG19

Case Study: VGGNet

- Why use smaller filters? (3x3 conv)
- Stack of three 3x3 conv (stride 1) layers has same effective receptive field as one 7x7 conv layer
- But deeper, more non-linearities
- And fewer parameters:
 - $3 * (3^2 C^2)$ vs. $7^2 C^2$ for C channels per layer

[Simonyan and Zisserman, 2014]



AlexNet



VGG16

VGG19

Case Study: VGGNet

INPUT: [224x224x3] memory: $224*224*3=150K$ params: 0 (not counting biases)

CONV3-64: [224x224x64] memory: $224*224*64=3.2M$ params: $(3*3*3)*64 = 1,728$

CONV3-64: [224x224x64] memory: $224*224*64=3.2M$ params: $(3*3*64)*64 = 36,864$

POOL2: [112x112x64] memory: $112*112*64=800K$ params: 0

CONV3-128: [112x112x128] memory: $112*112*128=1.6M$ params: $(3*3*64)*128 = 73,728$

CONV3-128: [112x112x128] memory: $112*112*128=1.6M$ params: $(3*3*128)*128 = 147,456$

POOL2: [56x56x128] memory: $56*56*128=400K$ params: 0

CONV3-256: [56x56x256] memory: $56*56*256=800K$ params: $(3*3*128)*256 = 294,912$

CONV3-256: [56x56x256] memory: $56*56*256=800K$ params: $(3*3*256)*256 = 589,824$

CONV3-256: [56x56x256] memory: $56*56*256=800K$ params: $(3*3*256)*256 = 589,824$

POOL2: [28x28x256] memory: $28*28*256=200K$ params: 0

CONV3-512: [28x28x512] memory: $28*28*512=400K$ params: $(3*3*256)*512 = 1,179,648$

CONV3-512: [28x28x512] memory: $28*28*512=400K$ params: $(3*3*512)*512 = 2,359,296$

CONV3-512: [28x28x512] memory: $28*28*512=400K$ params: $(3*3*512)*512 = 2,359,296$

POOL2: [14x14x512] memory: $14*14*512=100K$ params: 0

CONV3-512: [14x14x512] memory: $14*14*512=100K$ params: $(3*3*512)*512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14*14*512=100K$ params: $(3*3*512)*512 = 2,359,296$

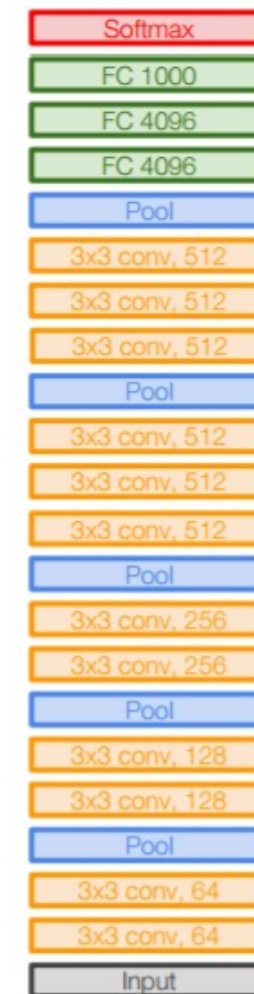
CONV3-512: [14x14x512] memory: $14*14*512=100K$ params: $(3*3*512)*512 = 2,359,296$

POOL2: [7x7x512] memory: $7*7*512=25K$ params: 0

FC: [1x1x4096] memory: 4096 params: $7*7*512*4096 = 102,760,448$

FC: [1x1x4096] memory: 4096 params: $4096*4096 = 16,777,216$

FC: [1x1x1000] memory: 1000 params: $4096*1000 = 4,096,000$



VGG16

Case Study: VGGNet

INPUT: [224x224x3] memory: $224*224*3=150\text{K}$ params: 0 (not counting biases)

CONV3-64: [224x224x64] memory: $224*224*64=3.2\text{M}$ params: $(3*3*3)*64 = 1,728$

CONV3-64: [224x224x64] memory: $224*224*64=3.2\text{M}$ params: $(3*3*64)*64 = 36,864$

POOL2: [112x112x64] memory: $112*112*64=800\text{K}$ params: 0

CONV3-128: [112x112x128] memory: $112*112*128=1.6\text{M}$ params: $(3*3*64)*128 = 73,728$

CONV3-128: [112x112x128] memory: $112*112*128=1.6\text{M}$ params: $(3*3*128)*128 = 147,456$

POOL2: [56x56x128] memory: $56*56*128=400\text{K}$ params: 0

CONV3-256: [56x56x256] memory: $56*56*256=800\text{K}$ params: $(3*3*128)*256 = 294,912$

CONV3-256: [56x56x256] memory: $56*56*256=800\text{K}$ params: $(3*3*256)*256 = 589,824$

CONV3-256: [56x56x256] memory: $56*56*256=800\text{K}$ params: $(3*3*256)*256 = 589,824$

POOL2: [28x28x256] memory: $28*28*256=200\text{K}$ params: 0

CONV3-512: [28x28x512] memory: $28*28*512=400\text{K}$ params: $(3*3*256)*512 = 1,179,648$

CONV3-512: [28x28x512] memory: $28*28*512=400\text{K}$ params: $(3*3*512)*512 = 2,359,296$

CONV3-512: [28x28x512] memory: $28*28*512=400\text{K}$ params: $(3*3*512)*512 = 2,359,296$

POOL2: [14x14x512] memory: $14*14*512=100\text{K}$ params: 0

CONV3-512: [14x14x512] memory: $14*14*512=100\text{K}$ params: $(3*3*512)*512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14*14*512=100\text{K}$ params: $(3*3*512)*512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14*14*512=100\text{K}$ params: $(3*3*512)*512 = 2,359,296$

POOL2: [7x7x512] memory: $7*7*512=25\text{K}$ params: 0

FC: [1x1x4096] memory: 4096 params: $7*7*512*4096 = 102,760,448$

FC: [1x1x4096] memory: 4096 params: $4096*4096 = 16,777,216$

FC: [1x1x1000] memory: 1000 params: $4096*1000 = 4,096,000$

Note:

Most memory is in early CONV

Most params are in late FC

TOTAL memory: $24\text{M} * 4 \text{ bytes} \approx 96\text{MB} / \text{image}$ (only forward! ~ 2 for bwd)

TOTAL params: 138M parameters

Case Study: VGGNet

INPUT: [224x224x3] memory: $224*224*3=150K$ params: 0 (not counting biases)

CONV3-64: [224x224x64] memory: $224*224*64=3.2M$ params: $(3*3*3)*64 = 1,728$

CONV3-64: [224x224x64] memory: $224*224*64=3.2M$ params: $(3*3*64)*64 = 36,864$

POOL2: [112x112x64] memory: $112*112*64=800K$ params: 0

CONV3-128: [112x112x128] memory: $112*112*128=1.6M$ params: $(3*3*64)*128 = 73,728$

CONV3-128: [112x112x128] memory: $112*112*128=1.6M$ params: $(3*3*128)*128 = 147,456$

POOL2: [56x56x128] memory: $56*56*128=400K$ params: 0

CONV3-256: [56x56x256] memory: $56*56*256=800K$ params: $(3*3*128)*256 = 294,912$

CONV3-256: [56x56x256] memory: $56*56*256=800K$ params: $(3*3*256)*256 = 589,824$

CONV3-256: [56x56x256] memory: $56*56*256=800K$ params: $(3*3*256)*256 = 589,824$

POOL2: [28x28x256] memory: $28*28*256=200K$ params: 0

CONV3-512: [28x28x512] memory: $28*28*512=400K$ params: $(3*3*256)*512 = 1,179,648$

CONV3-512: [28x28x512] memory: $28*28*512=400K$ params: $(3*3*512)*512 = 2,359,296$

CONV3-512: [28x28x512] memory: $28*28*512=400K$ params: $(3*3*512)*512 = 2,359,296$

POOL2: [14x14x512] memory: $14*14*512=100K$ params: 0

CONV3-512: [14x14x512] memory: $14*14*512=100K$ params: $(3*3*512)*512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14*14*512=100K$ params: $(3*3*512)*512 = 2,359,296$

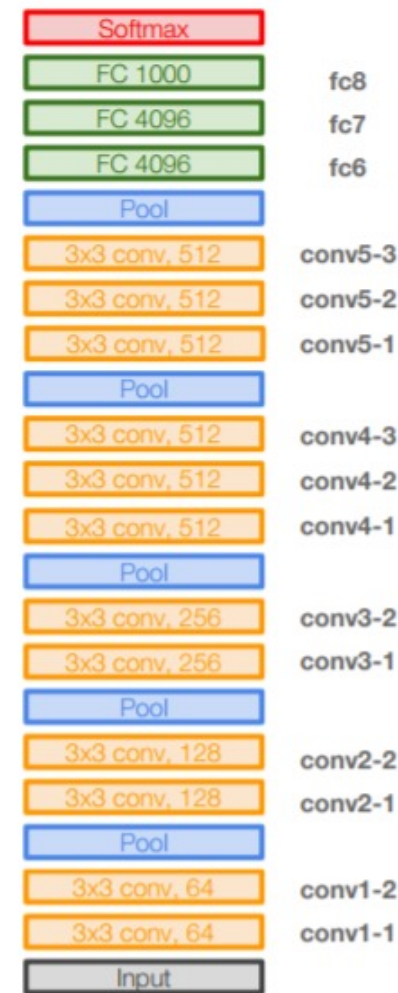
CONV3-512: [14x14x512] memory: $14*14*512=100K$ params: $(3*3*512)*512 = 2,359,296$

POOL2: [7x7x512] memory: $7*7*512=25K$ params: 0

FC: [1x1x4096] memory: 4096 params: $7*7*512*4096 = 102,760,448$

FC: [1x1x4096] memory: 4096 params: $4096*4096 = 16,777,216$

FC: [1x1x1000] memory: 1000 params: $4096*1000 = 4,096,000$



VGG16

Common names

Case Study: VGGNet

INPUT: [224x224x3] memory: $224*224*3=150K$ params: 0 (not counting biases)

CONV3-64: [224x224x64] memory: $224*224*64=3.2M$ params: $(3*3*3)*64 = 1,728$

CONV3-64: [224x224x64] memory: $224*224*64=3.2M$ params: $(3*3*64)*64 = 36,864$

POOL2: [112x112x64] memory: $112*112*64=800K$ params: 0

CONV3-128: [112x112x128] memory: $112*112*128=1.6M$ params: $(3*3*64)*128 = 73,728$

CONV3-128: [112x112x128] memory: $112*112*128=1.6M$ params: $(3*3*128)*128 = 147,456$

POOL2: [56x56x128] memory: $56*56*128=400K$ params: 0

CONV3-256: [56x56x256] memory: $56*56*256=800K$ params: $(3*3*128)*256 = 294,912$

CONV3-256: [56x56x256] memory: $56*56*256=800K$ params: $(3*3*256)*256 = 589,824$

CONV3-256: [56x56x256] memory: $56*56*256=800K$ params: $(3*3*256)*256 = 589,824$

POOL2: [28x28x256] memory: $28*28*256=200K$ params: 0

CONV3-512: [28x28x512] memory: $28*28*512=400K$ params: $(3*3*256)*512 = 1,179,648$

CONV3-512: [28x28x512] memory: $28*28*512=400K$ params: $(3*3*512)*512 = 2,359,296$

CONV3-512: [28x28x512] memory: $28*28*512=400K$ params: $(3*3*512)*512 = 2,359,296$

POOL2: [14x14x512] memory: $14*14*512=100K$ params: 0

CONV3-512: [14x14x512] memory: $14*14*512=100K$ params: $(3*3*512)*512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14*14*512=100K$ params: $(3*3*512)*512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14*14*512=100K$ params: $(3*3*512)*512 = 2,359,296$

POOL2: [7x7x512] memory: $7*7*512=25K$ params: 0

FC: [1x1x4096] memory: 4096 params: $7*7*512*4096 = 102,760,448$

FC: [1x1x4096] memory: 4096 params: $4096*4096 = 16,777,216$

FC: [1x1x1000] memory: 1000 params: $4096*1000 = 4,096,000$

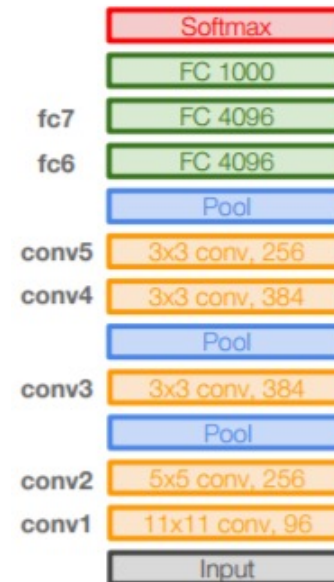


VGG16

Common names

Case Study: VGGNet

- Details:
 - ILSVRC'14 2nd in classification, 1st in localization
 - Similar training procedure as Krizhevsky 2012
 - No Local Response Normalisation (LRN)
 - Use VGG16 or VGG19 (VGG19 only slightly better, more memory)
 - Use ensembles for best results
 - FC7 features generalize well to other tasks



AlexNet



VGG16

VGG19

ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

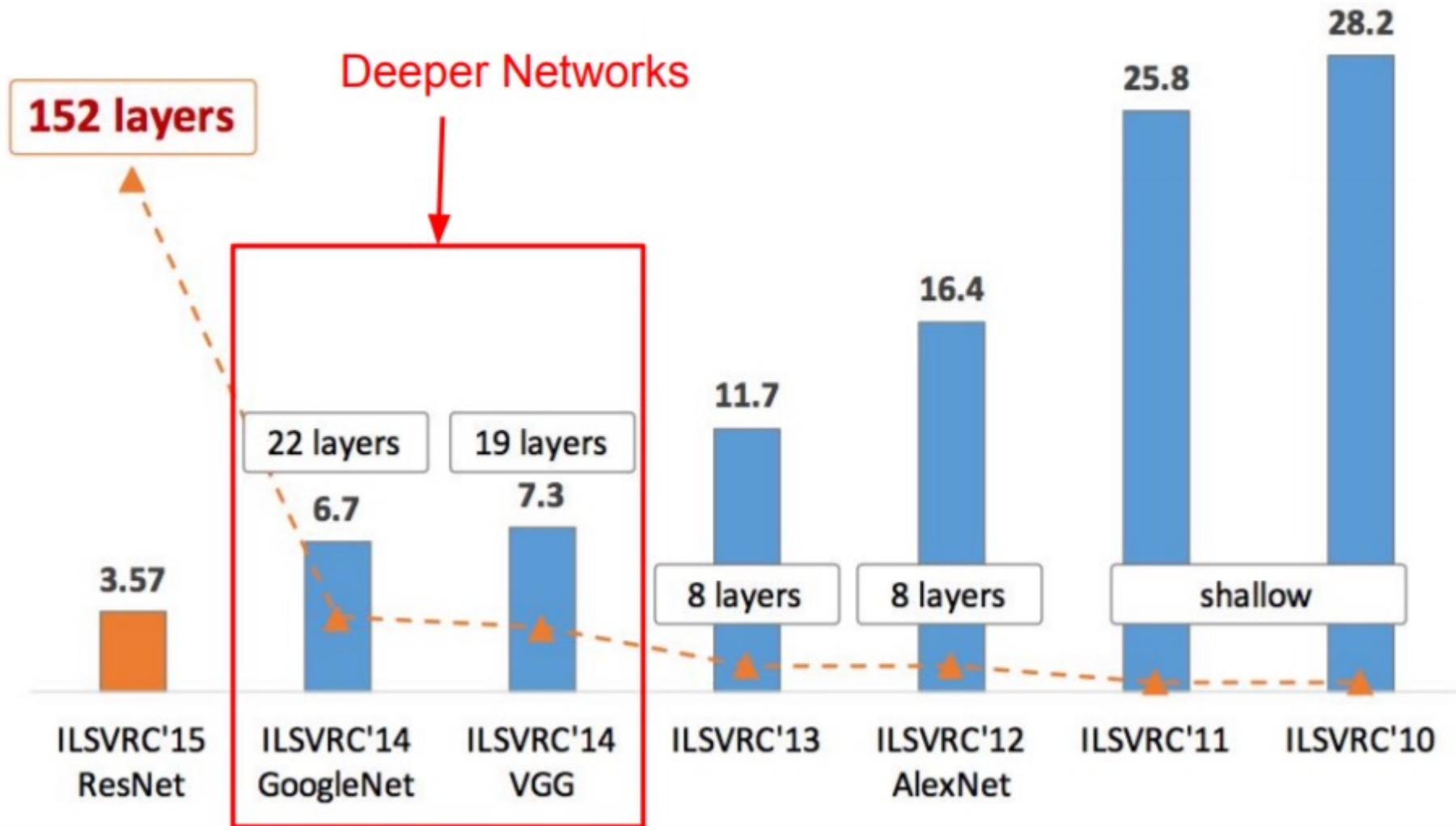
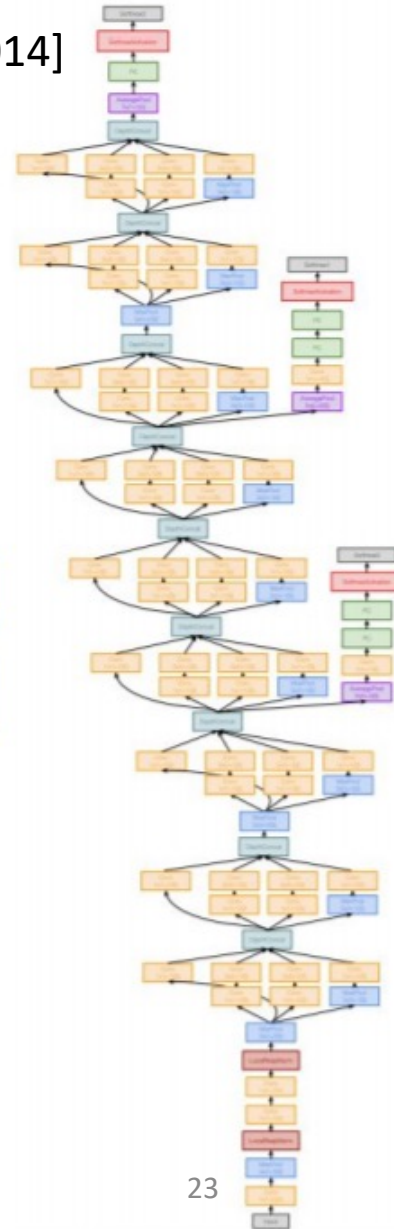
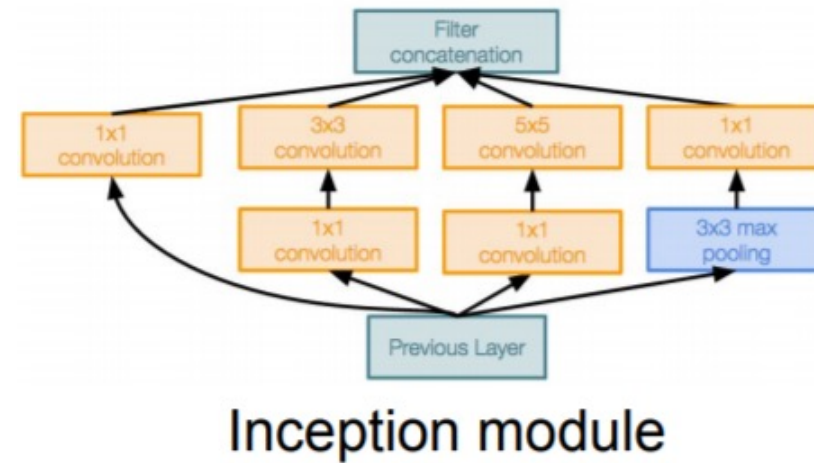


Figure copyright Kaiming He, 2016. Reproduced with permission.

Case Study: GoogLeNet

- 22 layers
- Deeper networks, with computational efficiency
 - Efficient “Inception” module
 - No FC layers
 - Only 5 million parameters!
 - 12x less than AlexNet
- ILSVRC’14 classification winner (6.7% top 5 error)

[Szegedy et al., 2014]

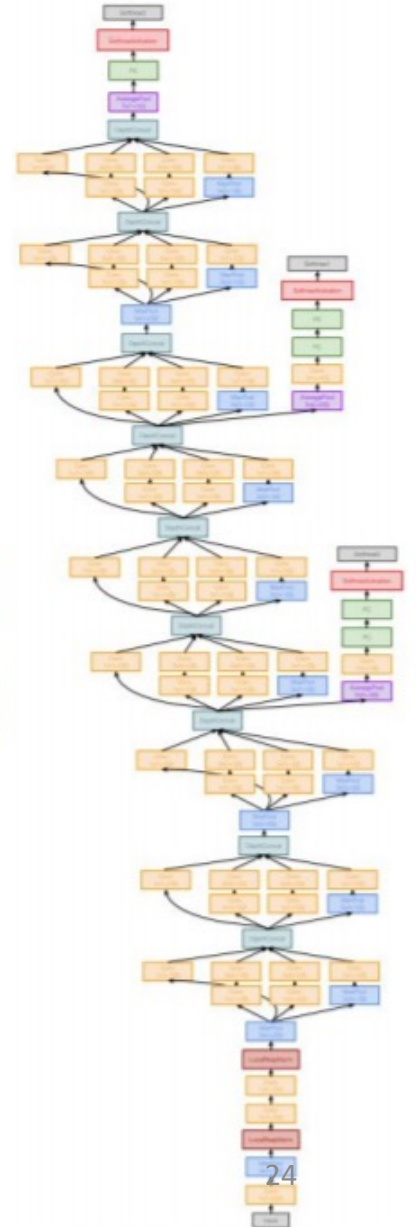
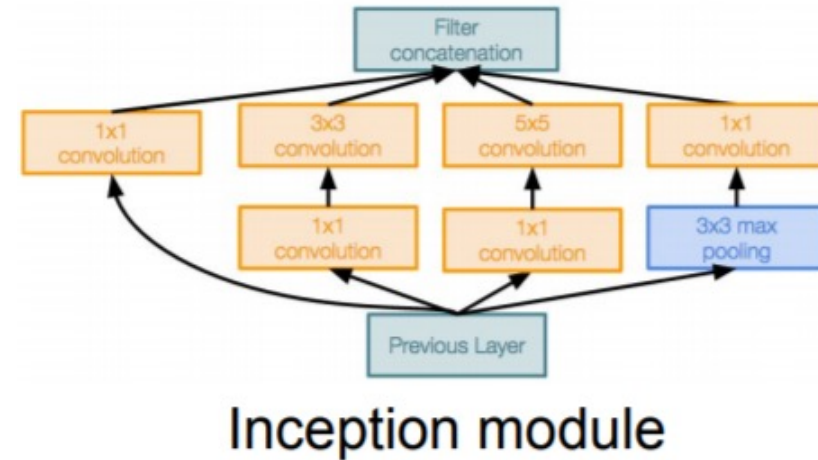


Case Study: GoogLeNet

[Szegedy et al., 2014]

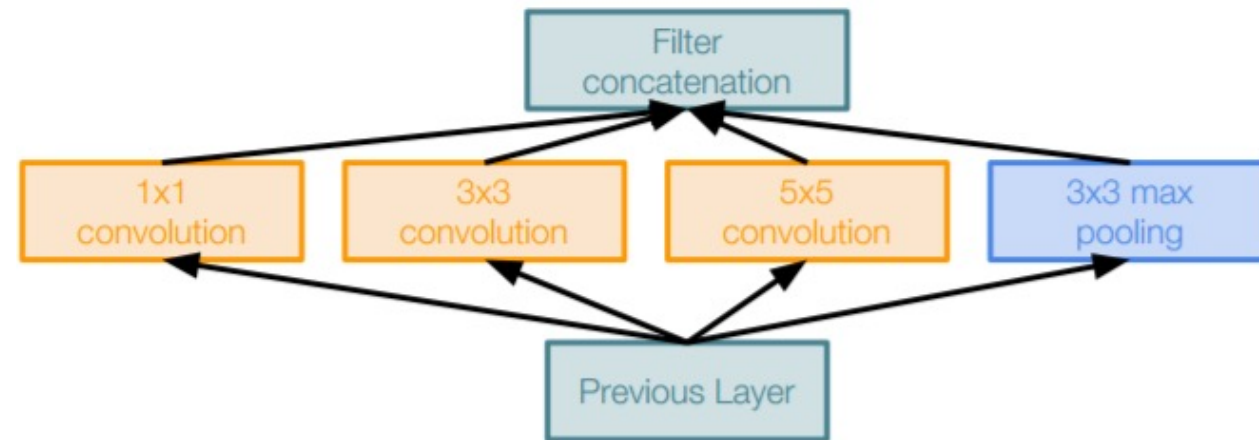
Inception module: a good local network topology
(network within a network)

GoogLeNet stack these modules on top of each other



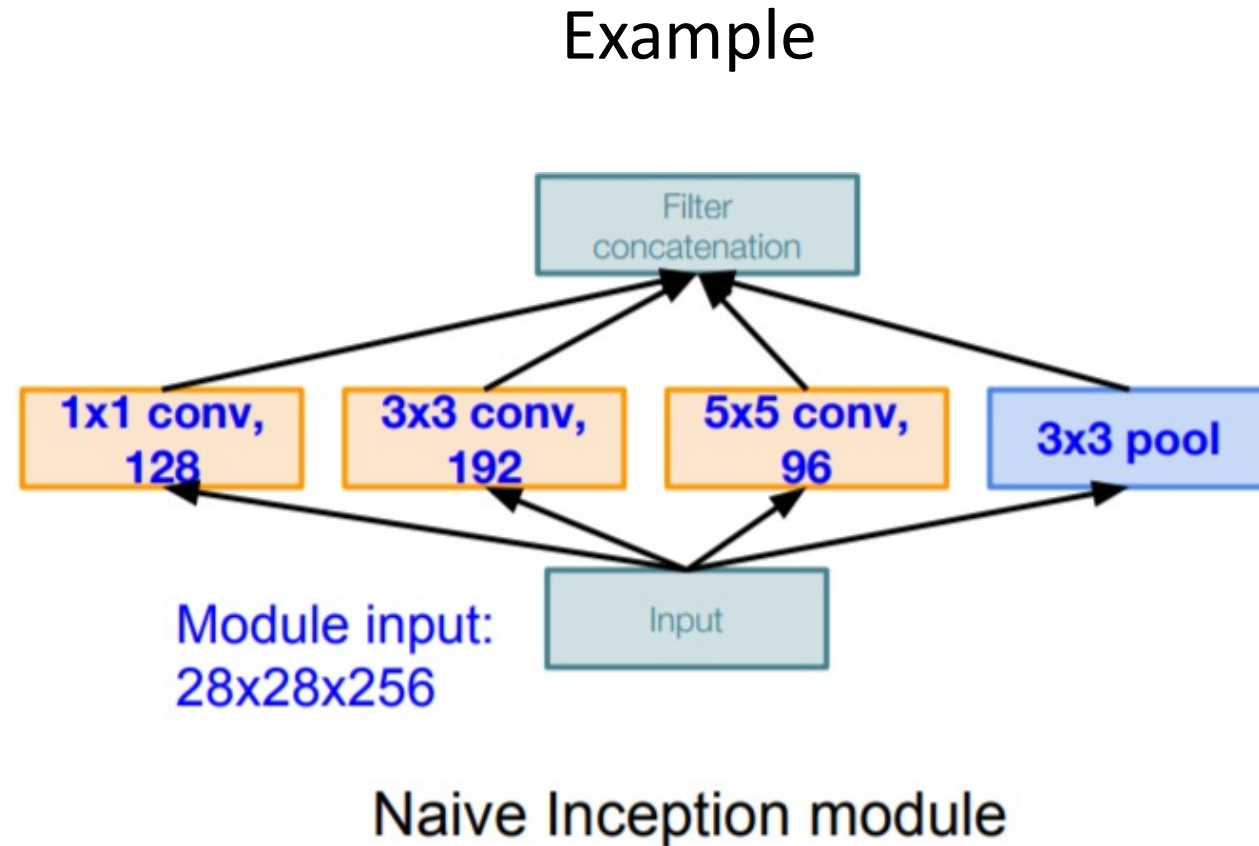
Case Study: GoogLeNet

- 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]



