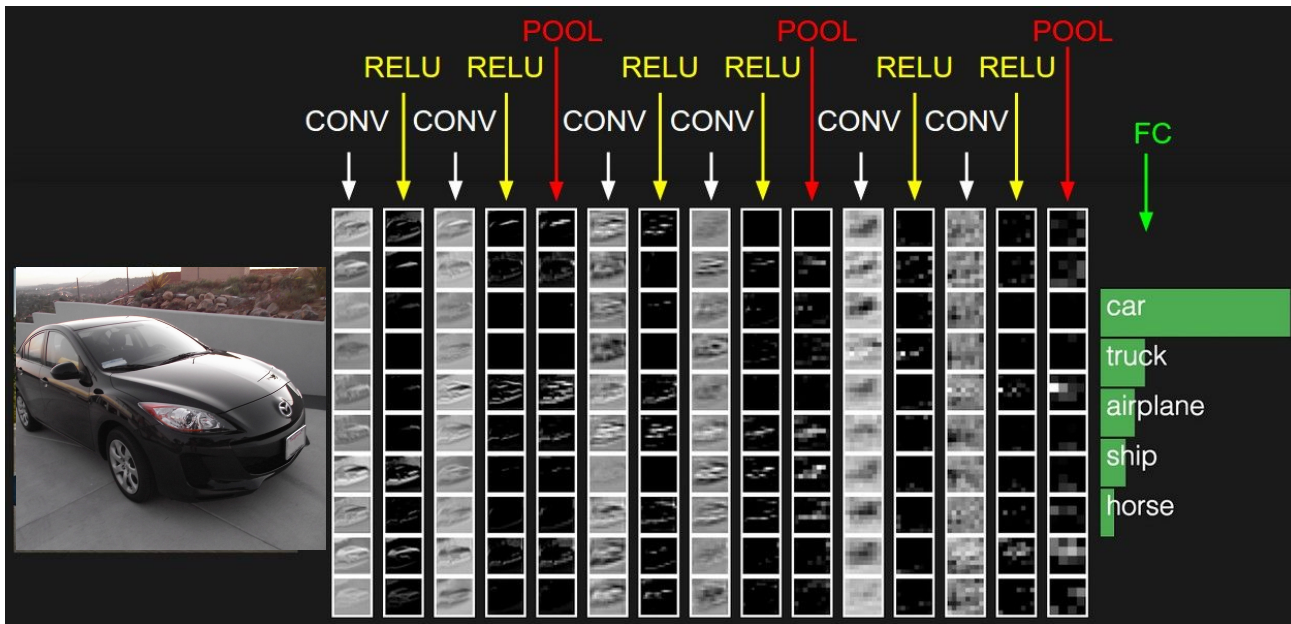


# Lecture 6:

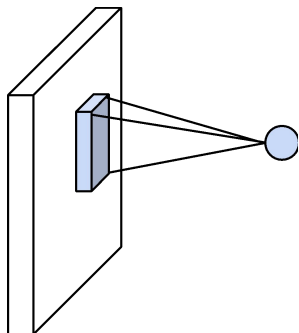
# CNN Architectures

# Recap: Convolutional Neural Networks

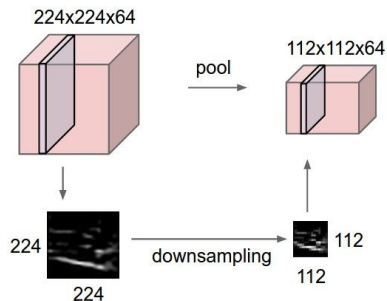


# Components of CNNs

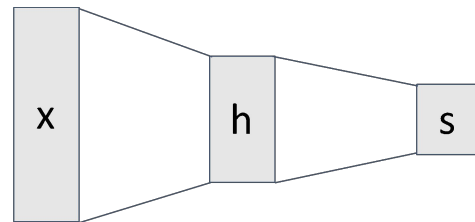
## Convolution Layers



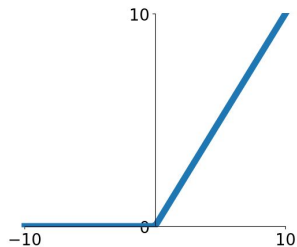
## Pooling Layers



## Fully-Connected Layers



## Activation Function



## Normalization

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

# Batch Normalization

Consider a single layer  $y = Wx$

The following could lead to tough optimization:

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

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

# Batch Normalization

[Ioffe and Szegedy, 2015]

“you want zero-mean unit-variance activations? just make them so.”

consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

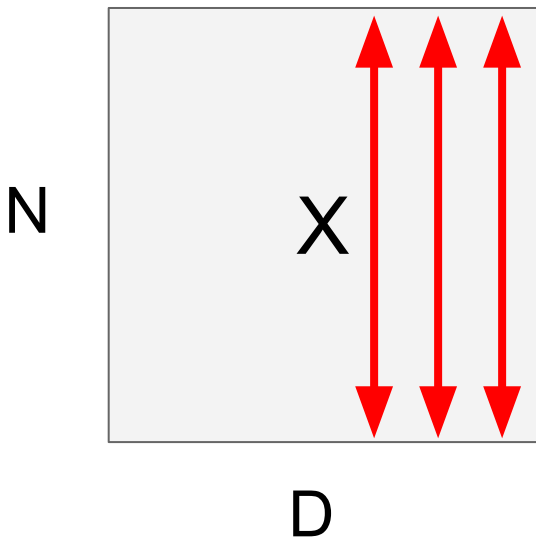
$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

this is a vanilla  
differentiable function...

# Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:**  $x : N \times D$



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

Per-channel mean,  
shape is  $D$

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

Per-channel var,  
shape is  $D$

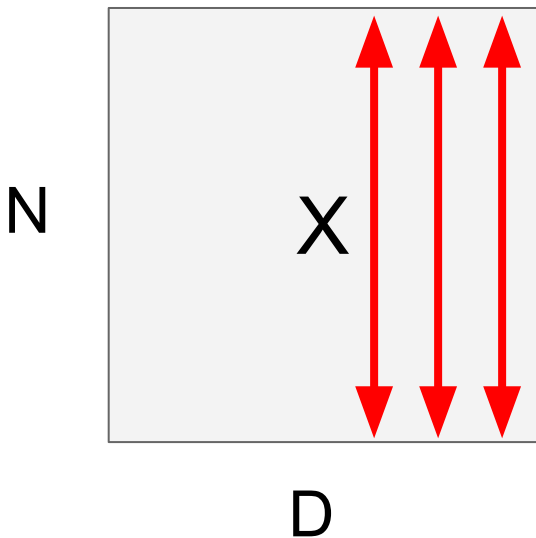
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized  $x$ ,  
Shape is  $N \times D$

# Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:**  $x : N \times D$



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

Per-channel mean,  
shape is  $D$

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

Per-channel var,  
shape is  $D$

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

Normalized  $x$ ,  
Shape is  $N \times D$

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

# Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:**  $x : N \times D$

**Learnable scale and shift parameters:**

$$\gamma, \beta : D$$

Learning  $\gamma = \sigma$ ,  
 $\beta = \mu$ , will recover the  
identity function!

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

Per-channel mean,  
shape is D

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

Per-channel var,  
shape is D

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

Normalized x,  
Shape is N x D

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

Output,  
Shape is N x D



# Batch Normalization: Test-Time

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

**Input:**  $x : N \times D$

**Learnable scale and shift parameters:**

$$\gamma, \beta : D$$

Learning  $\gamma = \sigma$ ,  
 $\beta = \mu$ , will recover the  
identity function!

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j} \quad \text{Per-channel mean, shape is D}$$
$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2 \quad \text{Per-channel var, shape is D}$$

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

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

# Batch Normalization: Test-Time

**Input:**  $x : N \times D$

$$\mu_j = \text{(Running) average of values seen during training}$$

Per-channel mean, shape is D

**Learnable scale and shift parameters:**

$$\gamma, \beta : D$$

$$\sigma_j^2 = \text{(Running) average of values seen during training}$$

Per-channel var, shape is D

During testing batchnorm becomes a linear operator!  
Can be fused with the previous fully-connected or conv layer

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

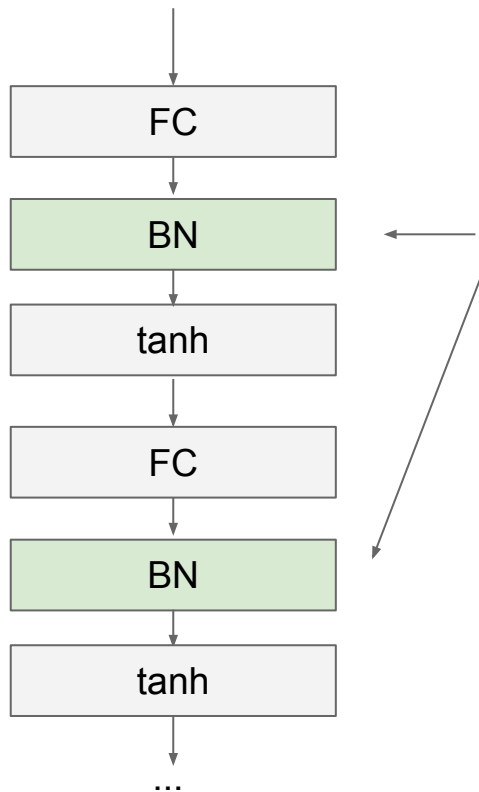
Normalized x, Shape is N x D

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

Output, Shape is N x D

# Batch Normalization

[Ioffe and Szegedy, 2015]

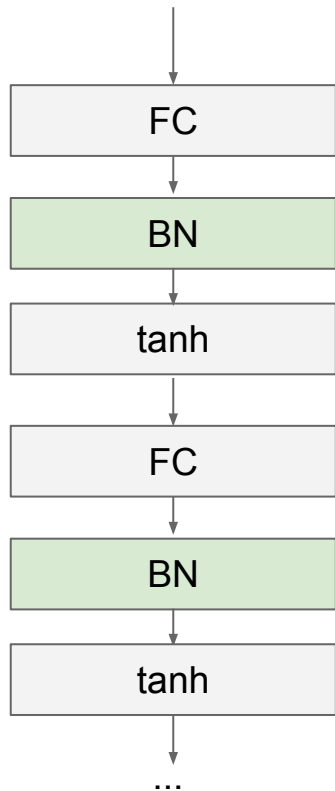


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

# Batch Normalization

[Ioffe and Szegedy, 2015]



- Makes deep networks **much** easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this is a very common source of bugs!

# Batch Normalization for ConvNets

Batch Normalization for  
**fully-connected** networks

$$\mathbf{x}: \mathbf{N} \times \mathbf{D}$$

Normalize



$$\boldsymbol{\mu}, \boldsymbol{\sigma}: 1 \times \mathbf{D}$$

$$\gamma, \beta: 1 \times \mathbf{D}$$

$$\mathbf{y} = \gamma (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \beta$$

Batch Normalization for  
**convolutional** networks  
(Spatial Batchnorm, BatchNorm2D)

$$\mathbf{x}: \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W}$$

Normalize



$$\boldsymbol{\mu}, \boldsymbol{\sigma}: 1 \times \mathbf{C} \times 1 \times 1$$

$$\gamma, \beta: 1 \times \mathbf{C} \times 1 \times 1$$

$$\mathbf{y} = \gamma (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \beta$$

# Layer Normalization

**Batch Normalization** for  
fully-connected networks

$$\mathbf{x}: \mathbf{N} \times \mathbf{D}$$

Normalize



$$\boldsymbol{\mu}, \boldsymbol{\sigma}: 1 \times \mathbf{D}$$

$$\boldsymbol{\gamma}, \boldsymbol{\beta}: 1 \times \mathbf{D}$$

$$\mathbf{y} = \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

**Layer Normalization** for  
fully-connected networks  
Same behavior at train and test!  
Can be used in recurrent networks

$$\mathbf{x}: \mathbf{N} \times \mathbf{D}$$

Normalize



$$\boldsymbol{\mu}, \boldsymbol{\sigma}: \mathbf{N} \times 1$$

$$\boldsymbol{\gamma}, \boldsymbol{\beta}: 1 \times \mathbf{D}$$

$$\mathbf{y} = \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

# Instance Normalization

**Batch Normalization** for  
convolutional networks

$\mathbf{x} : \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W}$

Normalize



$\boldsymbol{\mu}, \boldsymbol{\sigma} : 1 \times \mathbf{C} \times 1 \times 1$

$\boldsymbol{\gamma}, \boldsymbol{\beta} : 1 \times \mathbf{C} \times 1 \times 1$

$$\mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

**Instance Normalization** for  
convolutional networks  
Same behavior at train / test!

$\mathbf{x} : \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W}$

Normalize

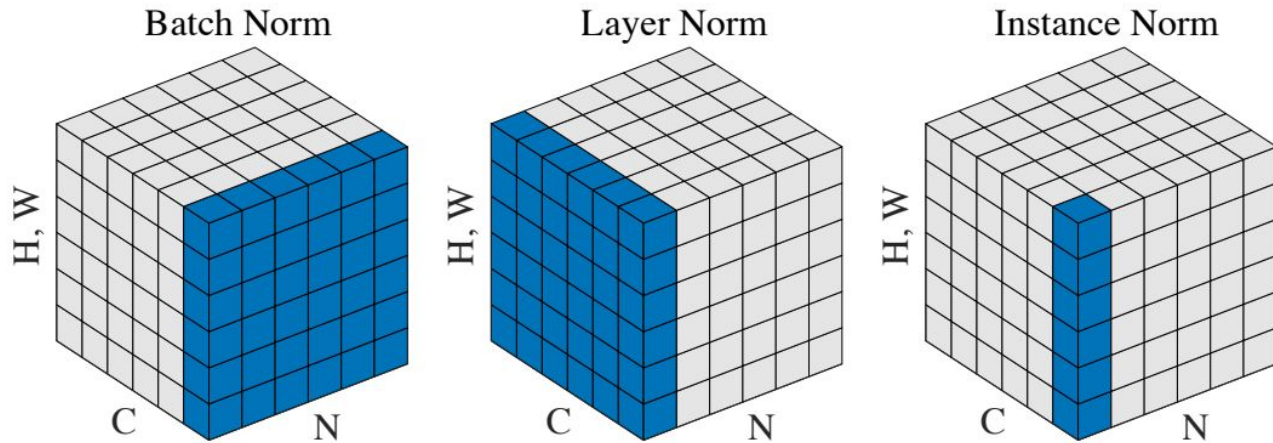


$\boldsymbol{\mu}, \boldsymbol{\sigma} : \mathbf{N} \times \mathbf{C} \times 1 \times 1$

$\boldsymbol{\gamma}, \boldsymbol{\beta} : 1 \times \mathbf{C} \times 1 \times 1$

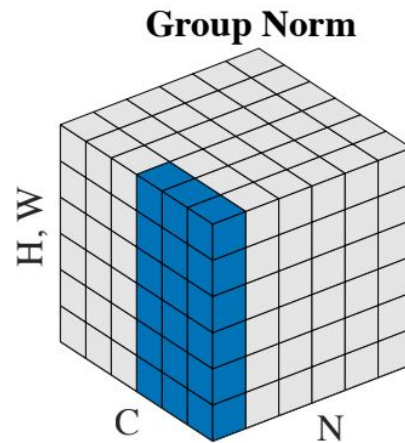
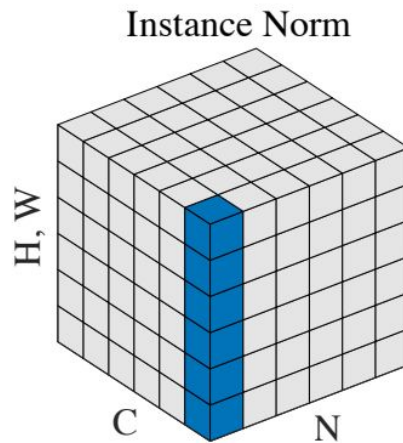
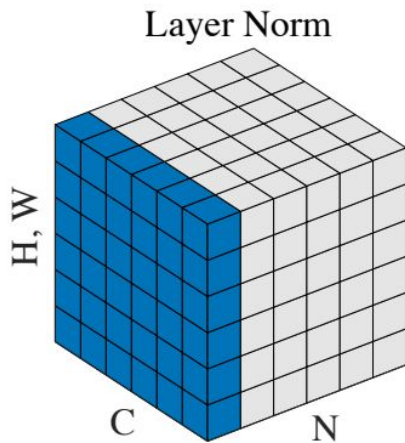
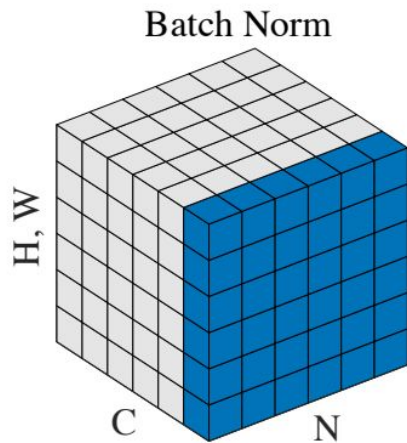
$$\mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

# Comparison of Normalization Layers



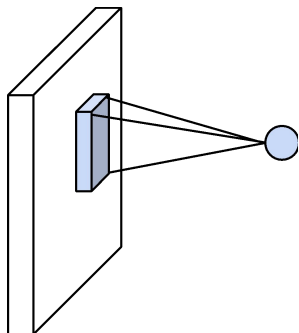


# Group Normalization

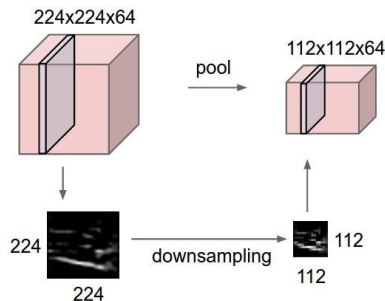


# Components of CNNs

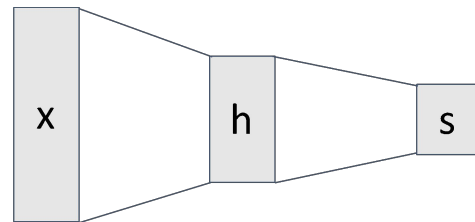
## Convolution Layers



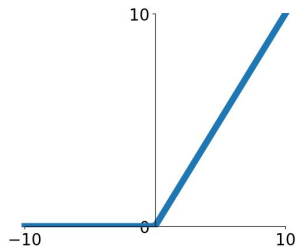
## Pooling Layers



## Fully-Connected Layers



## Activation Function

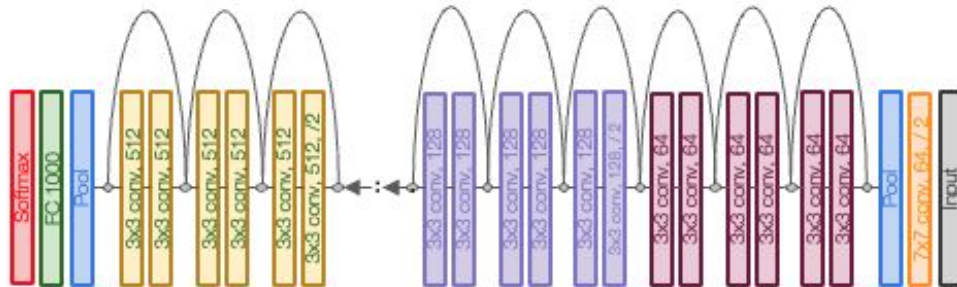
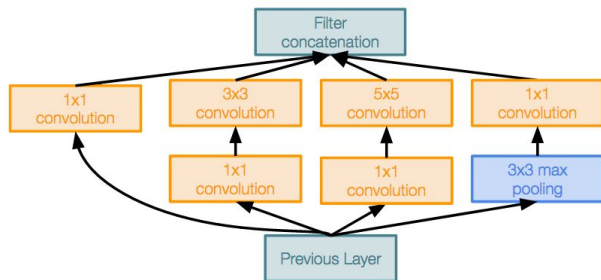
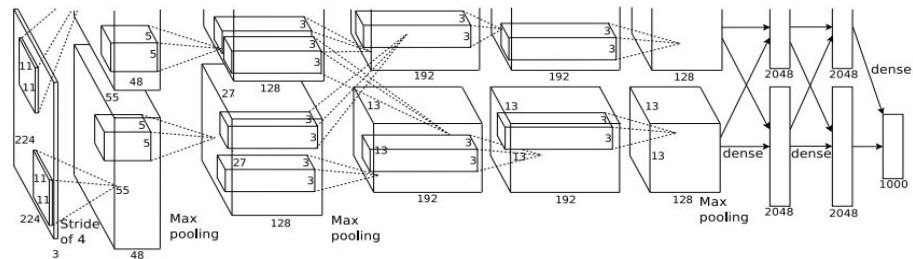
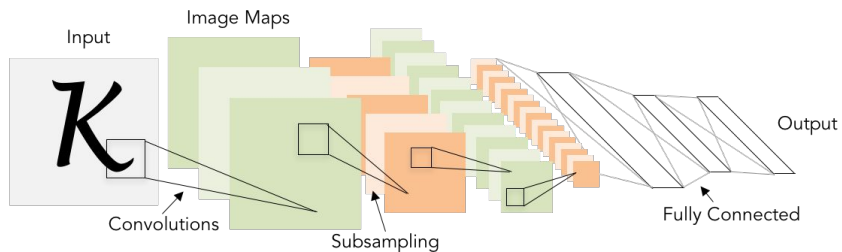


## Normalization

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

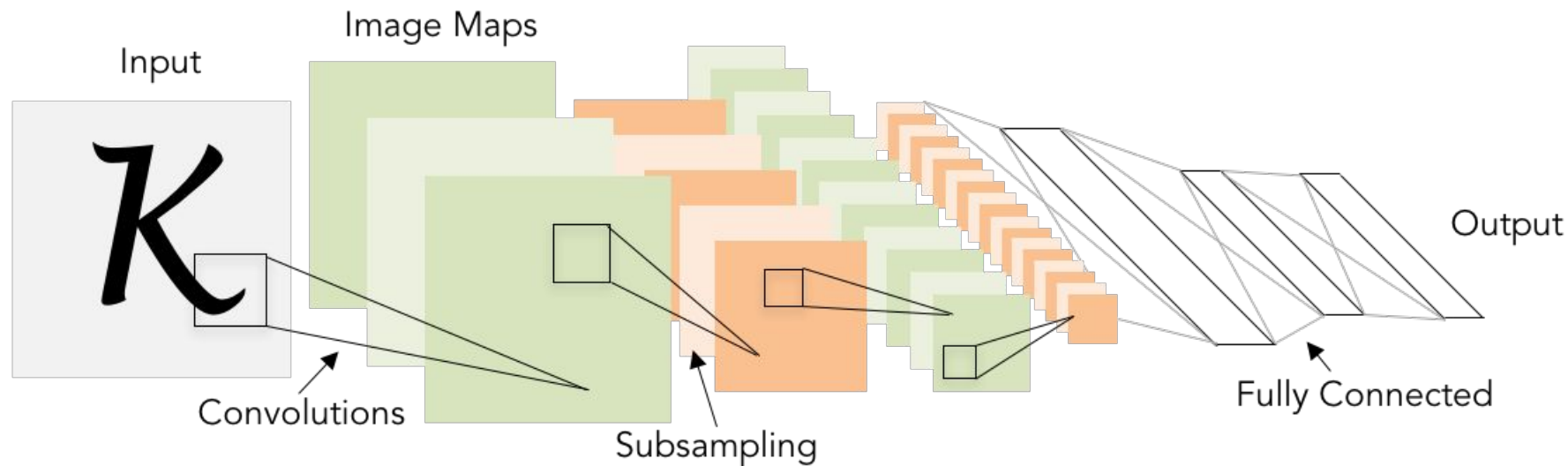
**Question:** How should we put them together?

# Today: CNN Architectures



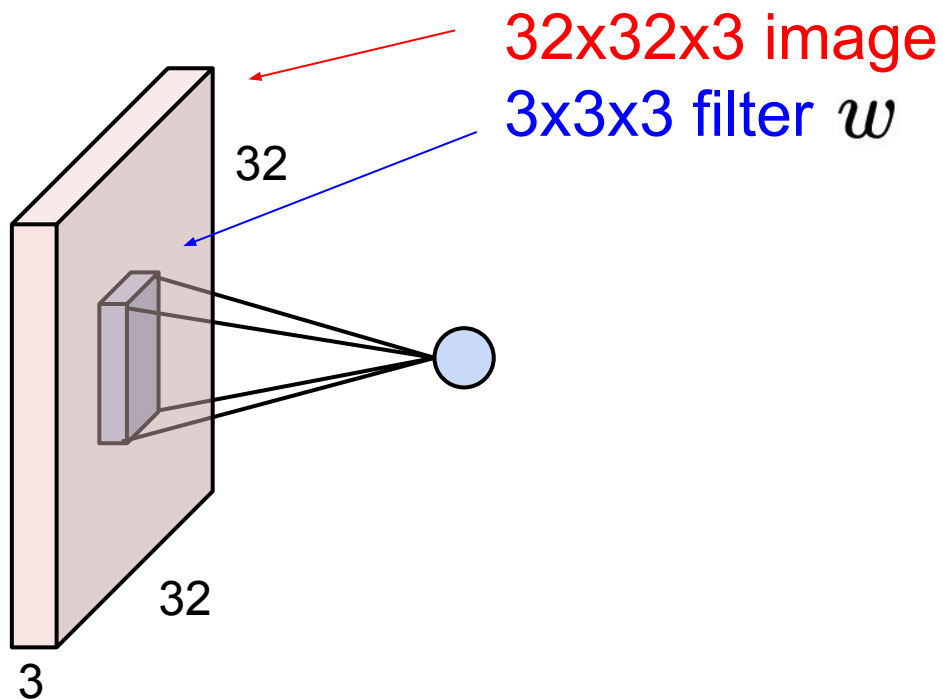
# Review: LeNet-5

[LeCun et al., 1998]



Conv filters were 5x5, applied at stride 1  
Subsampling (Pooling) layers were 2x2 applied at stride 2  
i.e. architecture is [CONV-POOL-CONV-POOL-FC-FC]

# Review: Convolution



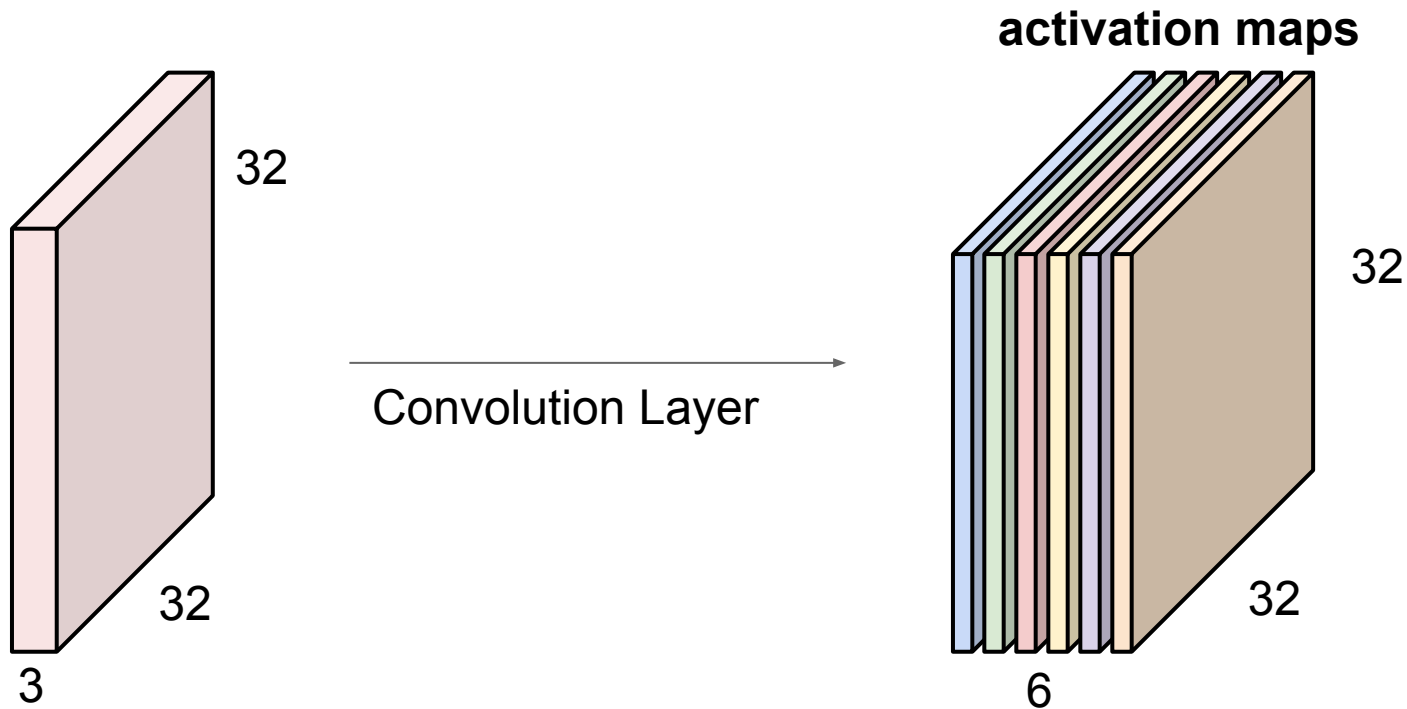
0	0	0	0	0	0				
0									
0									
0									
0									