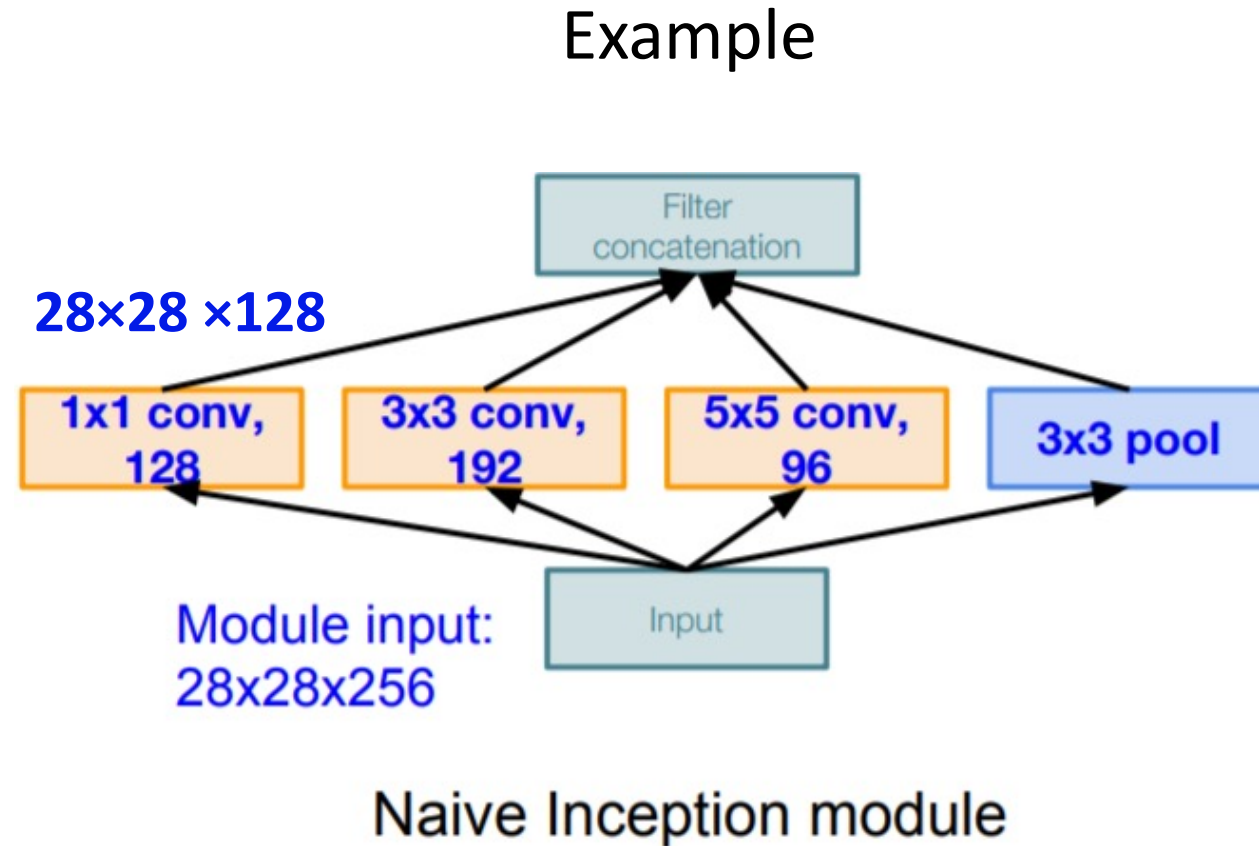
Case Study: GoogLeNet

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



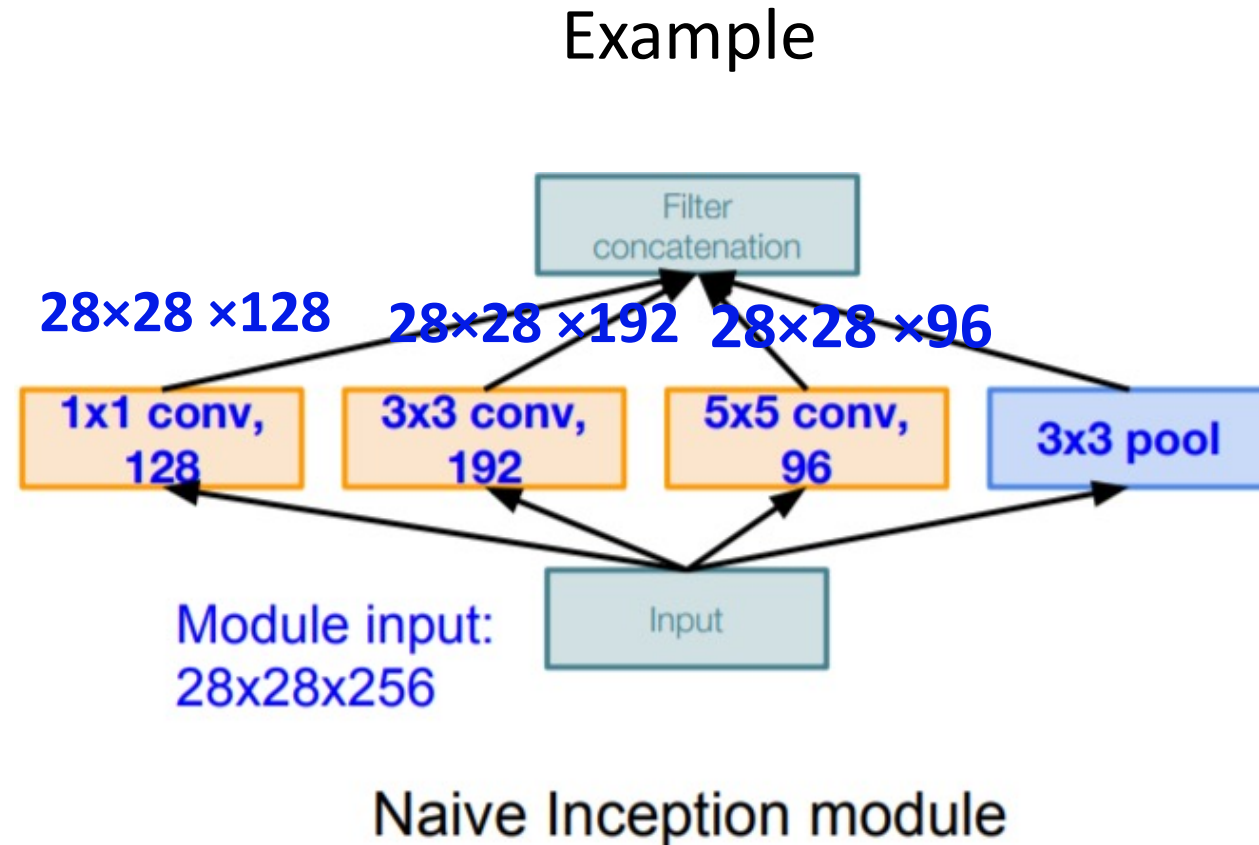
Case Study: GoogLeNet

- Q: What is the problem with this?
[Hint: Computational complexity]
- Example:
 - Q1: What is the output size of the 1x1 conv, with 128 filters?



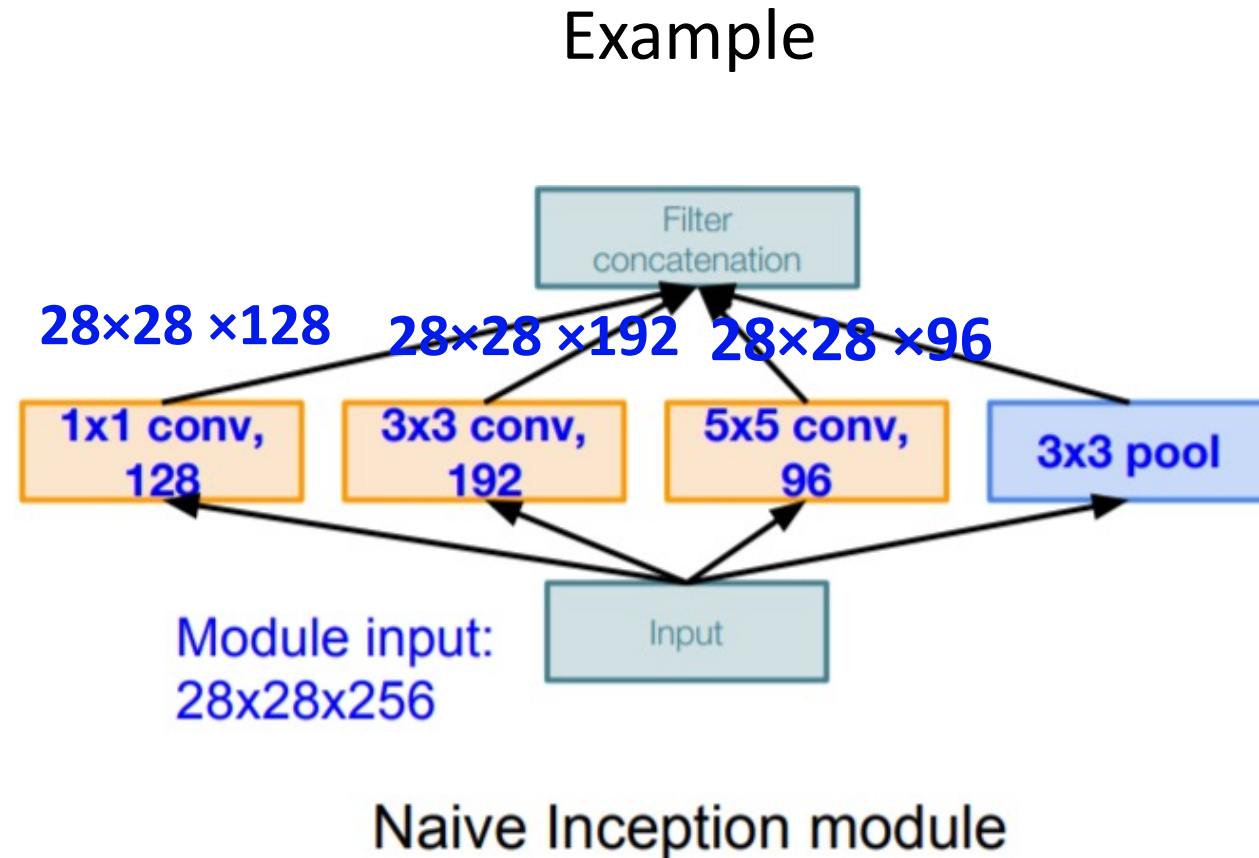
Case Study: GoogLeNet

- Q: What is the problem with this?
[Hint: Computational complexity]
- Example:
 - Q1: What is the output size of the 1x1 conv, with 128 filters?
 - Q2: What are the output sizes of all different filter operations?



Case Study: GoogLeNet

- Q: What is the problem with this?
[Hint: Computational complexity]
- Example:
 - Q1: What is the output size of the 1x1 conv, with 128 filters?
 - Q2: What are the output sizes of all different filter operations?
 - Q3: What is output size after filter concatenation?



Case Study: GoogLeNet

- Conv Ops:

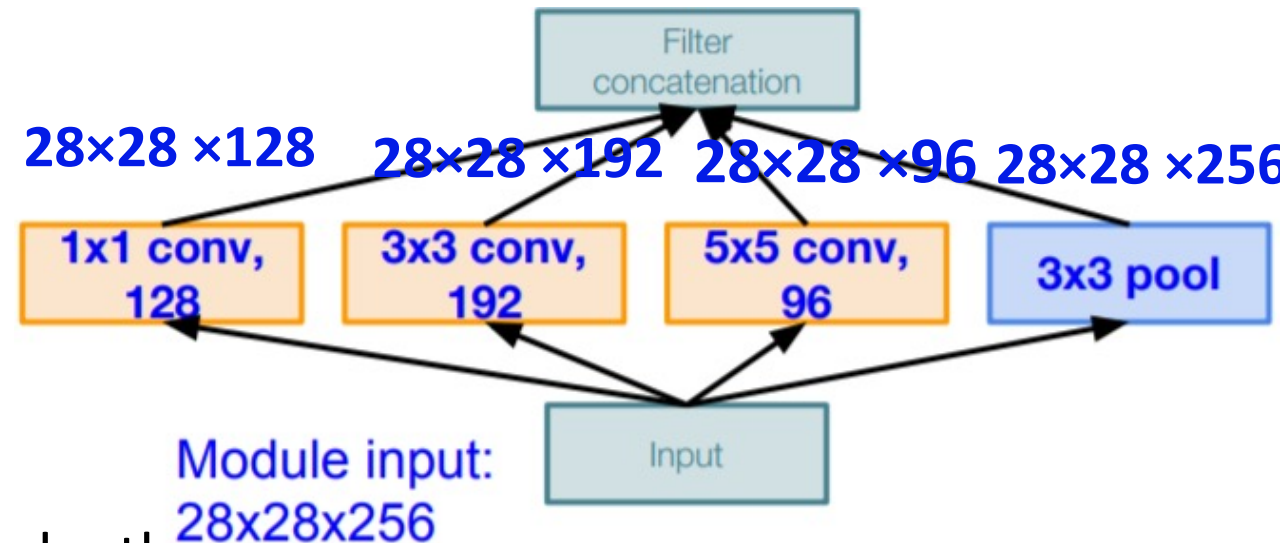
- [1x1 conv, 128] $28 \times 28 \times 128 \times 1 \times 1 \times 256$
- [3x3 conv, 192] $28 \times 28 \times 192 \times 3 \times 3 \times 256$
- [5x5 conv, 96] $28 \times 28 \times 96 \times 5 \times 5 \times 256$

- Total: **854M** ops

- Very expensive computations

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

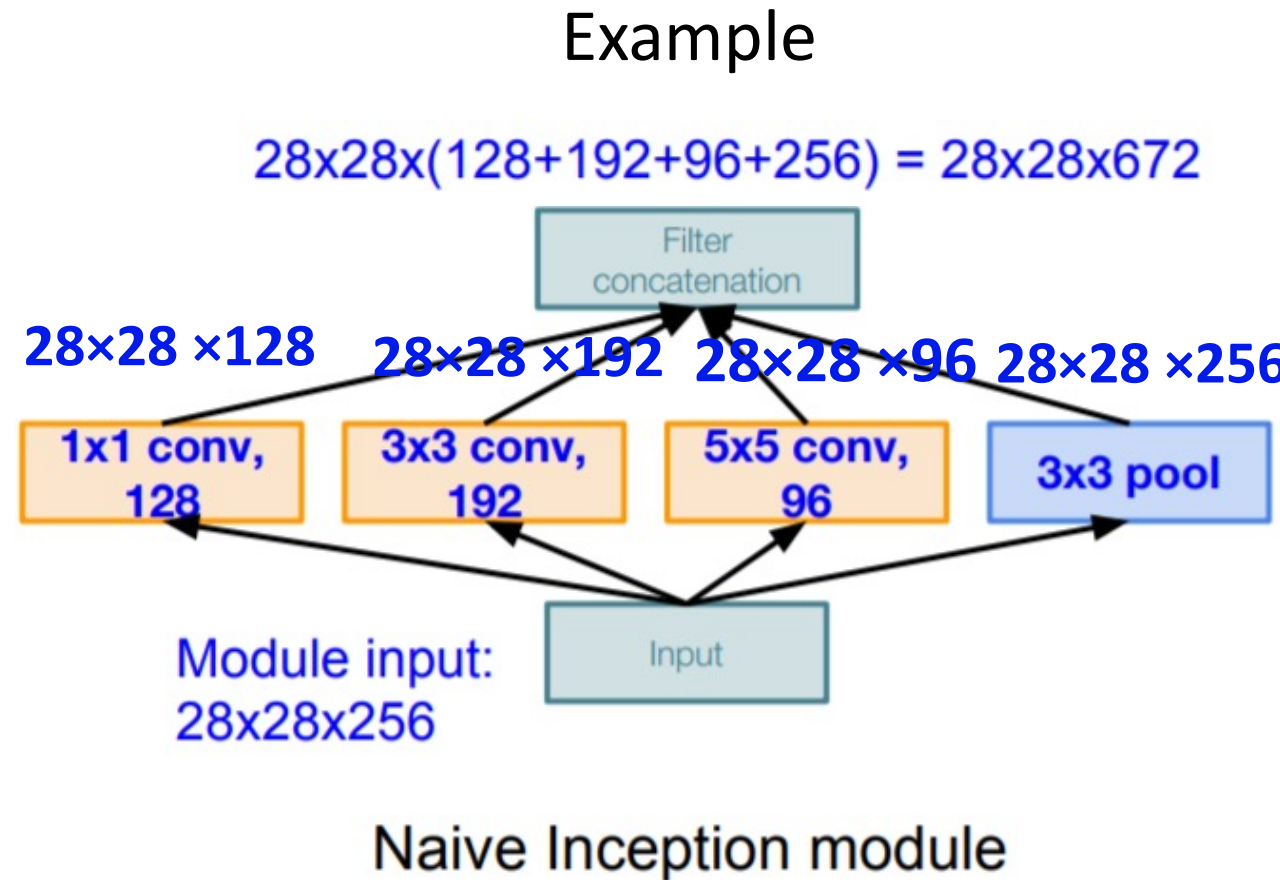
Example



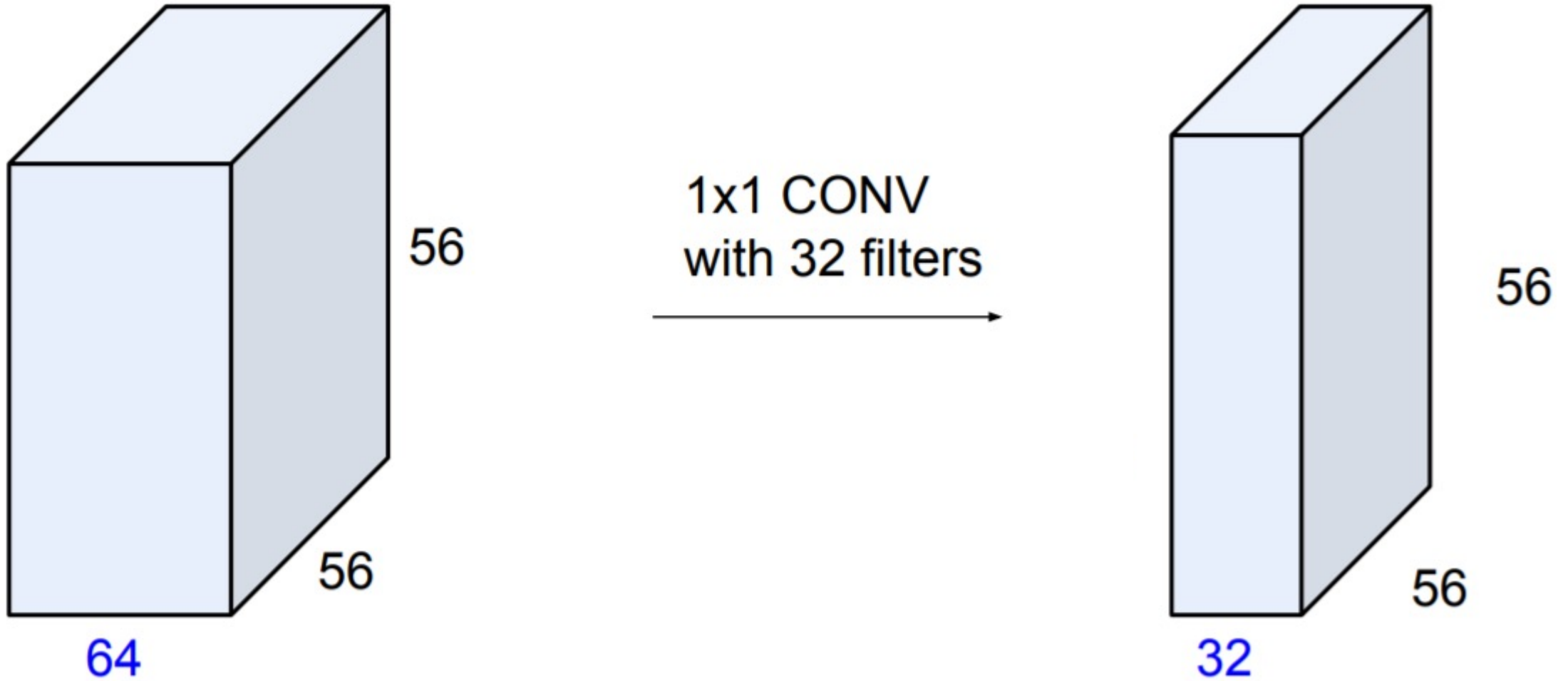
Naive Inception module

Case Study: GoogLeNet

- Solution: “bottleneck” layers that use 1x1 convolutions to reduce feature depth

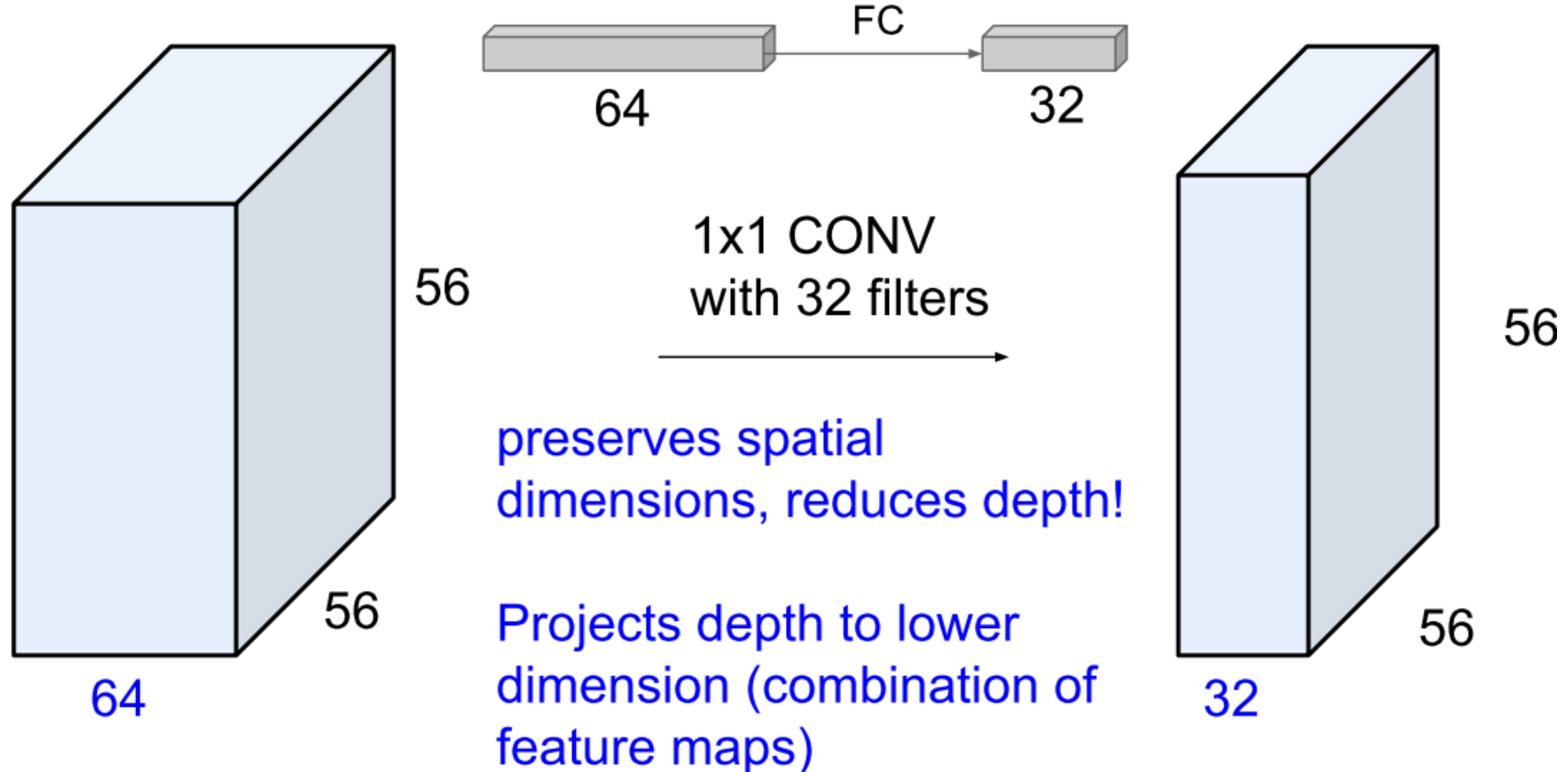


1x1 convolutions

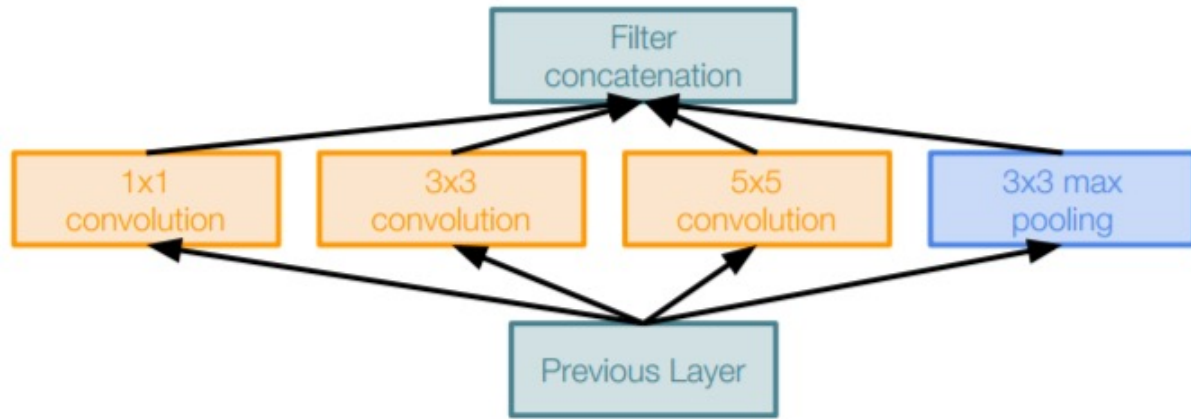


1x1 convolutions

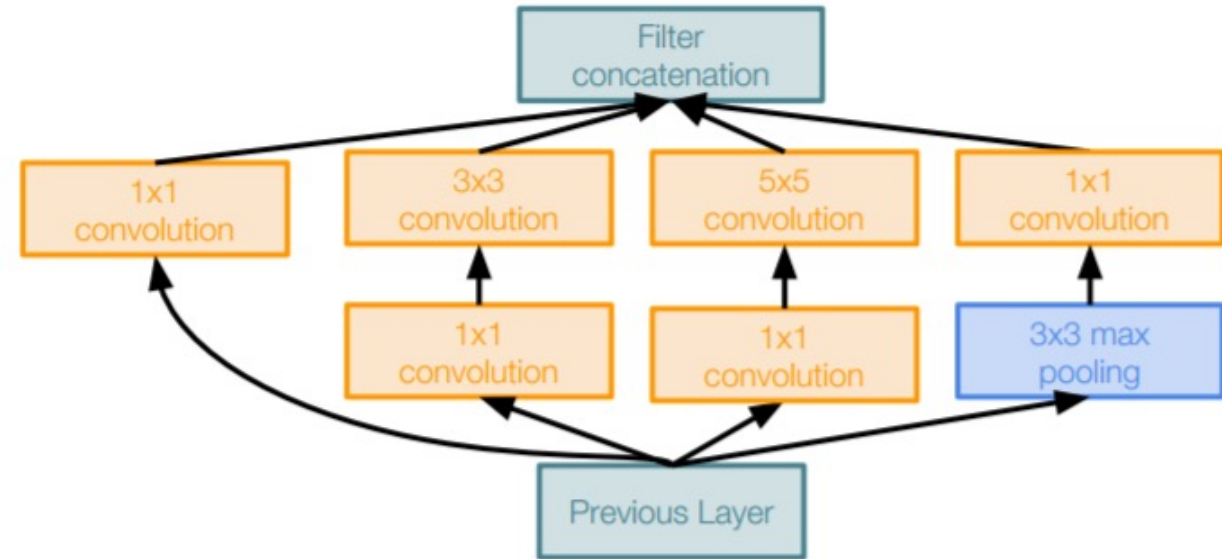
Alternatively, interpret it as applying the same FC layer on each input pixel



Case Study: GoogLeNet

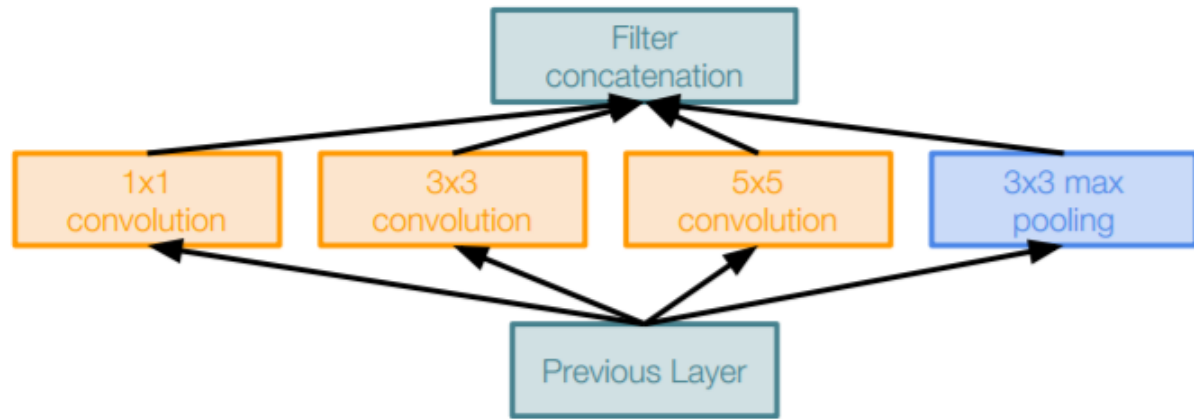


Naive Inception module



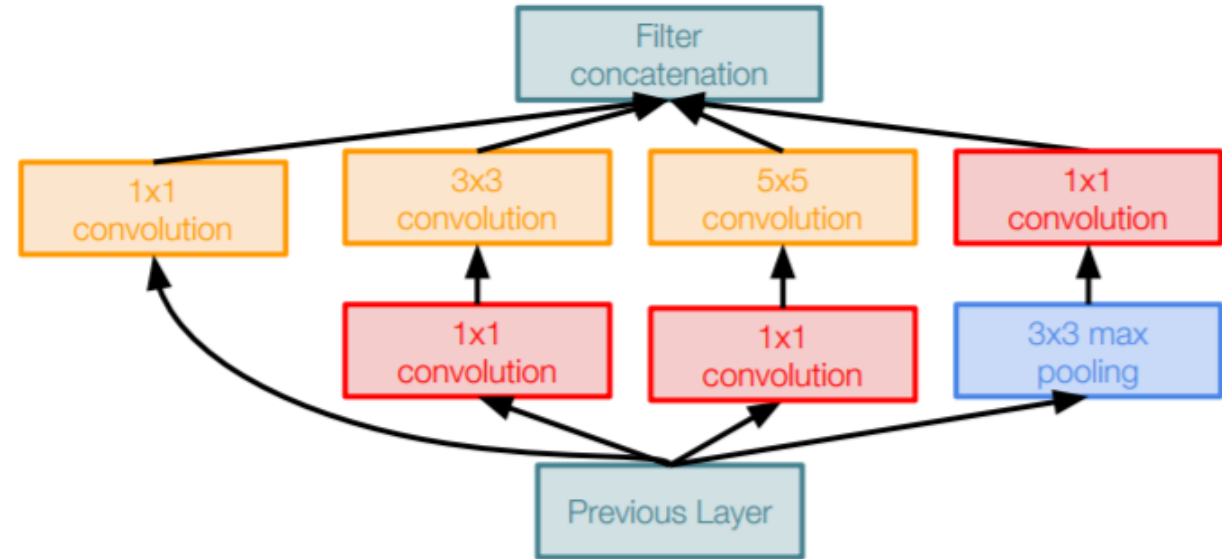
Inception module with dimension reduction

Case Study: GoogLeNet



Naive Inception module

1x1 conv “bottleneck”
layers



Inception module with dimension reduction

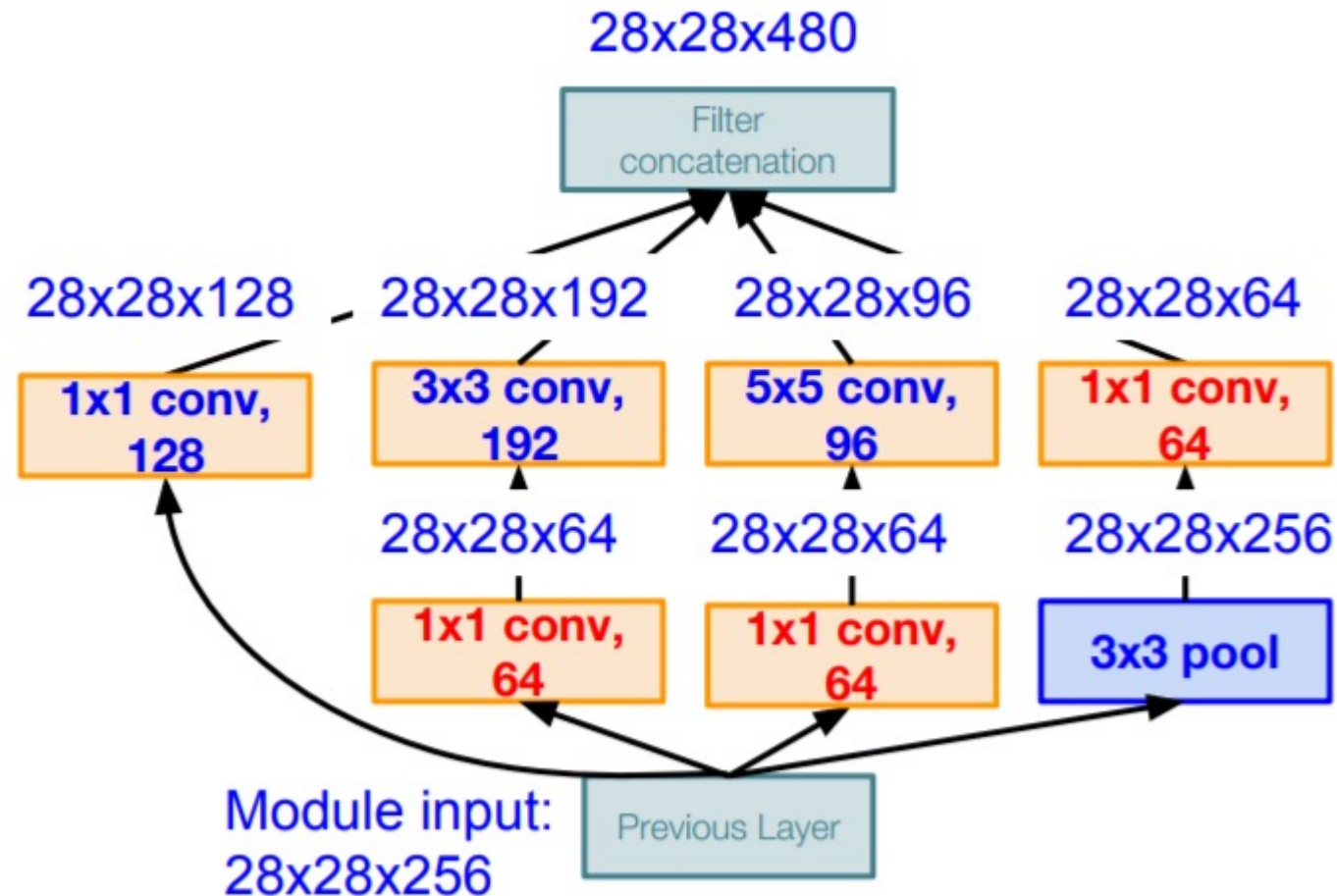
Case Study: GoogLeNet

- Conv Ops:

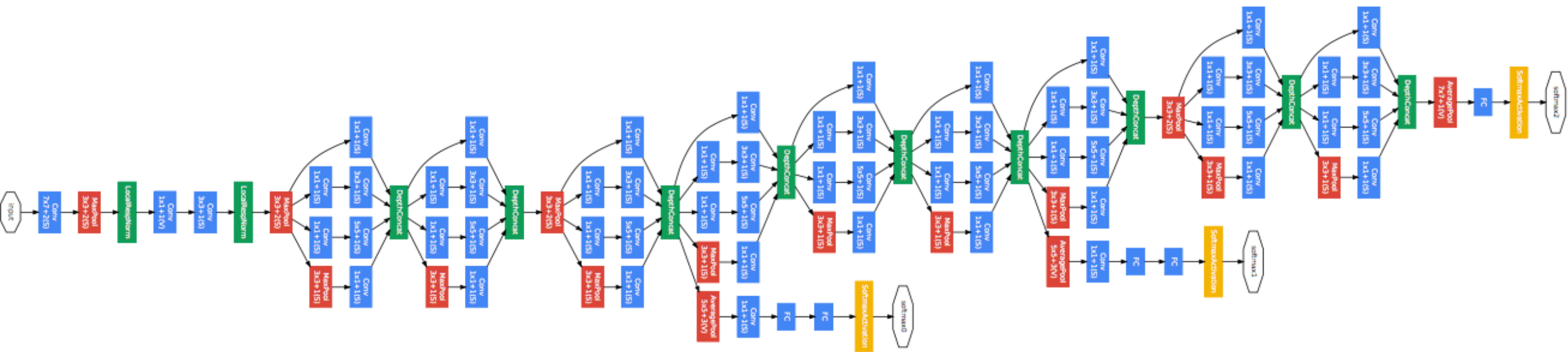
- [1x1 conv, 64] $28 \times 28 \times 64 \times 1 \times 1 \times 256$
- [1x1 conv, 64] $28 \times 28 \times 64 \times 1 \times 1 \times 256$
- [1x1 conv, 128] $28 \times 28 \times 128 \times 1 \times 1 \times 256$
- [3x3 conv, 192] $28 \times 28 \times 192 \times 3 \times 3 \times 64$
- [5x5 conv, 96] $28 \times 28 \times 96 \times 5 \times 5 \times 64$
- [1x1 conv, 64] $28 \times 28 \times 64 \times 1 \times 1 \times 256$

- Total: 358M ops

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



GoogLeNet

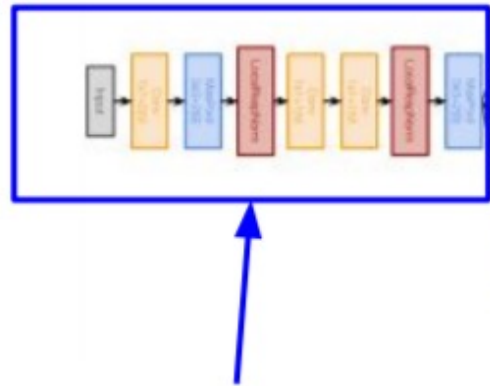


Case Study: GoogLeNet

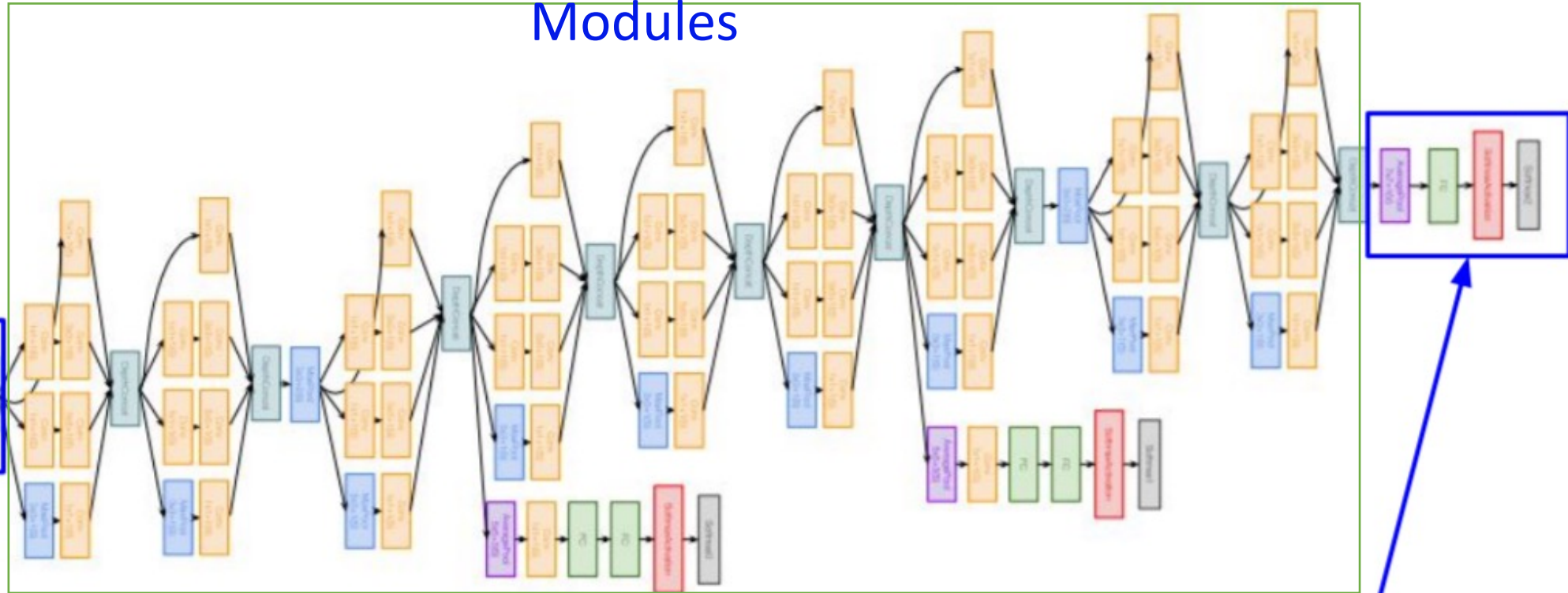
Stacked Inception Modules

[Szegedy et al., 2014]

Full GoogLeNet architecture



Stem Network:
Conv-Pool-
2x Conv-Pool



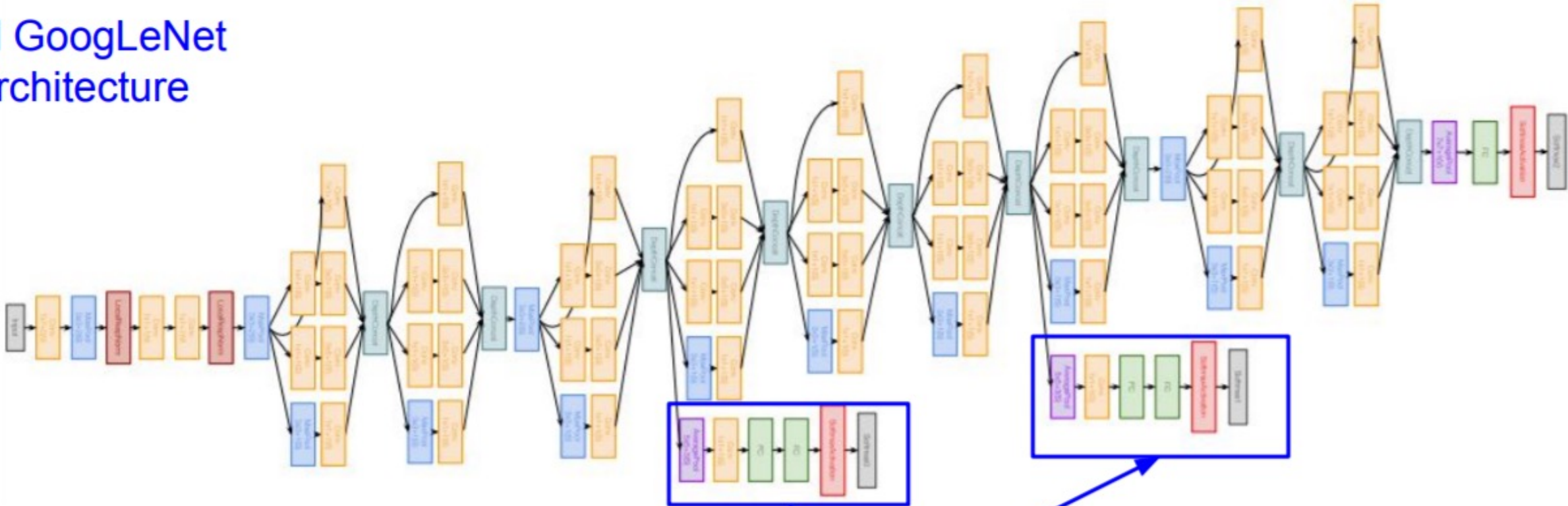
Note: after the last convolutional layer, a global average pooling layer is used that spatially averages across each feature map, before final FC layer. No longer multiple expensive FC layers!

Classifier output
(removed expensive FC layers!)

Case Study: GoogLeNet

[Szegedy et al., 2014]

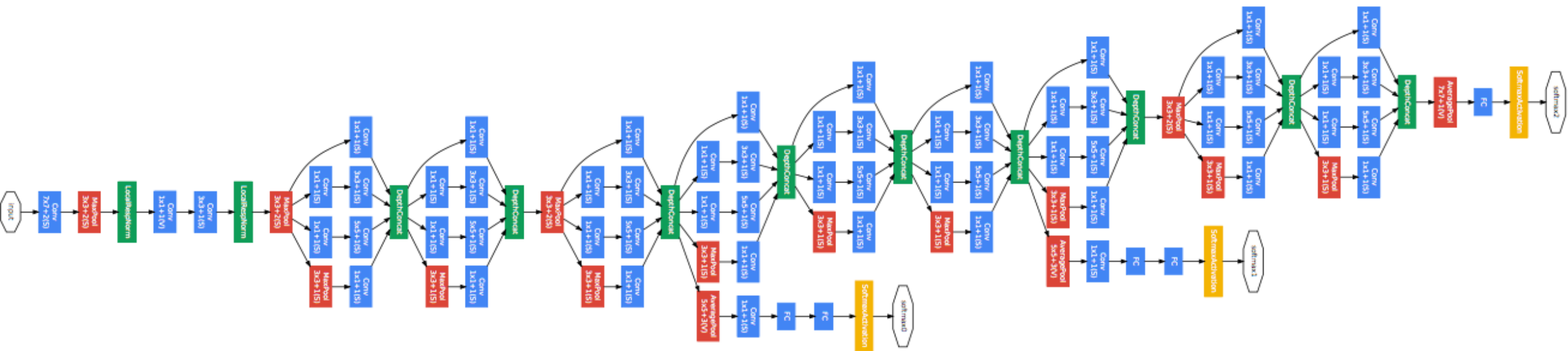
Full GoogLeNet
architecture



Auxiliary classification outputs to inject additional gradient at lower layers
(AvgPool-1x1Conv-FC-FC-Softmax)

Case Study: GoogLeNet

[Szegedy et al., 2014]

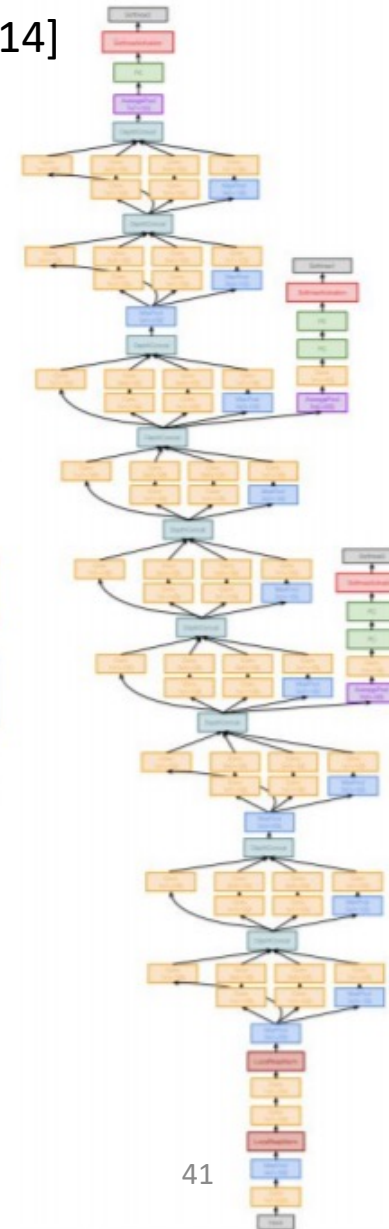
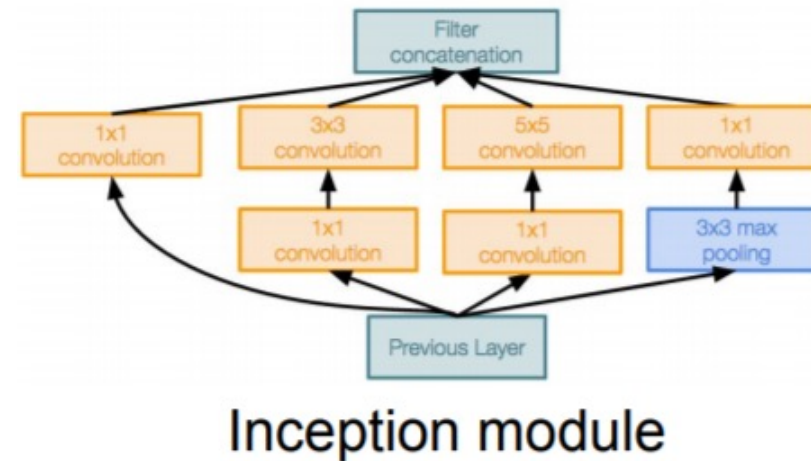


22 total layers with weights
(parallel layers count as 1 layer => 2 layers per Inception module.
Don't count auxiliary output layers)

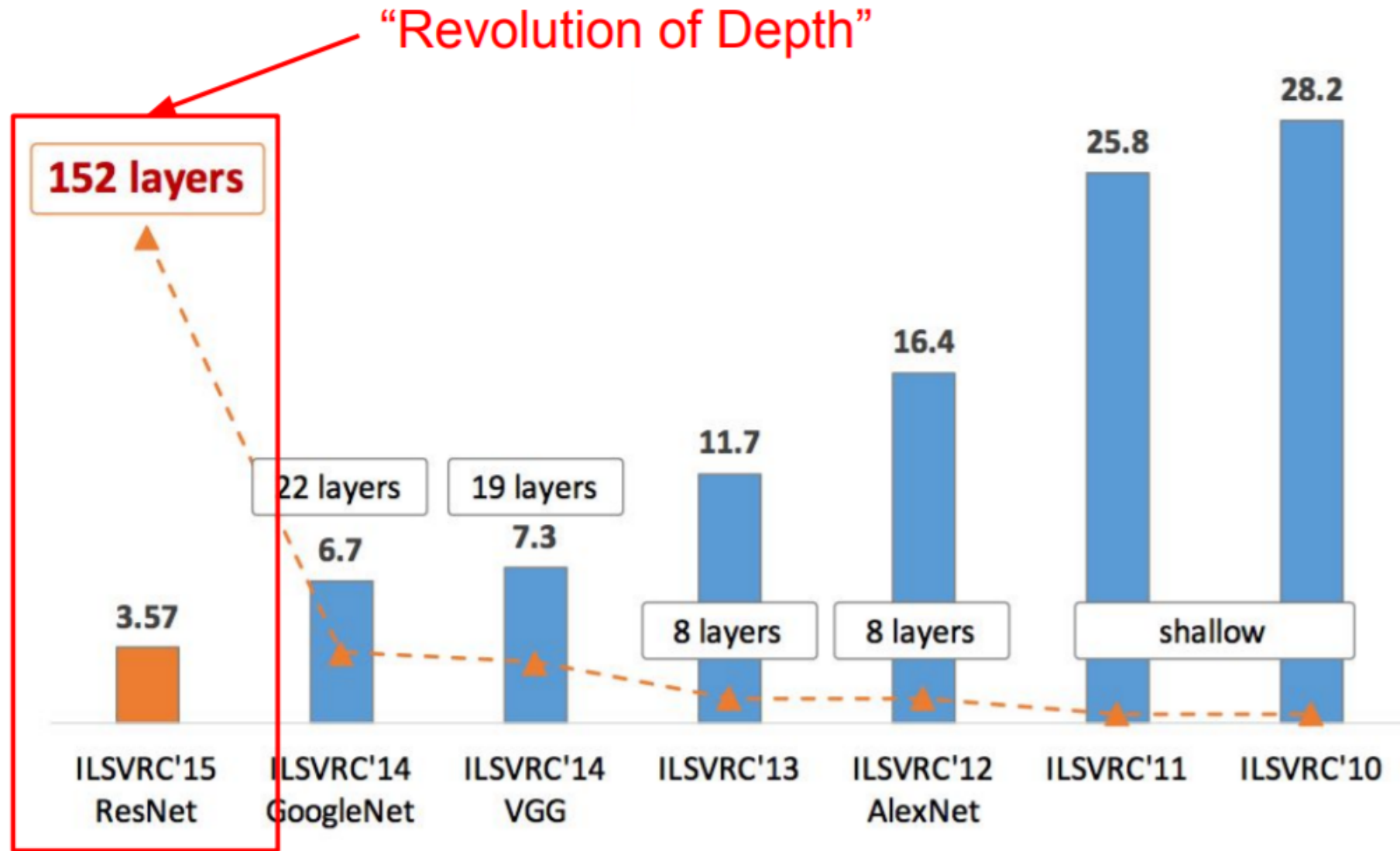
Case Study: GoogLeNet

- Deeper networks, with computation
 - 22 layers
 - Efficient “Inception” module
 - No FC layers
 - Only 5 million parameters!
 - 12x less than AlexNet
 - ILSVRC’14 classification winner (6.7% top 5 error)

[Szegedy et al., 2014]