**Stride:**  
Downsample  
output activations

0	0	0	0	0	0	0			
0									
0									
0									
0									

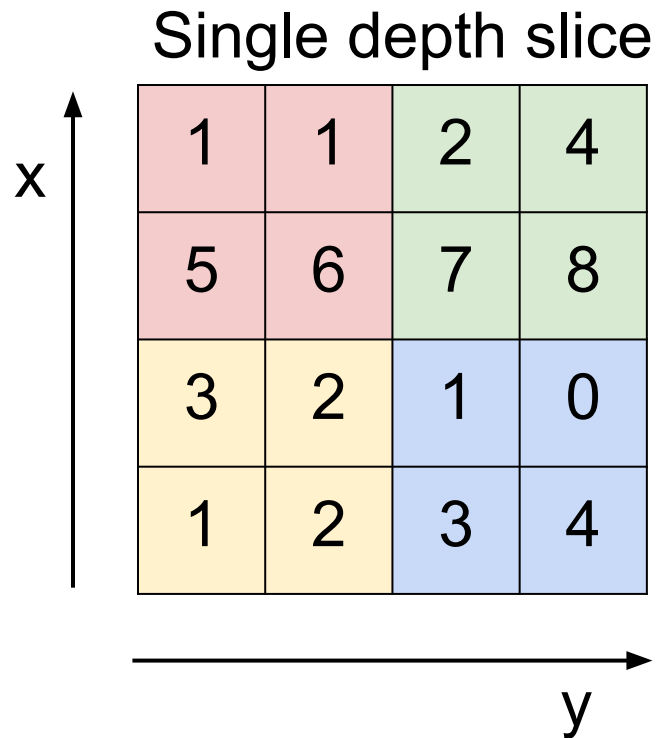
**Padding:**  
Preserve  
input spatial  
dimensions in  
output activations

# Review: Convolution

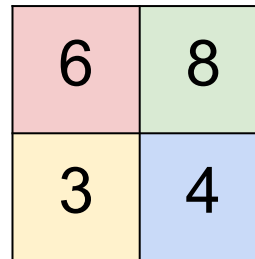


Each conv filter outputs a “slice” in the activation

# Review: Pooling



max pool with 2x2 filters  
and stride 2



# Today: CNN Architectures

## Case Studies

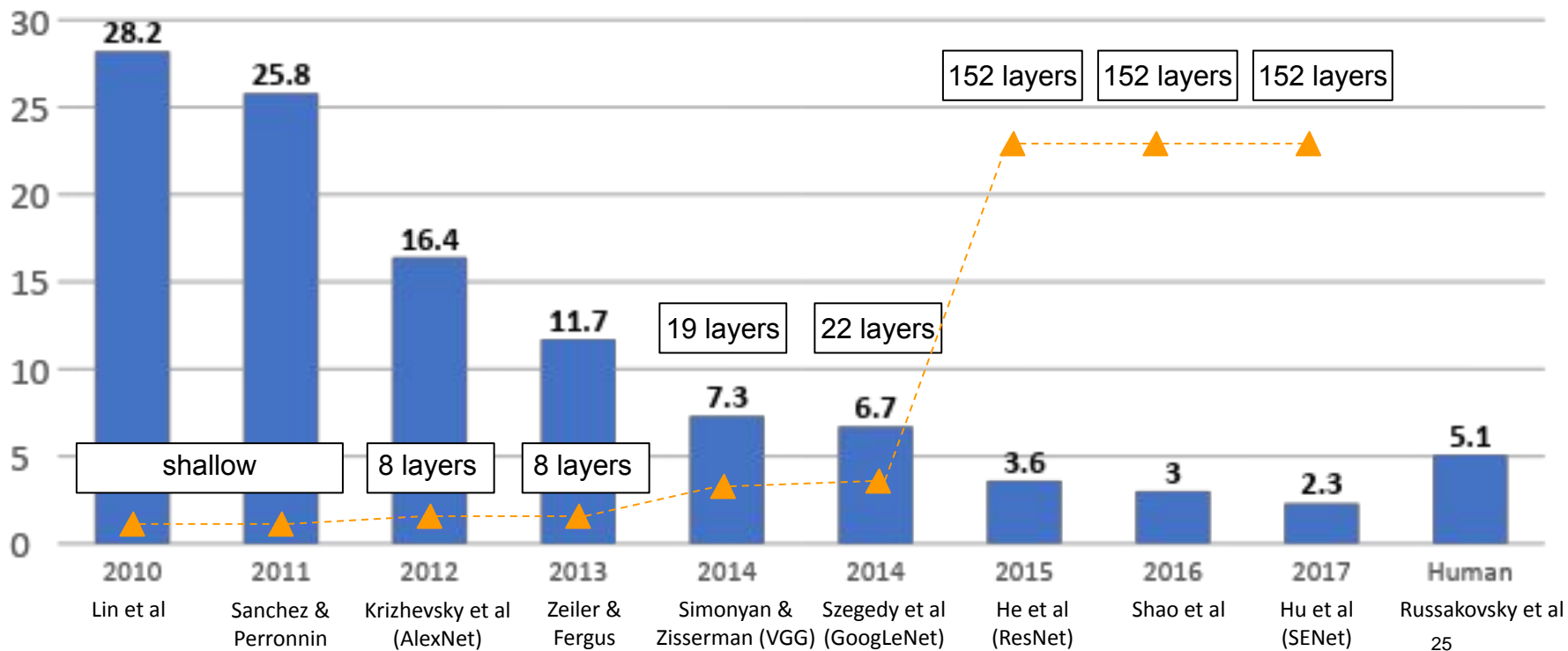
- AlexNet
- VGG
- GoogLeNet
- ResNet

## Also....

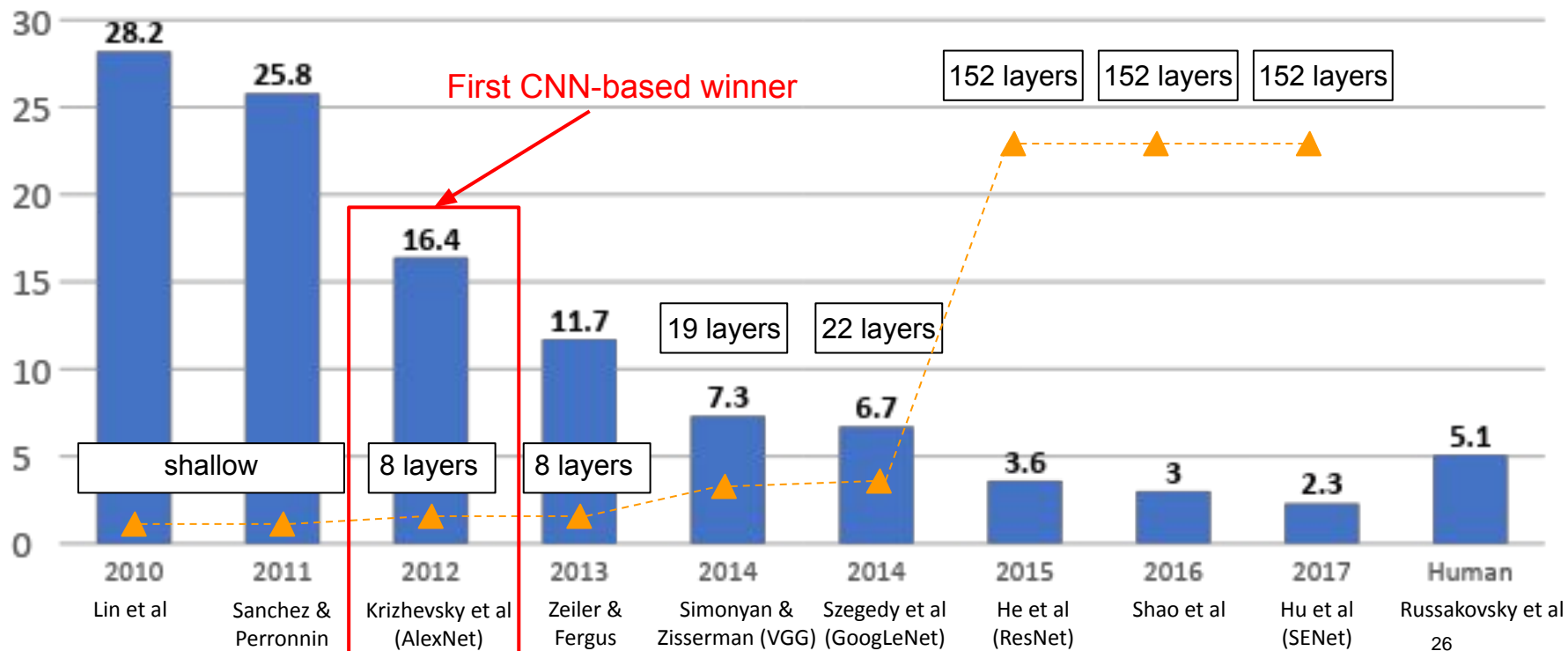
- SENet
- Wide ResNet
- ResNeXT
- DenseNet
- MobileNets
- NASNet
- EfficientNet



# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



# Case Study: AlexNet

[Krizhevsky et al. 2012]

## Architecture:

CONV1

MAX POOL1

NORM1

CONV2

MAX POOL2

NORM2

CONV3

CONV4

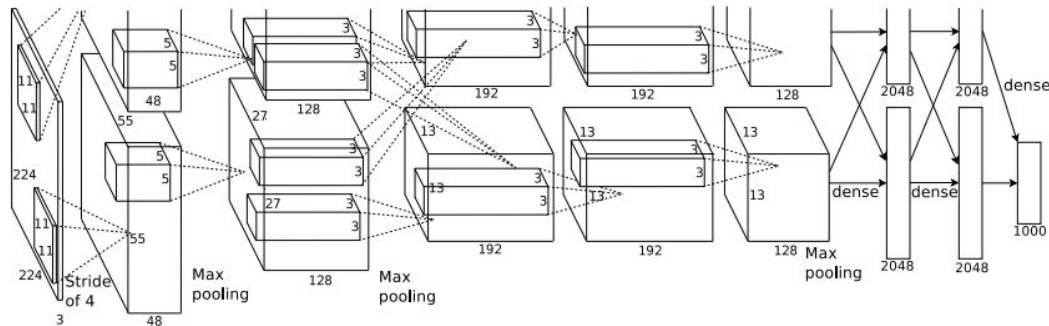
CONV5

Max POOL3

FC6

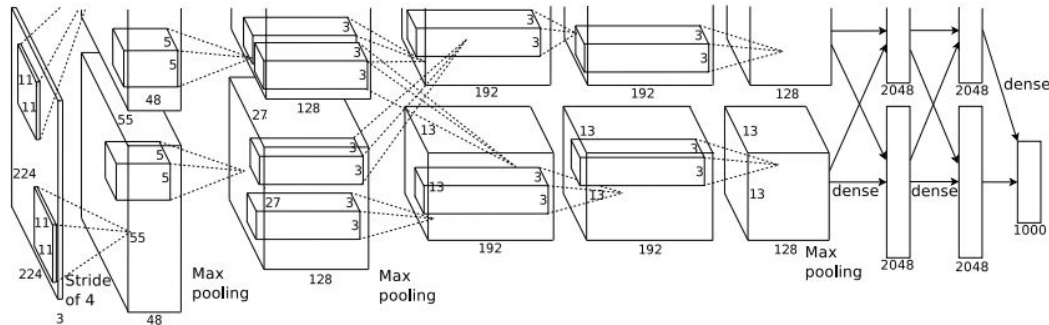
FC7

FC8



# Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

**First layer (CONV1):** 96 11x11 filters applied at stride 4

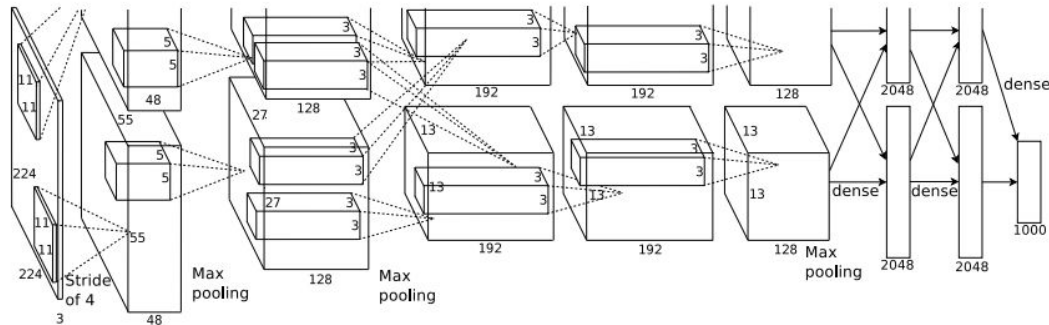
=>

Q: what is the output volume size? Hint:  $(227-11)/4+1 = 55$

$$W' = (W - F + 2P) / S + 1$$

# Case Study: AlexNet

[Krizhevsky et al. 2012]



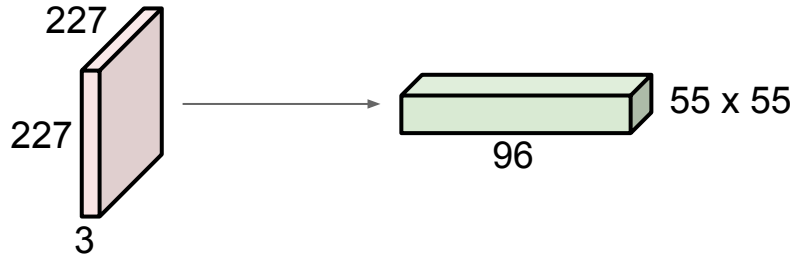
Input: 227x227x3 images

**First layer (CONV1):** 96 11x11 filters applied at stride 4

=>

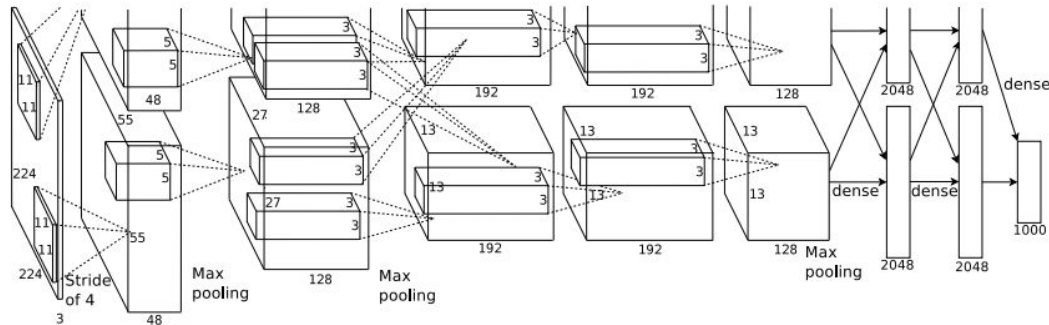
Output volume **[55x55x96]**

$$W' = (W - F + 2P) / S + 1$$



# Case Study: AlexNet

[Krizhevsky et al. 2012]



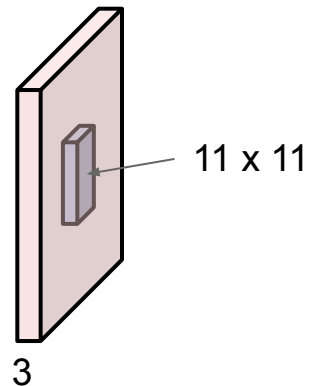
Input: 227x227x3 images

**First layer (CONV1):** 96 11x11 filters applied at stride 4

=>

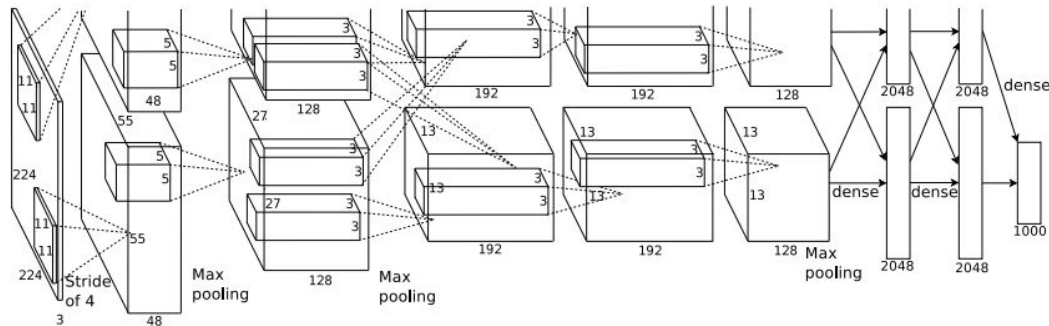
Output volume **[55x55x96]**

Q: What is the total number of parameters in this layer?



# Case Study: AlexNet

[Krizhevsky et al. 2012]



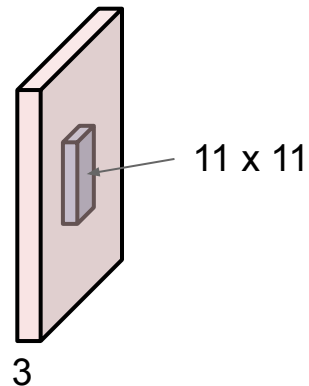
Input: 227x227x3 images

**First layer (CONV1):** 96 11x11 filters applied at stride 4

=>

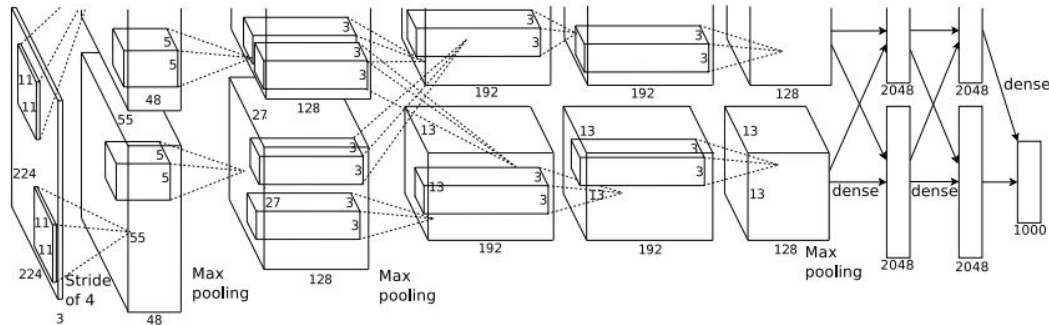
Output volume **[55x55x96]**

Parameters:  $(11 \cdot 11 \cdot 3 + 1) \cdot 96 = \mathbf{35K}$



# Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

After CONV1: 55x55x96

$$W' = (W - F + 2P) / S + 1$$

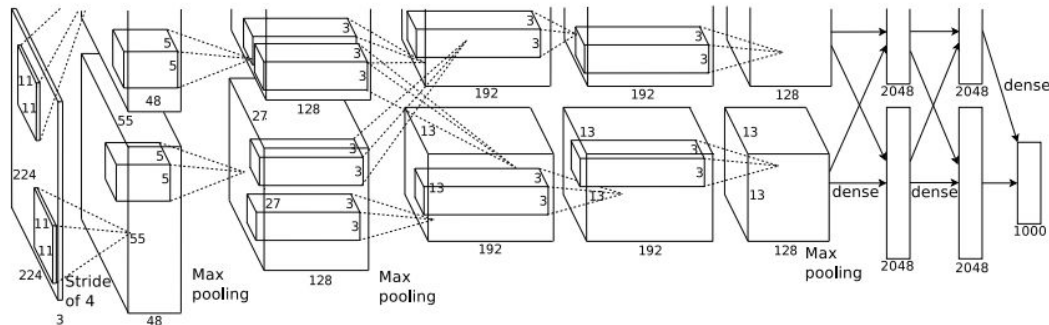
**Second layer (POOL1):** 3x3 filters applied at stride 2

Q: what is the output volume size? Hint:  $(55-3)/2+1 = 27$



# Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

After CONV1: 55x55x96

$$W' = (W - F + 2P) / S + 1$$

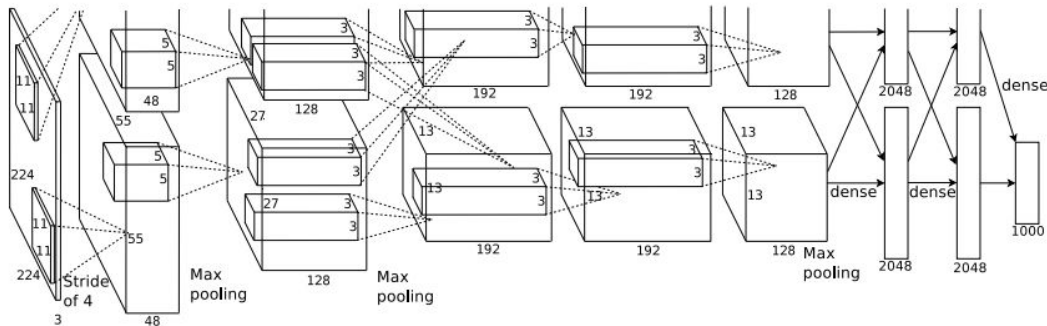
**Second layer (POOL1):** 3x3 filters applied at stride 2

Output volume: 27x27x96

Q: what is the number of parameters in this layer?

# Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

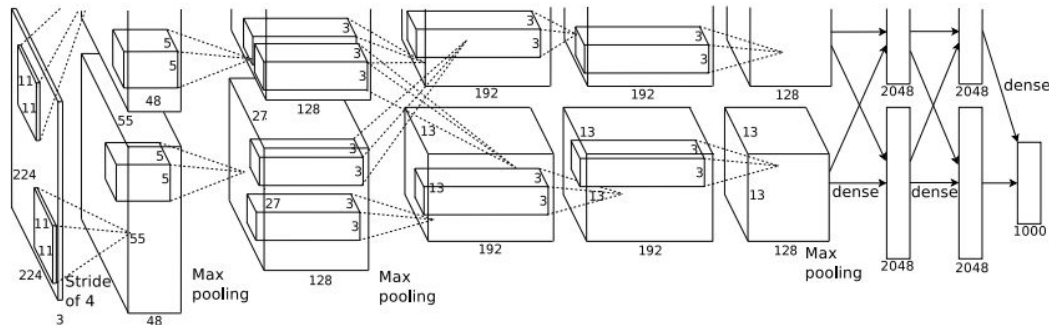
After CONV1: 55x55x96

**Second layer (POOL1):** 3x3 filters applied at stride 2

Output volume: 27x27x96

Parameters: 0!

[Krizhevsky et al. 2012]



After CONV1: 55x55x96

After POOL1: 27x27x96

■ ■ ■

# Case Study: AlexNet

[Krizhevsky et al. 2012]

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

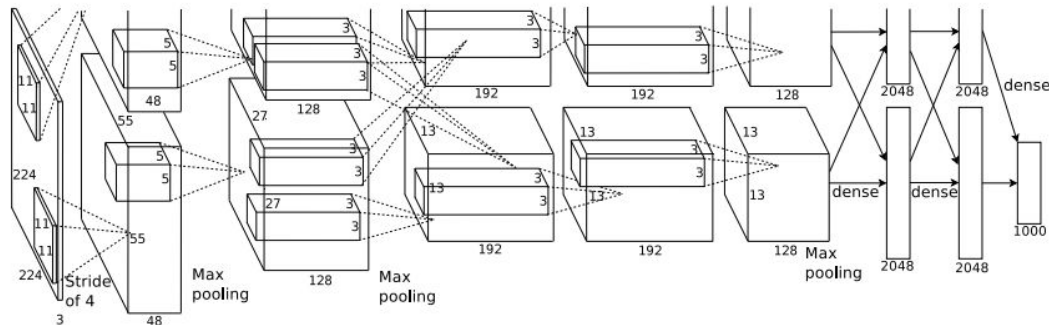
[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

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

[4096] **FC6**: 4096 neurons

[4096] **FC7**: 4096 neurons

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



# Case Study: AlexNet

[Krizhevsky et al. 2012]

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

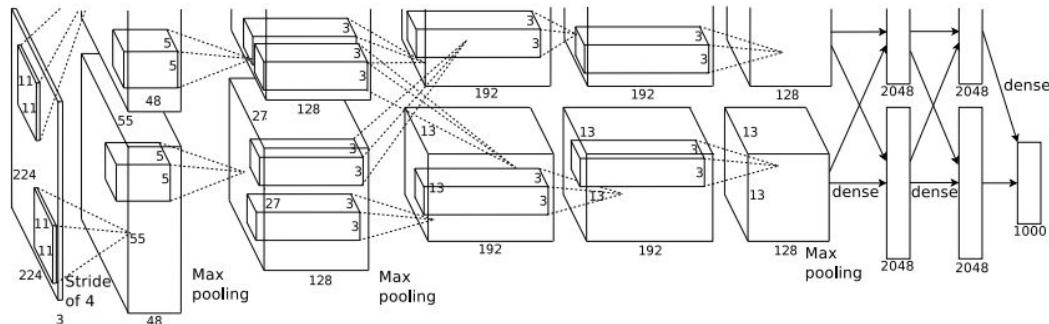
[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

[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 LRN 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

[Krizhevsky et al. 2012]

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

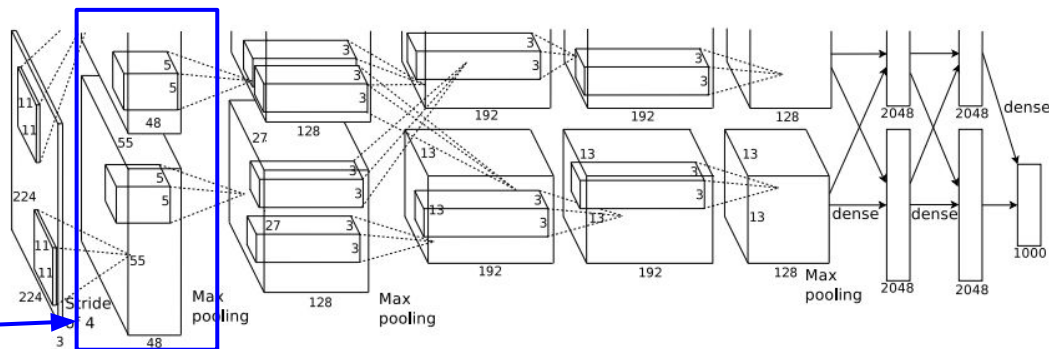
[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

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

[4096] **FC6**: 4096 neurons

[4096] **FC7**: 4096 neurons

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



[55x55x48] x 2

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

[Krizhevsky et al. 2012]

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

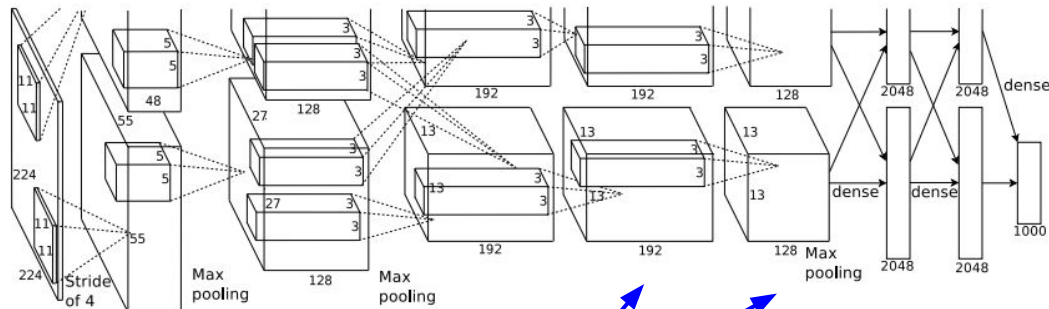
[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

[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

[Krizhevsky et al. 2012]

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

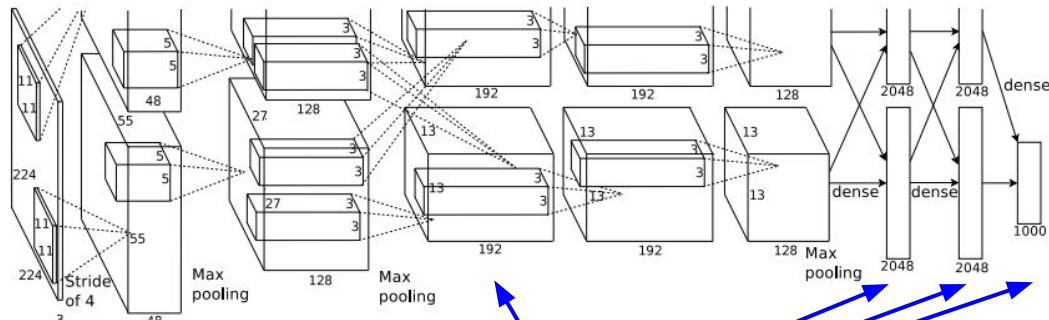
[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