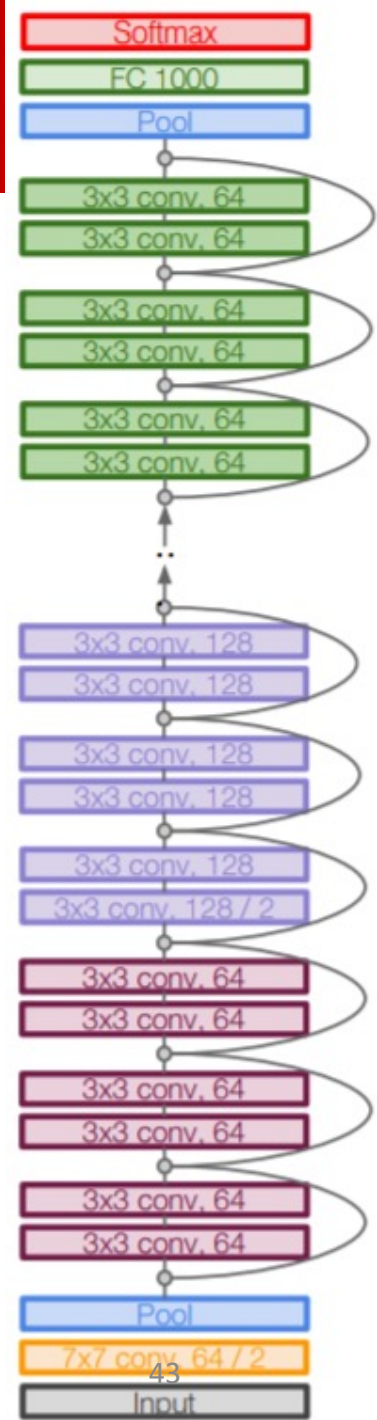
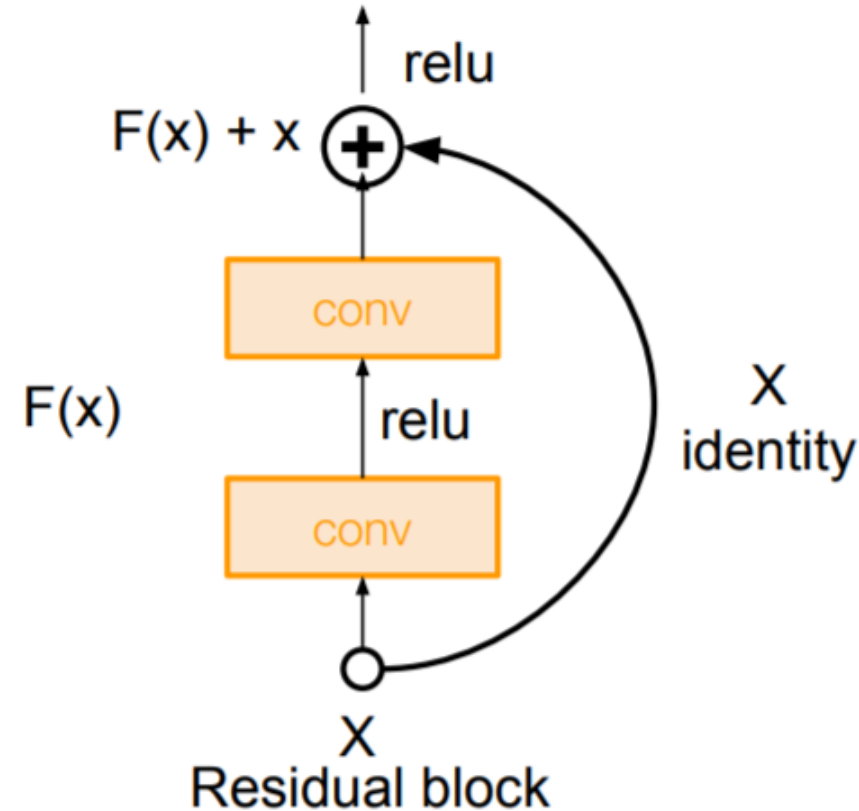
ImageNet Large Scale Visual Recognition Challenge (ILSVRC)



Case Study: ResNet

[He et al., 2015]

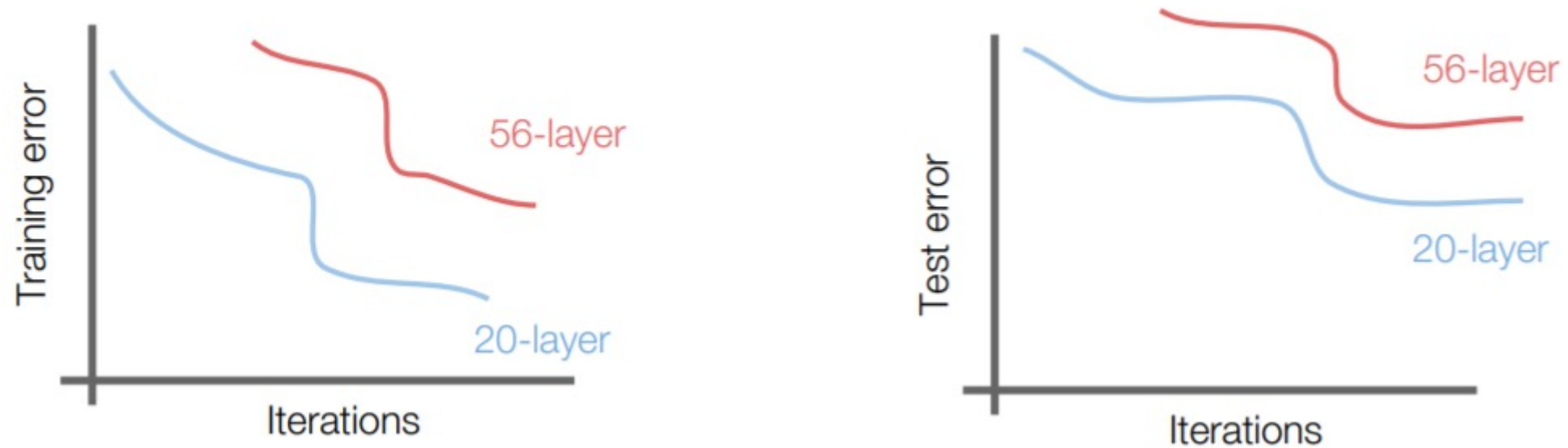
- Very deep networks using residual connections
 - 152-layer model for ImageNet
 - ILSVRC'15 classification winner (3.57% top 5 error)
 - Swept all classification and detection competitions in ILSVRC'15 and COCO'15!



Case Study: ResNet

[He et al., 2015]

- What happens when we continue stacking deeper layers on a “plain” convolutional neural network?

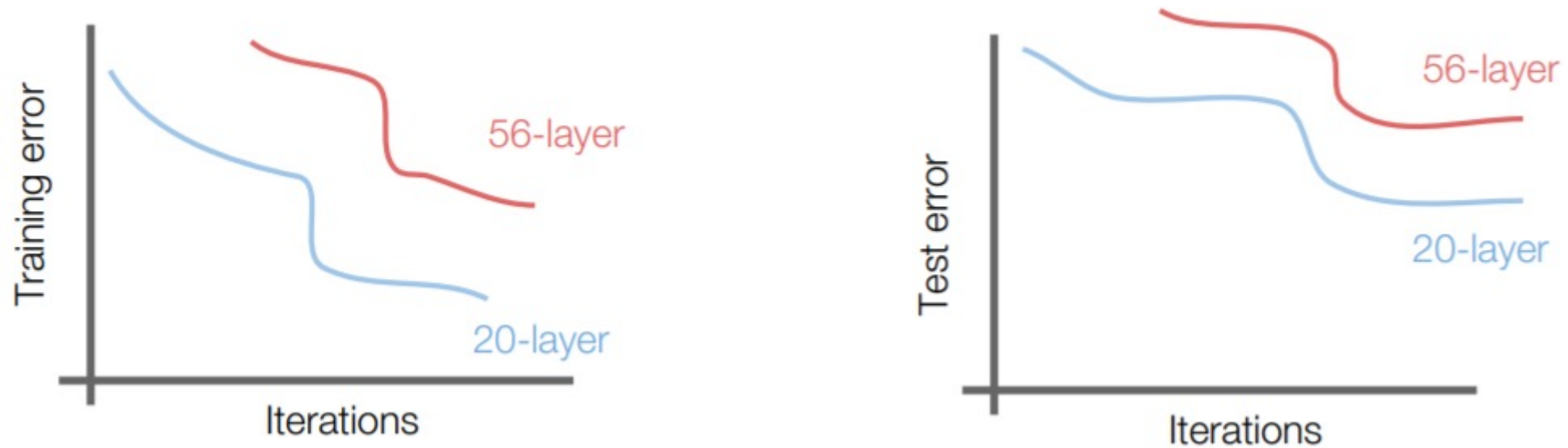


- Q: What's strange about these training and test curves?

Case Study: ResNet

[He et al., 2015]

- What happens when we continue stacking deeper layers on a “plain” convolutional neural network?



- 56-layer model performs worse on both training and test error
 - A deeper model should not have higher training error
 - The deeper model performs worse, but it's not caused by overfitting!

Case Study: ResNet

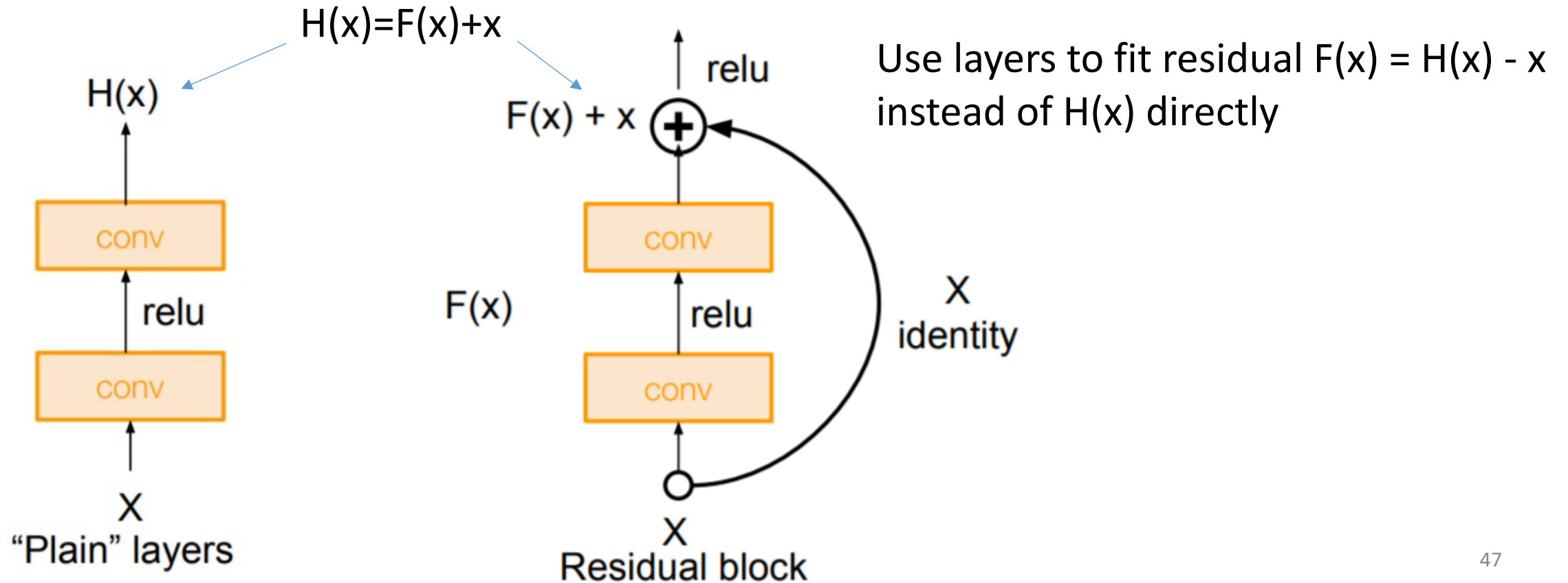
[He et al., 2015]

- Hypothesis: the problem is an optimization problem, deeper models are harder to optimize
- The deeper model should be able to perform at least as well as the shallower model.
 - A solution by construction is copying the learned layers from the shallower model and setting additional layers to identity mapping.

Case Study: ResNet

[He et al., 2015]

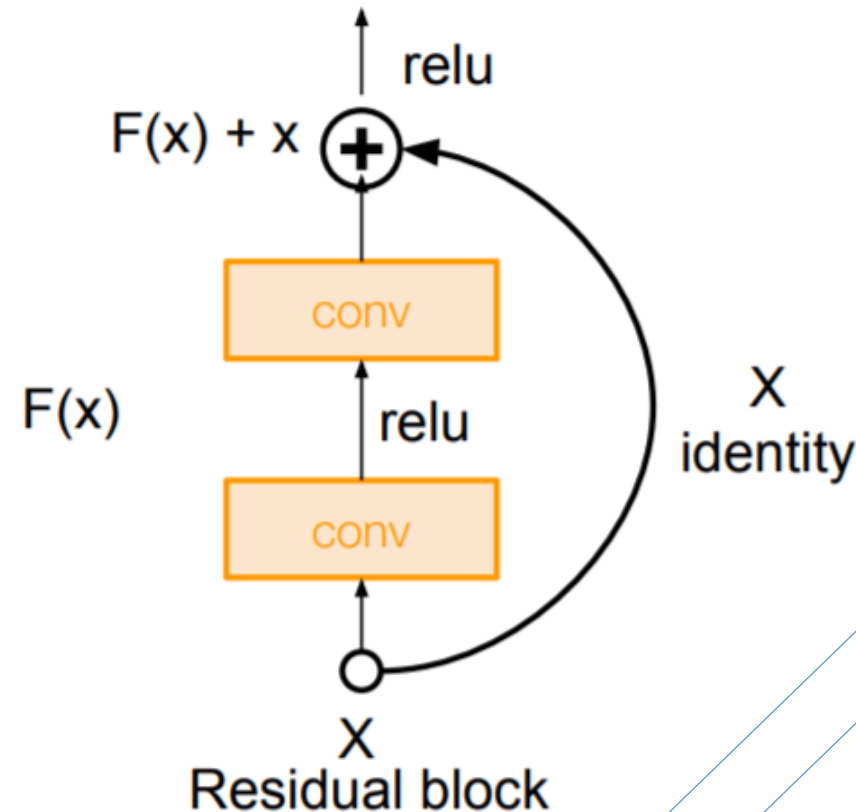
- Solution: Use network layers to fit a residual mapping $F(x)$ instead of directly trying to fit a desired underlying mapping $H(x)$



Case Study: ResNet

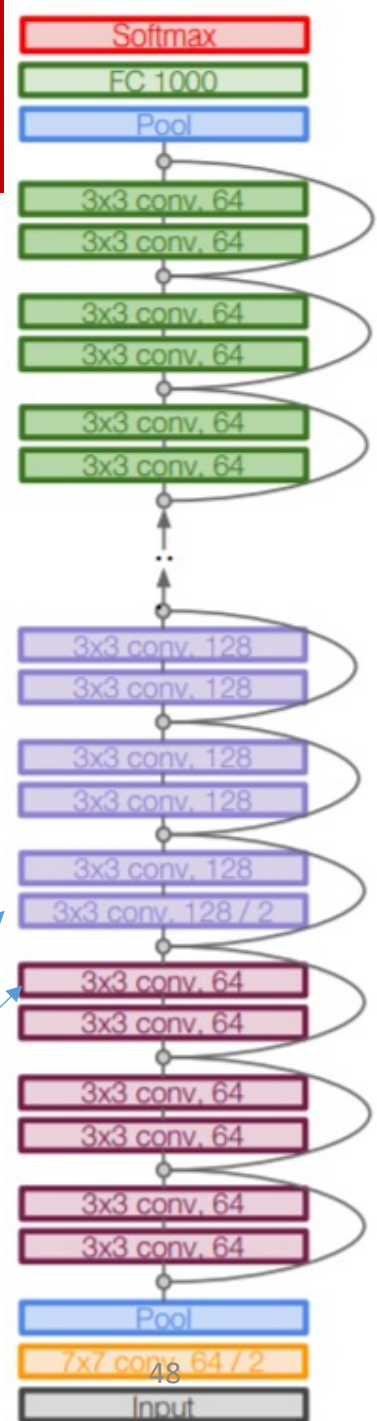
[He et al., 2015]

- Full ResNet architecture:
 - Stack residual blocks
 - Every residual block has two 3x3 conv layers
 - Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)



128 filters, spatially with stride 2

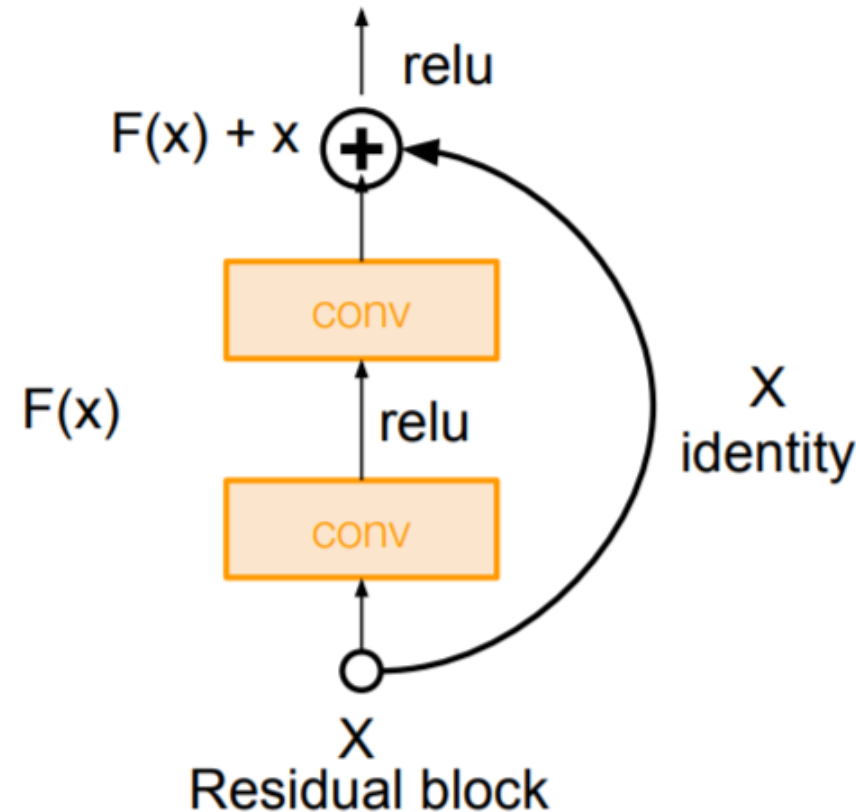
64 filters, spatially with stride 1



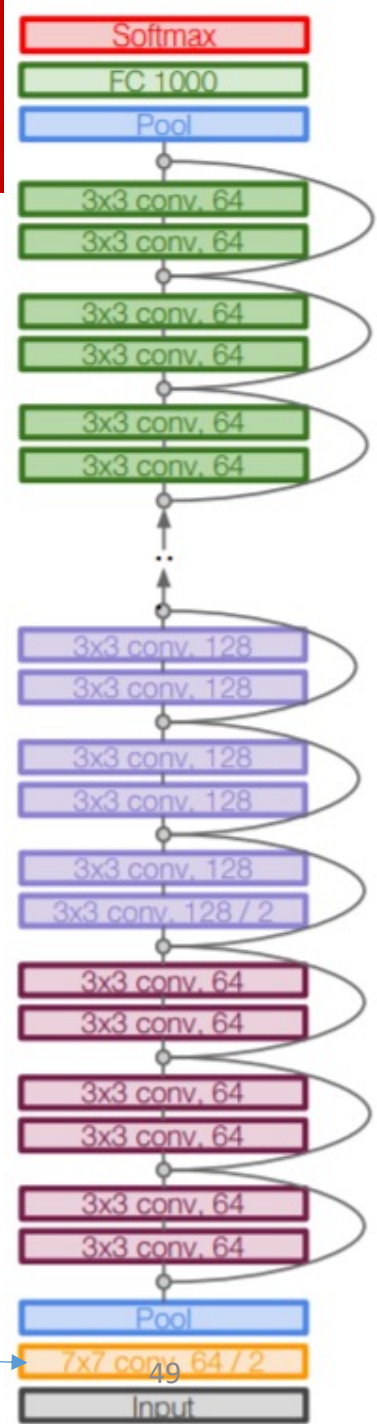
Case Study: ResNet

[He et al., 2015]

- Full ResNet architecture:
 - Stack residual blocks
 - Every residual block has two 3x3 conv layers
 - Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
 - Additional conv layer at the beginning



Beginning conv layer



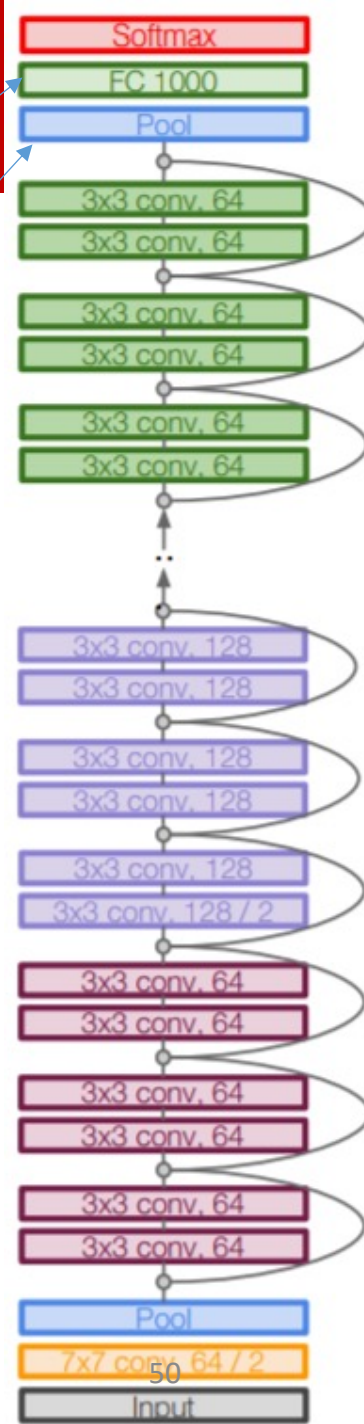
Case Study: ResNet

- Full ResNet architecture:
 - Stack residual blocks
 - Every residual block has two 3x3 conv layers
 - Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
 - Additional conv layer at the beginning
 - No FC layers at the end (only FC 1000 to output classes)

No FC layers besides
FC 1000 to output classes

Global average pooling
layer after last conv layer

[He et al., 2015]

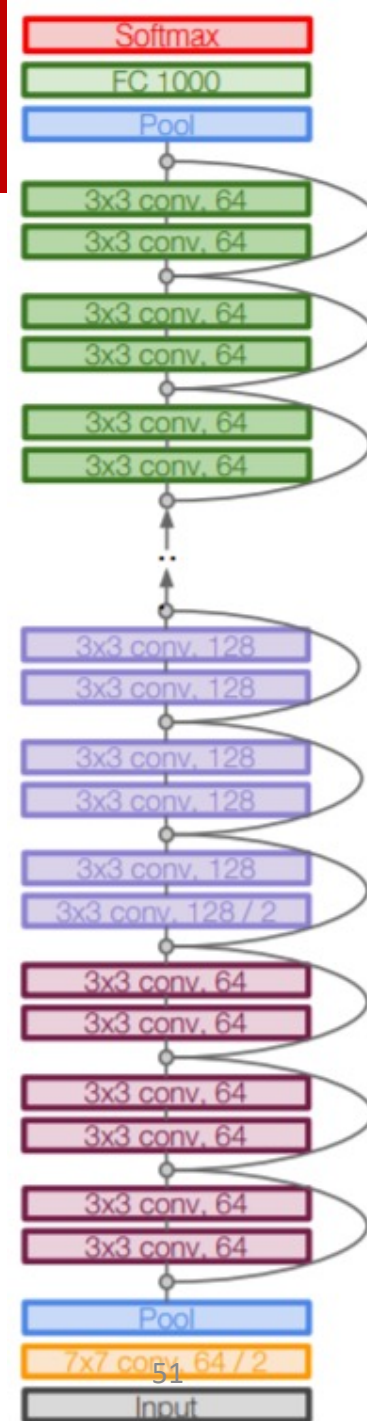


Case Study: ResNet

[He et al., 2015]

For deeper networks (ResNet-50+),
use “bottleneck” layer to improve
efficiency (similar to GoogLeNet)

Total depths of 34,
50, 101, or 152 layers
for ImageNet



Case Study: ResNet

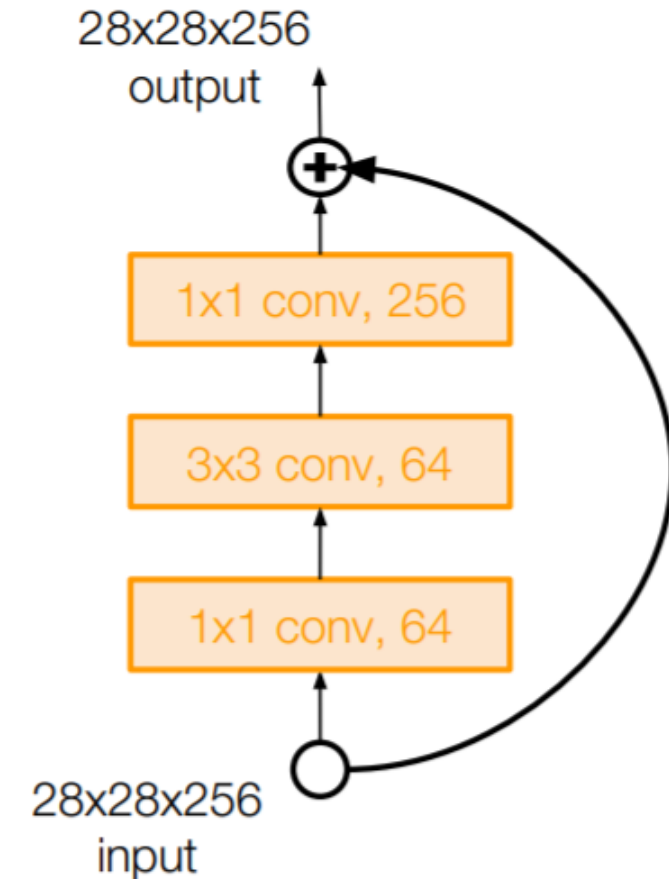
[He et al., 2015]

For deeper networks (ResNet-50+), use “bottleneck” layer to improve efficiency (similar to GoogLeNet)

1x1 conv, 256 filters projects back to 256 feature maps (28x28x256)