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

[4096] **FC6**: 4096 neurons

[4096] **FC7**: 4096 neurons

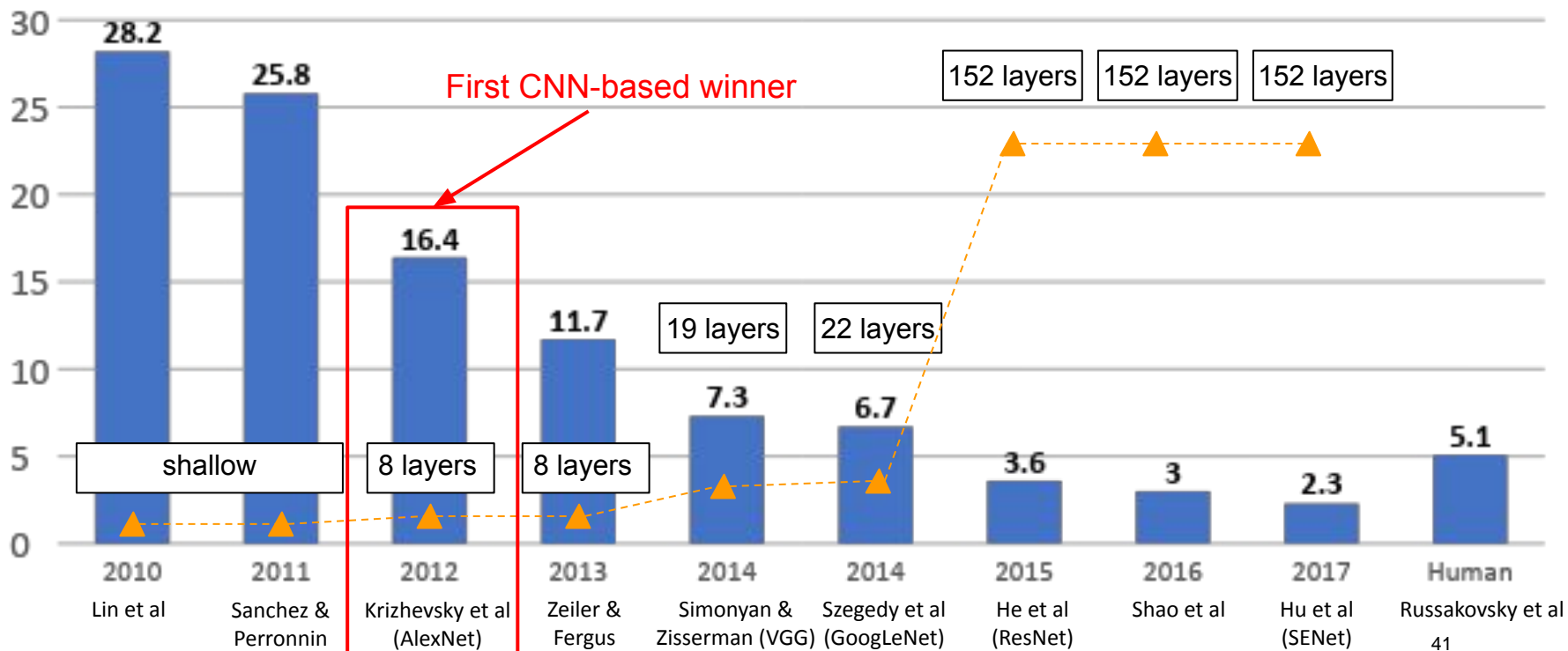
[1000] **FC8**: 1000 neurons (class scores)



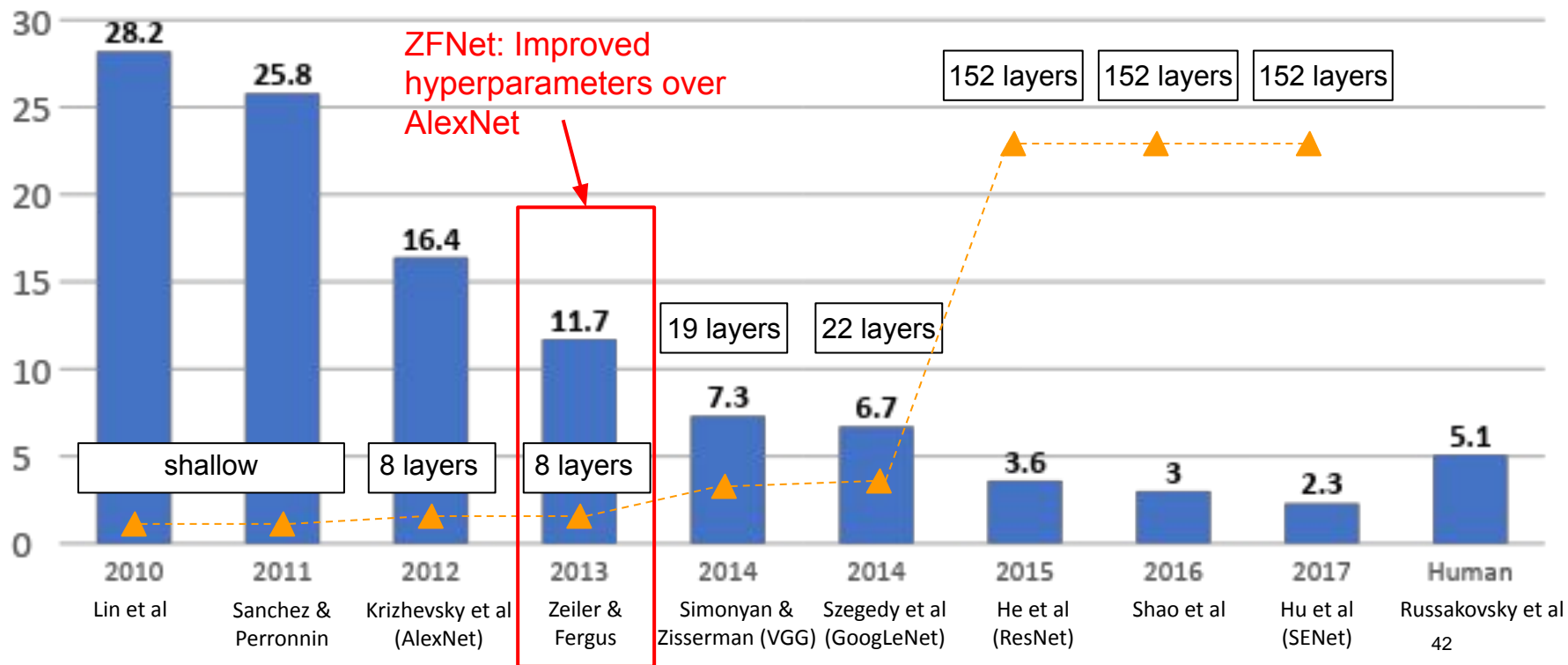
**CONV3, FC6, FC7, FC8:**  
Connections with all feature maps in preceding layer, communication across GPUs



# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

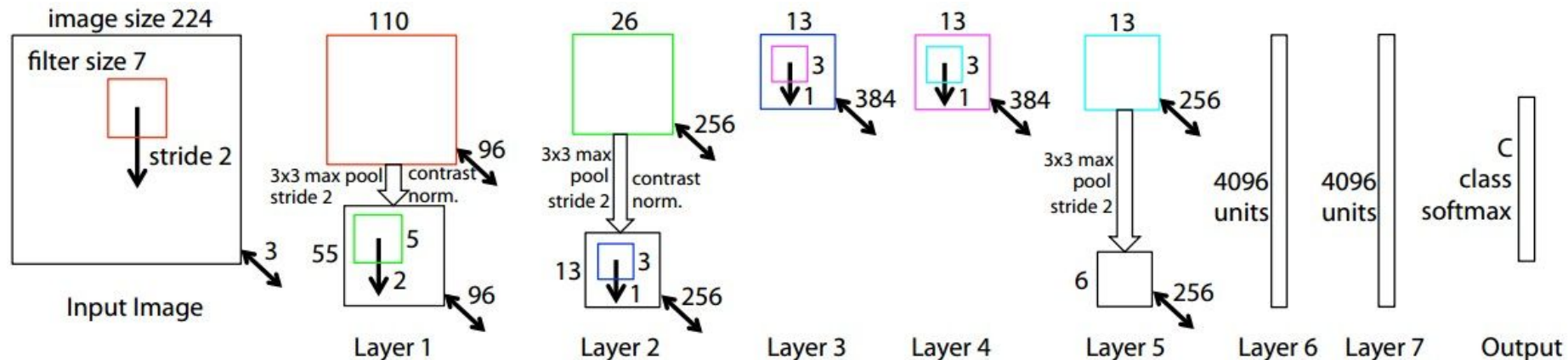


# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



# ZFNet

[Zeiler and Fergus, 2013]



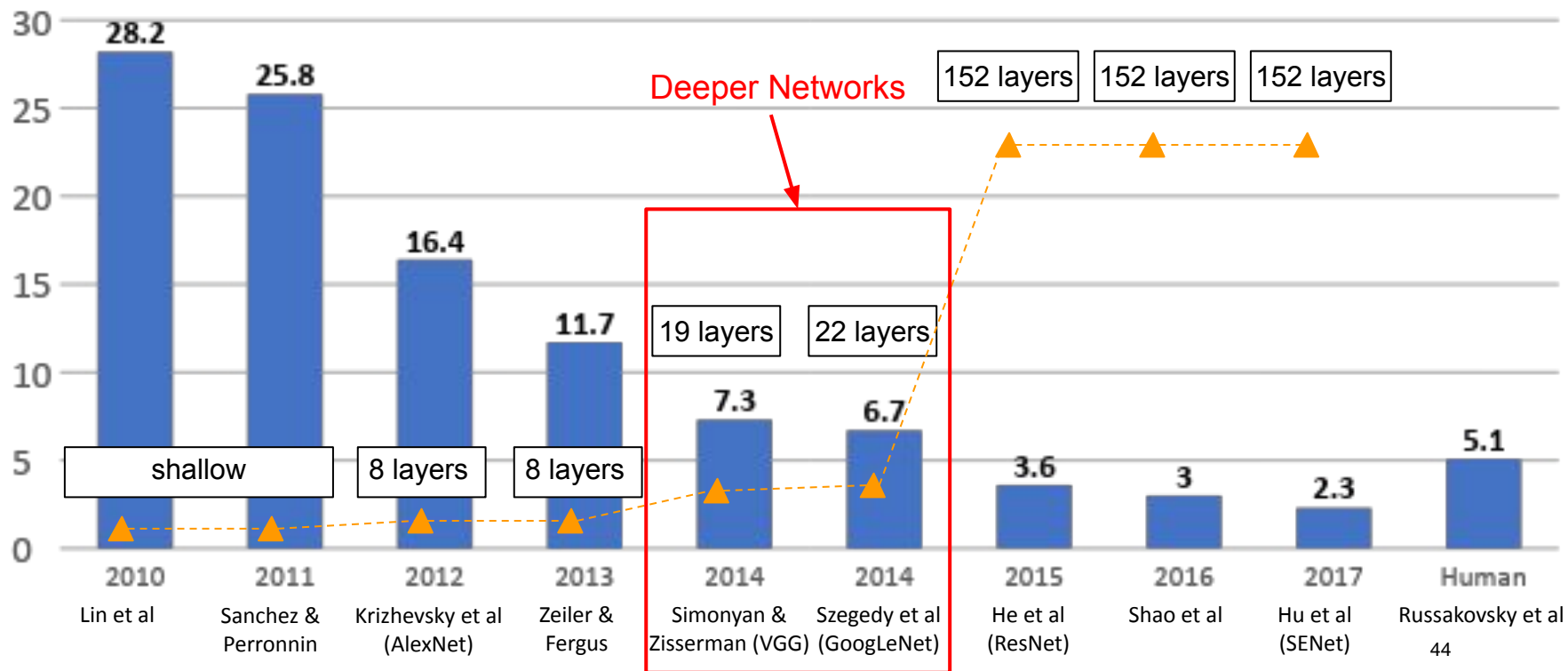
AlexNet but:

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% <sup>43</sup> > 11.7%

# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

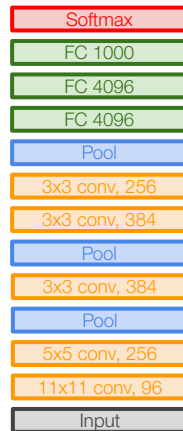
8 layers (AlexNet)

-> 16 - 19 layers (VGG16Net)

Only 3x3 CONV stride 1, pad 1  
and 2x2 MAX POOL stride 2

11.7% top 5 error in ILSVRC'13 (ZFNet)

-> 7.3% top 5 error in ILSVRC'14



AlexNet



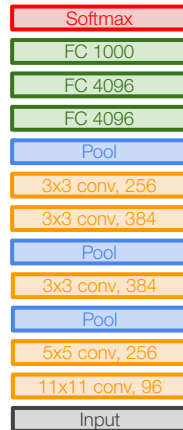
VGG16

VGG19  
45

# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)



AlexNet



VGG16

VGG19  
46

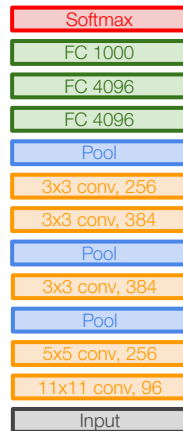
# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

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



AlexNet



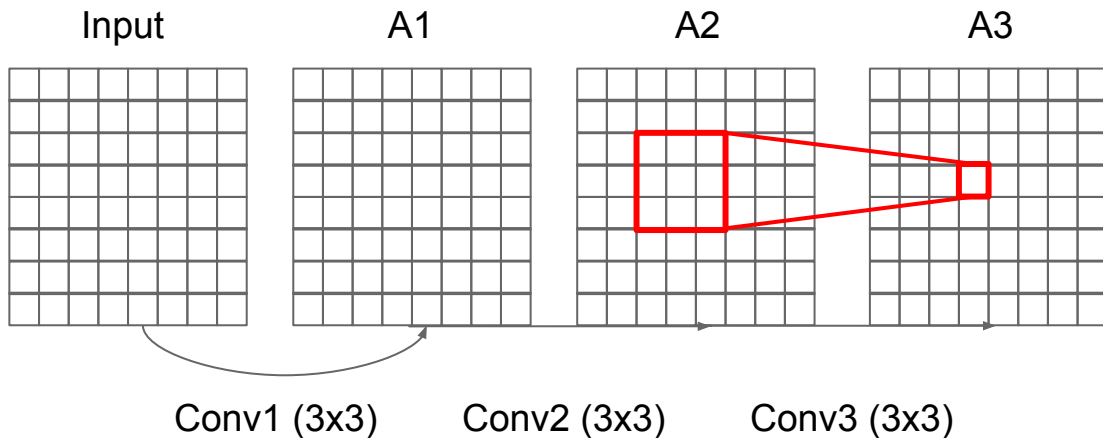
VGG16

VGG19<sub>47</sub>

# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

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

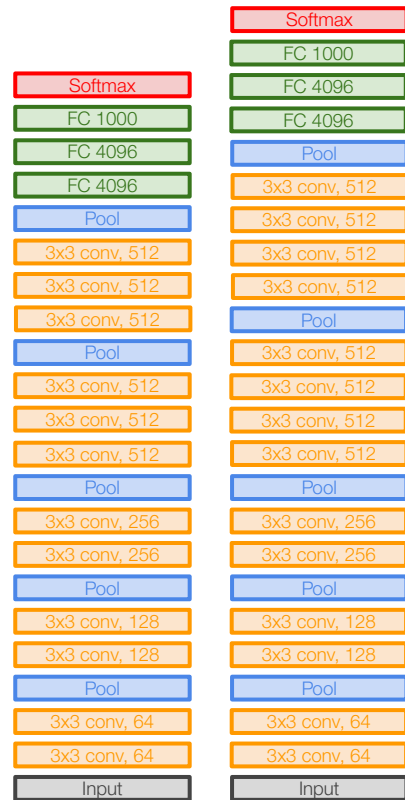
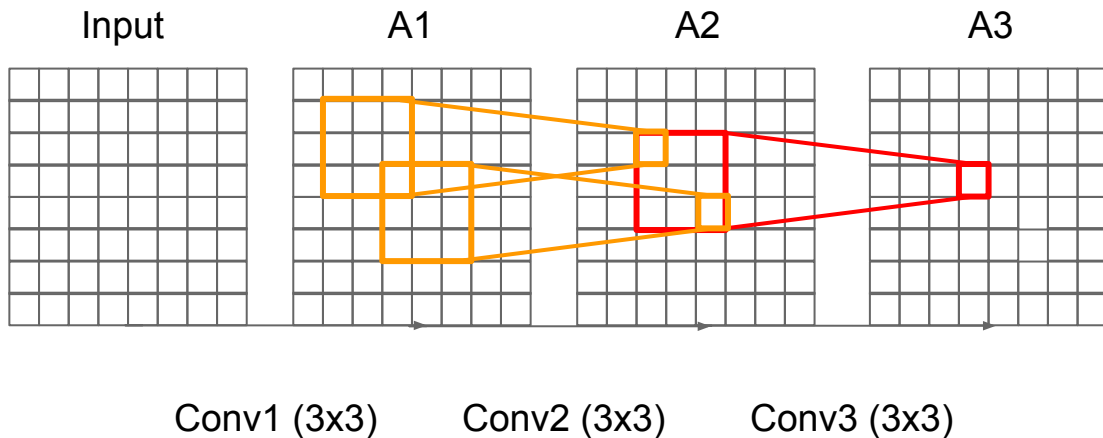




# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

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



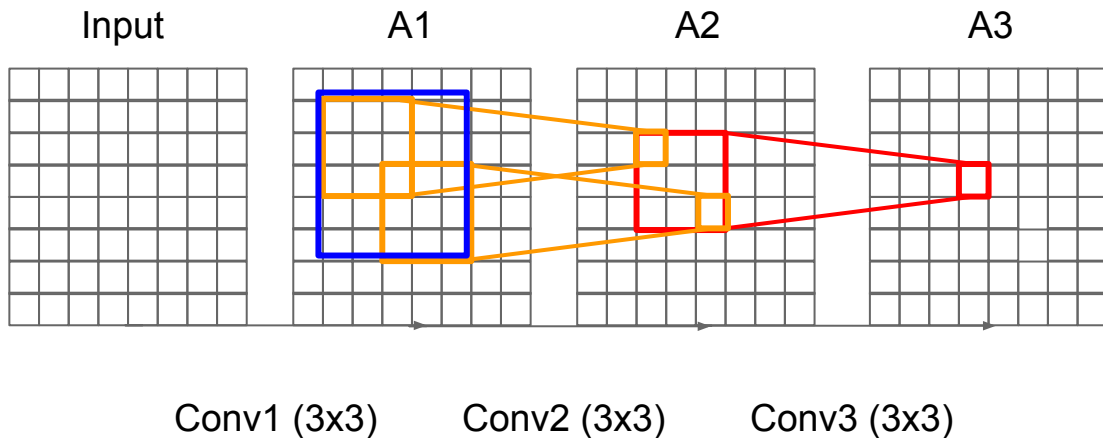
VGG16

VGG19

# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

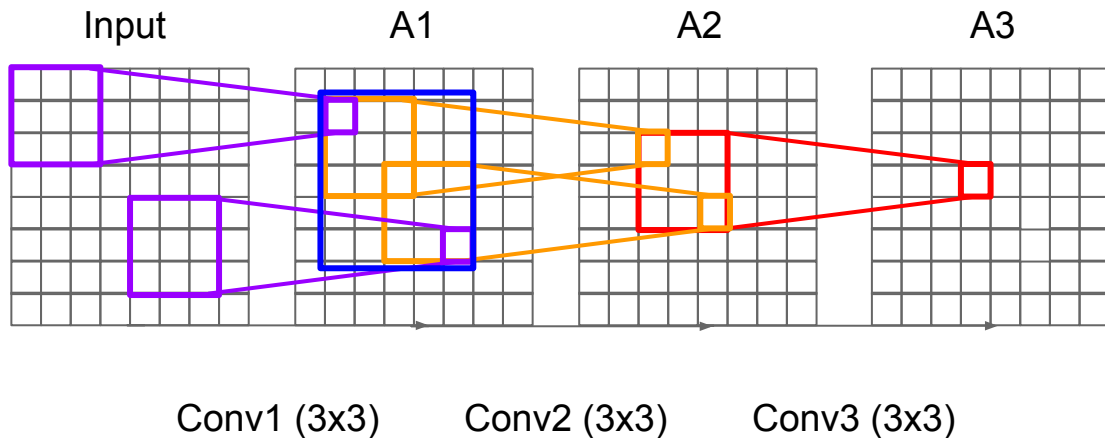
Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?



# Case Study: VGGNet

*[Simonyan and Zisserman, 2014]*

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



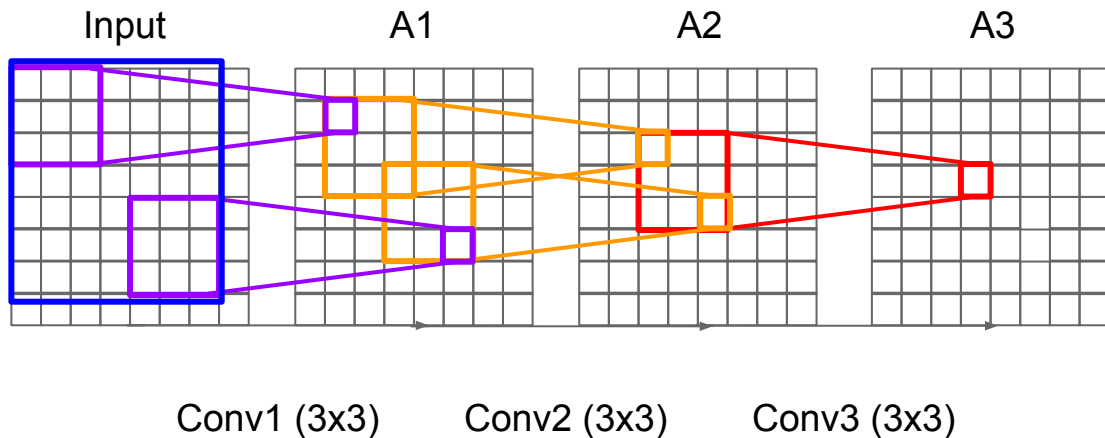
## VGG16

## VGG19

# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

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



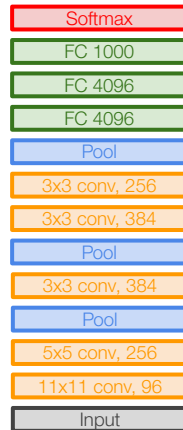
# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers  
has same **effective receptive field** as  
one 7x7 conv layer

[7x7]



AlexNet



VGG16

VGG19  
53

# Case Study: VGGNet

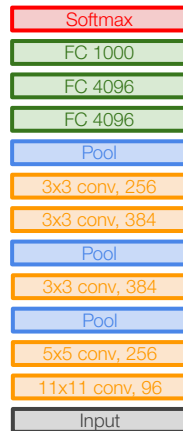
[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

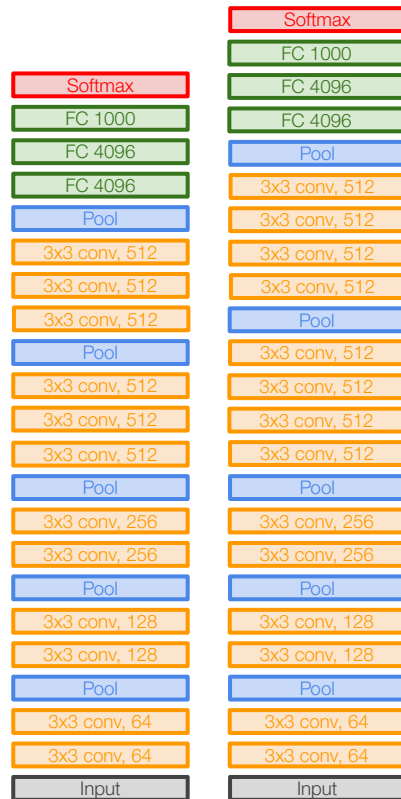
Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

But deeper, more non-linearities

And fewer parameters:  $3 * (3^2 C^2)$  vs.  $7^2 C^2$  for C channels per layer



AlexNet



VGG16

VGG19  
54

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:  $(3*3*512)*512 = 2,359,296$

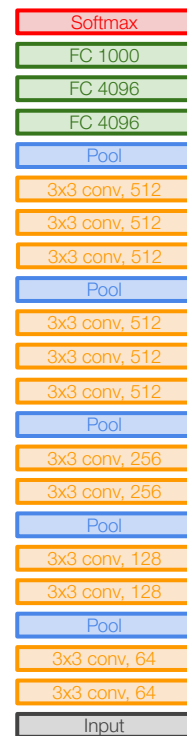
CONV3-512: [14x14x512] memory: 14\*14\*512=100K params:  $(3*3*512)*512 = 2,359,296$

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

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

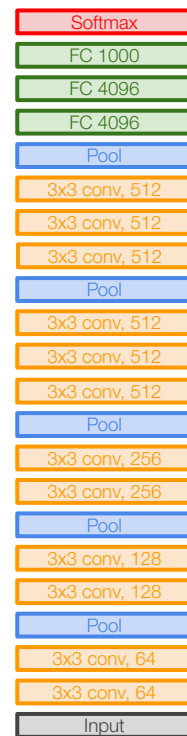
FC: [1x1x4096] memory: 4096 params:  $7*7*512*4096 = 102,760,448$

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

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

TOTAL memory: 24M \* 4 bytes ~= 96MB / image (for a forward pass)

TOTAL params: 138M parameters



VGG16



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:  $4096*1000 = 4,096,000$

Note:

Most memory is in  
early CONV

Most params are  
in late FC

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

TOTAL params: 138M parameters

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:  $4096*1000 = 4,096,000$

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

**TOTAL params: 138M parameters**



VGG16

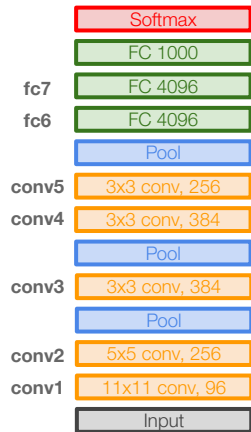
Common names

# Case Study: VGGNet

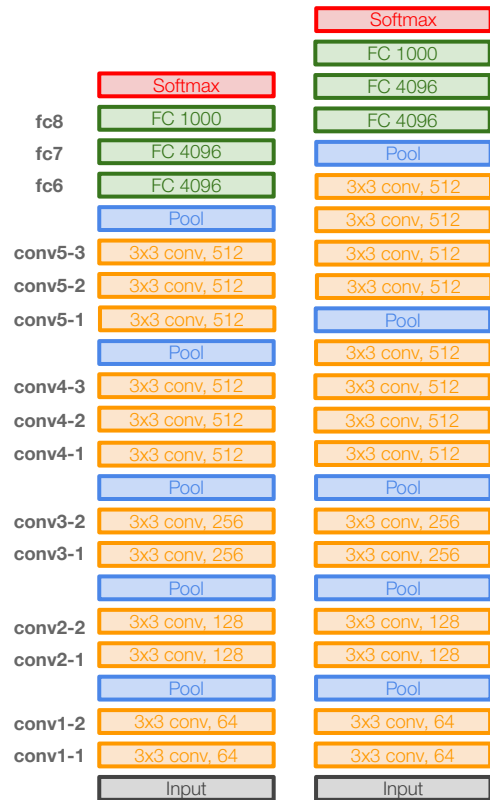
[Simonyan and Zisserman, 2014]

## 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



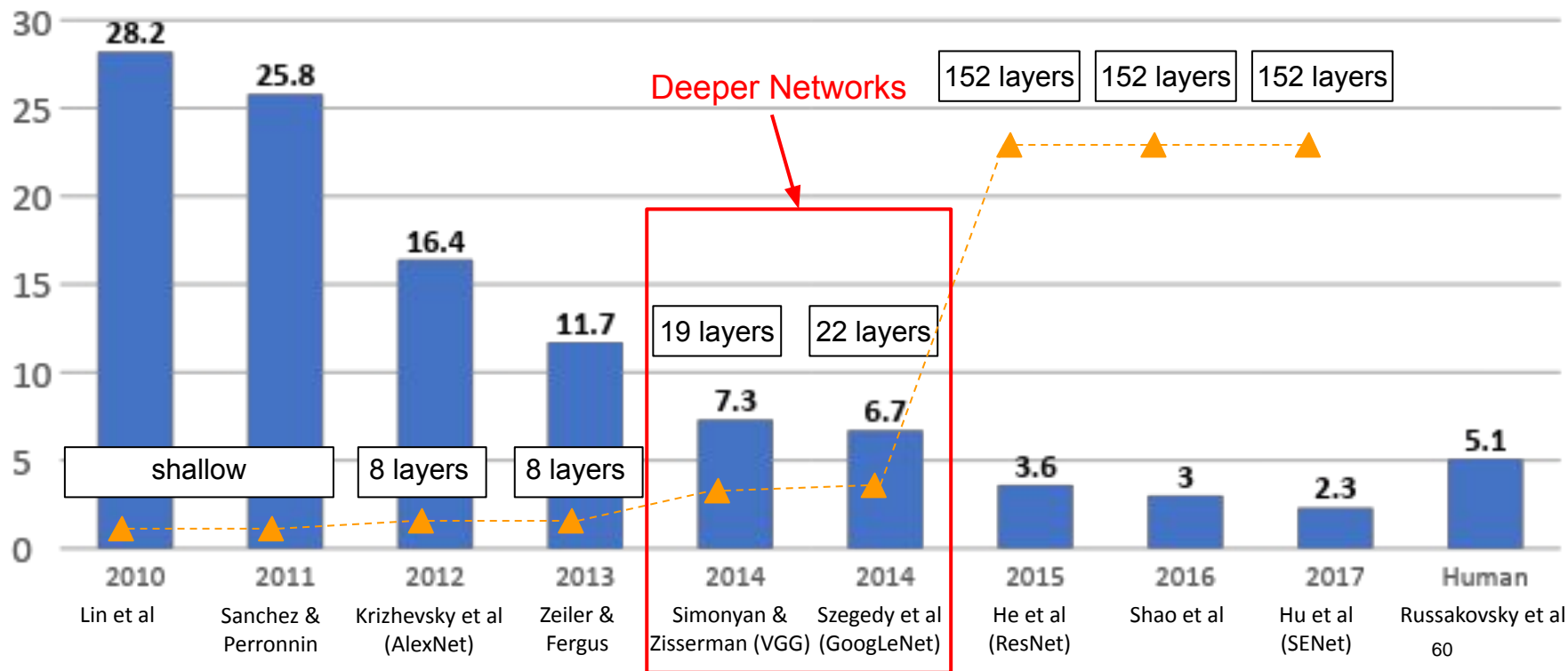
AlexNet



VGG16

VGG19  
59

# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

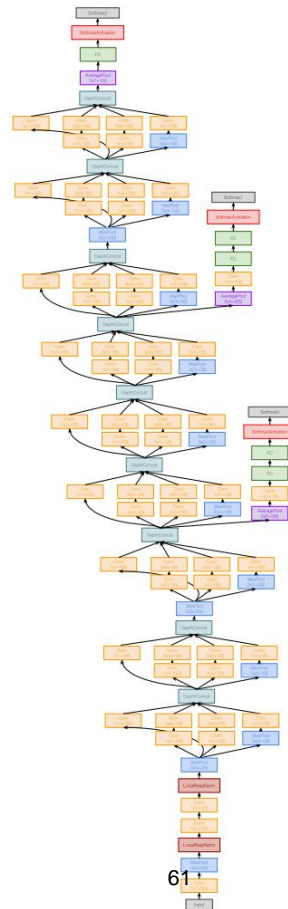
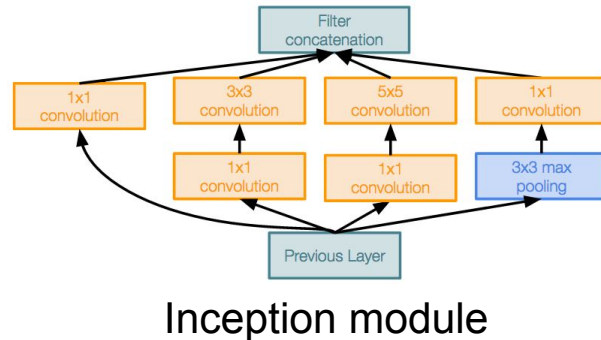


# Case Study: GoogLeNet

[Szegedy et al., 2014]

Deeper networks, with computational efficiency

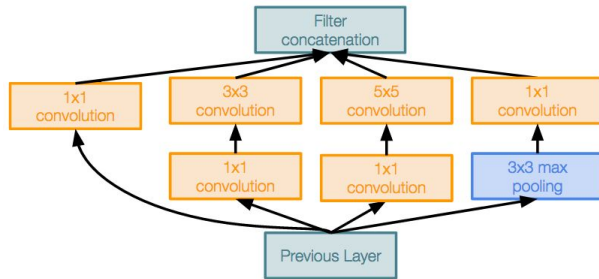
- ILSVRC'14 classification winner (6.7% top 5 error)
- 22 layers
- Only 5 million parameters!  
12x less than AlexNet  
27x less than VGG-16
- Efficient “Inception” module
- No FC layers



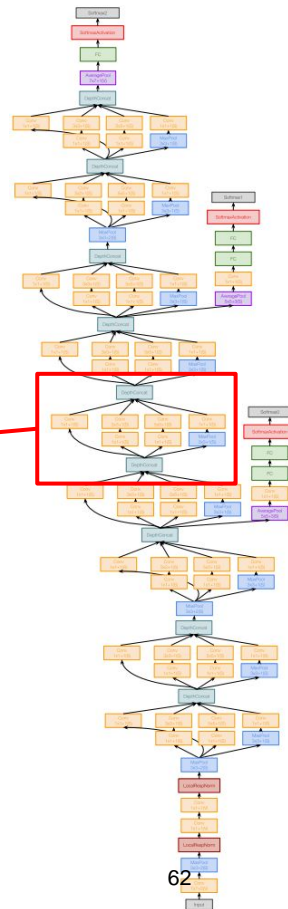
# Case Study: GoogLeNet

[Szegedy et al., 2014]

“Inception module”: design a good local network topology (network within a network) and then stack these modules on top of each other

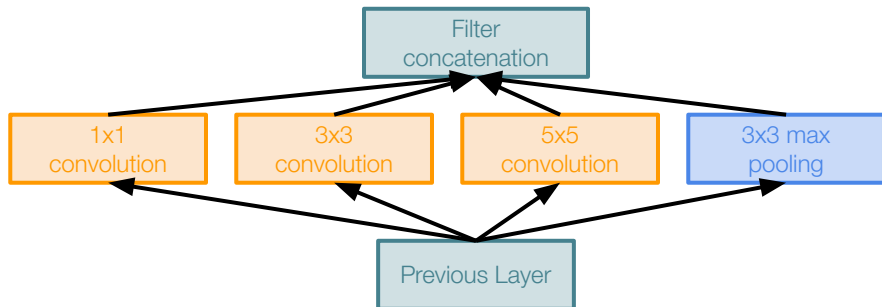


Inception module



# Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

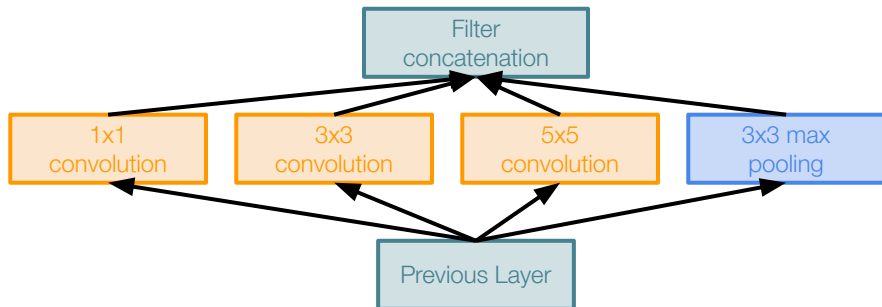
Apply parallel filter operations on the input from previous layer:

- Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
- Pooling operation (3x3)

Concatenate all filter outputs together channel-wise

# Case Study: GoogLeNet

[Szegedy et al., 2014]



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 channel-wise

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

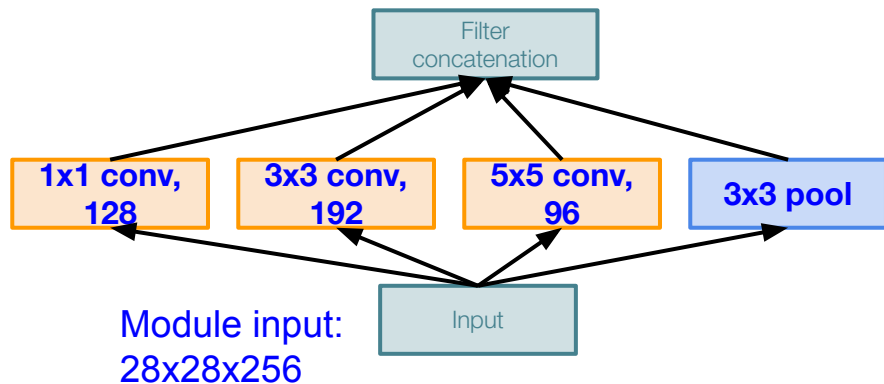


# Case Study: GoogLeNet

[Szegedy et al., 2014]

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

Example:



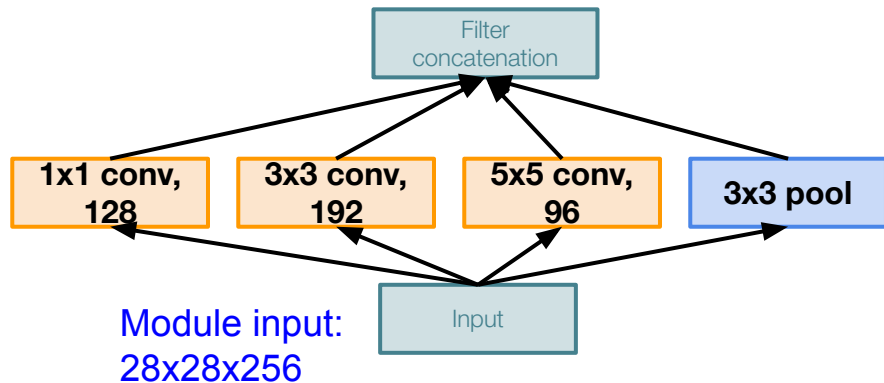
Naive Inception module

# Case Study: GoogLeNet

[Szegedy et al., 2014]

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

Example: Q1: What are the output sizes of all different filter operations?



Naive Inception module

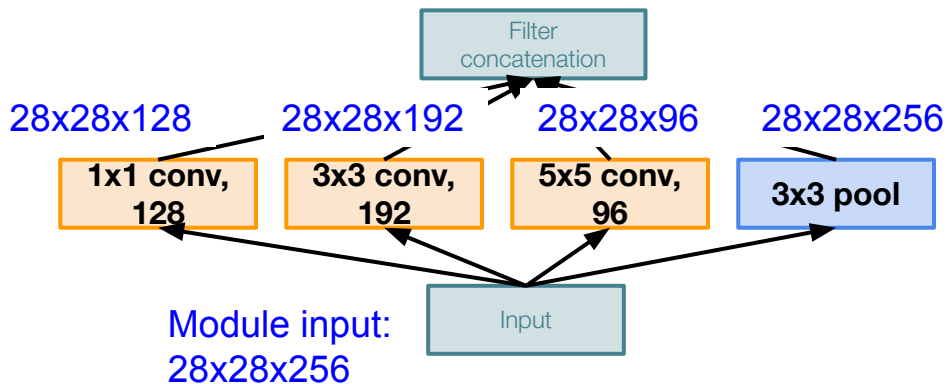
# Case Study: GoogLeNet

[Szegedy et al., 2014]

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

Example:

Q1: What are the output sizes of all different filter operations?



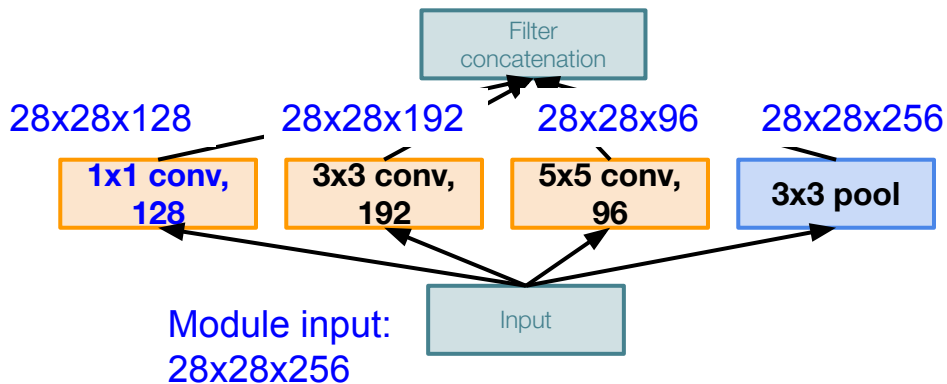
Naive Inception module

# Case Study: GoogLeNet

[Szegedy et al., 2014]

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

Example: Q2: What is output size after filter concatenation?



Naive Inception module

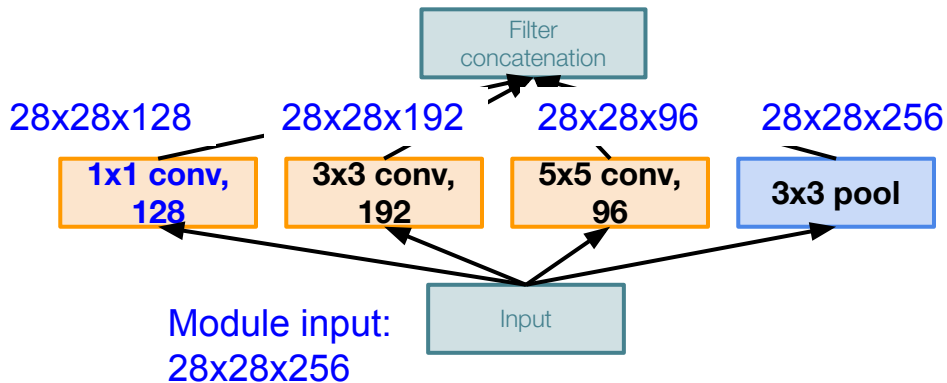
# Case Study: GoogLeNet

[Szegedy et al., 2014]

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

**Example:** Q2: What is output size after filter concatenation?

$$28 \times 28 \times (128 + 192 + 96 + 256) = 28 \times 28 \times 672$$



Naive Inception module

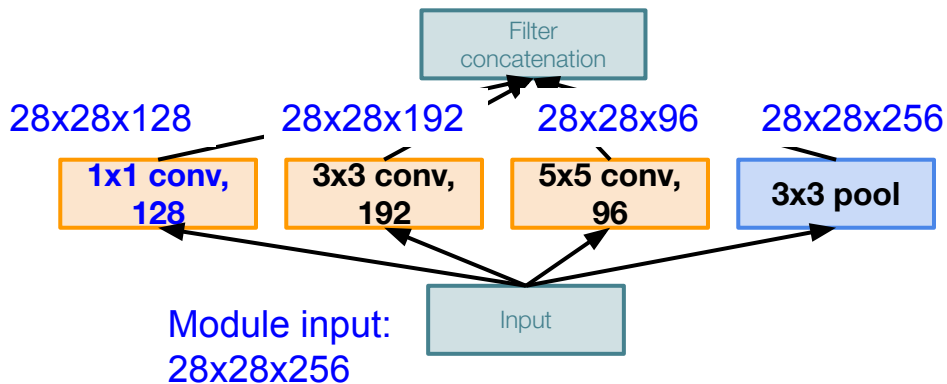
# Case Study: GoogLeNet

[Szegedy et al., 2014]

**Example:**

Q2: What is output size after filter concatenation?

$$28 \times 28 \times (128 + 192 + 96 + 256) = 28 \times 28 \times 672$$



Naive Inception module

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

**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**

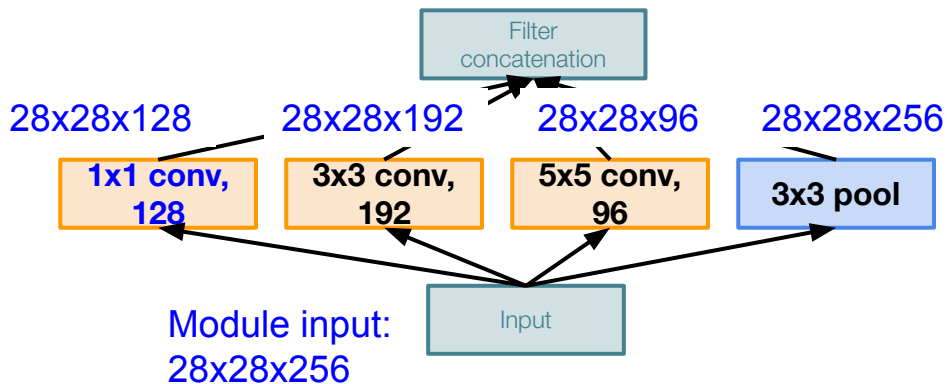
# Case Study: GoogLeNet

[Szegedy et al., 2014]

**Example:**

Q2: What is output size after filter concatenation?

$$28 \times 28 \times (128 + 192 + 96 + 256) = 28 \times 28 \times 672$$



Naive Inception module

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

**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 compute

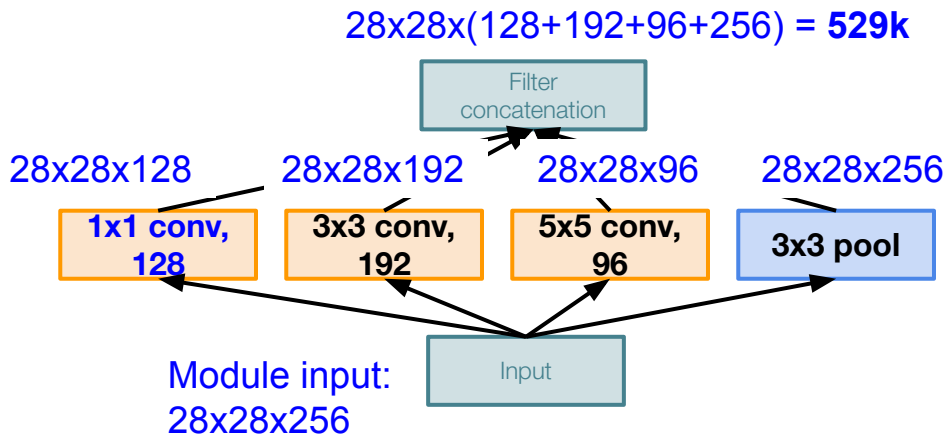
Pooling layer also preserves feature depth, which means total depth after concatenation can only grow at every layer!

# Case Study: GoogLeNet

[Szegedy et al., 2014]

**Example:**

Q2: What is output size after filter concatenation?



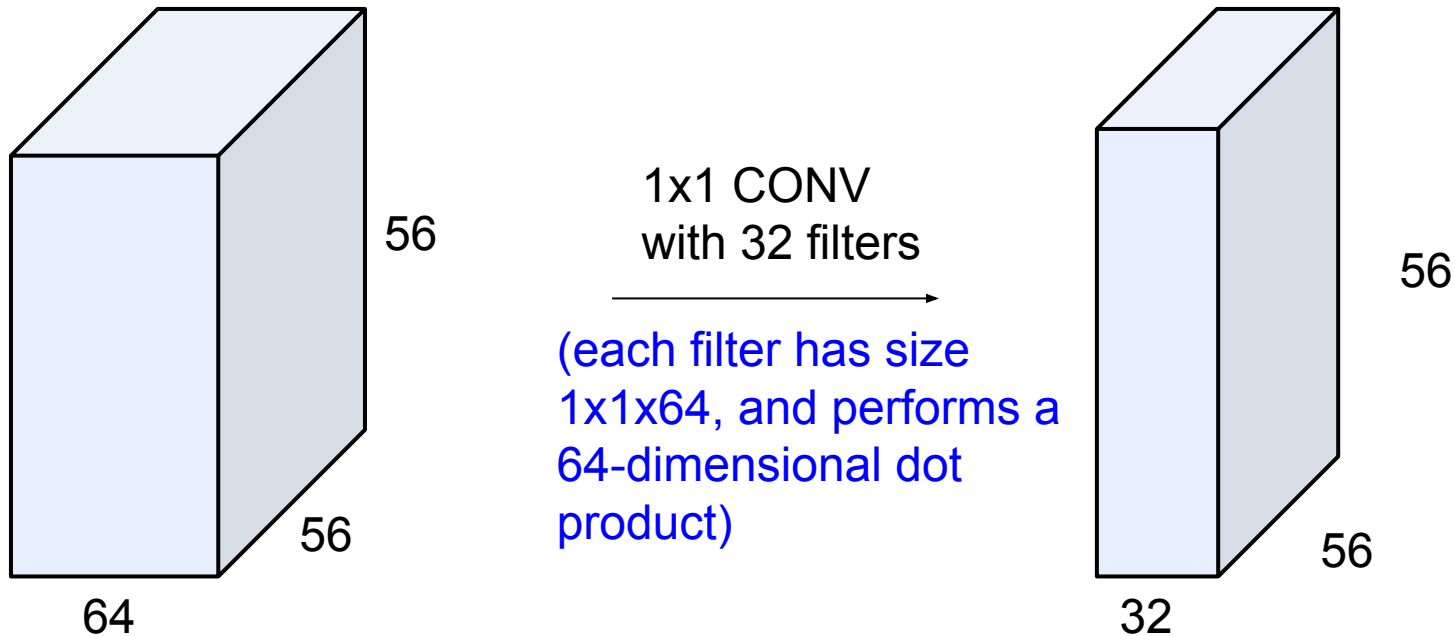
Naive Inception module

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

Solution: “bottleneck” layers that use  $1 \times 1$  convolutions to reduce feature channel size

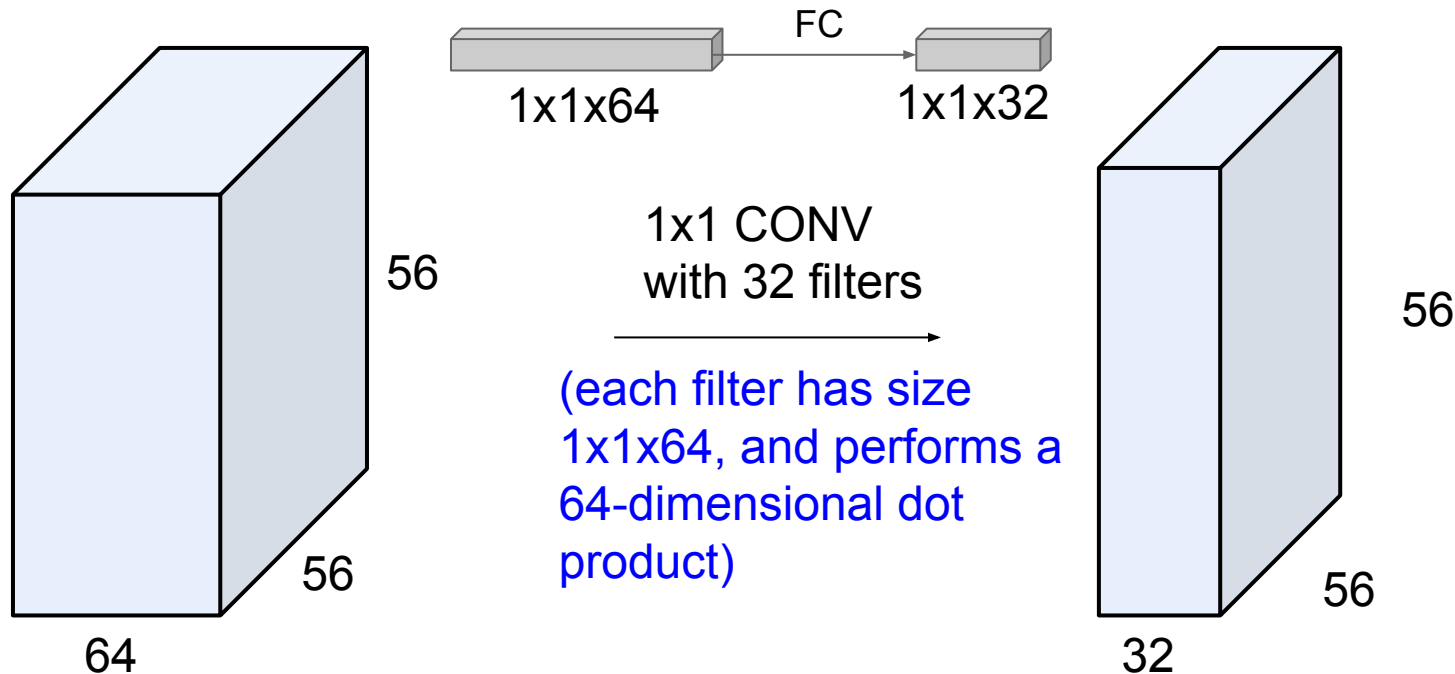


# Review: 1x1 convolutions



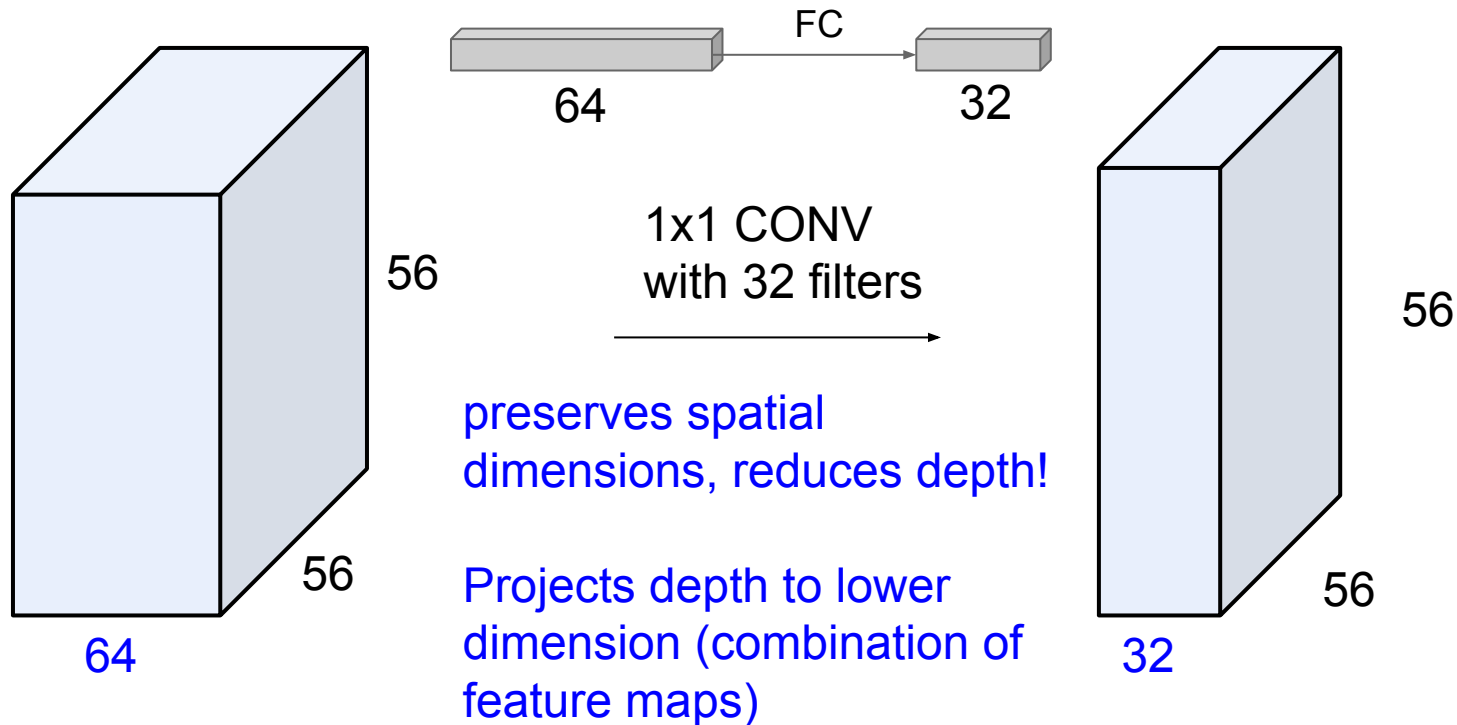
# Review: 1x1 convolutions

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



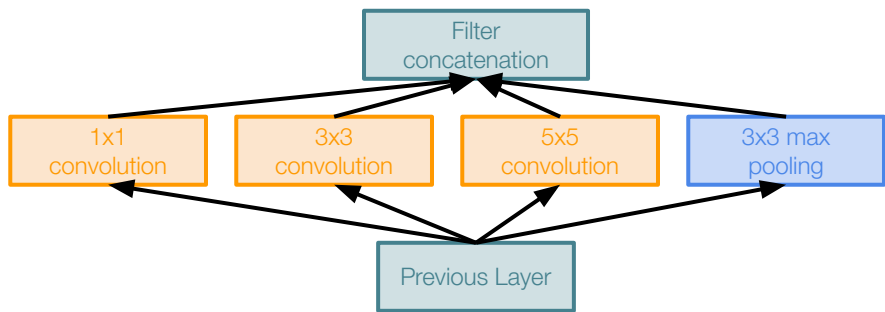
# Review: 1x1 convolutions

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

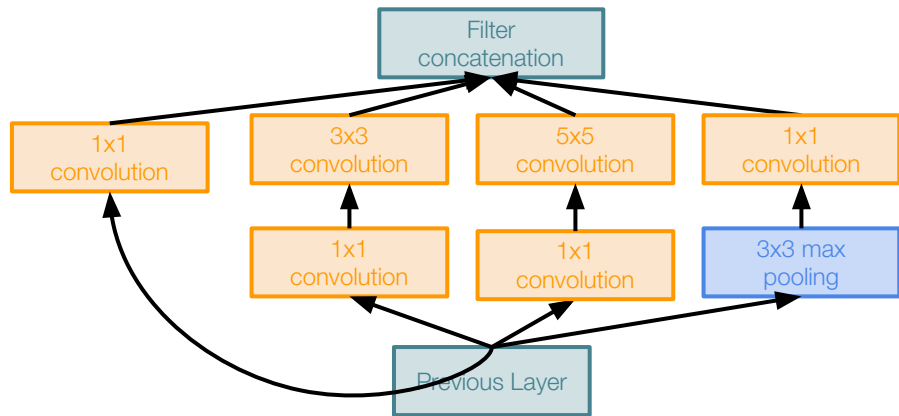


# Case Study: GoogLeNet

[Szegedy et al., 2014]