3x3 conv operates over only 64 feature maps

1x1 conv, 64 filters to project to 28x28x64

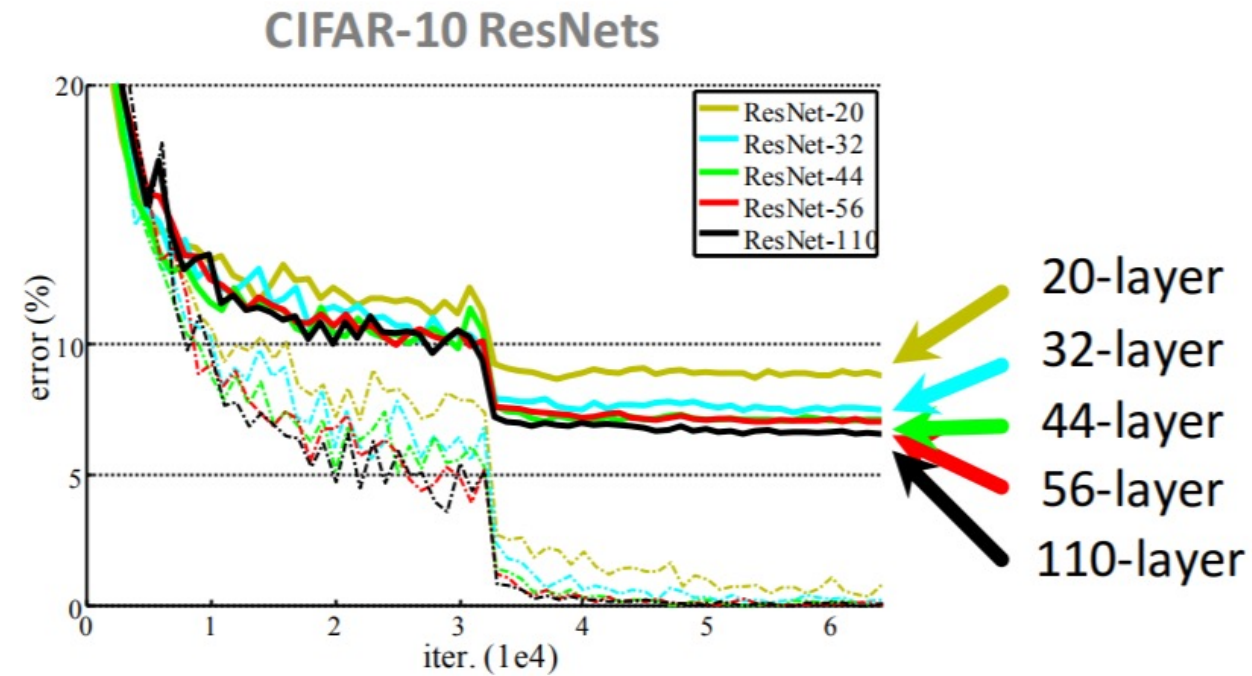
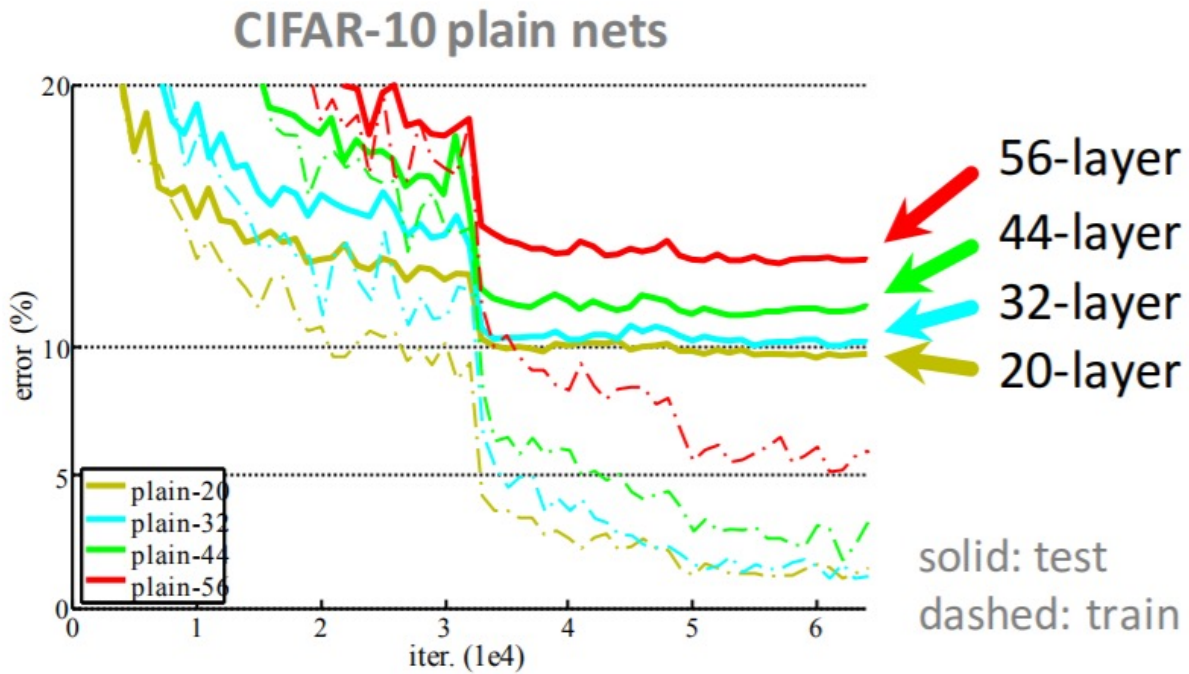


Case Study: ResNet

[He et al., 2015]

- Training ResNet in practice:
 - Batch Normalization after every CONV layer
 - Xavier/2 initialization from He et al.
 - SGD + Momentum (0.9)
 - Learning rate: 0.1, divided by 10 when validation error plateaus
 - Mini-batch size 256
 - Weight decay of $1e-5$
 - No dropout used

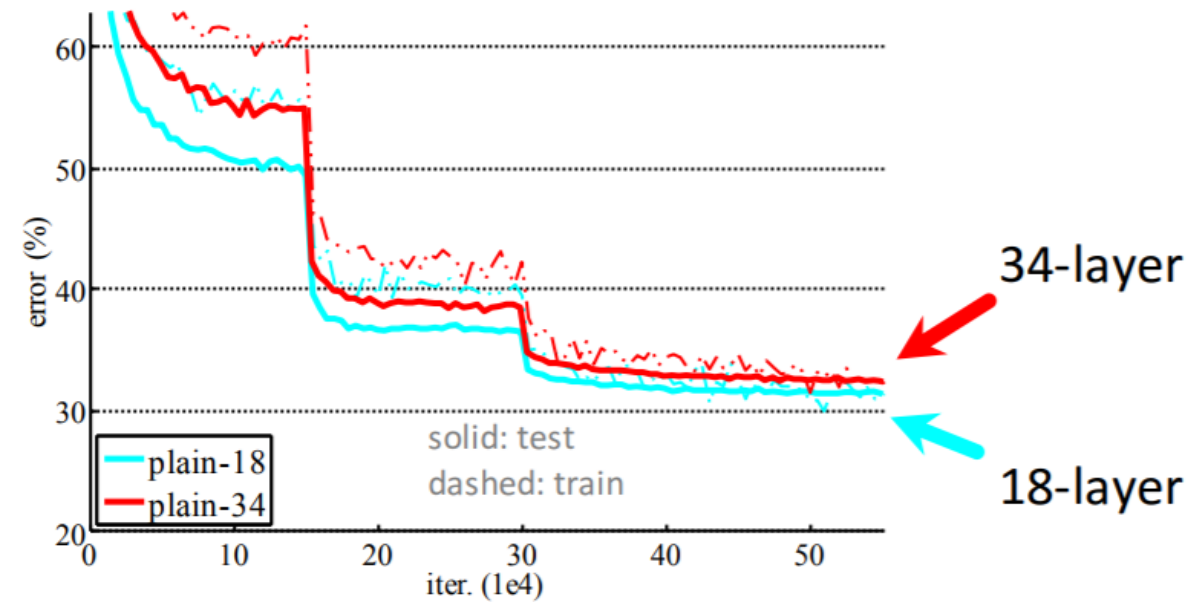
ResNet: CIFAR-10 experiments



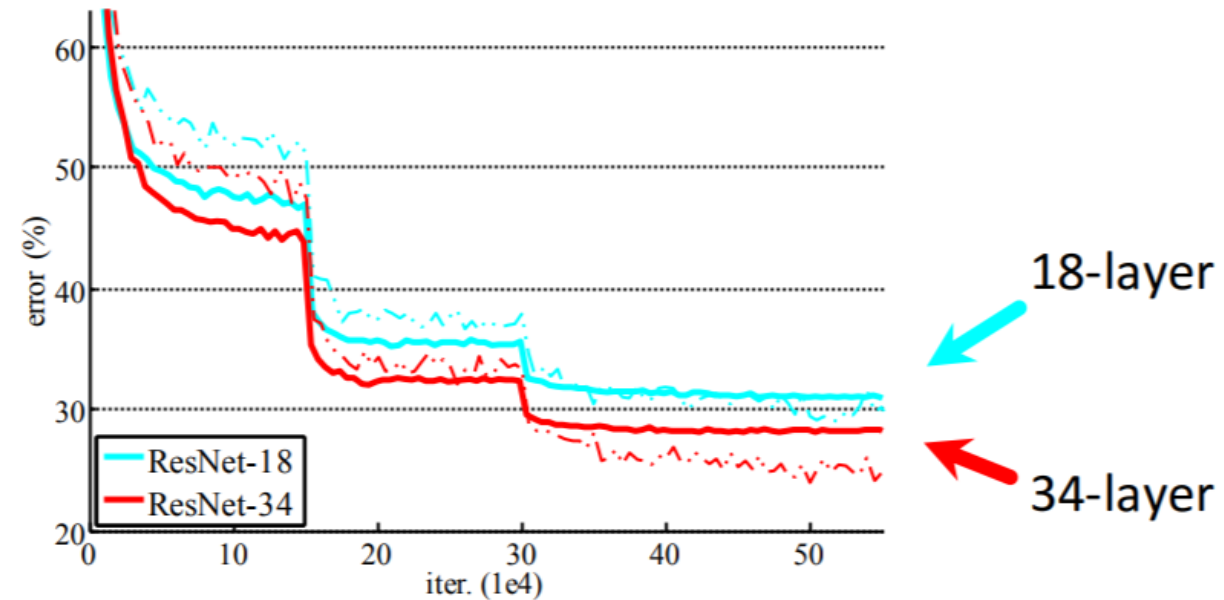
- Deeper ResNets have lower training error, and also lower test error
 - Not explicitly address generalization, but deeper+thinner shows good generalization

ResNet: ImageNet experiments

ImageNet plain nets



ImageNet ResNets



Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

Case Study: ResNet

[He et al., 2015]

- Experimental Results
 - Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
 - Deeper networks now achieve lowering training error as expected
 - Swept 1st place in all ILSVRC and COCO 2015 competitions

MSRA @ ILSVRC & COCO 2015 Competitions

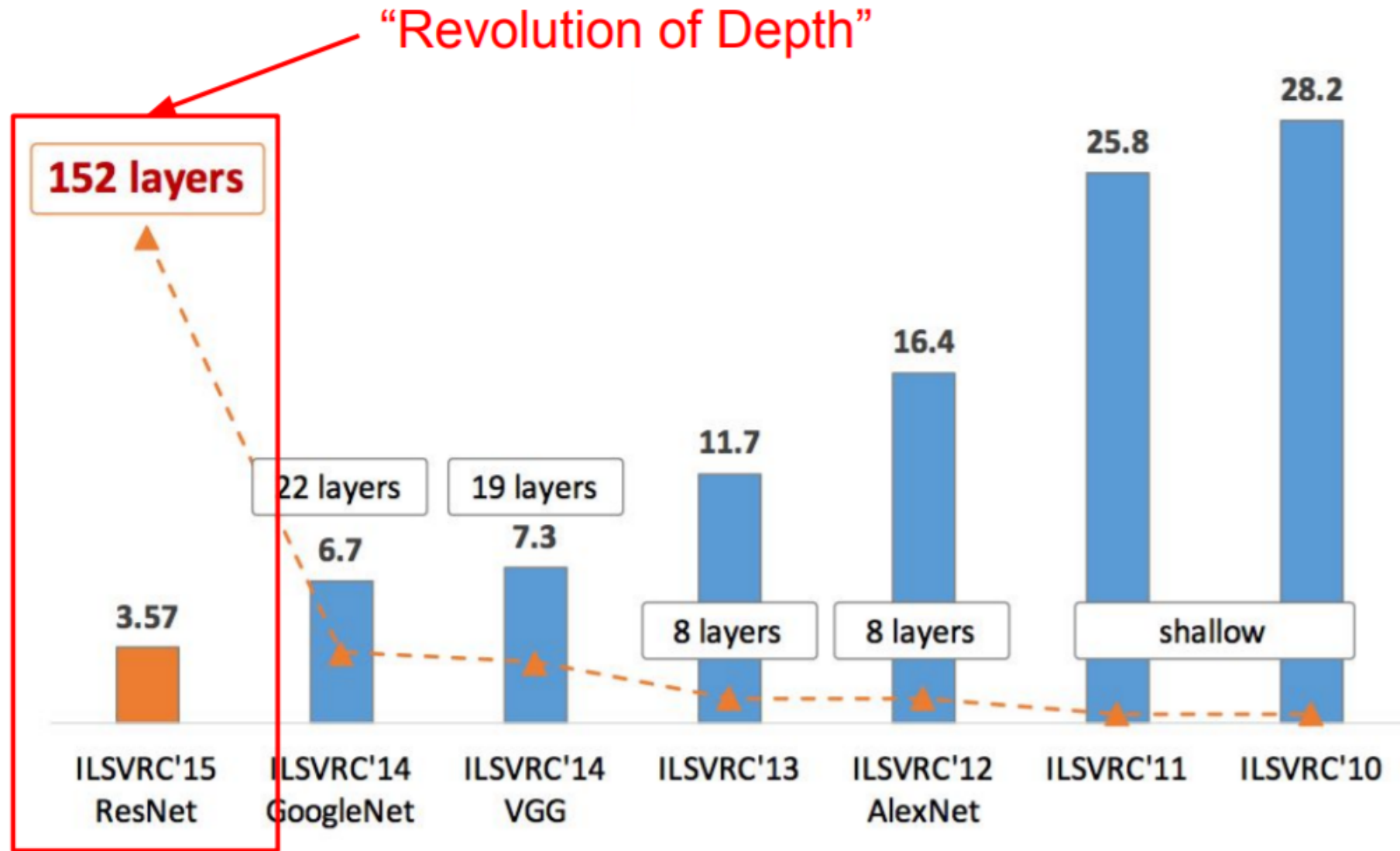
- **1st places in all five main tracks**

- 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

ILSVRC 2015 classification winner (3.6% top 5 error)

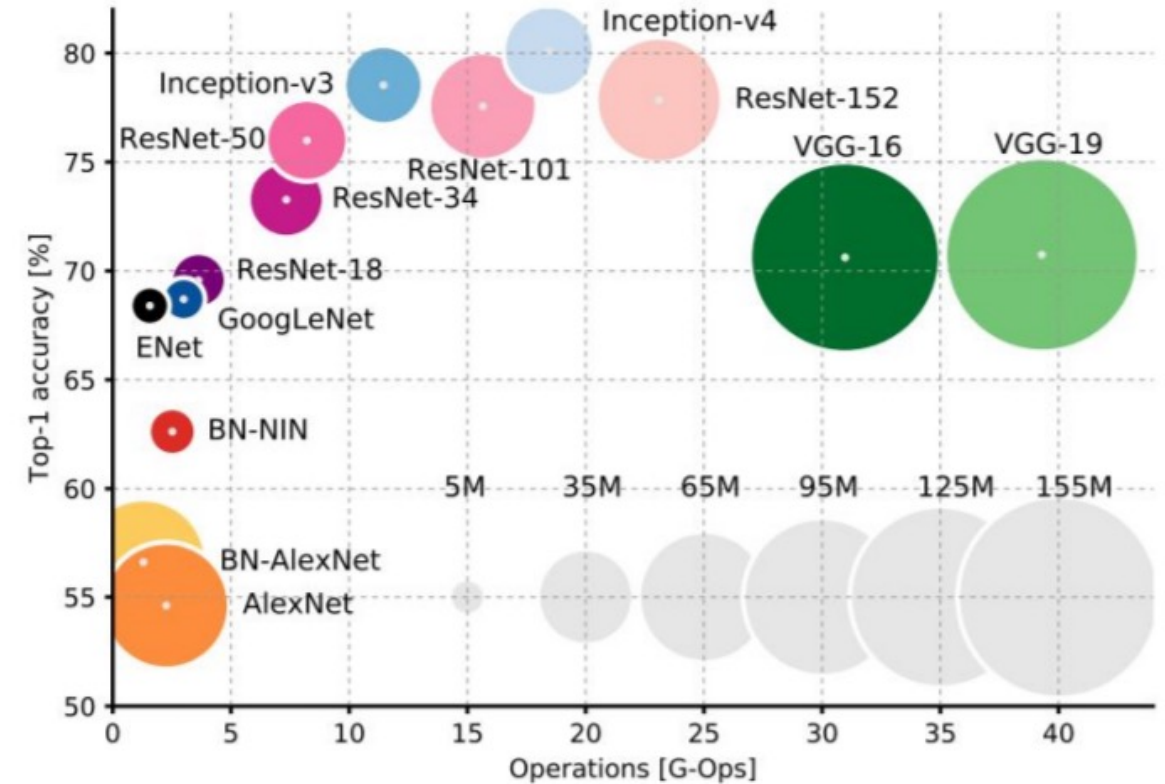
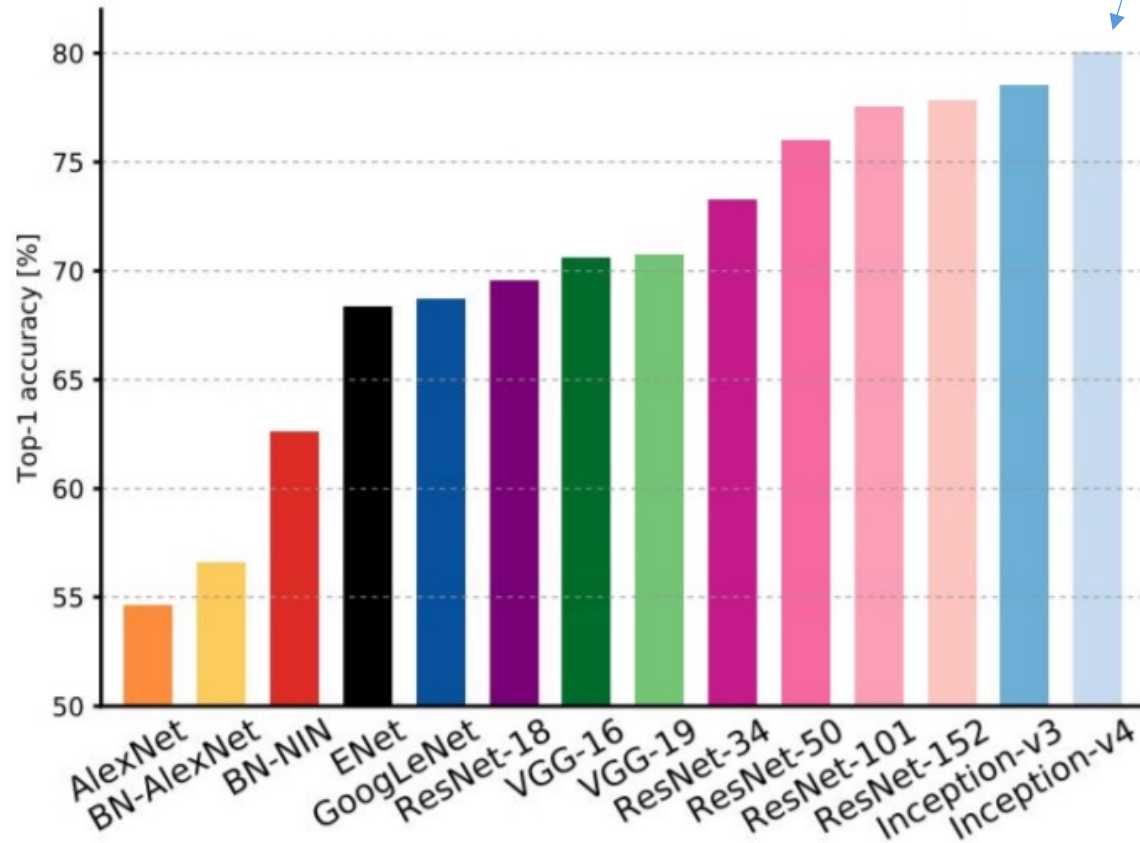
better than “human performance”! (Russakovsky 2014)

ImageNet Large Scale Visual Recognition Challenge (ILSVRC)



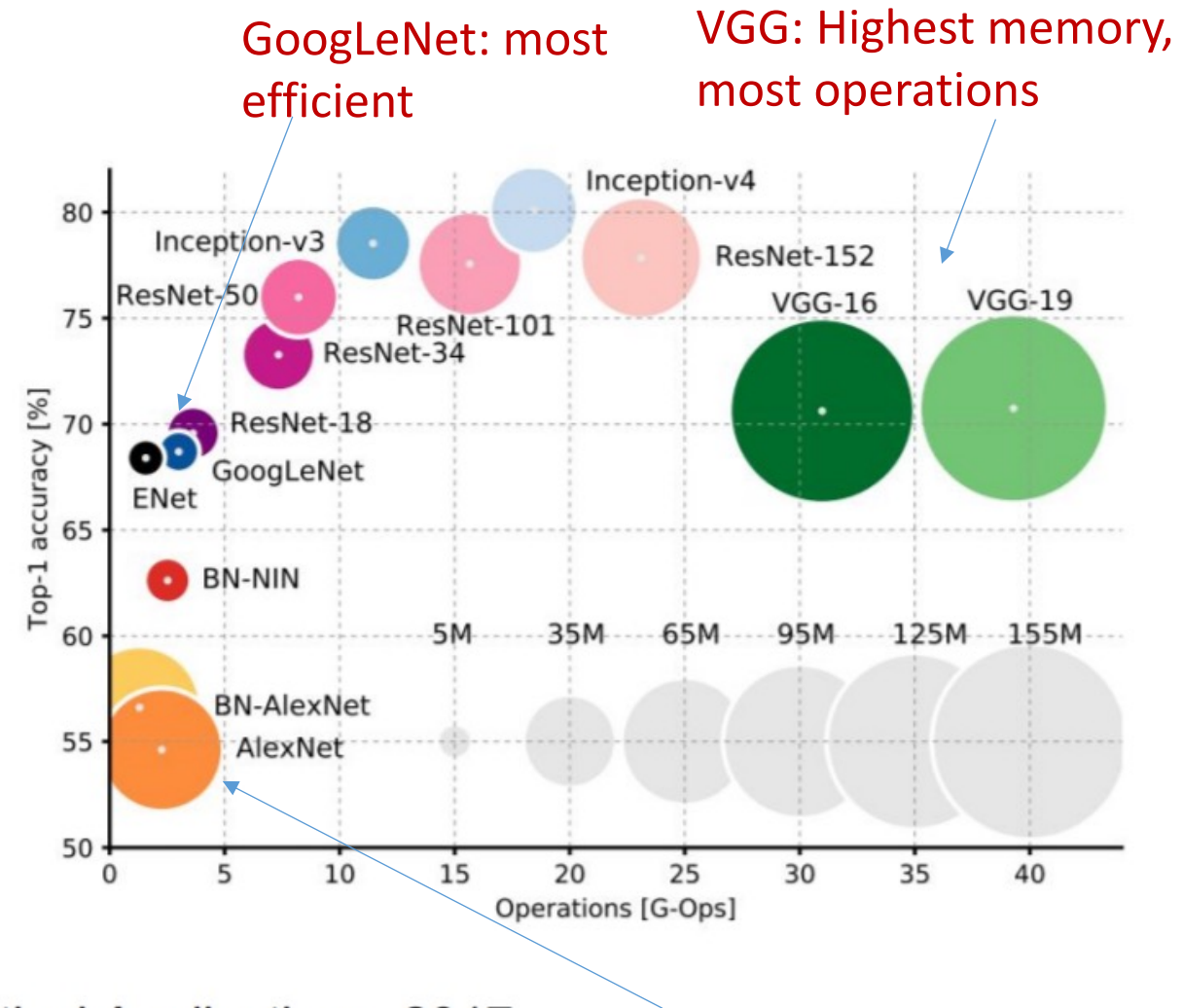
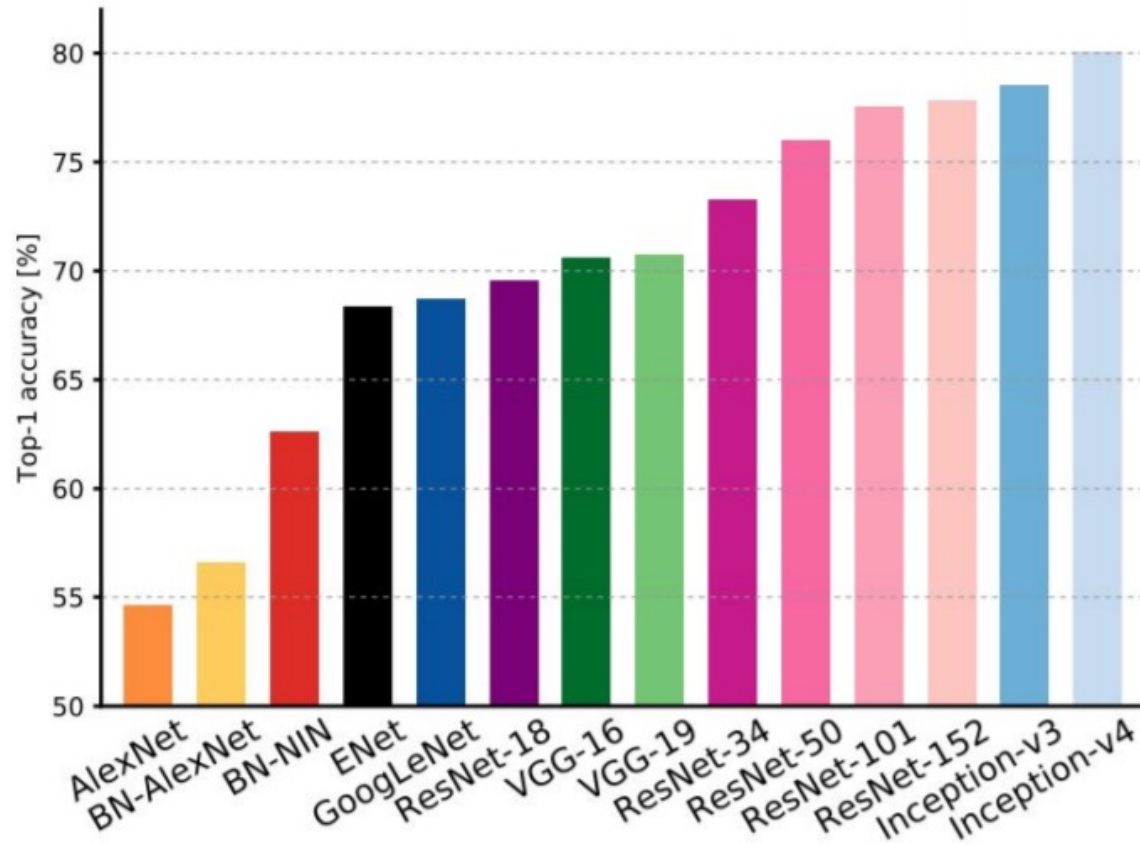
Comparing complexity

Inception-v4: Resnet + Inception!



An Analysis of Deep Neural Network Models for Practical Applications, 2017.