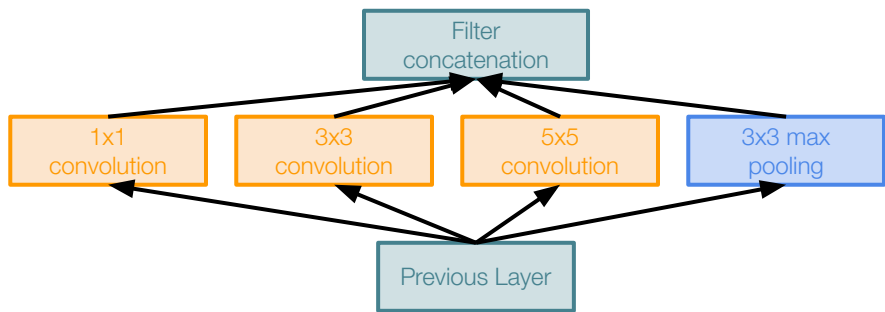
Naive Inception module



Inception module with dimension reduction

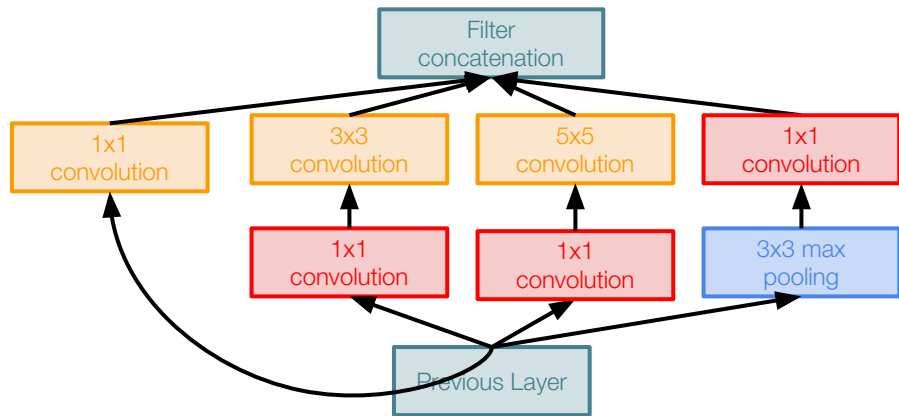
# Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

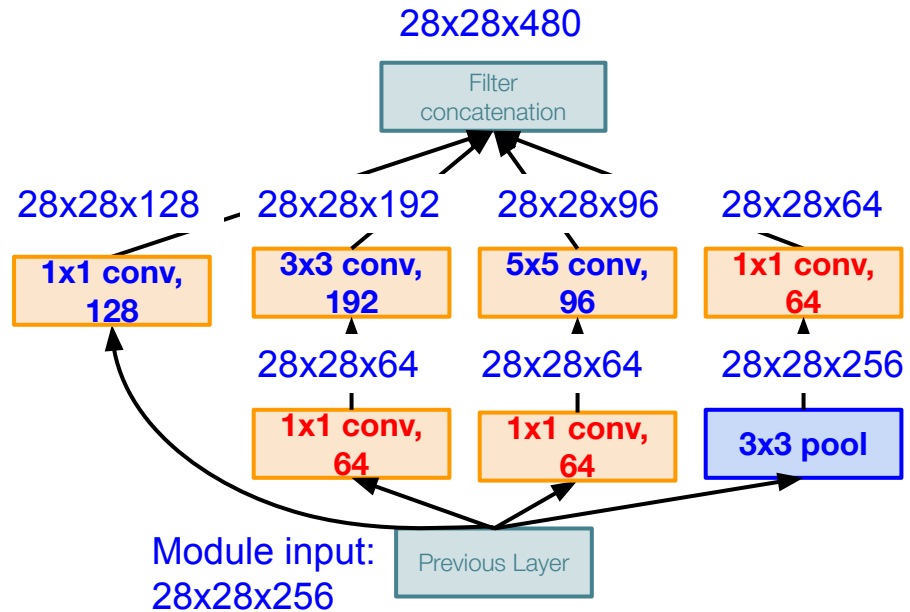
1x1 conv “bottleneck”  
layers



Inception module with dimension reduction

# Case Study: GoogLeNet

[Szegedy et al., 2014]



Inception module with dimension reduction

Using same parallel layers as naive example, and adding “ $1 \times 1$  conv, 64 filter” bottlenecks:

## Conv Ops:

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

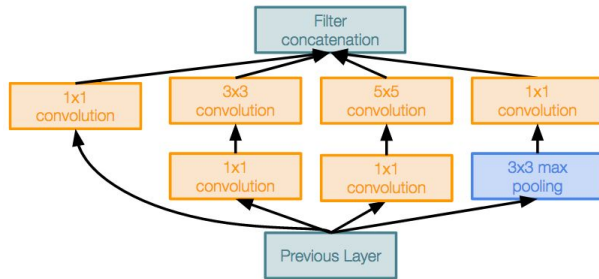
**Total: 358M ops**

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

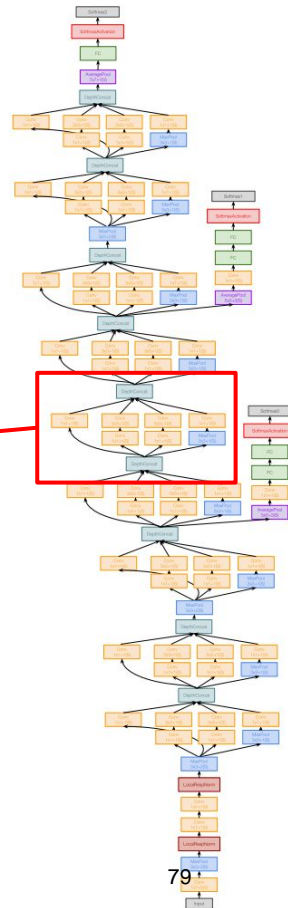
# Case Study: GoogLeNet

[Szegedy et al., 2014]

Stack Inception modules  
with dimension reduction  
on top of each other



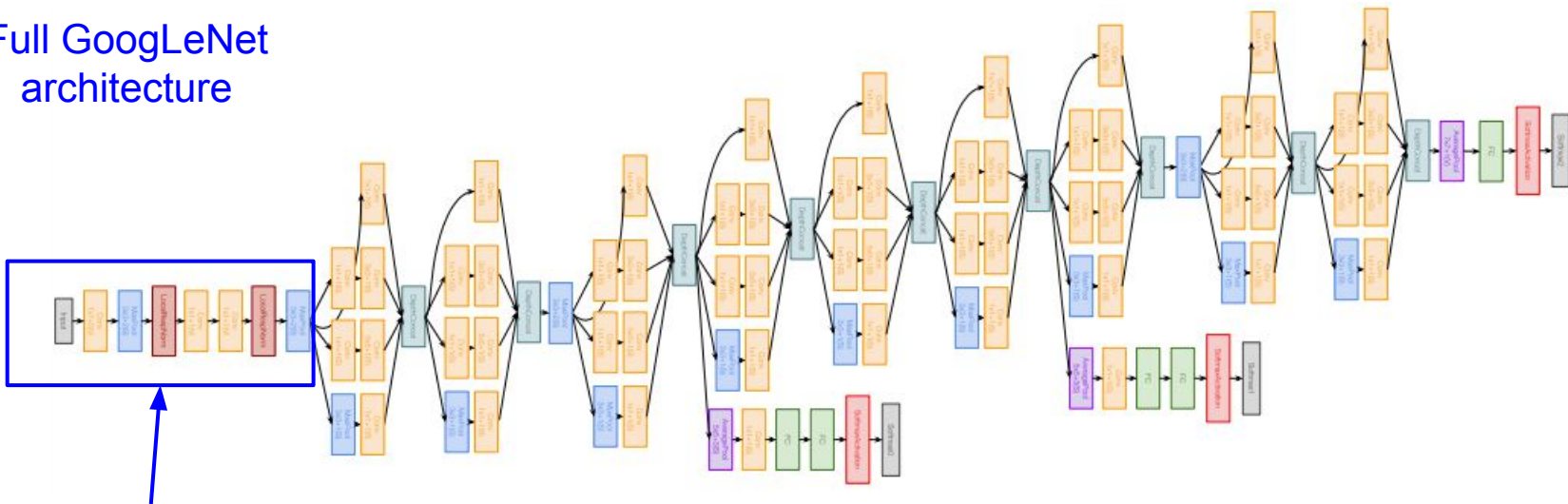
Inception module



# Case Study: GoogLeNet

[Szegedy et al., 2014]

# Full GoogLeNet architecture



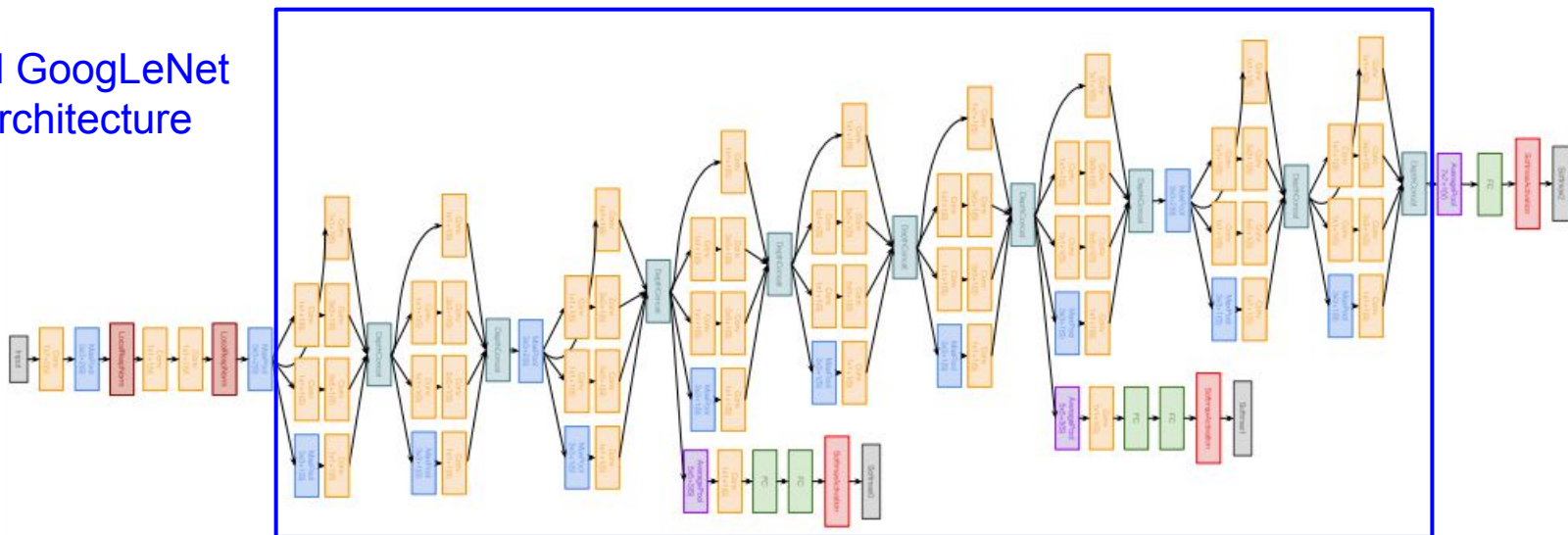
Stem Network:  
Conv-Pool-  
2x Conv-Pool



# Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet  
architecture

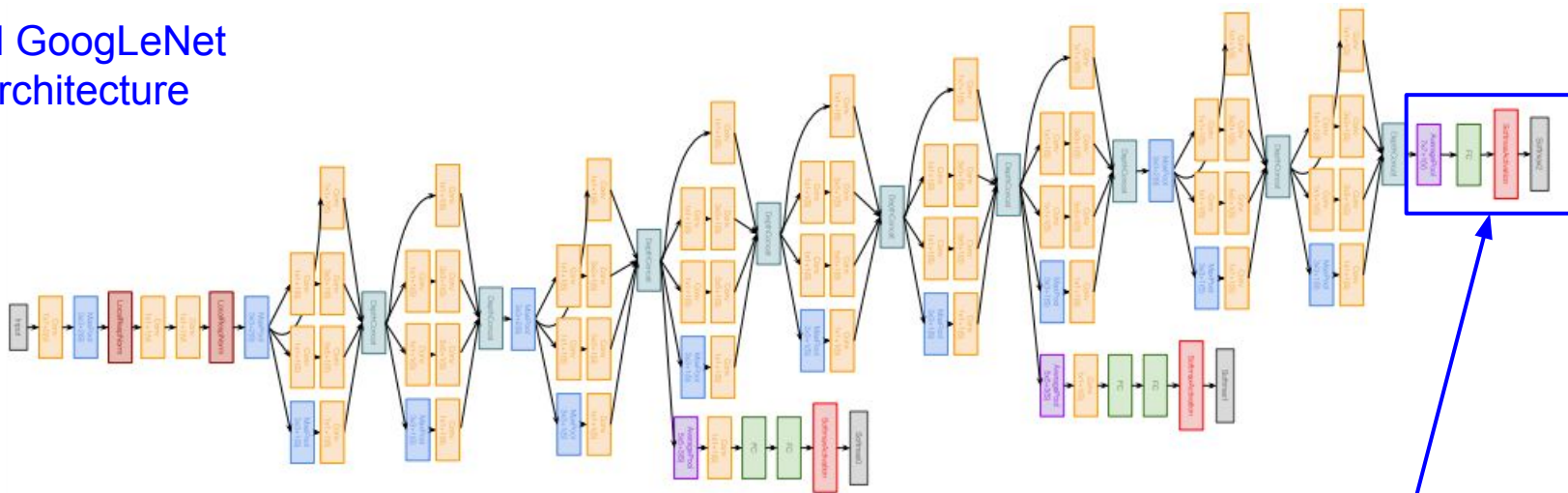


Stacked Inception  
Modules

# Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet  
architecture

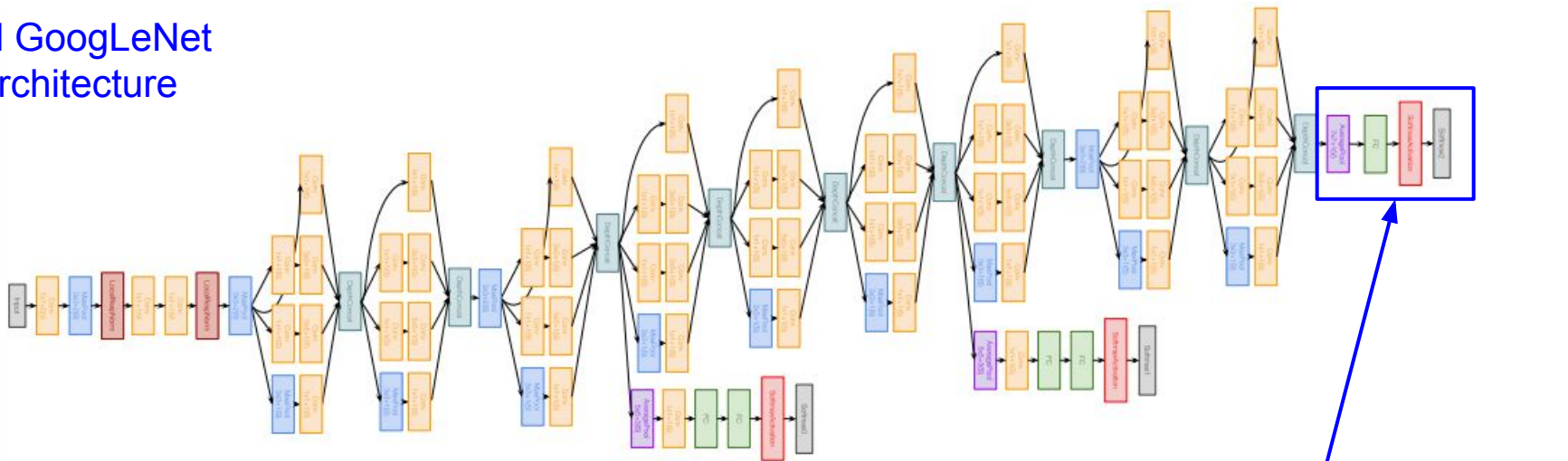


Classifier output

# Case Study: GoogLeNet

[Szegedy et al., 2014]

## Full GoogLeNet architecture



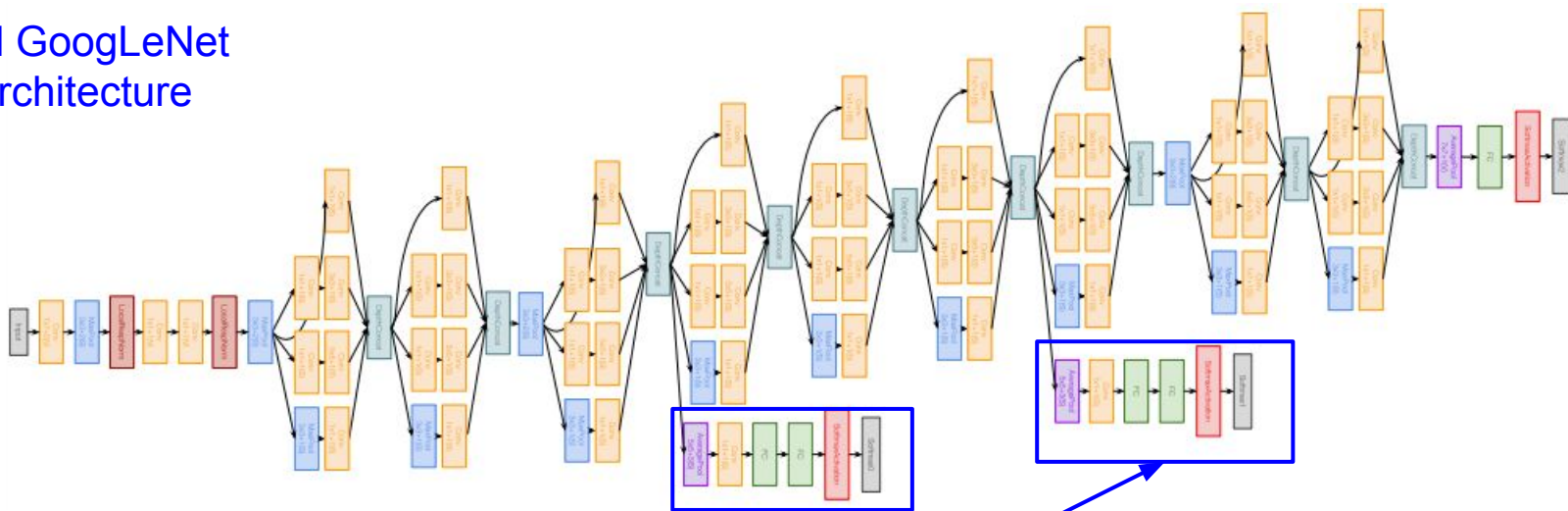
Note: after the last convolutional layer, a global average pooling layer is used that spatially averages across each feature map, before final FC layer. No longer multiple expensive FC layers!

Classifier output

# Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet  
architecture

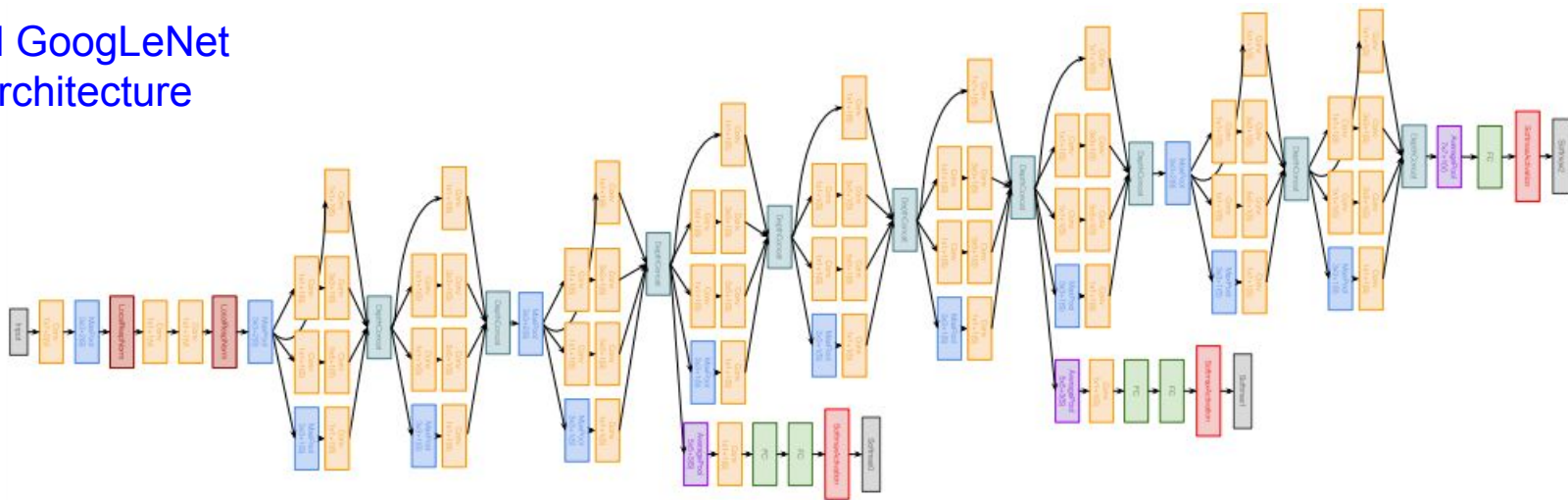


Auxiliary classification outputs to inject additional gradient at lower layers  
(AvgPool-1x1Conv-FC-FC-Softmax)

# Case Study: GoogLeNet

[Szegedy et al., 2014]

## Full GoogLeNet architecture



22 total layers with weights

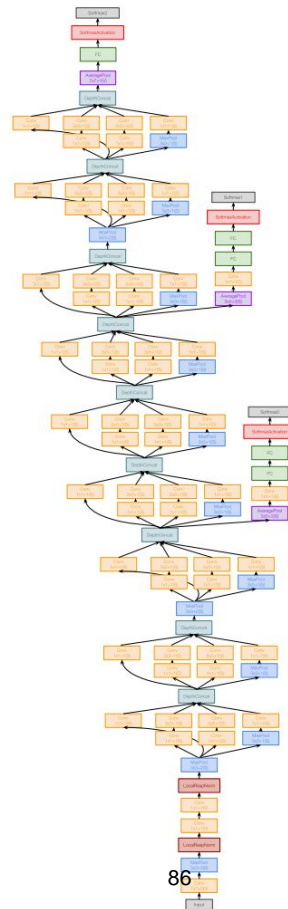
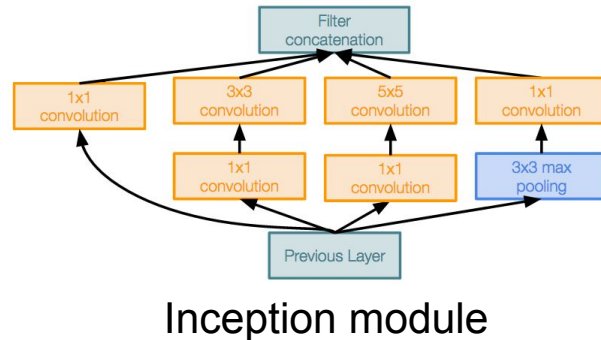
(parallel layers count as 1 layer => 2 layers per Inception module. Don't count auxiliary output layers)

# Case Study: GoogLeNet

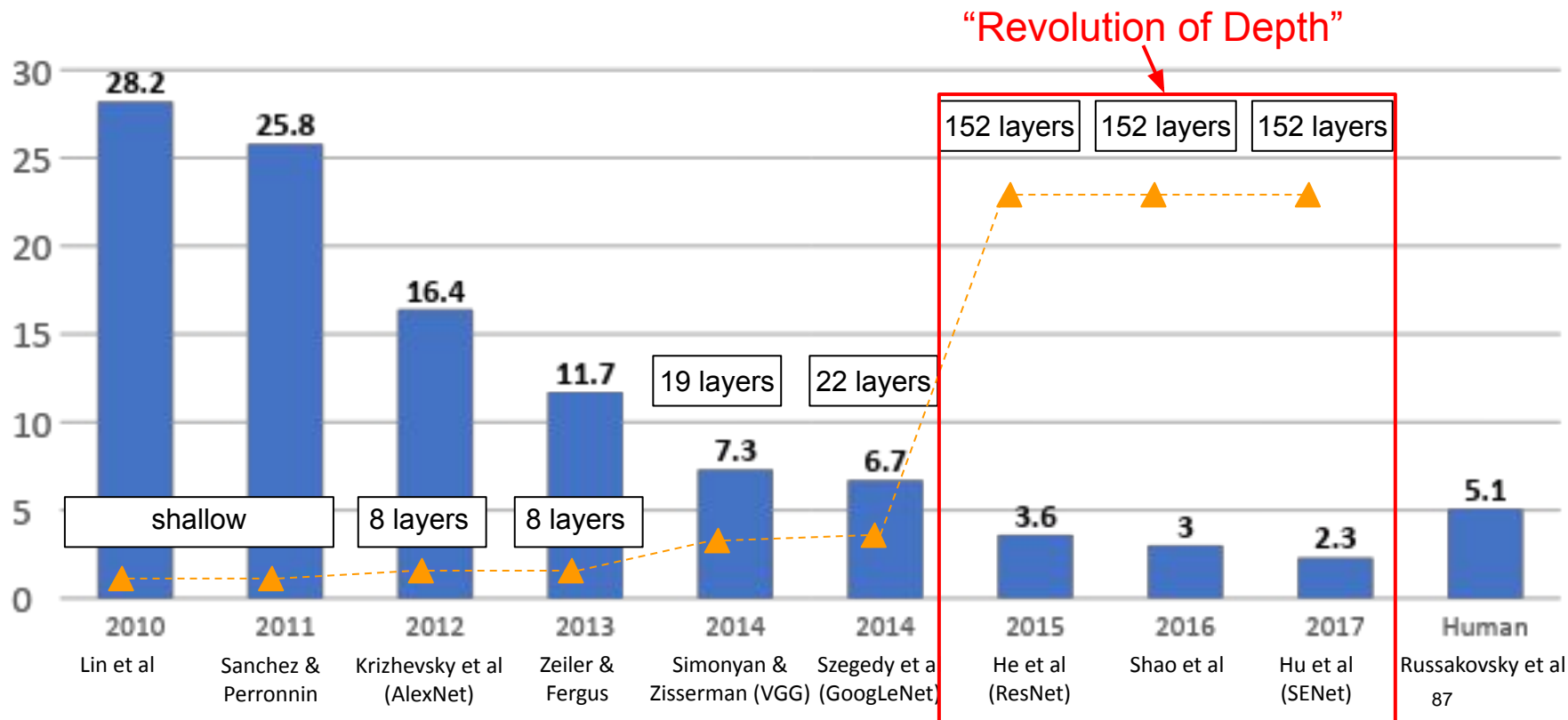
[Szegedy et al., 2014]

Deeper networks, with computational efficiency

- 22 layers
- Efficient “Inception” module
- Avoids expensive FC layers
- 12x less params than AlexNet
- 27x less params than VGG-16
- ILSVRC’14 classification winner (6.7% top 5 error)



# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

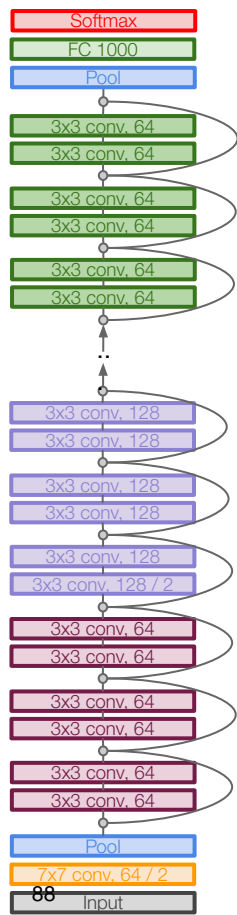
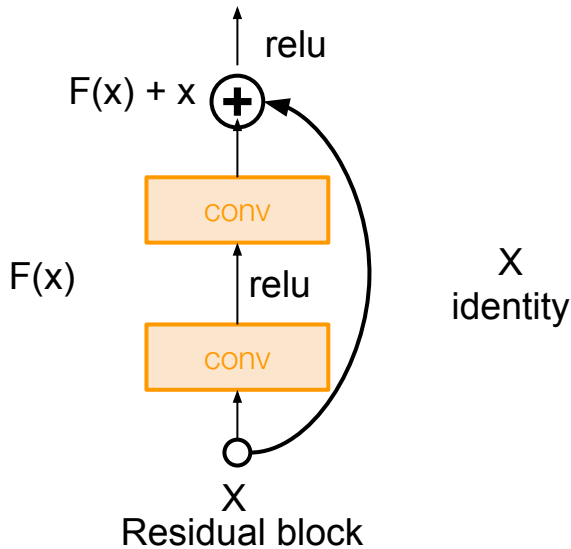


# Case Study: ResNet

*[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

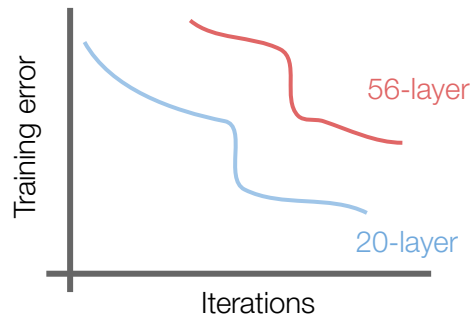
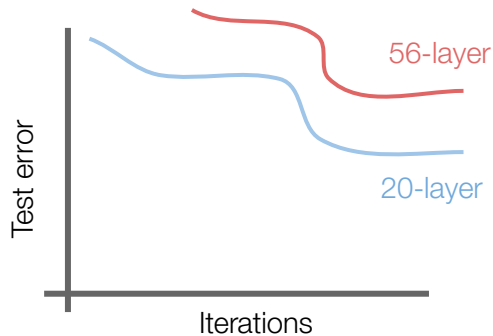
*[He et al., 2015]*

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

# Case Study: ResNet

[He et al., 2015]

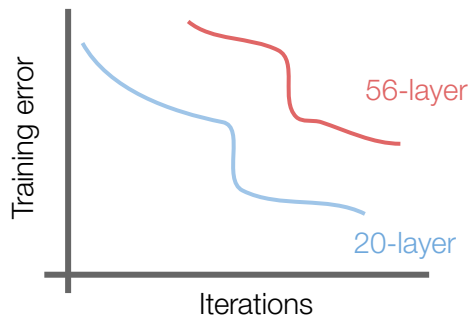
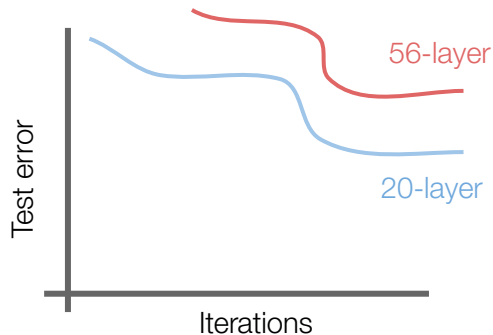
What happens when we continue stacking deeper layers on a “plain” convolutional neural network?