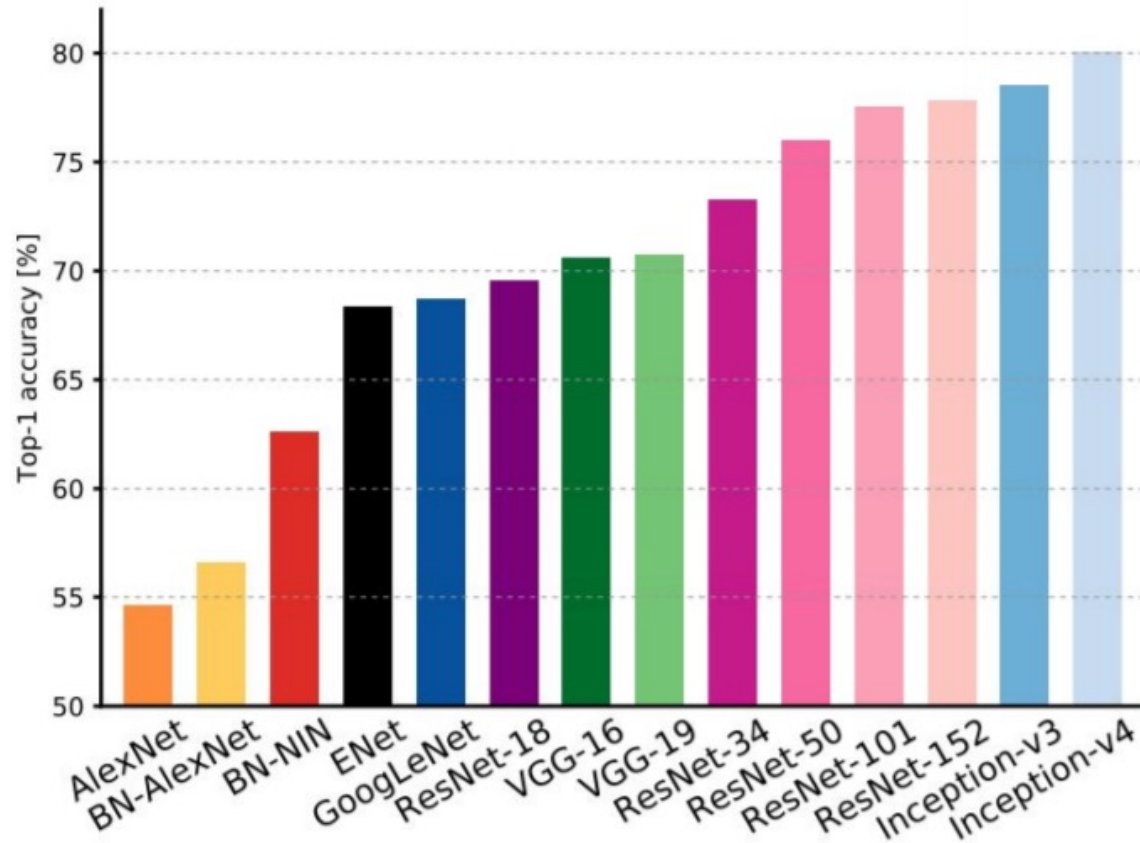
Comparing complexity



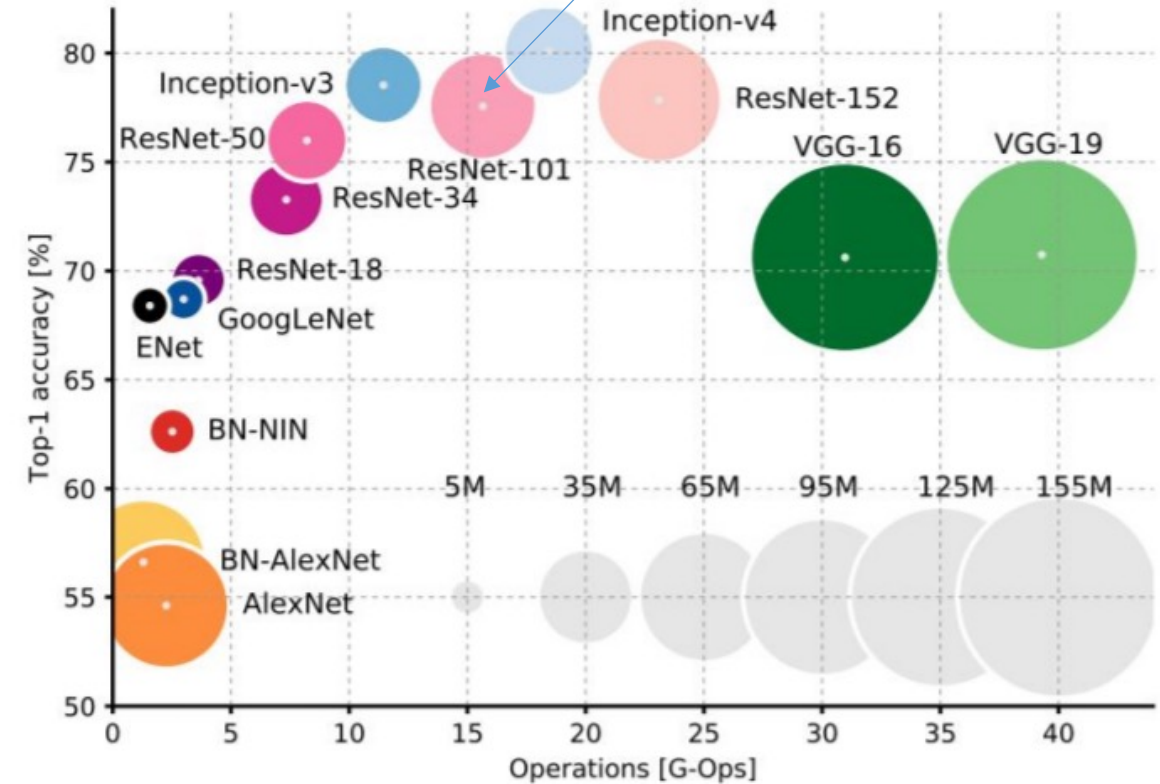
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

AlexNet: Smaller ops, still memory heavy, lower accuracy

Comparing complexity



ResNet: Moderate efficiency
depending on model, highest accuracy



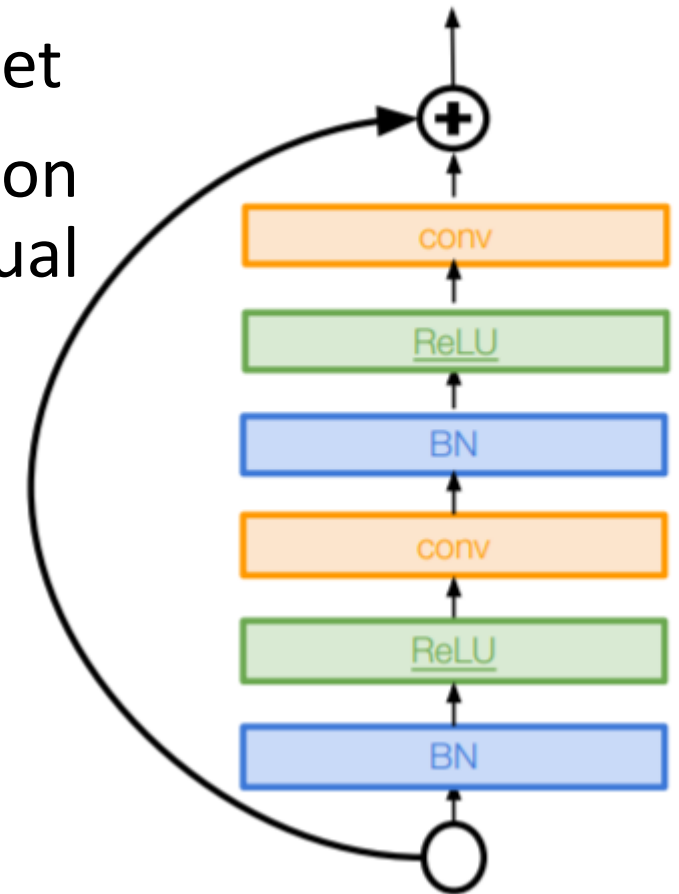
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Improving ResNets...

[He et al. 2016]

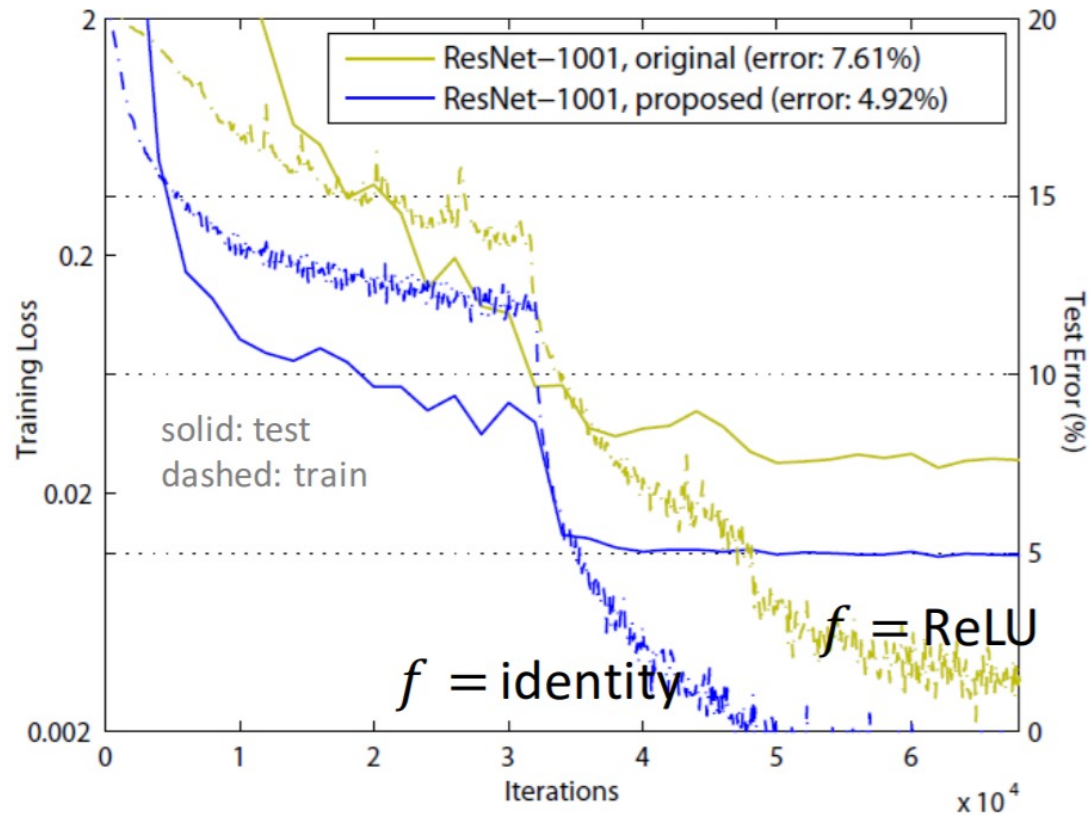
Identity Mappings in Deep Residual Networks

- Improved ResNet block design from creators of ResNet
- Creates a more direct path for propagating information throughout network (moves activation to residual mapping pathway)
- Gives better performance

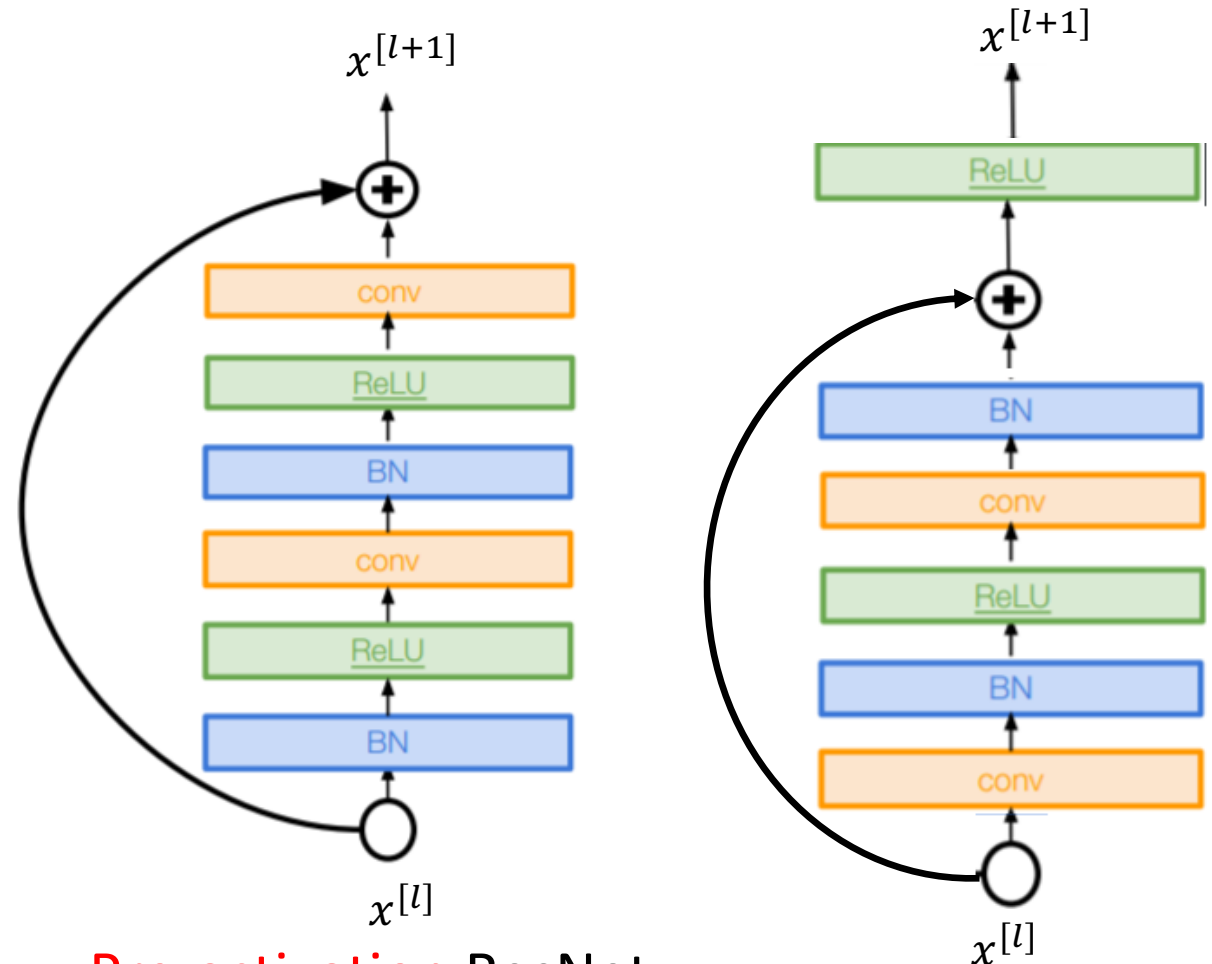


Improving ResNets...

- Identity Mappings in Deep Residual Networks



ReLU could block back prop for very deep networks



Pre-activation ResNet

Backprop on ResNet

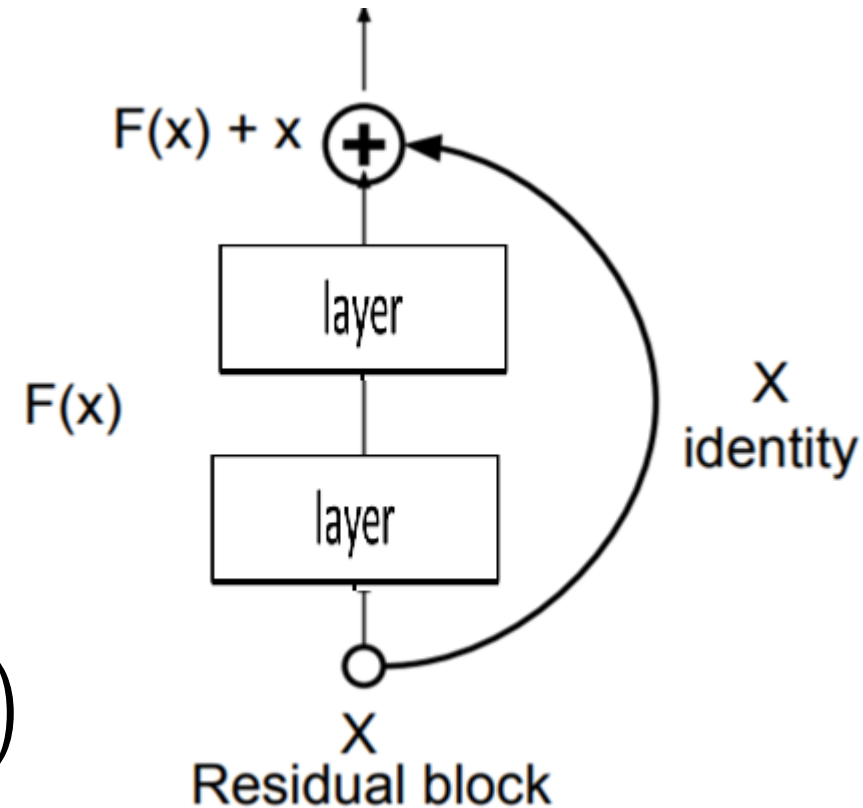
$$x^{[l+1]} = x^{[l]} + F(x^{[l]})$$

$$x^{[l+2]} = x^{[l+1]} + F(x^{[l+1]})$$

$$x^{[l+2]} = x^{[l]} + F(x^{[l]}) + F(x^{[l+1]})$$

$$x^{[L]} = x^{[l]} + \sum_{i=l}^{L-1} F(x^{[i]})$$

$$\frac{\partial E}{\partial x^{[l]}} = \frac{\partial E}{\partial x^{[L]}} \frac{\partial x^{[L]}}{\partial x^{[l]}} = \frac{\partial E}{\partial x^{[L]}} \left(1 + \frac{\partial}{\partial x^{[l]}} \sum_{i=l}^{L-1} F(x^{[i]}) \right)$$



Any $\frac{\partial E}{\partial x^{[L]}}$ is **directly back-prop** to any $\frac{\partial E}{\partial x^{[l]}}$, plus residual.

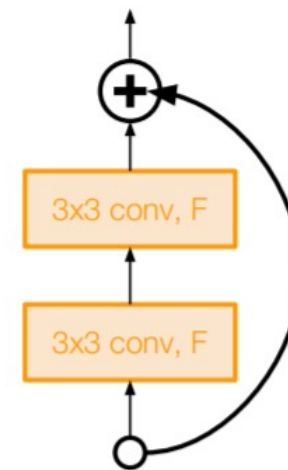
Any $\frac{\partial E}{\partial x^{[l]}}$ is **additive**; unlikely to vanish.

Improving ResNets...

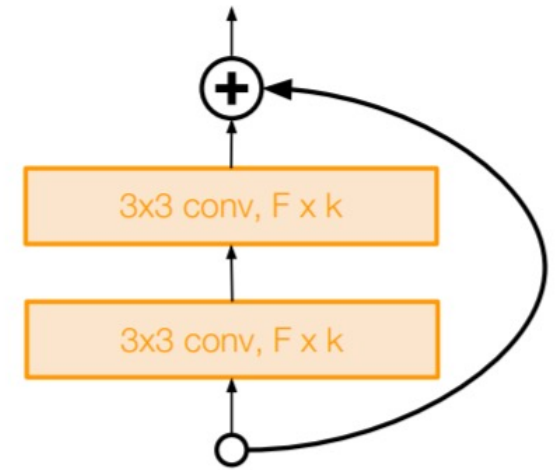
[Zagoruyko et al. 2016]

Wide Residual Networks

- Argues that residuals are the important factor, not depth
- Uses wider residual blocks ($F \times k$ filters instead of F filters in each layer)
- 50-layer wide ResNet outperforms 152-layer original ResNet
- Increasing width instead of depth more computationally efficient (parallelizable)



Basic residual block



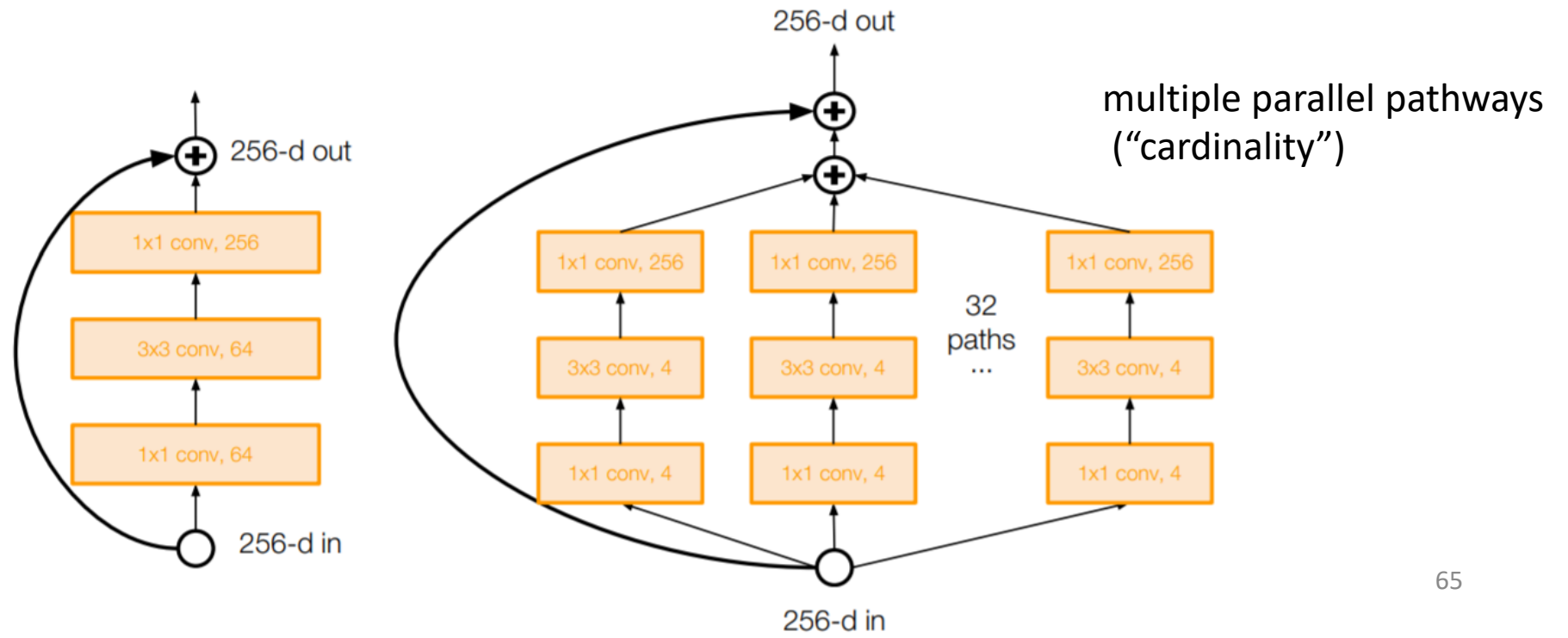
Wide residual block

Improving ResNets...

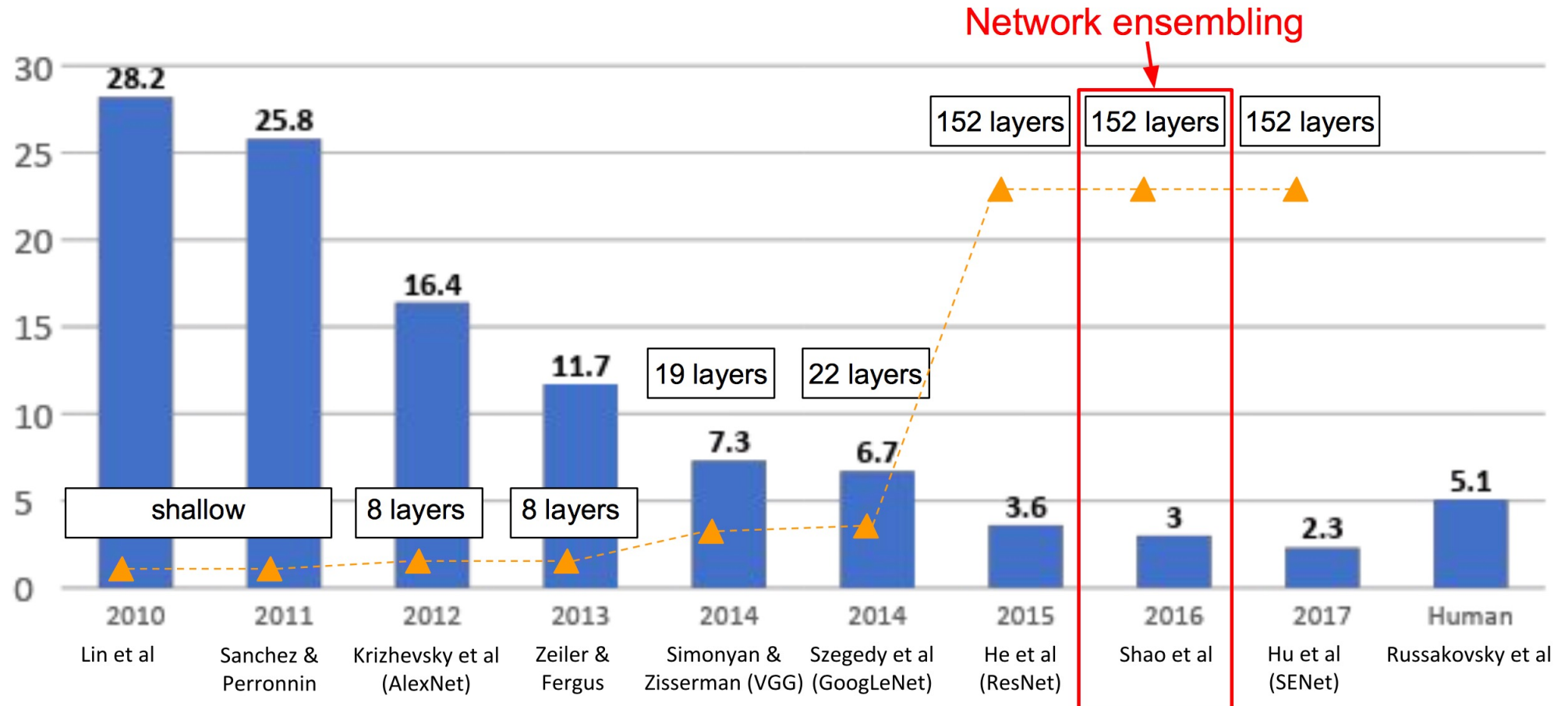
[Xie et al. 2016]

Aggregated Residual Transformations for Deep Neural Networks (ResNeXt)

- Also from creators of ResNet
- Increases width of residual block through multiple parallel pathways
- Parallel pathways similar in spirit to Inception module



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Improving ResNets...

“Good Practices for Deep Feature Fusion”

[Shao et al. 2016]

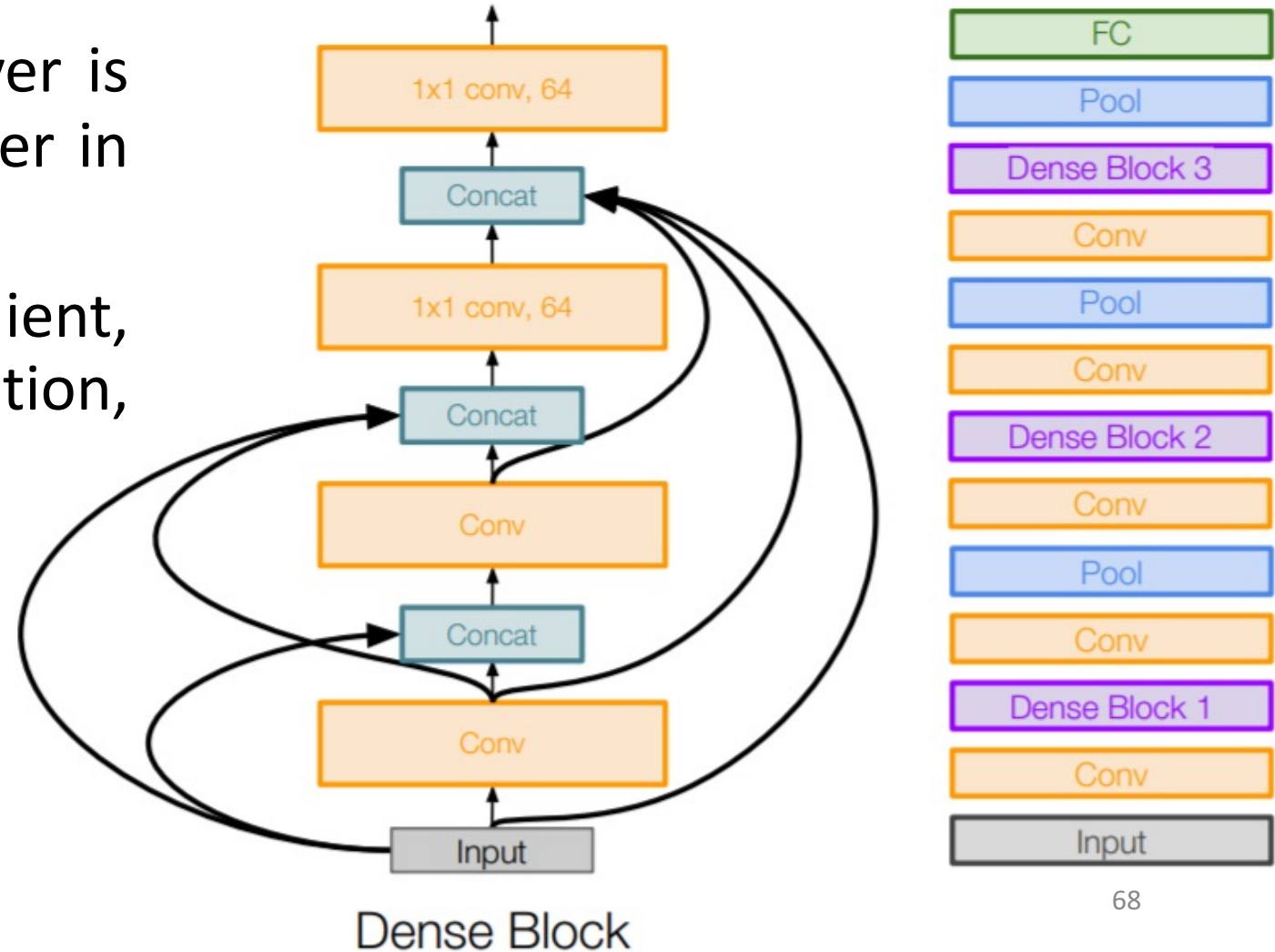
- Multi-scale ensembling of Inception, Inception-Resnet, Resnet, Wide Resnet models
- ILSVRC'16 classification winner

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

Beyond ResNets...