# 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 test and training error

-> The deeper model performs worse, but it's **not caused by overfitting!**

# Case Study: ResNet

[He et al., 2015]

Fact: Deep models have more representation power (more parameters) than shallower models.

Hypothesis: the problem is an *optimization* problem,  
**deeper models are harder to optimize**

# Case Study: ResNet

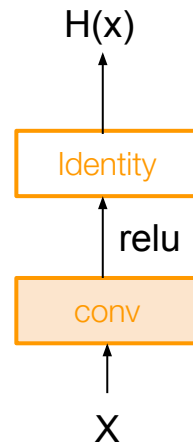
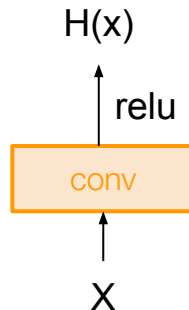
[He et al., 2015]

Fact: Deep models have more representation power (more parameters) than shallower models.

Hypothesis: the problem is an *optimization* problem, deeper models are harder to optimize

What should the deeper model learn to be at least as good as the shallower model?

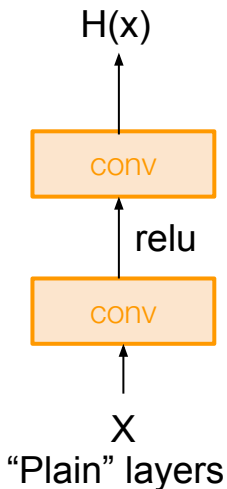
A solution by construction is copying the learned layers from the shallower model and setting additional layers to identity mapping.



# Case Study: ResNet

*[He et al., 2015]*

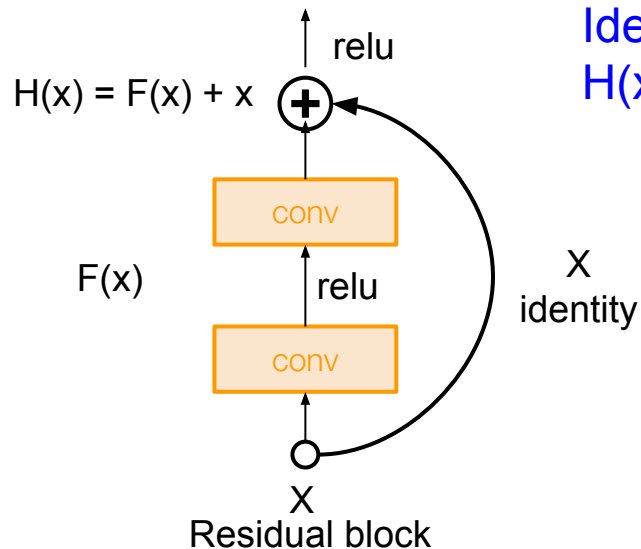
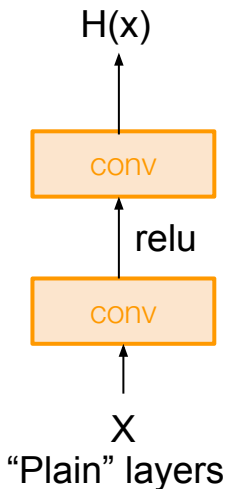
Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



# Case Study: ResNet

[He et al., 2015]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping

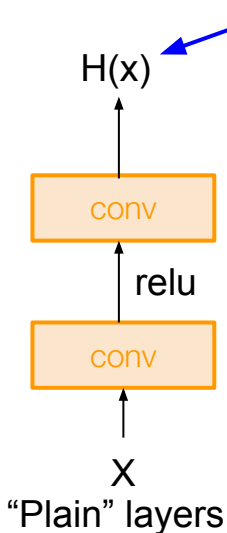


Identity mapping:  
 $H(x) = x$  if  $F(x) = 0$

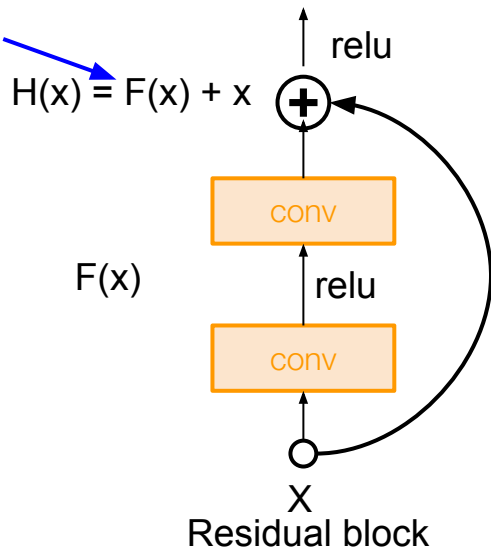
# Case Study: ResNet

[He et al., 2015]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



$$H(x) = F(x) + x$$



Identity mapping:  
 $H(x) = x$  if  $F(x) = 0$

Use layers to fit **residual**  
 $F(x) = H(x) - x$   
instead of  
 $H(x)$  directly

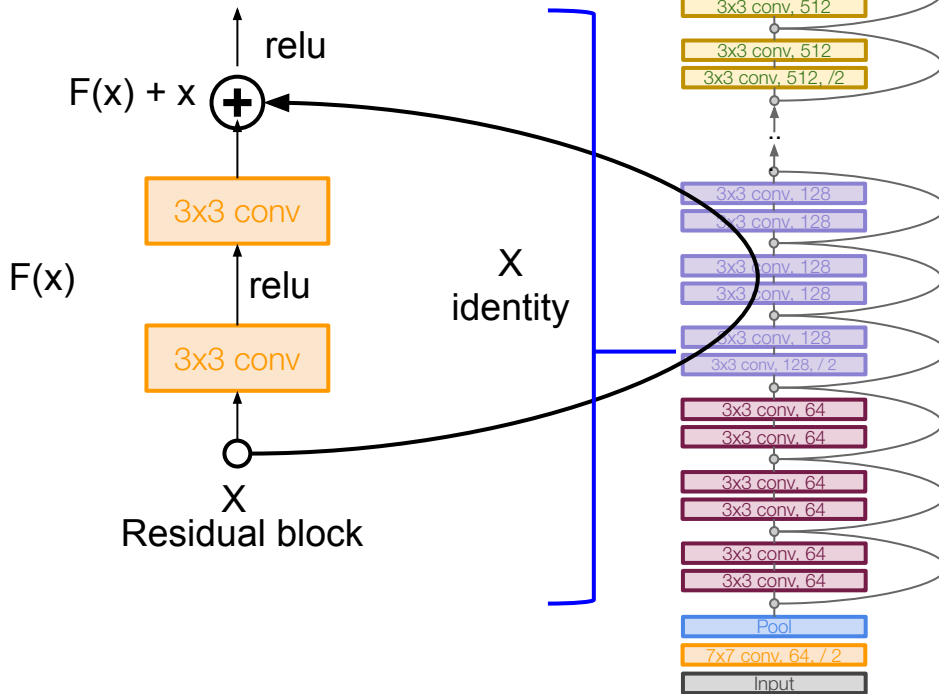


# Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers

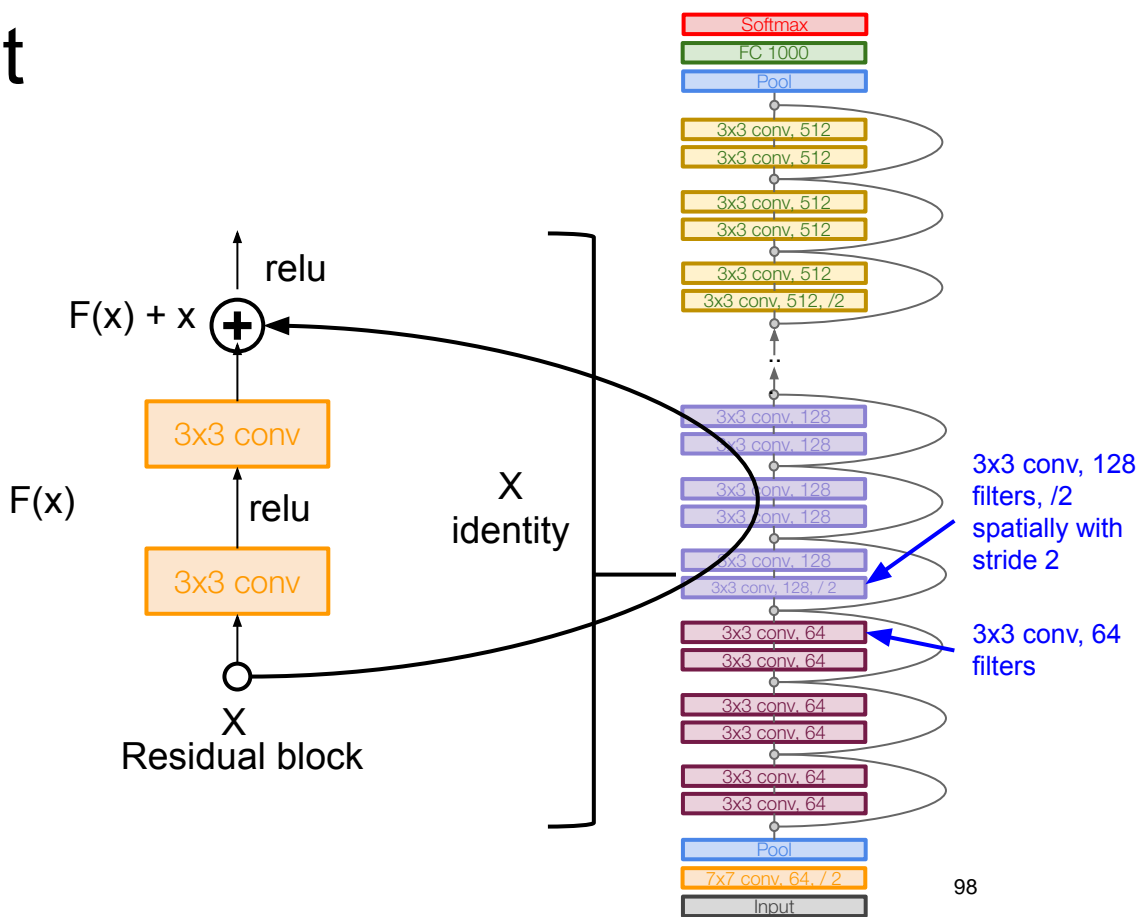


# Case Study: ResNet

[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)
- Reduce the activation volume by half.

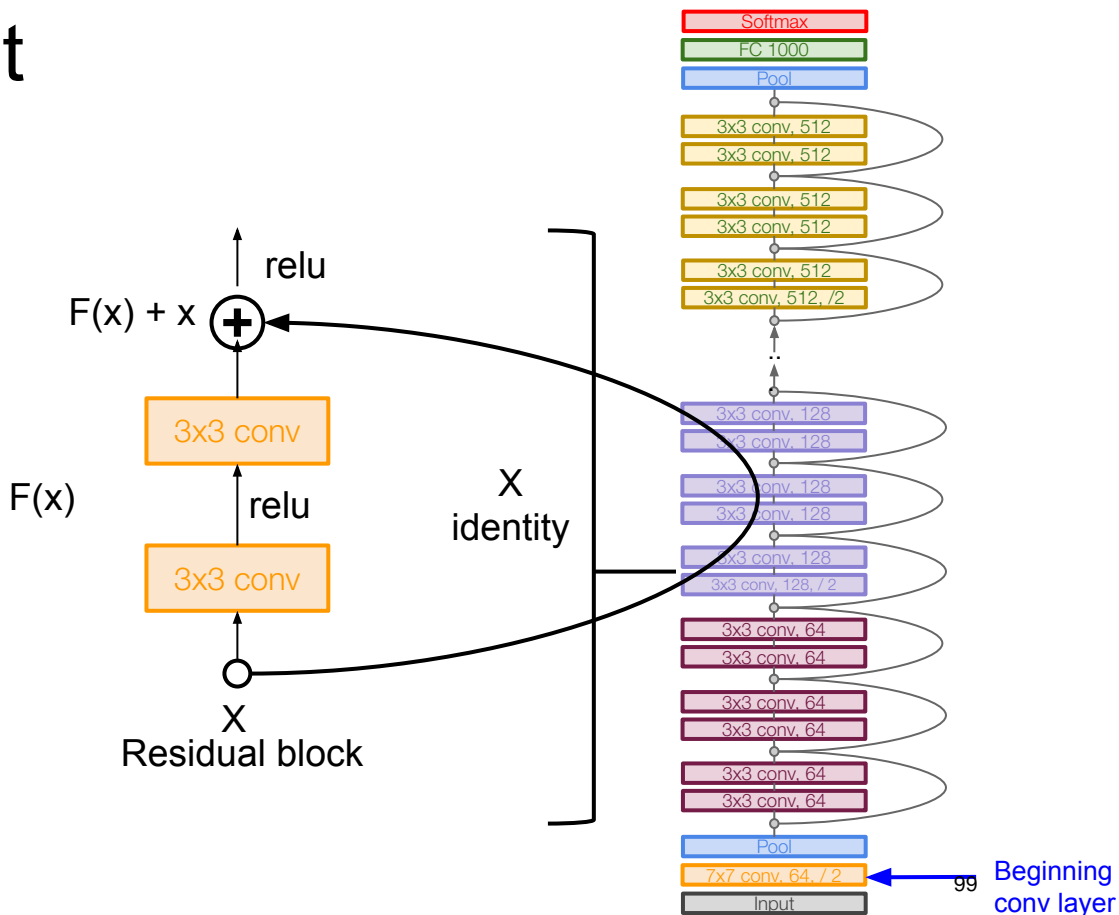


# Case Study: ResNet

[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 (stem)

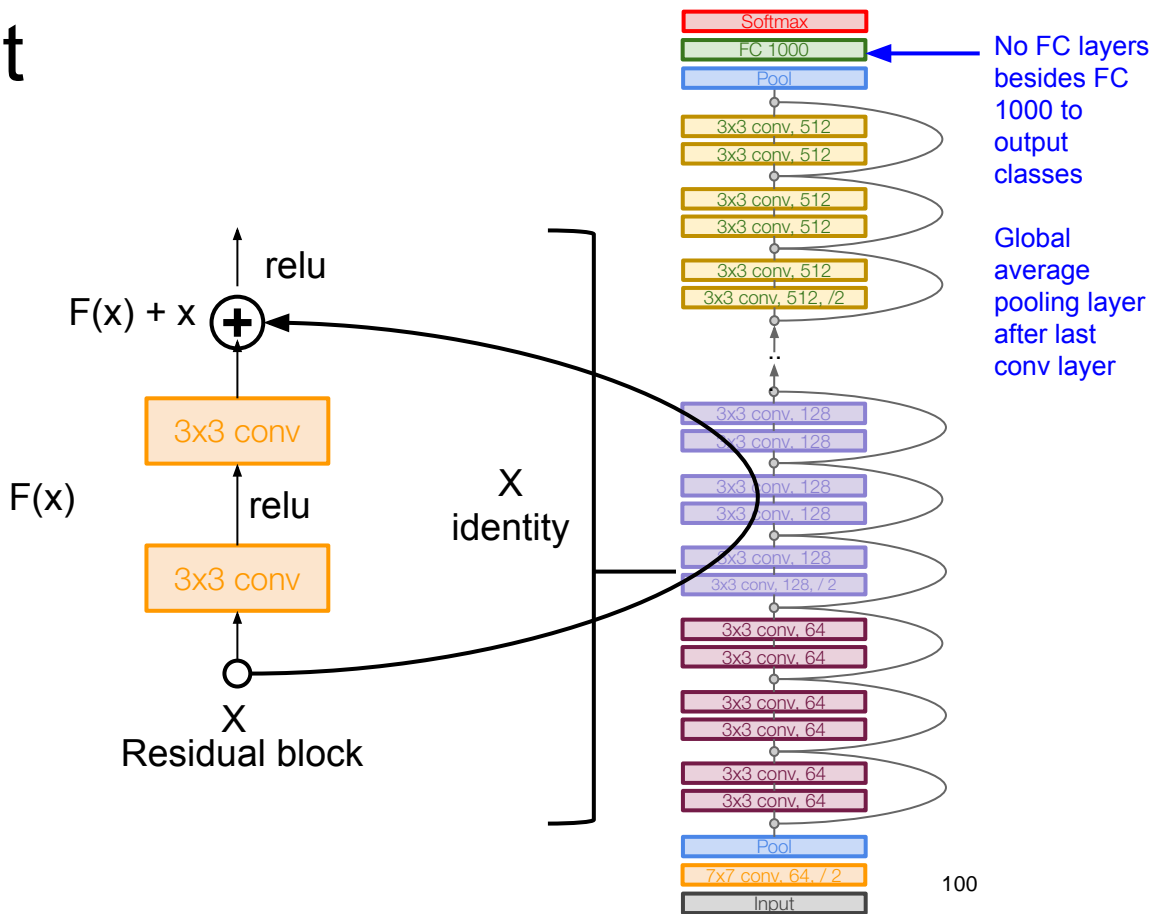


# Case Study: ResNet

[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 (stem)
- No FC layers at the end (only FC 1000 to output classes)
- (In theory, you can train a ResNet with input image of variable sizes)



# Case Study: ResNet

[He et al., 2015]

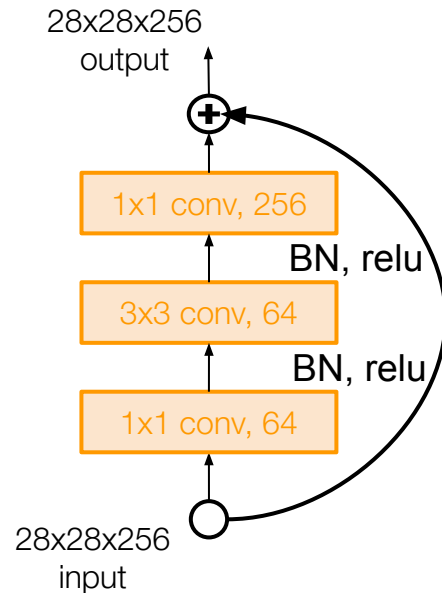
Total depths of 18, 34, 50,  
101, or 152 layers for  
ImageNet



# Case Study: ResNet

[He et al., 2015]

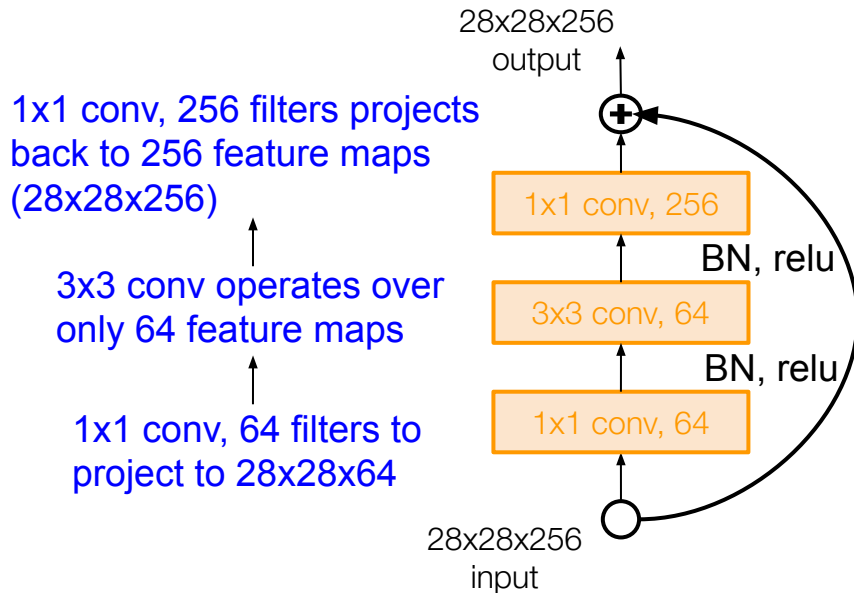
For deeper networks  
(ResNet-50+), use “bottleneck”  
layer to improve efficiency  
(similar to GoogLeNet)



# Case Study: ResNet

[He et al., 2015]

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



# Case Study: ResNet

*[He et al., 2015]*

Training ResNet in practice:

- Batch Normalization after every CONV layer
- Xavier 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



# 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 lower 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

# 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 lower training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions

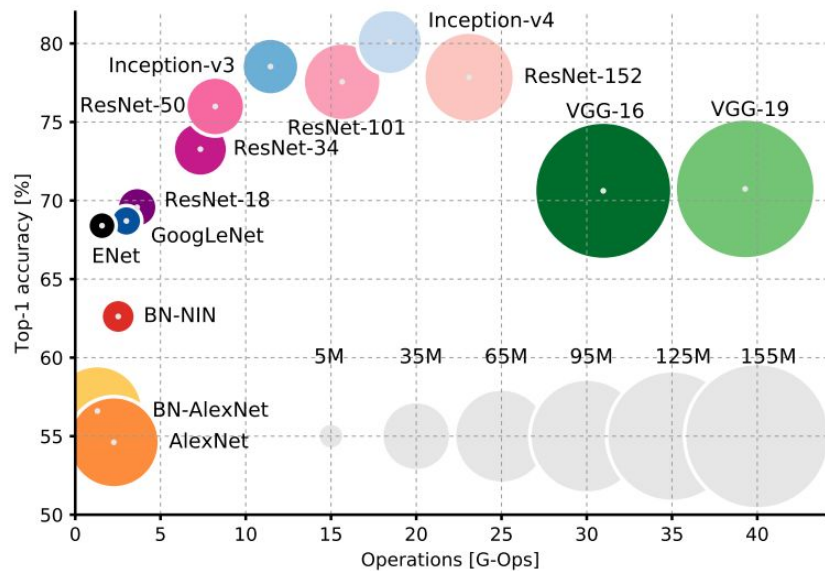
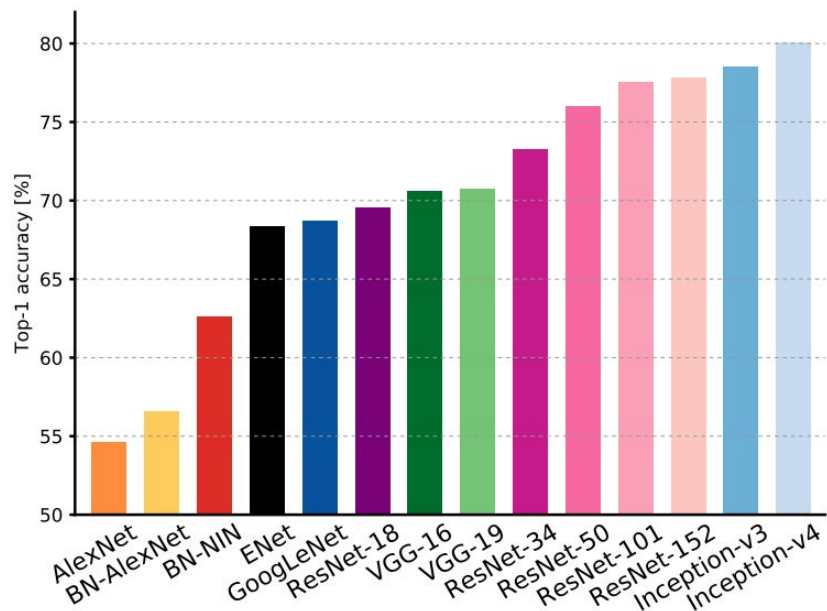
## MSRA @ ILSVRC & COCO 2015 Competitions

- **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)

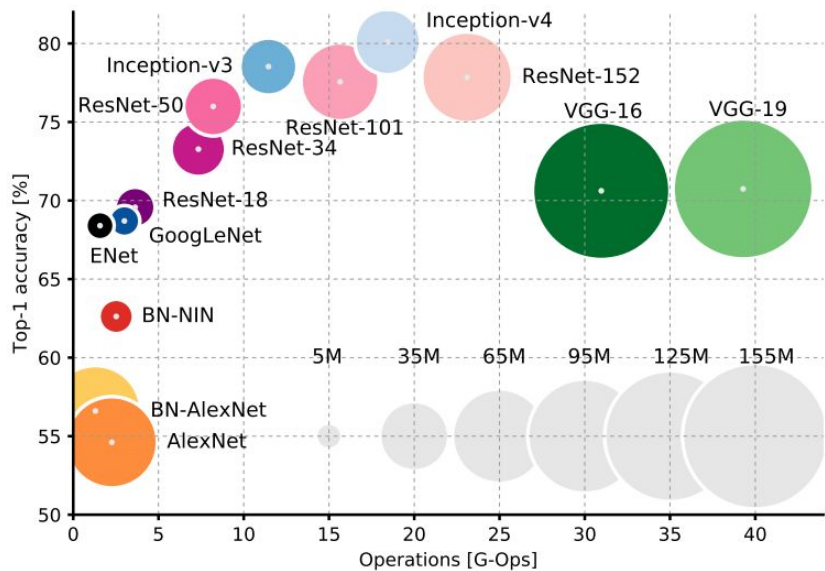
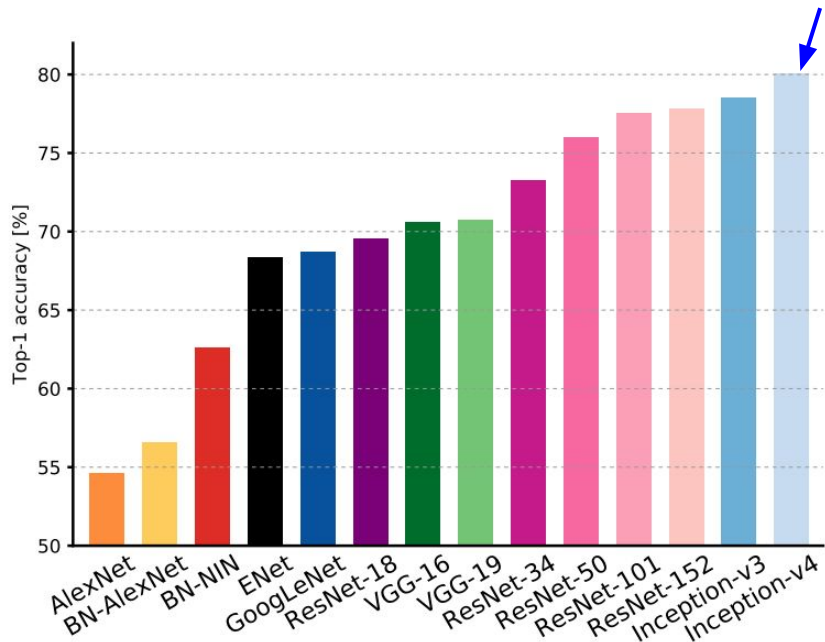
# Comparing complexity...