Densely Connected Convolutional Networks

- Dense blocks where each layer is connected to every other layer in feedforward fashion
- Alleviates vanishing gradient, strengthens feature propagation, encourages feature reuse

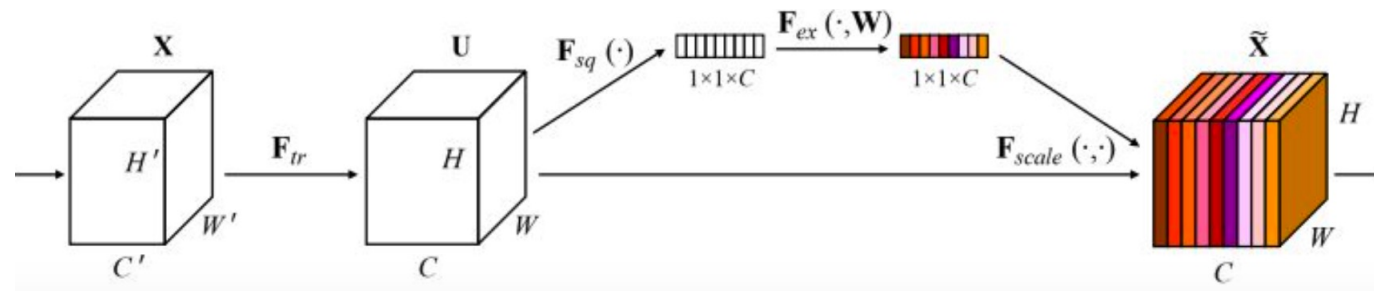
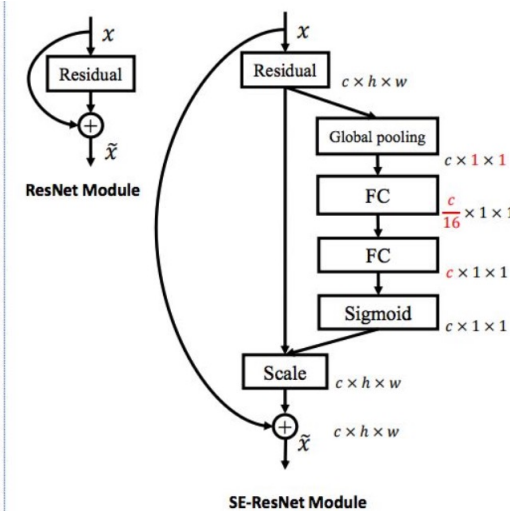
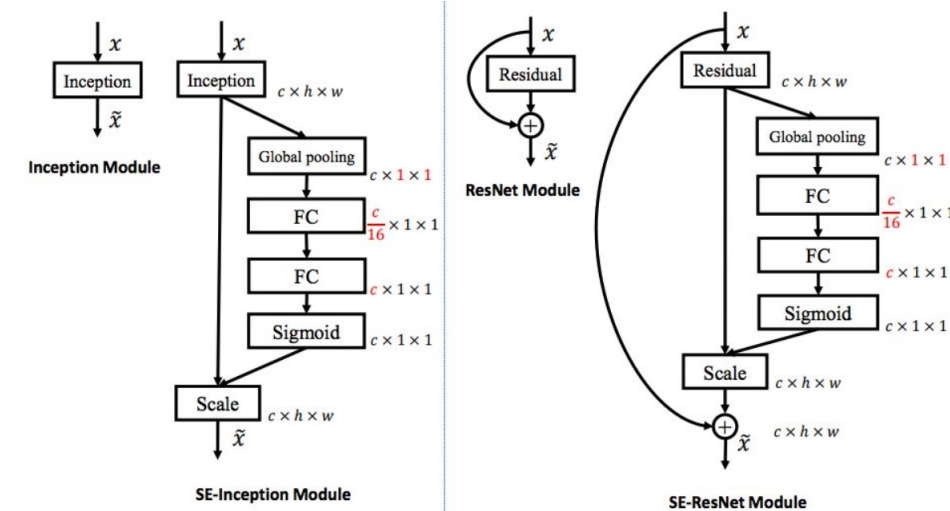


Improving ResNets...

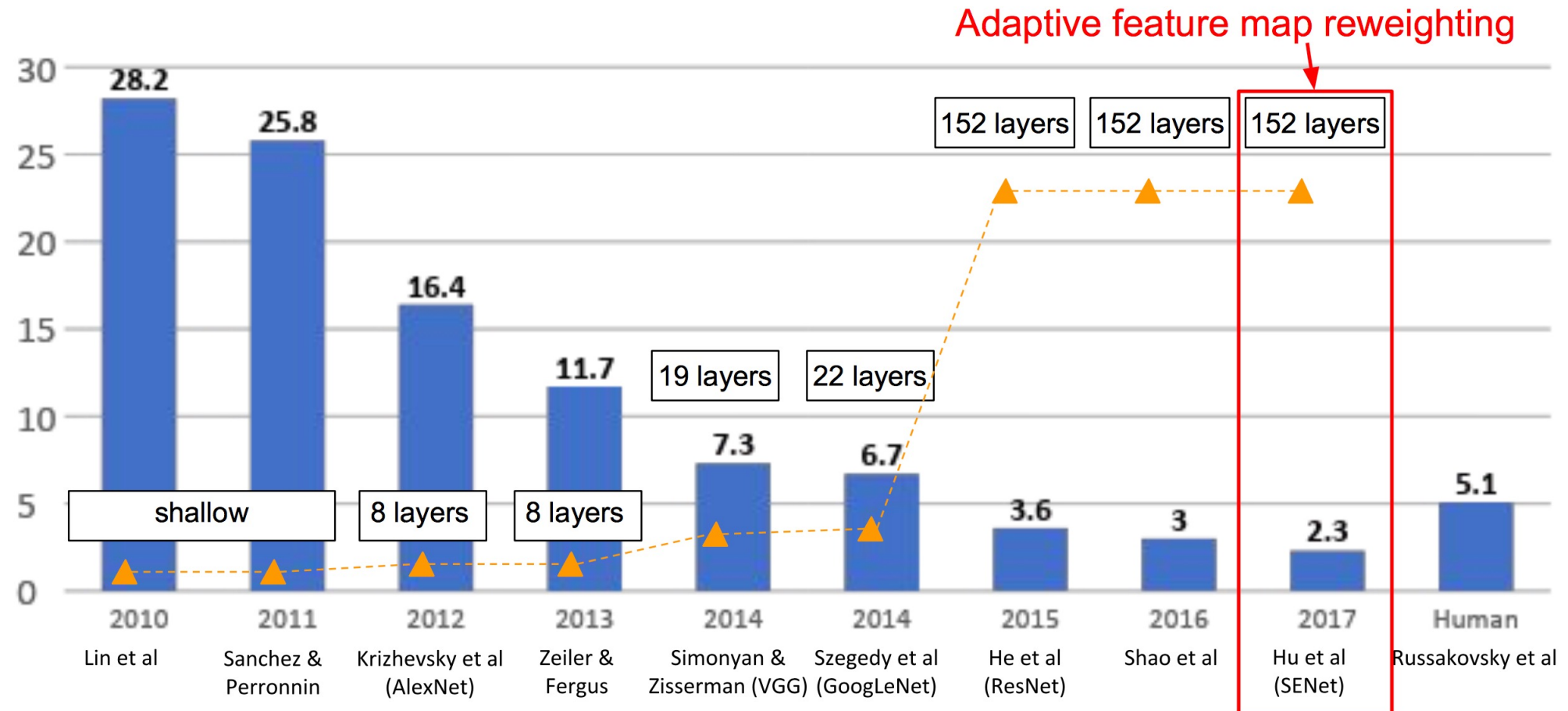
Squeeze-and-Excitation Networks (SENet)

[Hu et al. 2017]

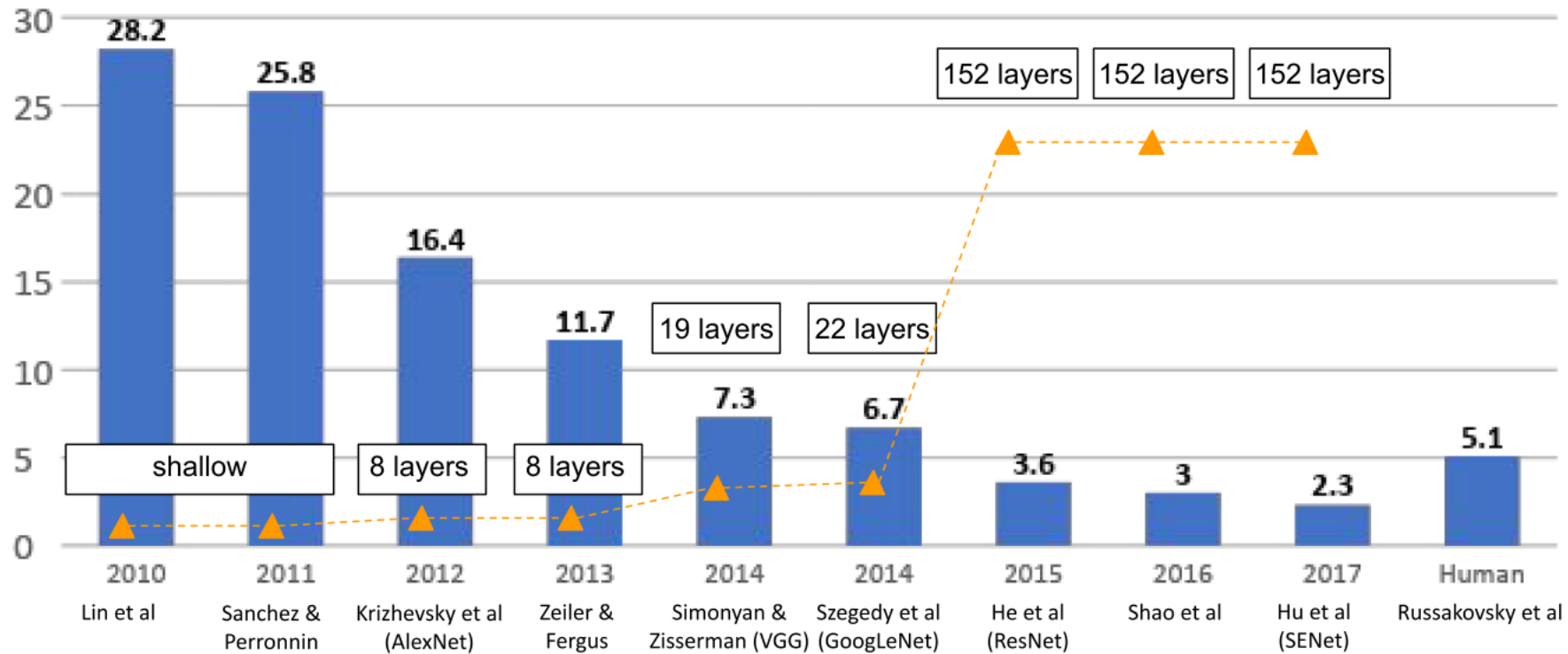
- Add a “feature recalibration” module that learns to adaptively reweight feature maps
- Global information (global avg. pooling layer) + 2 FC layers used to determine feature map weights
- ILSVRC'17 classification winner (using ResNeXt-152 as a base architecture)



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



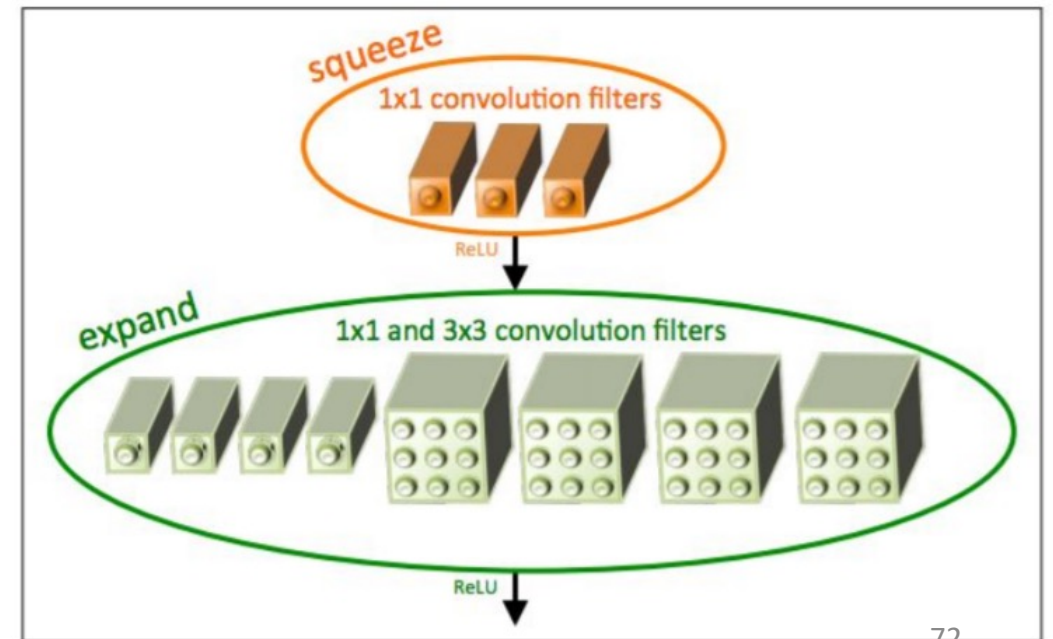
Completion of the challenge:
Annual ImageNet competition no longer
held after 2017 -> now moved to Kaggle.

Efficient networks...

[Iandola et al. 2017]

Squeeze Net:

- **Fire modules** consisting of a “squeeze” layer with 1x1 filters feeding an “expand” layer with 1x1 and 3x3 filters
- AlexNet level accuracy on ImageNet with 50x fewer parameters
- Can compress to 510x smaller than AlexNet (0.5Mb model size)

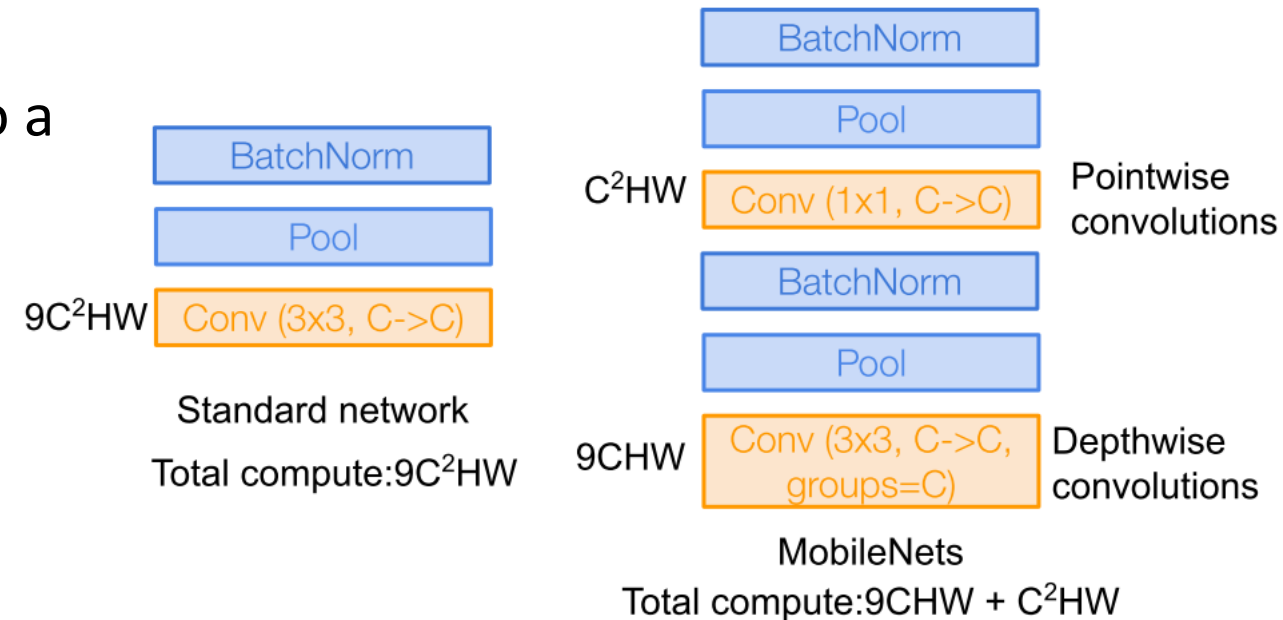


Efficient networks

[Howard et al. 2017]

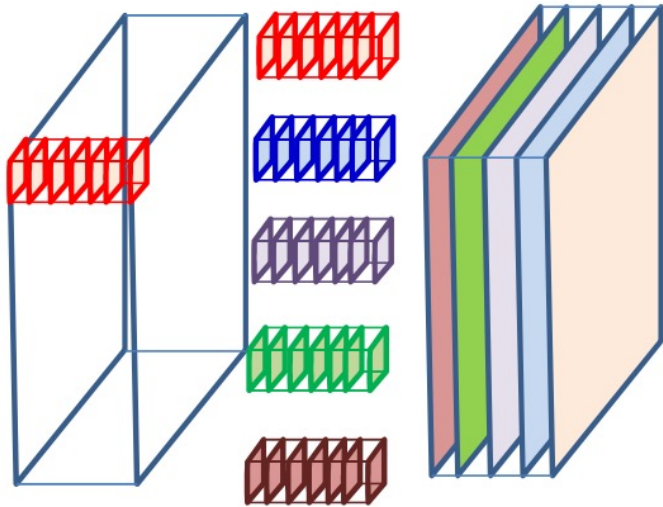
- **MobileNets:** Efficient Convolutional Neural Networks for Mobile Applications

- Depthwise separable convs. replace standard convs by factorizing them into a depthwise conv and a 1×1 conv
- Much more efficient, with little loss in accuracy
- Follow-up MobileNetV2 work in 2018 (Sandler et al.)
- ShuffleNet: Zhang et al, CVPR 2018



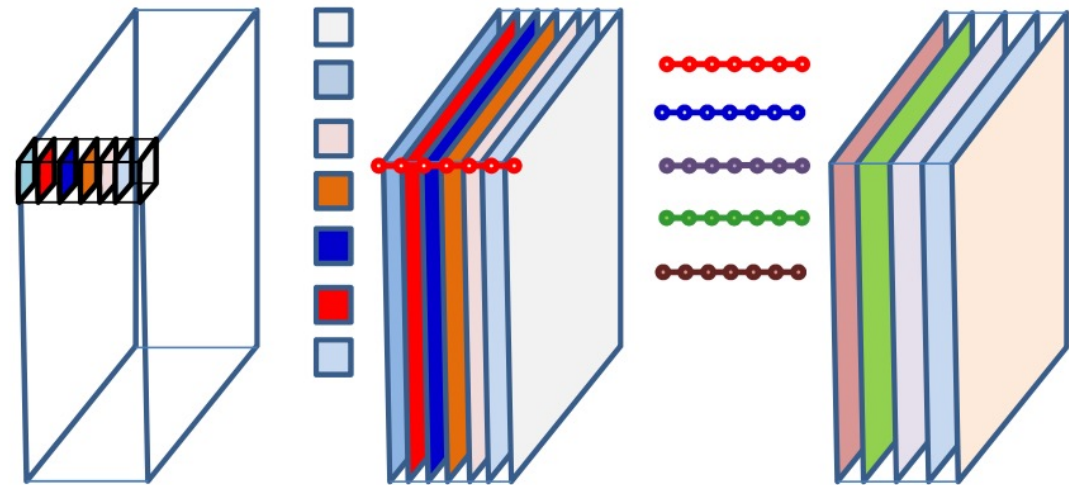
Conventional vs. depth-wise convolution

Conventional



- M input channels, N output channels:
- N independent $M \times K \times K$ **3D** filters, which span all M input channels
- Each filter produces one output channel
- Total NMK^2 parameters

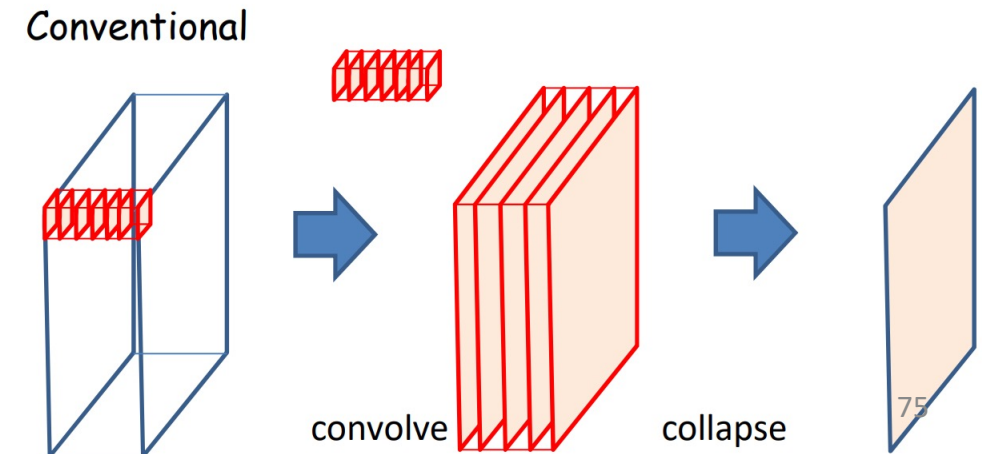
Depth-wise



- M input channels, N output channels in 2 stages:
- Stage 1:
 - M independent $K \times K$ **2D** filters, one per input channel
 - Each filter applies to only one input channel
 - No. of output channels = no. of input channels
- Stage 2:
 - N $M \times 1 \times 1$ 1D filters
 - Each applies to *one* 2D location across all M input channels
- Total $NM + MK^2$ parameters

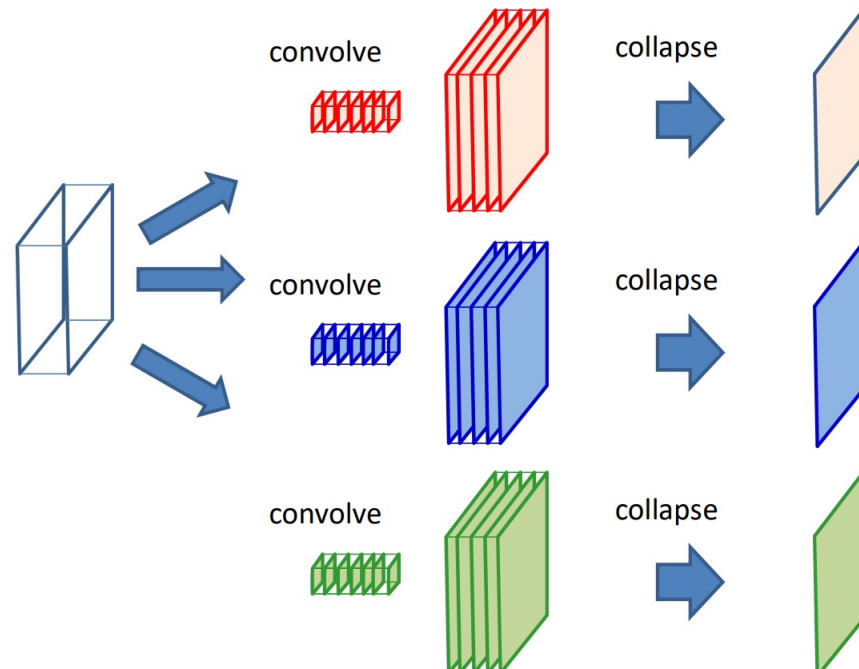
Depth-wise convolutions

- Alternate view of **conventional convolution**:
 - Each channel of each filter scans its corresponding map to produce a convolved map
 - N input channels will require a filter with N channels
 - The N convolved maps are added together to produce the final output map (or channel) for that filter



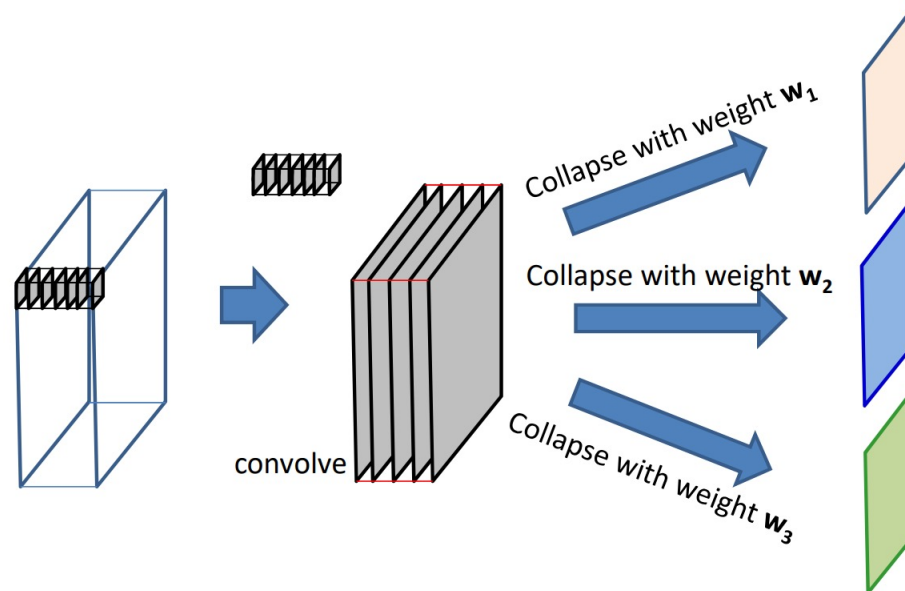
Conventional convolutions

- This is done separately for each of the M filters producing M output maps (channels)



Depth-wise convolution

- In depth-wise convolution the convolution step is performed only once
- The simple summation is replaced by a weighted sum across channels
 - Different weights (for summation) produce different output channels

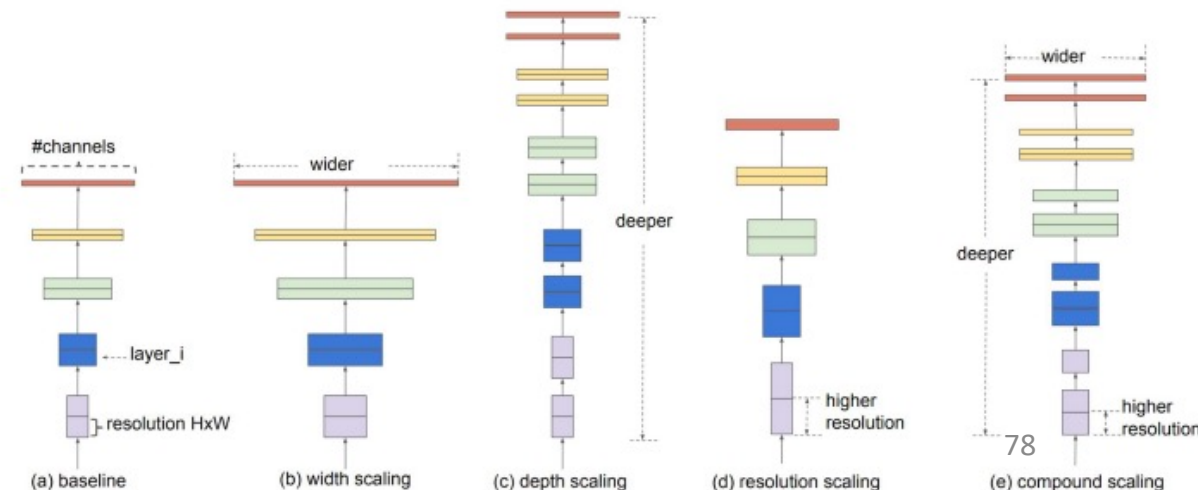
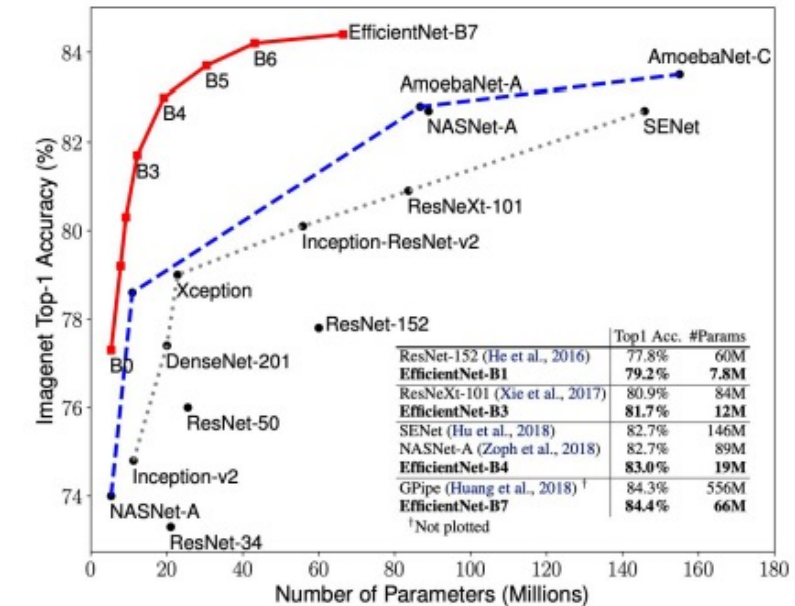


But sometimes smart heuristic is better than NAS ...