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

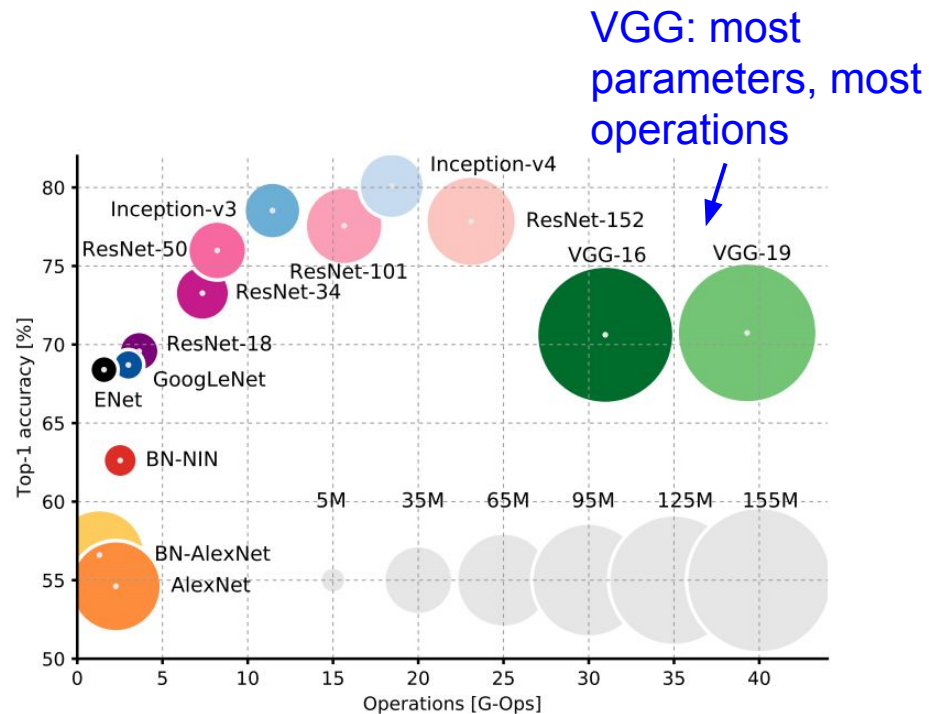
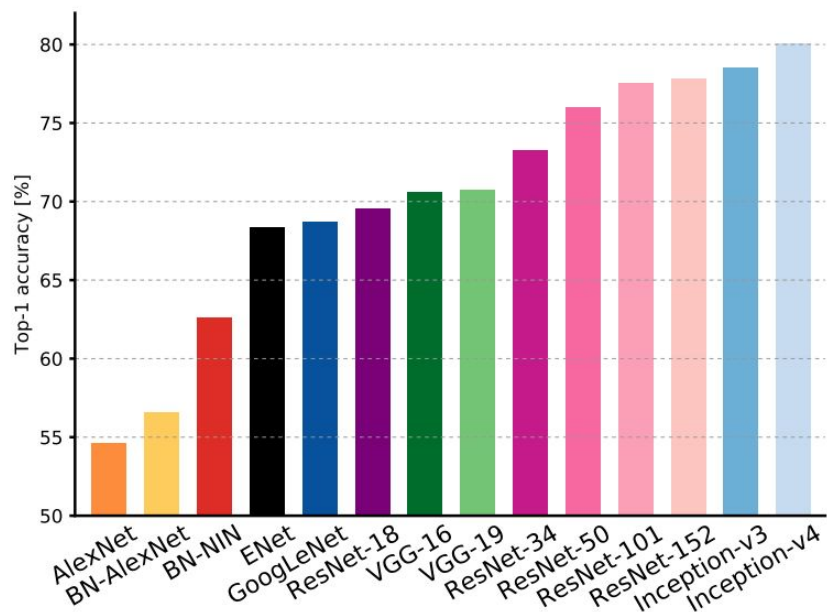
# Comparing complexity...

Inception-v4: Resnet + Inception!



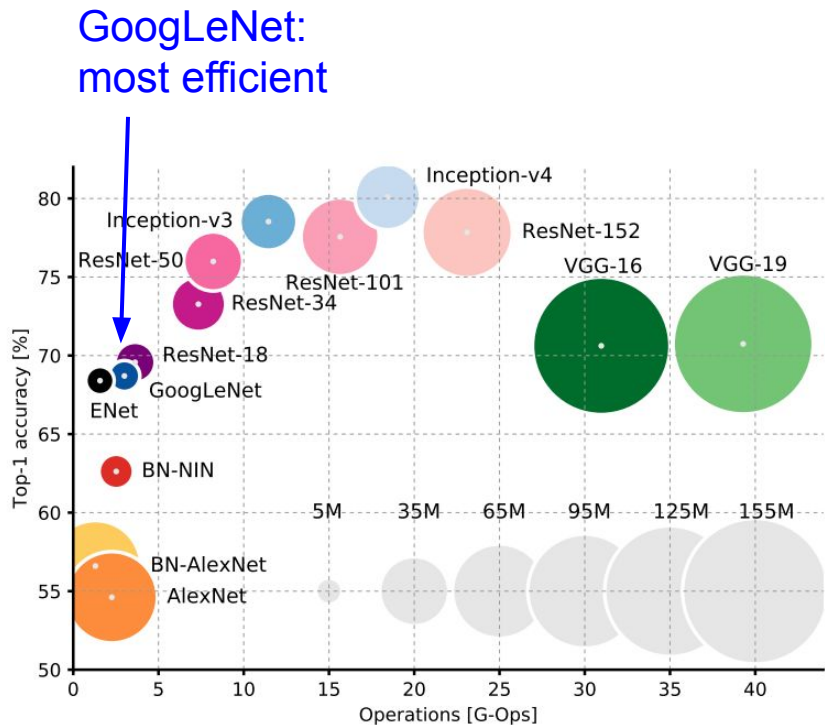
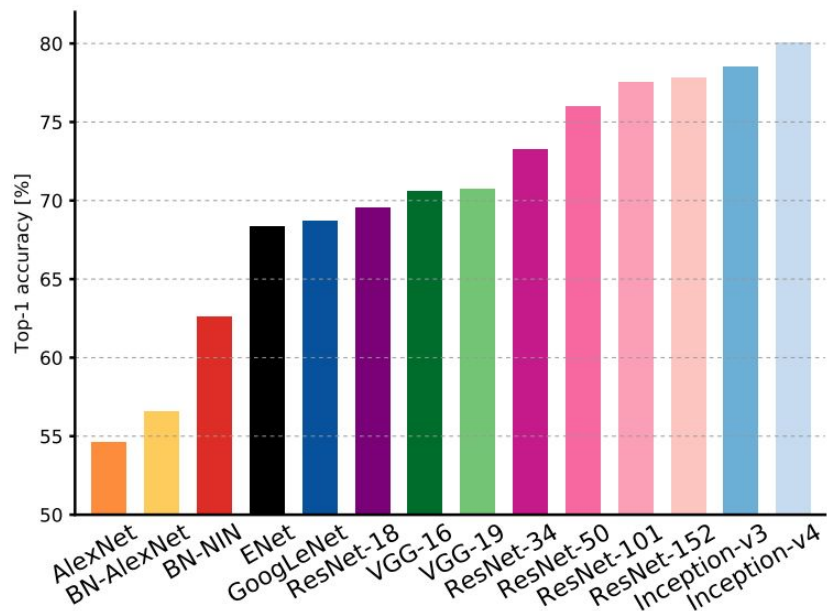
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

# Comparing complexity...



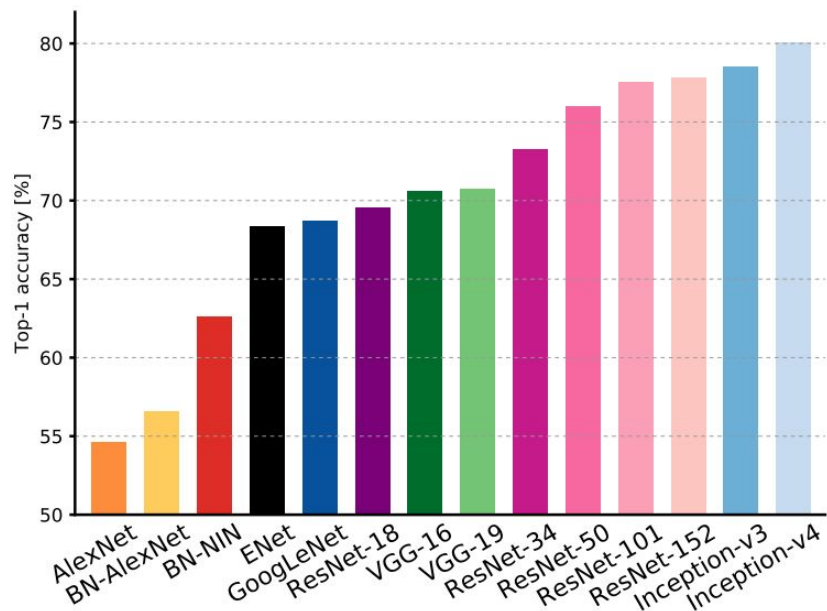
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

# Comparing complexity...

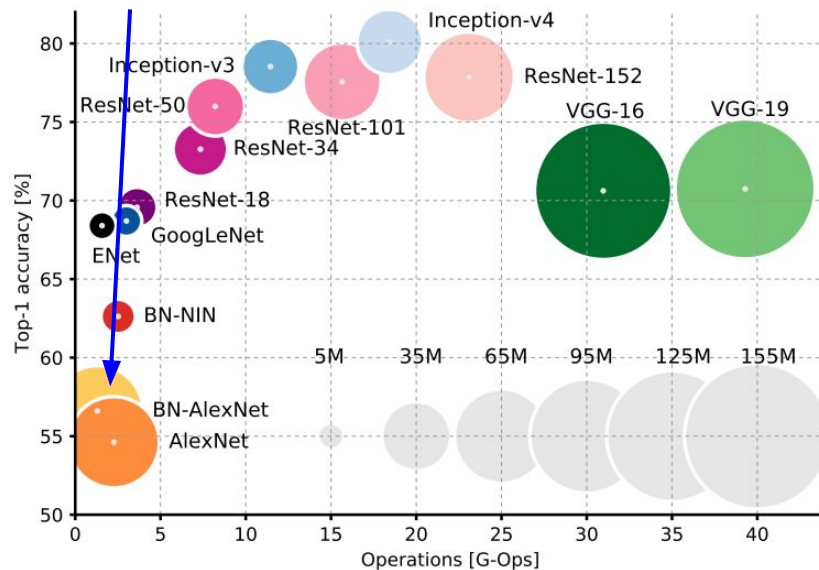


An Analysis of Deep Neural Network Models for Practical Applications, 2017.

# Comparing complexity...



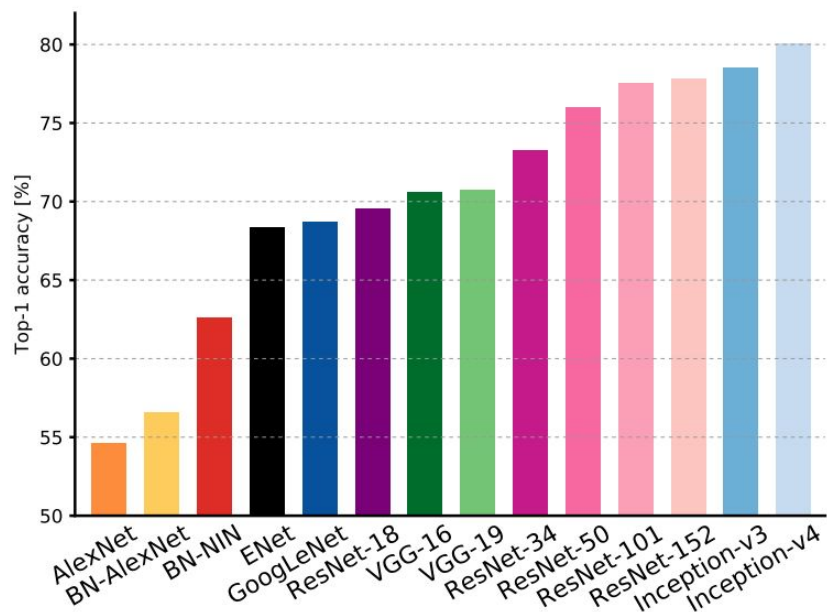
AlexNet:  
Smaller compute, still memory  
heavy, lower accuracy



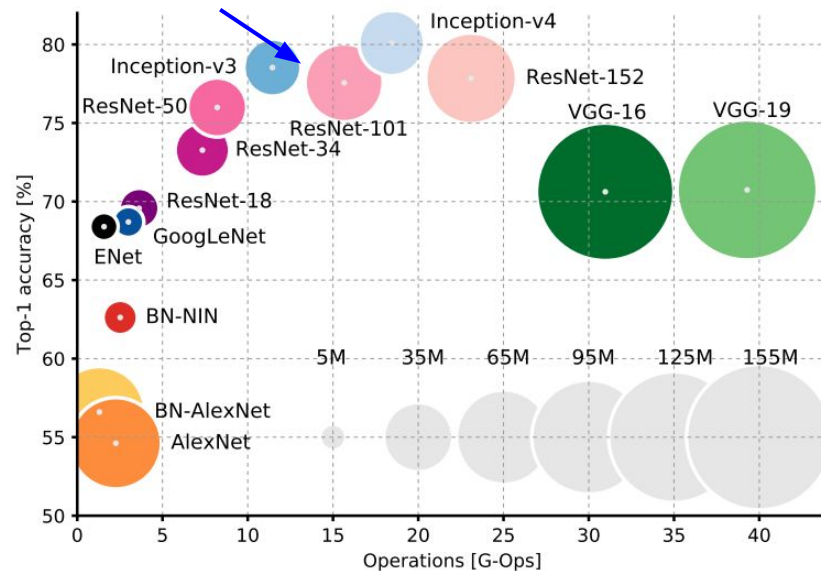
An Analysis of Deep Neural Network Models for Practical Applications, 2017.



# Comparing complexity...



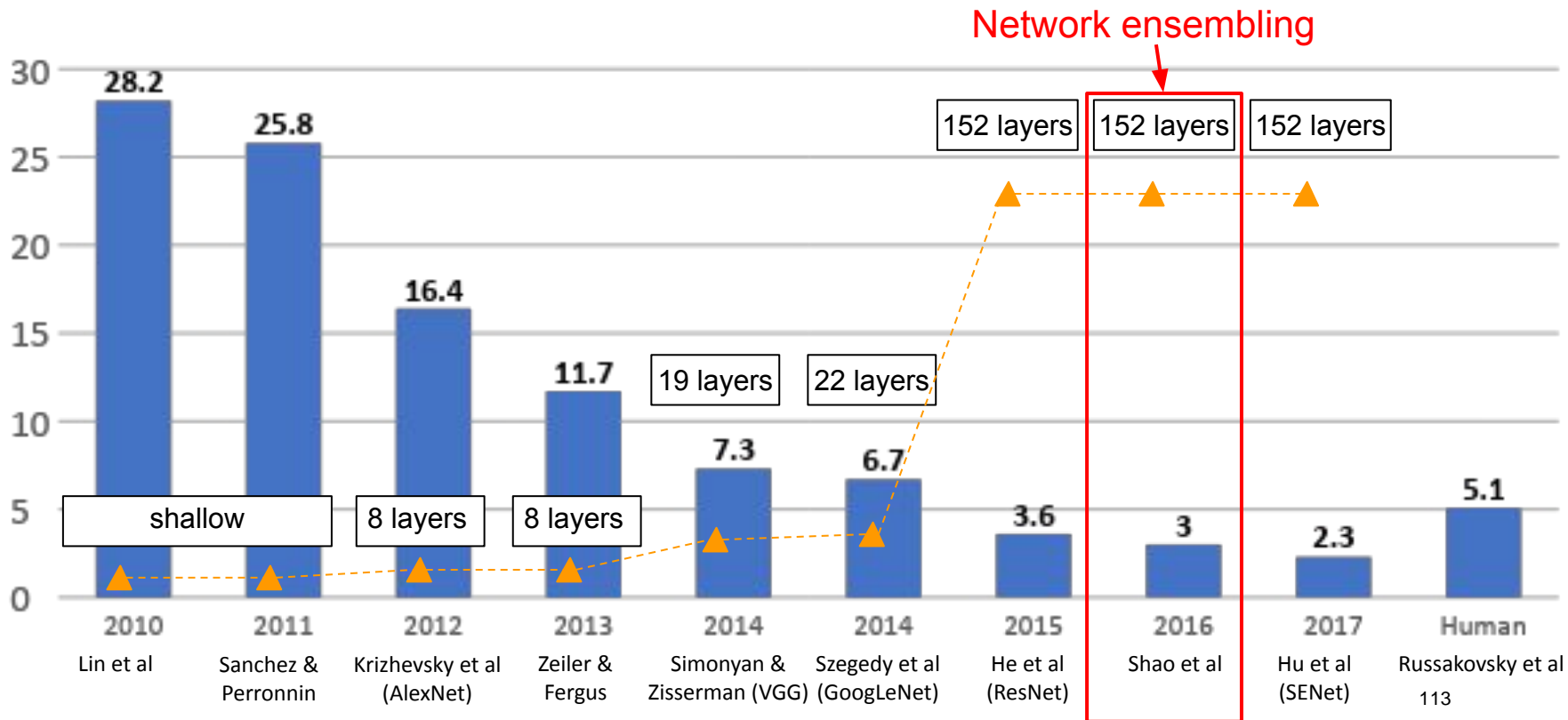
ResNet:  
Moderate efficiency depending on  
model, highest accuracy



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



# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



# Improving ResNets...

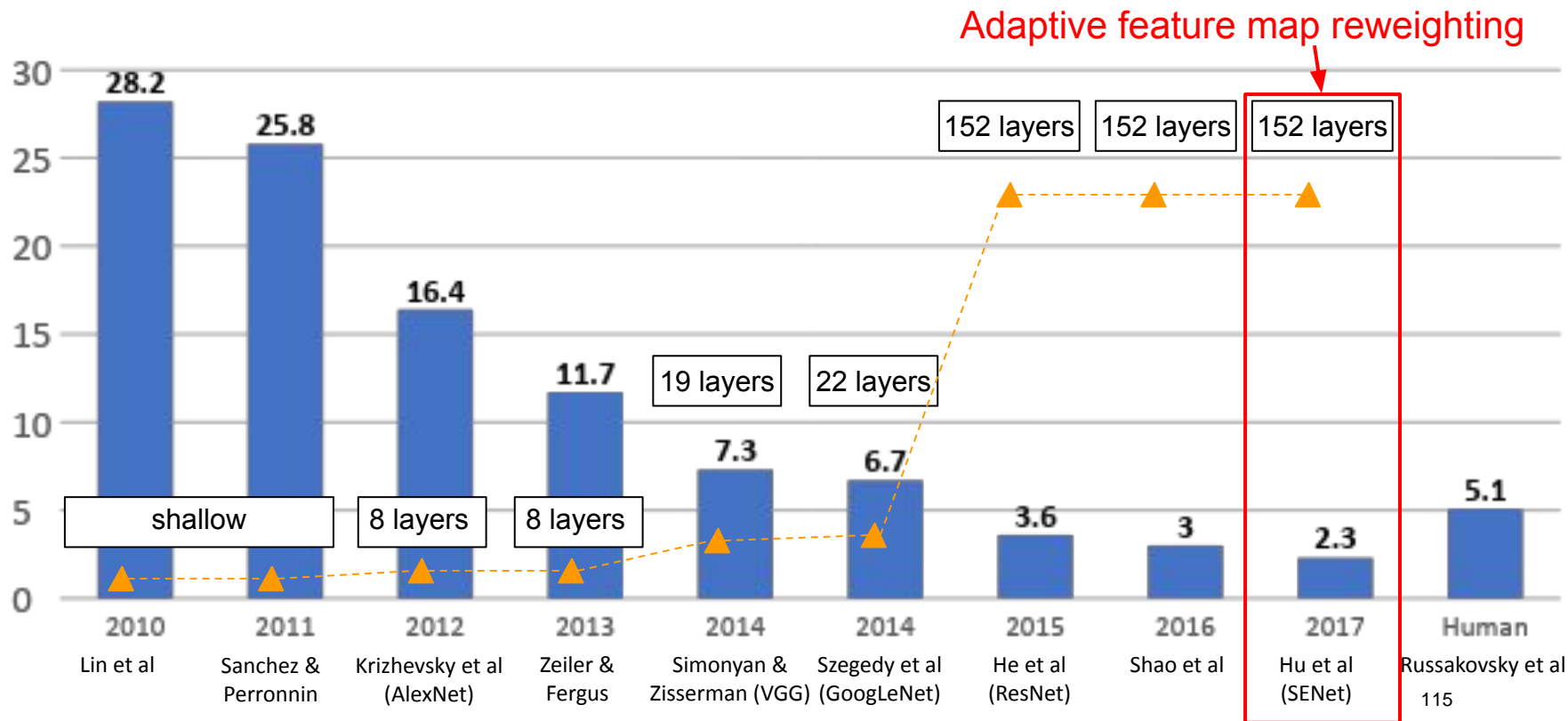
## “Good Practices for Deep Feature Fusion”

*[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

# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

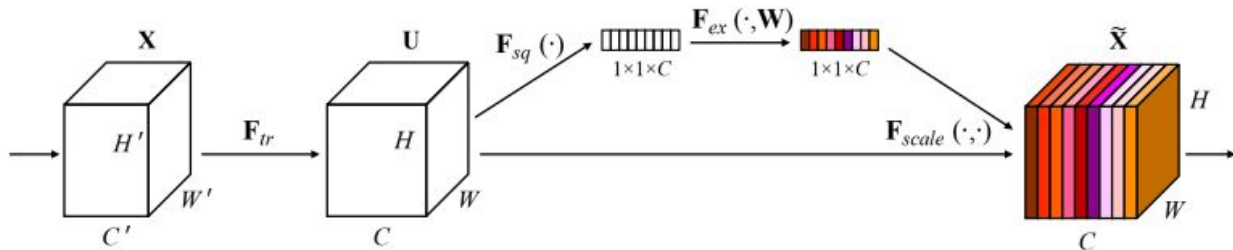
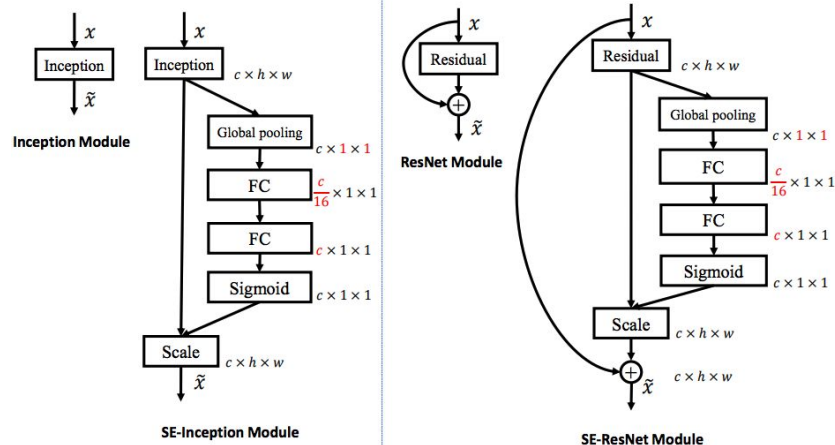


# Improving ResNets...

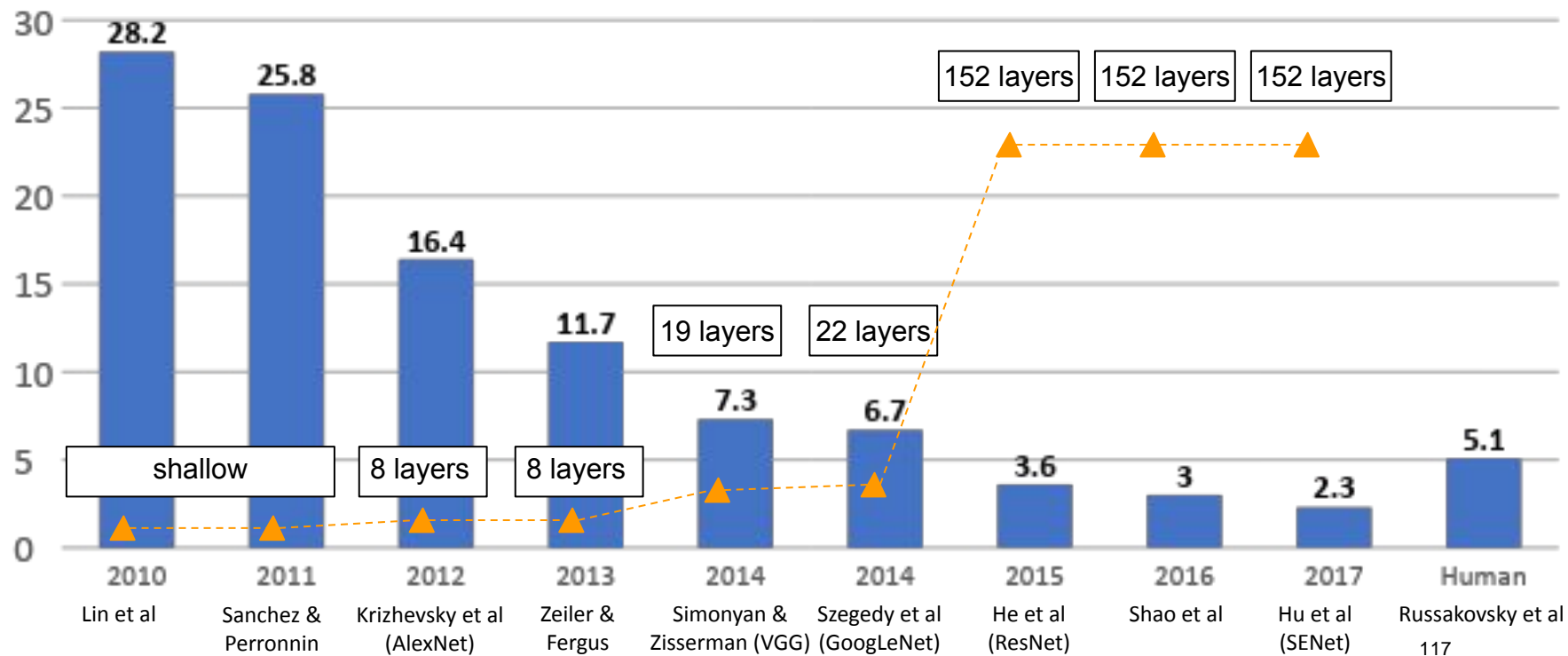
## Squeeze-and-Excitation Networks (SENet)

[Hu et al. 2017]