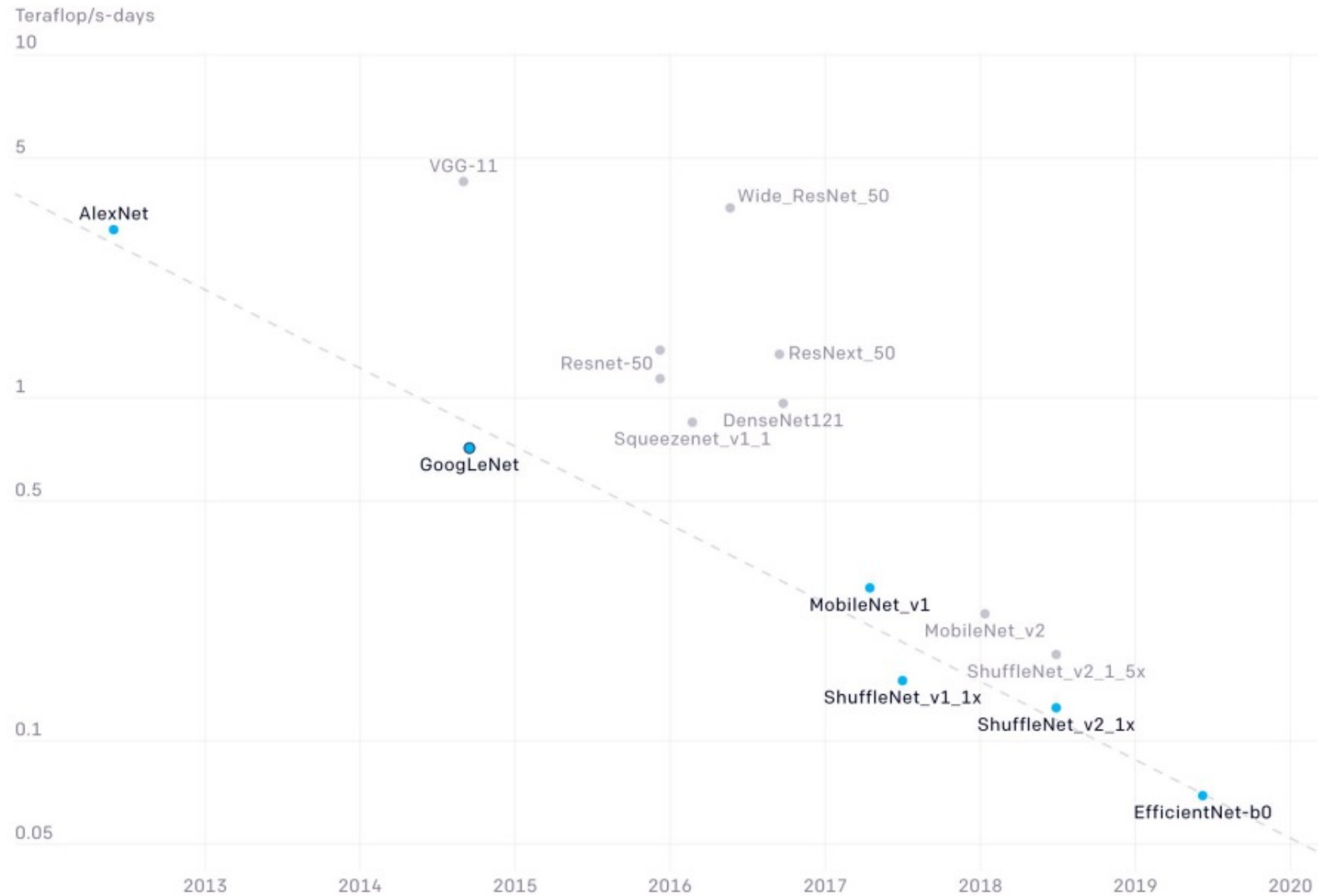
- **EfficientNet**: Smart Compound Scaling
- Increase network capacity by scaling width, depth, and resolution, while balancing accuracy and efficiency.
- Search for optimal set of compound scaling factors given a compute budget (target memory & flops).
- Scale up using smart heuristic rules

$$\begin{aligned} \text{depth: } d &= \alpha^\phi \\ \text{width: } w &= \beta^\phi \\ \text{resolution: } r &= \gamma^\phi \\ \text{s.t. } \alpha \cdot \beta^2 \cdot \gamma^2 &\approx 2 \\ \alpha \geq 1, \beta \geq 1, \gamma \geq 1 \end{aligned}$$

[Tan and Le. 2019]



Efficient networks



<https://openai.com/blog/ai-and-efficiency/>

Summary: CNN Architectures

- Case Studies
 - AlexNet
 - VGG
 - GoogLeNet
 - ResNet
- Also....
 - Improvement of ResNet
 - Wide ResNet
 - ResNeXT
 - Squeeze-and-Excitation Network
 - DenseNet
 - SqueezeNet, MobileNet, EfficientNet

Summary: CNN Architectures

- VGG, GoogLeNet, ResNet all in wide use, available in model zoos
- ResNet current best CNN default
- Trend towards extremely deep CNNs
- Significant research centers around design of layer / skip connections and improving gradient flow
- Efforts to investigate necessity of depth vs. width and residual connections
- Efficient architectures

Transfer Learning

- “You need a lot of a data if you want to train/use CNNs”
- However, by transfer learning you can use the learned features for a task (using a large amount of data) in another related task (for which you don't have enough data)

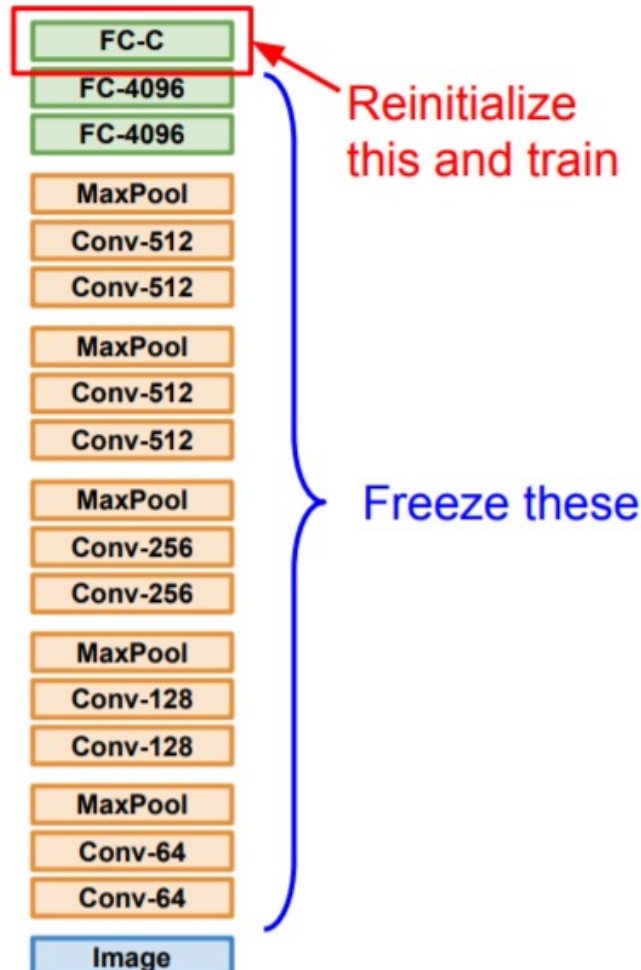
Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014.
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014.

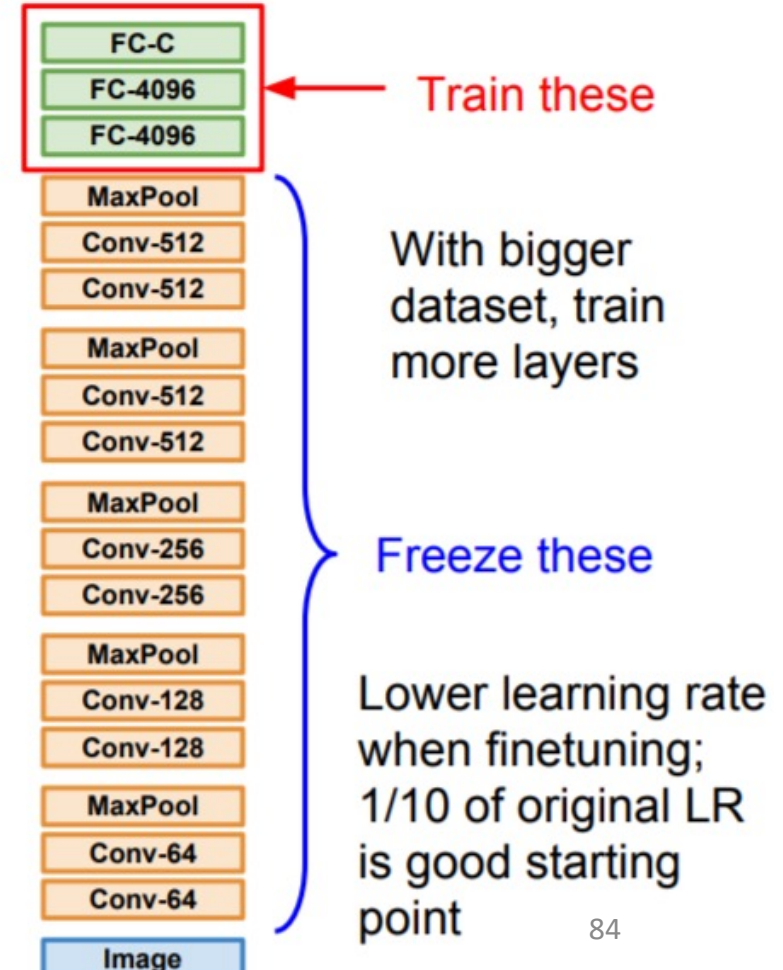
1. Train on Imagenet



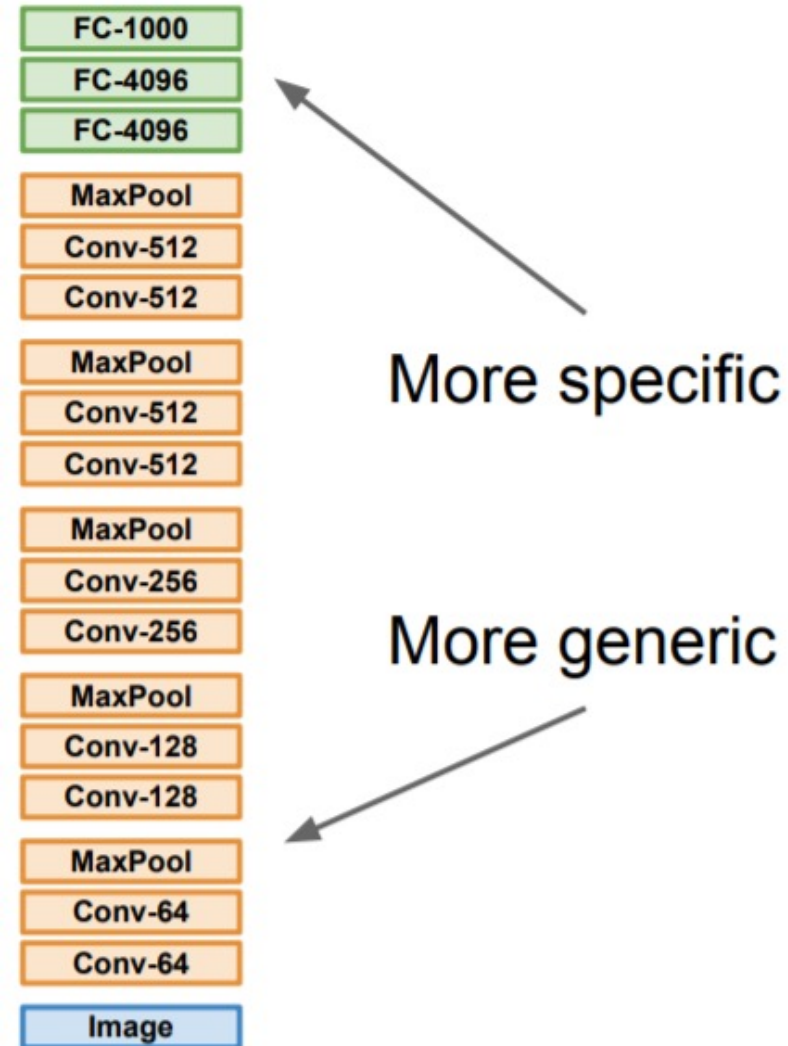
2. Small Dataset (C classes)



3. Bigger dataset

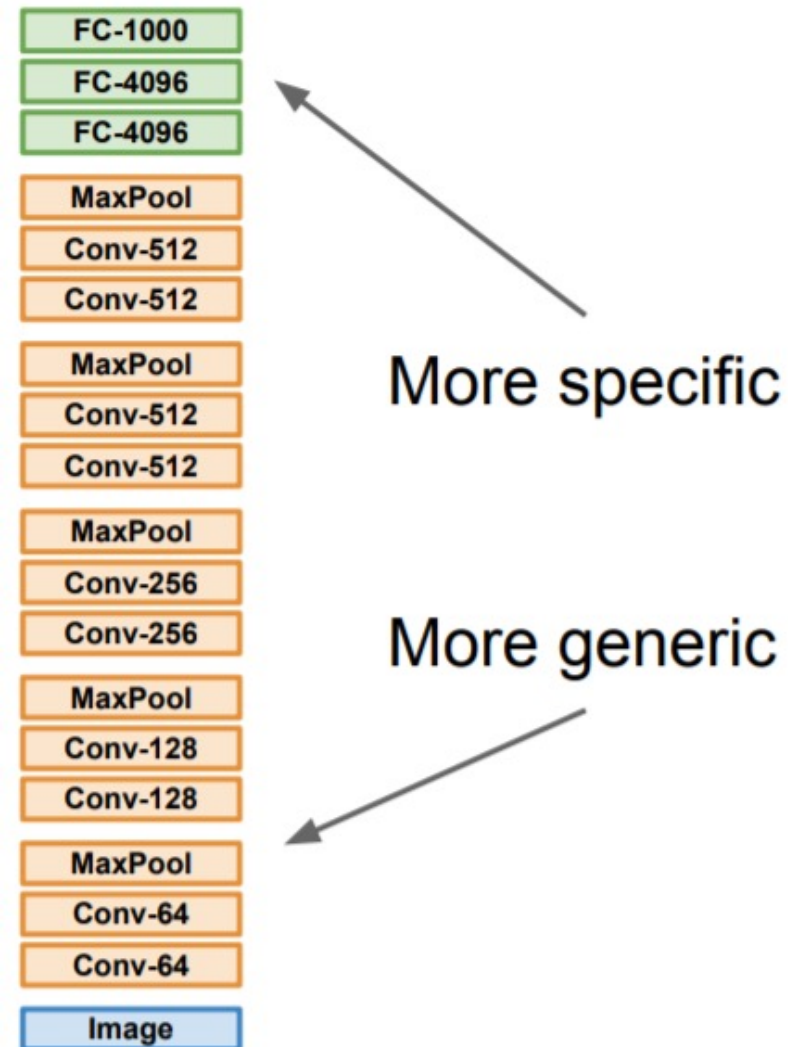


Transfer learning with CNNs



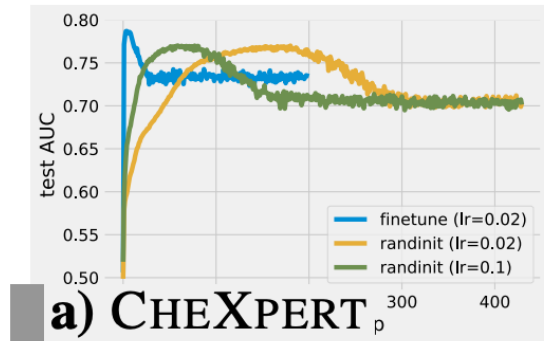
	very similar dataset	very different dataset
very little data		
quite a lot of data		

Transfer learning with CNNs

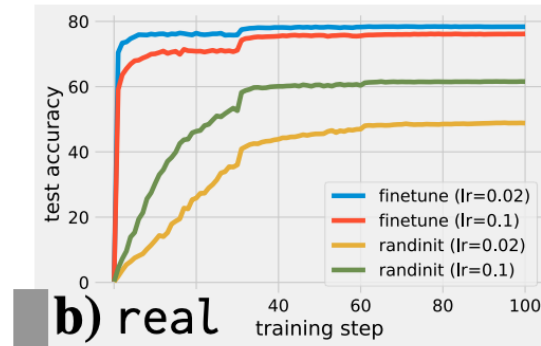


	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	You're in trouble... Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

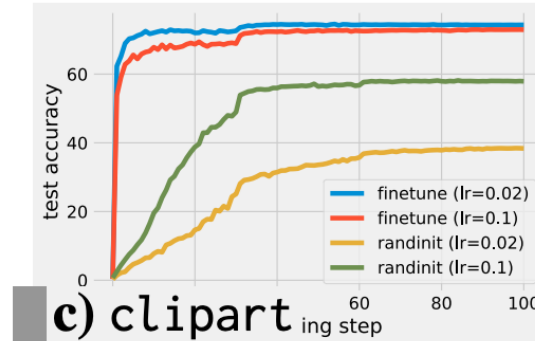
What is being transferred?



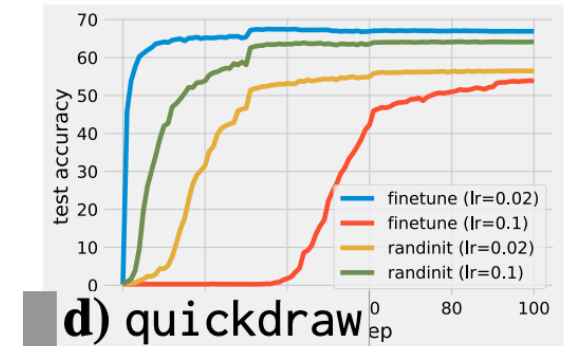
chexpert



airplane



axe

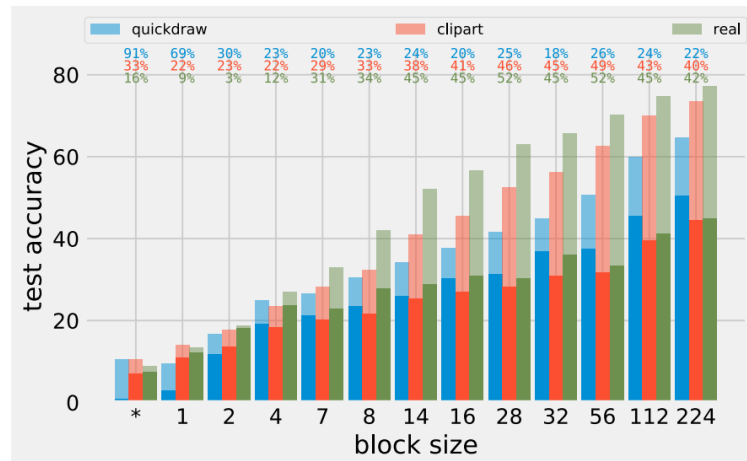


angel

Effects on final performance and optimization speed

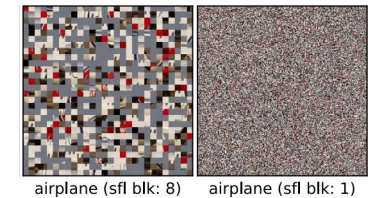
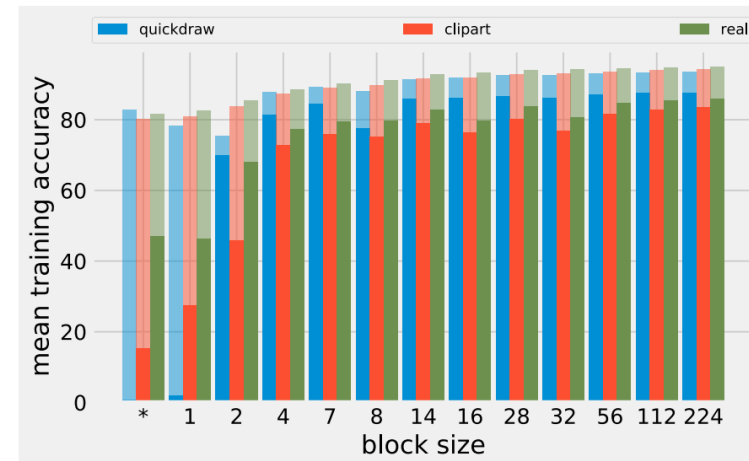
Semi-transparent and solid bar correspond to pre-trained and random weights

The numbers on the top of the left pane show the relative accuracy drop



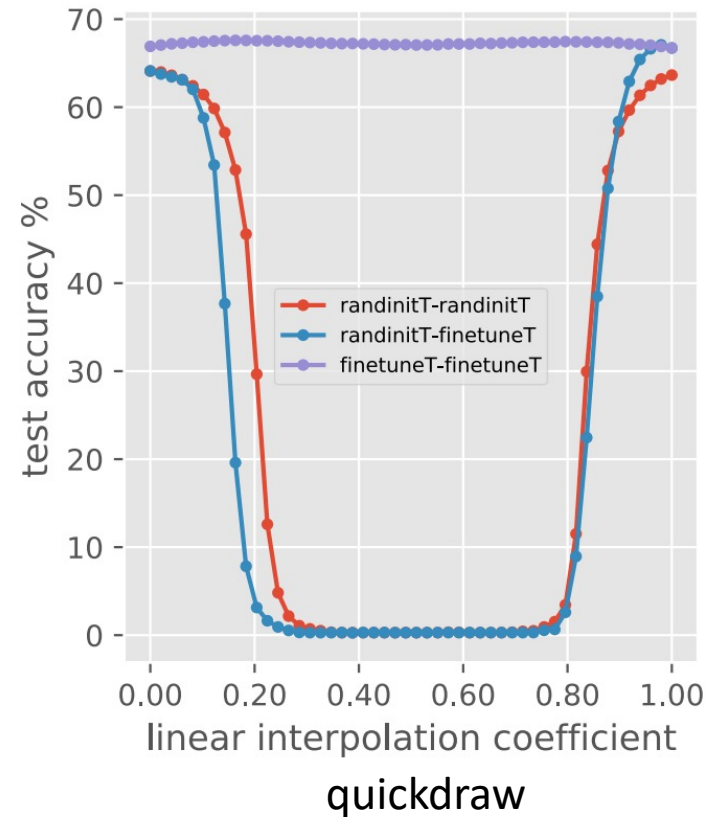
Block size '*' include pixel shuffling also across all RGB channels.

average training accuracy over 100 finetune epochs

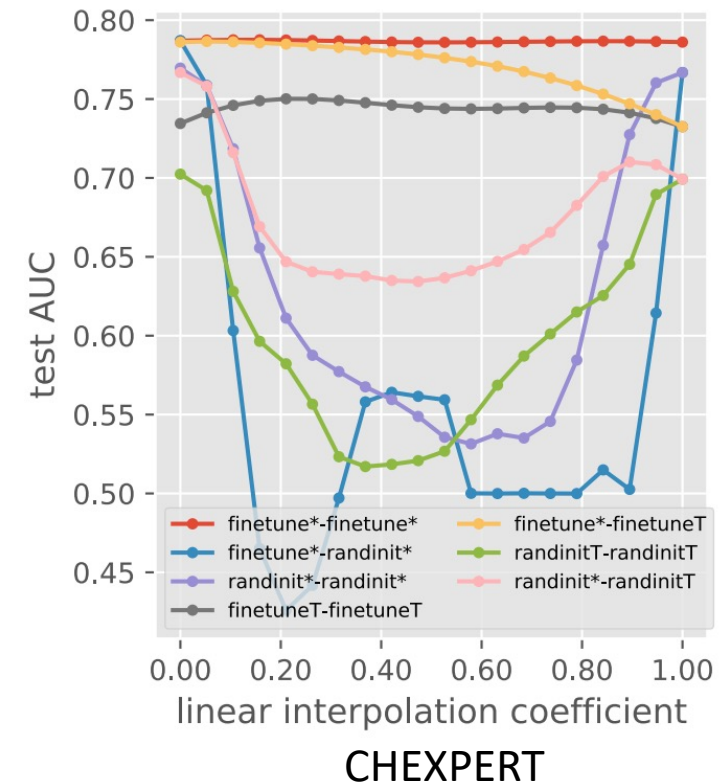


Performance barrier between different solutions

- There is no performance barriers between finetuned models
- Finetuned models reside in the same basin.
- RandInits end up in a different basin, even for the same random seed.



Since there is no overfitting from overtraining on quickdraw, we only show results for final checkpoints



'randinit*' and 'randinitT' means the best and final checkpoint in a RI-T training trajectory. Similarly, 'finetune*' and 'finetuneT' for P-T

Takeaway for your projects and beyond:

- Have some dataset of interest but it has $< \sim 1\text{M}$ images?
 1. Find a very large dataset that has similar data, train a big ConvNet there
 2. Transfer learn to your dataset
- Deep learning frameworks provide a “Model Zoo” of pretrained models so you don’t need to train your own