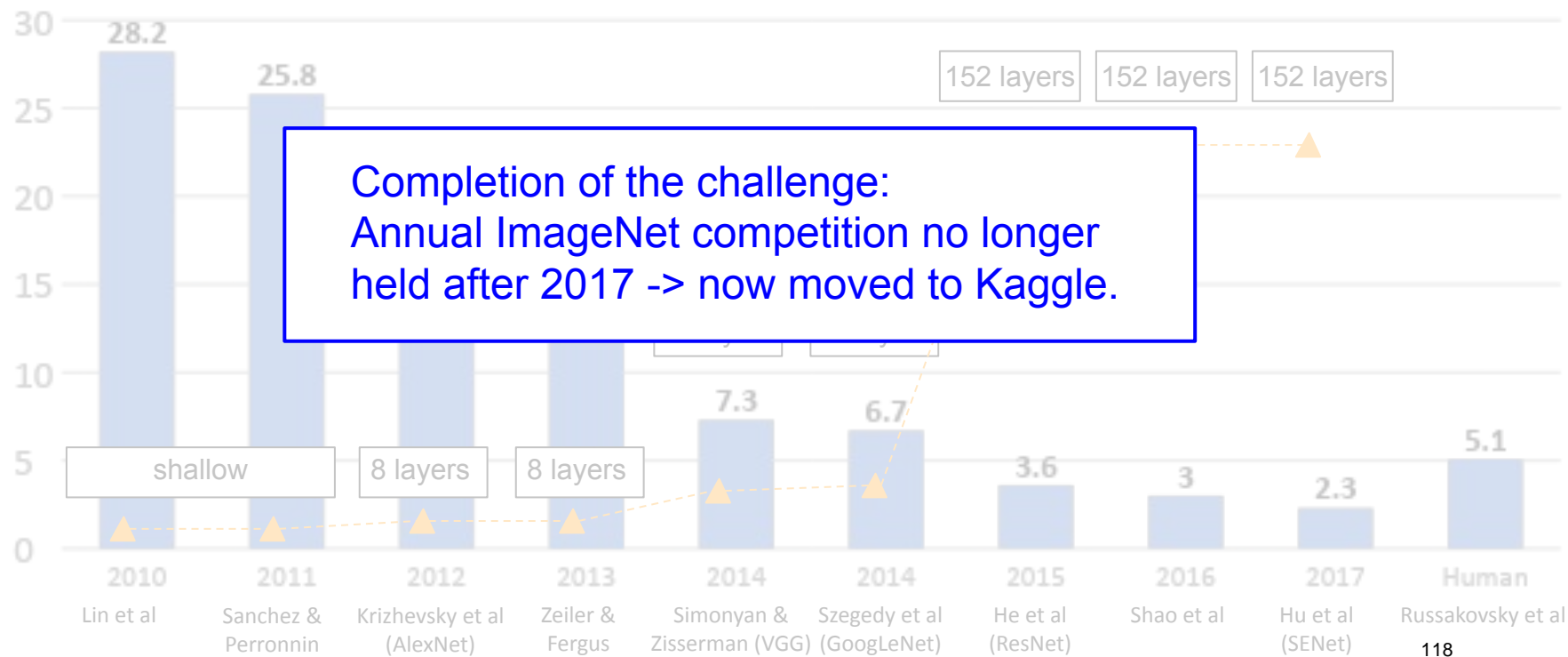
- Add a “feature recalibration” module that learns to adaptively reweight feature maps
- Global information (global avg. pooling layer) + 2 FC layers used to determine feature map weights
- ILSVRC'17 classification winner (using ResNeXt-152 as a base architecture)



# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



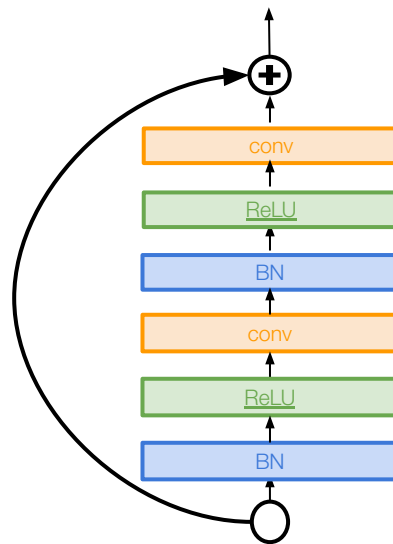
But research into CNN architectures is still flourishing

# Improving ResNets...

## Identity Mappings in Deep Residual Networks

[He et al. 2016]

- Improved ResNet block design from creators of ResNet
- Creates a more direct path for propagating information throughout network
- Gives better performance



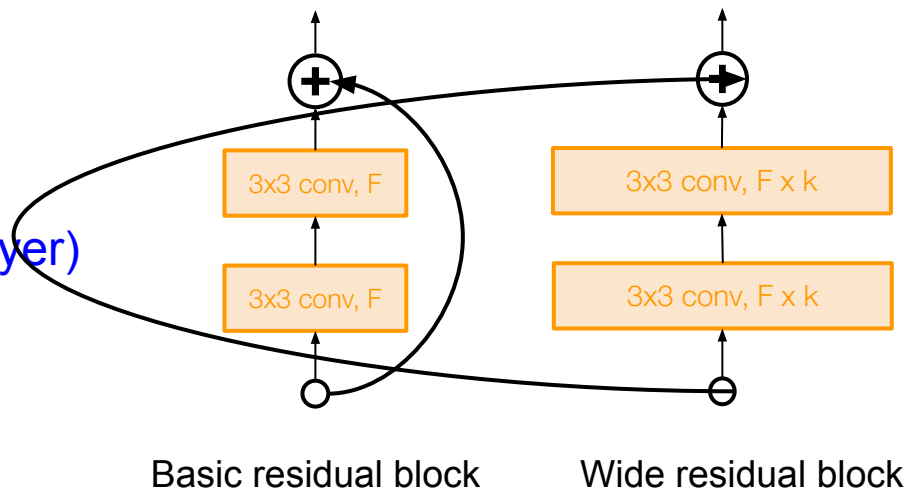


# Improving ResNets...

## Wide Residual Networks

[Zagoruyko et al. 2016]

- Argues that residuals are the important factor, not depth
- User 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)

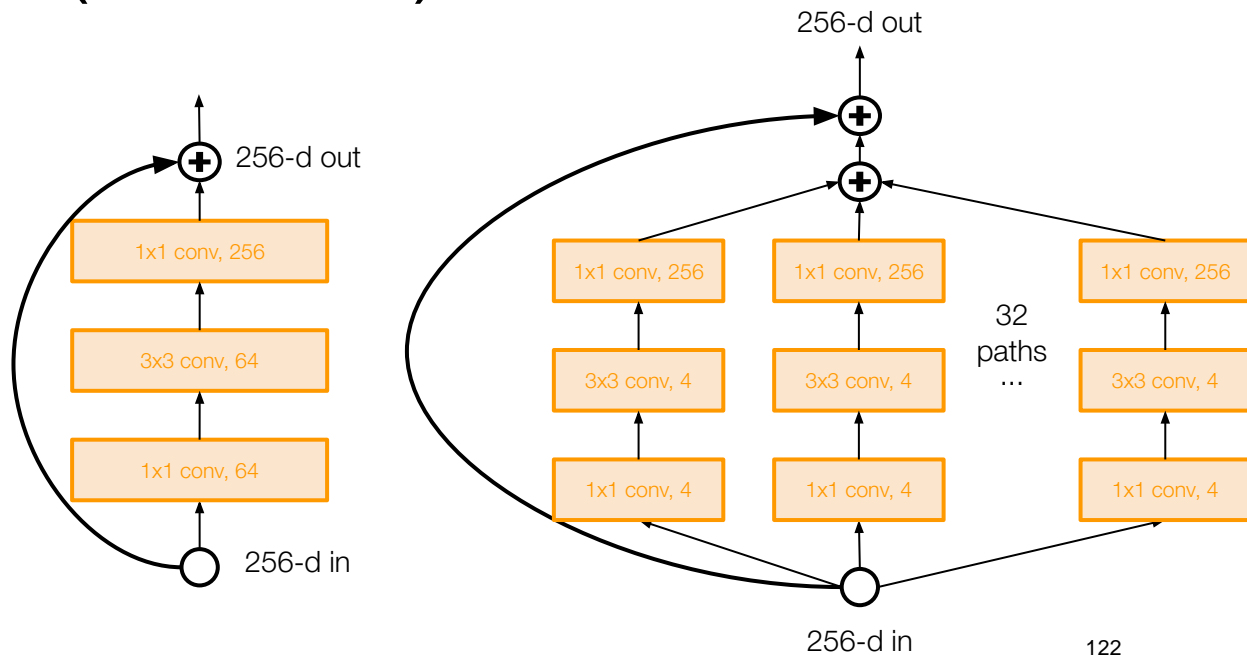


# Improving ResNets...

## Aggregated Residual Transformations for Deep Neural Networks (ResNeXt)

[Xie et al. 2016]

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

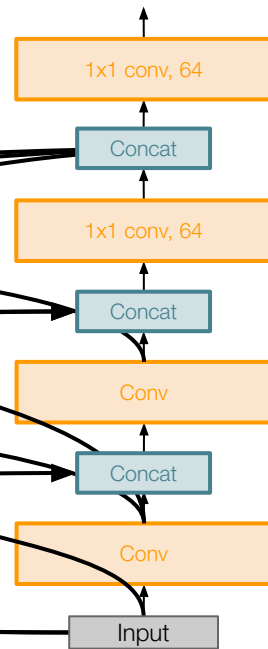


# Other ideas...

## Densely Connected Convolutional Networks (DenseNet)

[Huang et al. 2017]

- Dense blocks where each layer is connected to every other layer in feedforward fashion
- Alleviates vanishing gradient, strengthens feature propagation, encourages feature reuse
- Showed that shallow 50-layer network can outperform deeper 152 layer ResNet



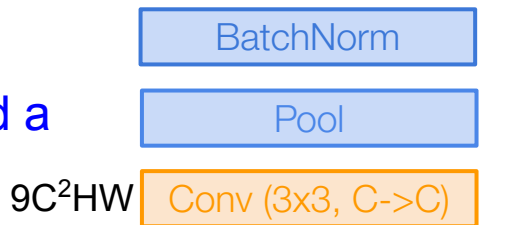
Dense Block



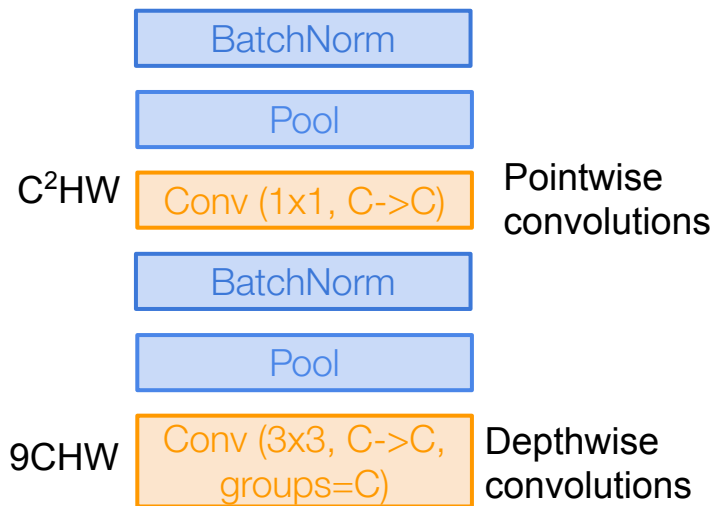
# Efficient networks...

## MobileNets: Efficient Convolutional Neural Networks for Mobile Applications *[Howard et al. 2017]*

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



Standard network  
Total compute:  $9C^2HW$



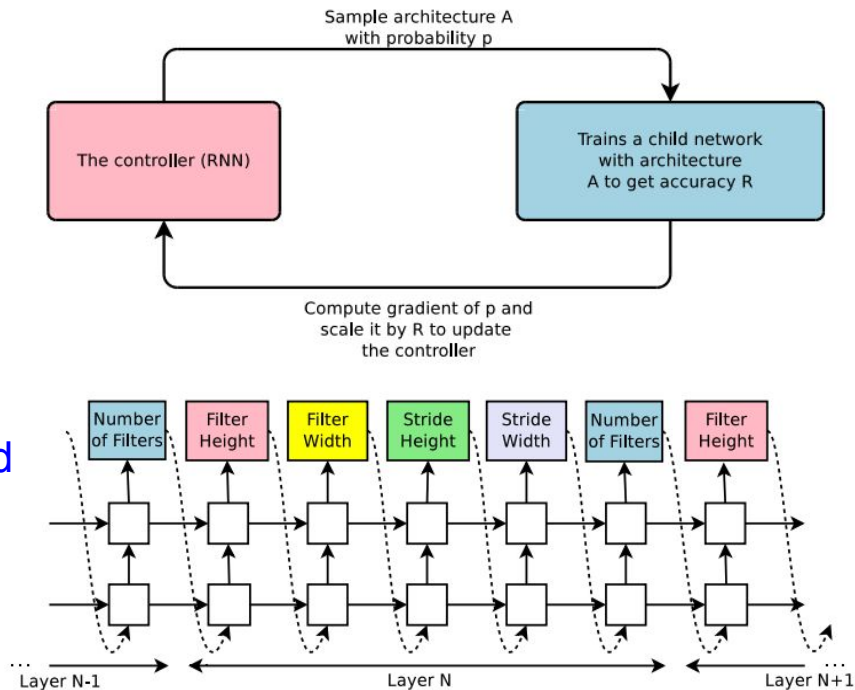
MobileNets  
Total compute:  $9CHW + C^{2,124}$

# Learning to search for network architectures...

## Neural Architecture Search with Reinforcement Learning (NAS)

[Zoph et al. 2016]

- “Controller” network that learns to design a good network architecture (output a string corresponding to network design)
- Iterate:
  - 1) Sample an architecture from search space
  - 2) Train the architecture to get a “reward”  $R$  corresponding to accuracy
  - 3) Compute gradient of sample probability, and scale by  $R$  to perform controller parameter update (i.e. increase likelihood of good architecture being sampled, decrease likelihood of bad architecture)

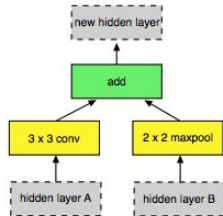
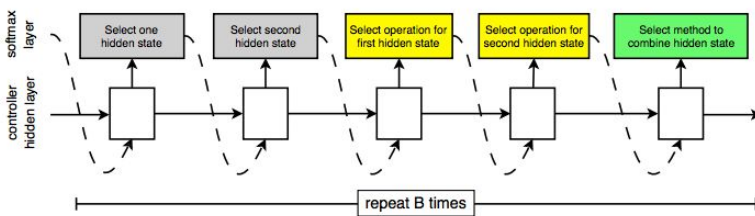
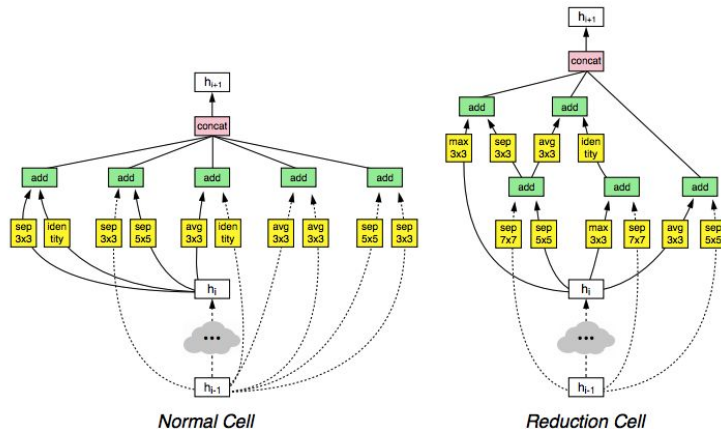


# Learning to search for network architectures...

## Learning Transferable Architectures for Scalable Image Recognition

[Zoph et al. 2017]

- Applying neural architecture search (NAS) to a large dataset like ImageNet is expensive
- Design a search space of building blocks (“cells”) that can be flexibly stacked
- NASNet: Use NAS to find best cell structure on smaller CIFAR-10 dataset, then transfer architecture to ImageNet
- Many follow-up works in this space e.g. AmoebaNet (Real et al. 2019) and ENAS (Pham, Guan et al. 2018)



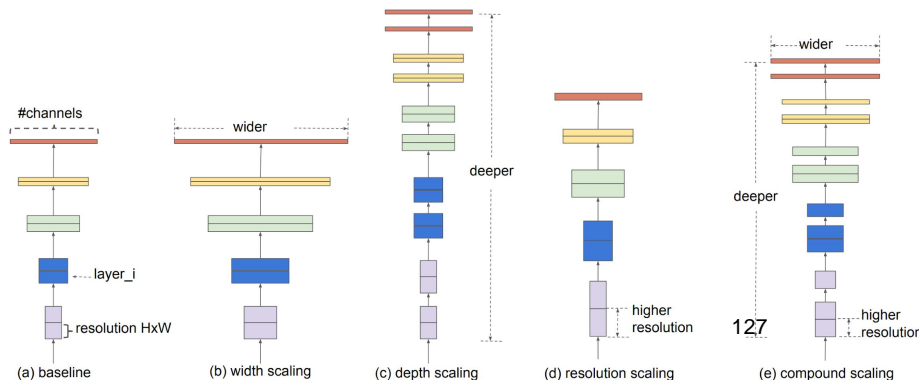
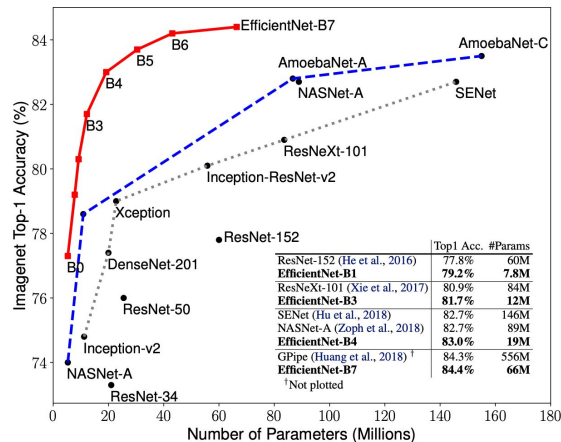
# But sometimes smart heuristic is better than NAS ...

## EfficientNet: Smart Compound Scaling

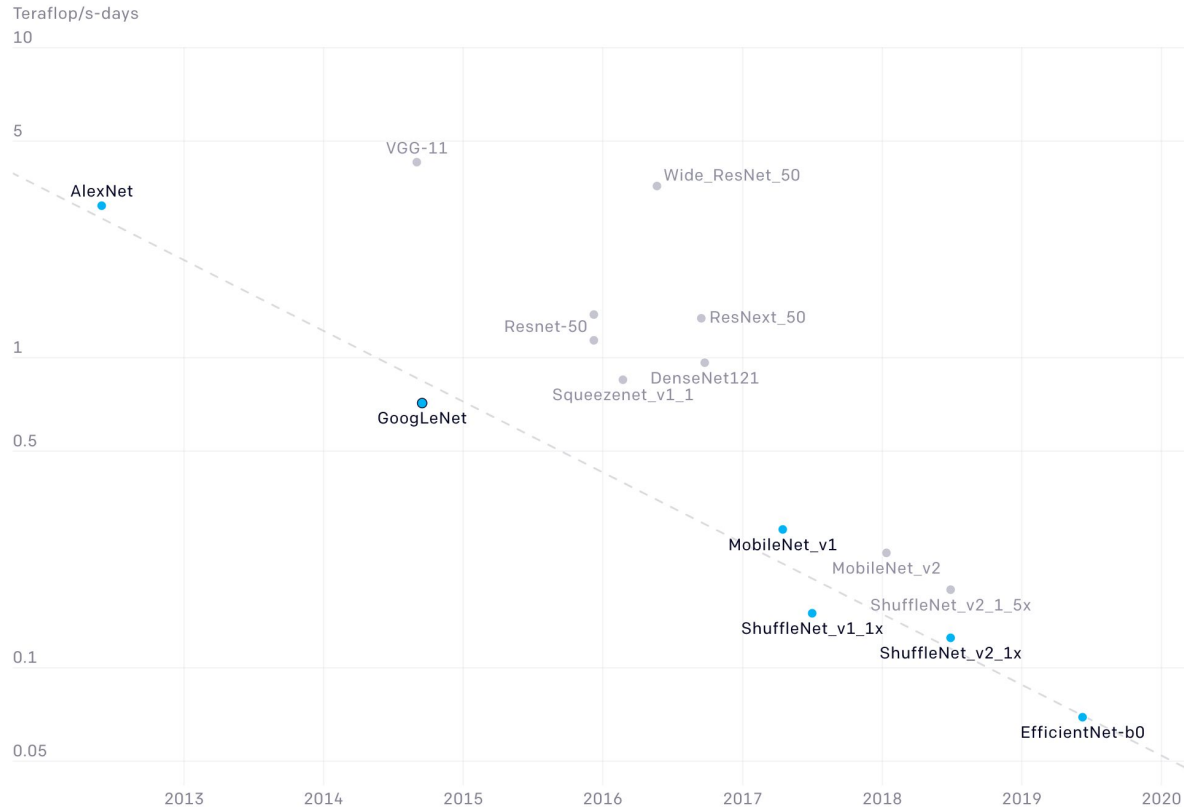
[Tan and Le. 2019]

- 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

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



# Efficient networks...



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



# Summary: CNN Architectures

## Case Studies

- AlexNet
- VGG
- GoogLeNet
- ResNet

## Also....

- SENet
- Wide ResNet
- ResNeXT
- DenseNet
- MobileNets
- NASNet

# Main takeaways

**AlexNet** showed that you can use CNNs to train Computer Vision models.

**ZFNet**, **VGG** shows that bigger networks work better

**GoogLeNet** is one of the first to focus on efficiency using 1x1 bottleneck convolutions and global avg pool instead of FC layers

**ResNet** showed us how to train extremely deep networks

- Limited only by GPU & memory!
- Showed diminishing returns as networks got bigger

After ResNet: CNNs were better than the human metric and focus shifted to Efficient networks:

- Lots of tiny networks aimed at mobile devices: **MobileNet**, **ShuffleNet**

**Neural Architecture Search** can now automate architecture design

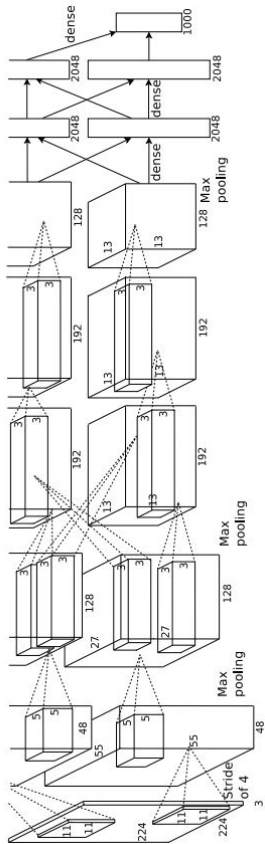
# Summary: CNN Architectures

- Many popular architectures are available in model zoos.
- ResNets are currently good defaults to use.
- Networks have gotten increasingly deep over time.
- Many other aspects of network architectures are also continuously being investigated and improved.

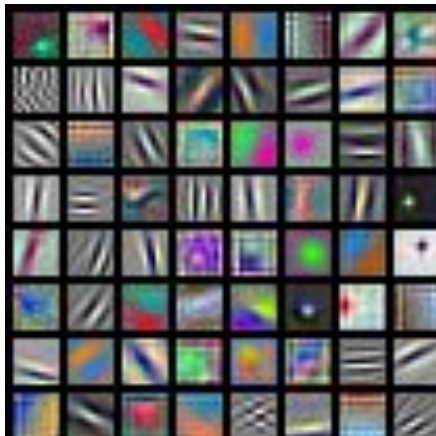
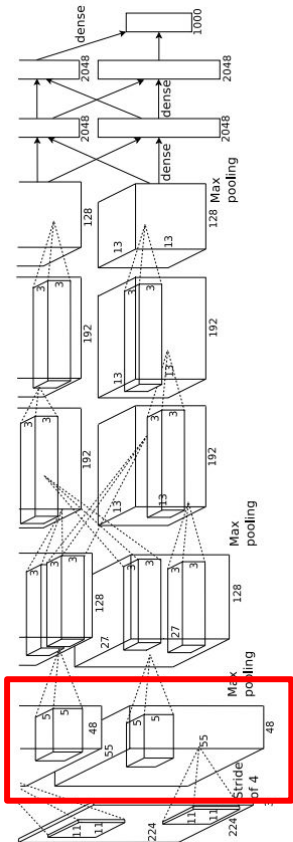
# Transfer learning

You need a lot of a data if you want to train/use CNNs?

# Transfer Learning with CNNs

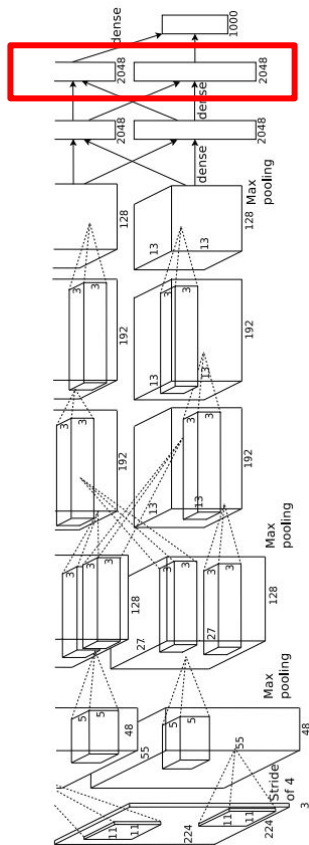


# Transfer Learning with CNNs

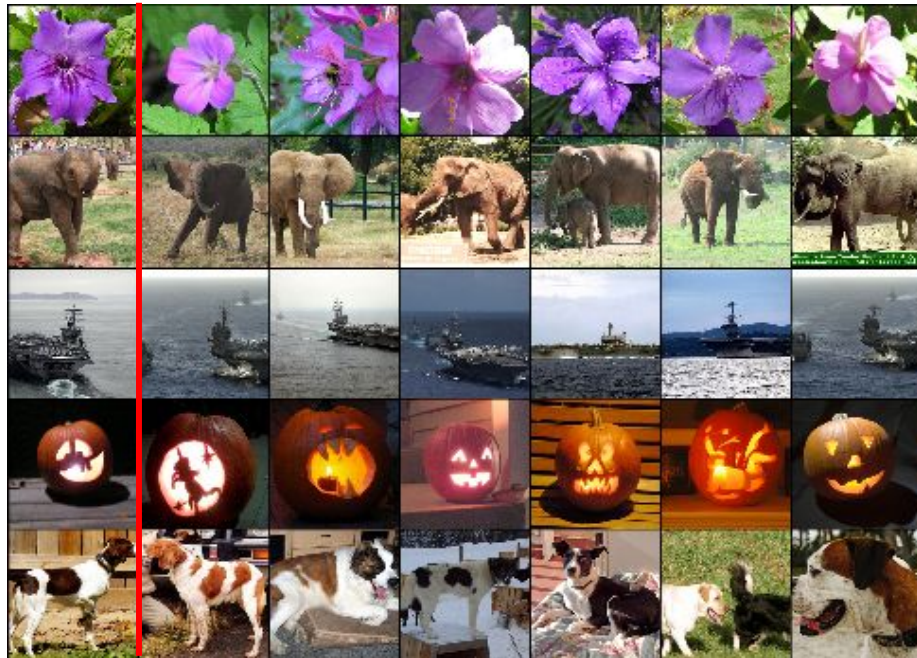


AlexNet:  
64 x 3 x 11 x 11

# Transfer Learning with CNNs



Test image    L2 Nearest neighbors in feature space





# Transfer Learning with CNNs

## 1. Train on Imagenet

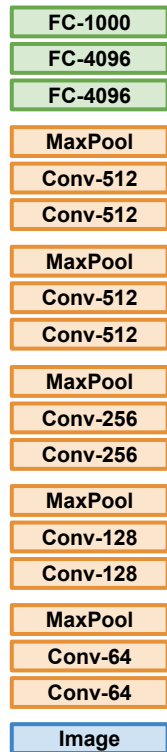


Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014  
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

# 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)



# 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

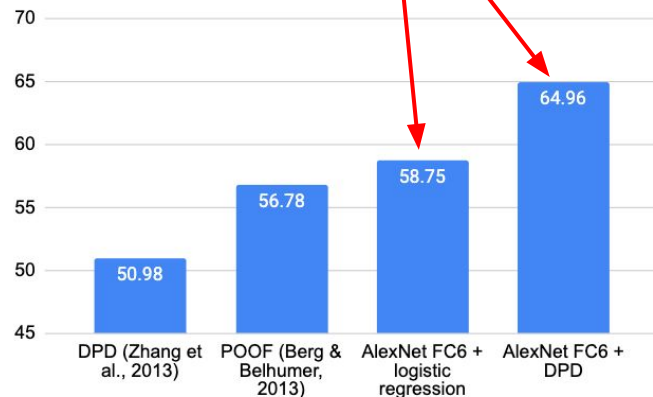
## 1. Train on Imagenet



## 2. Small Dataset (C classes)



Finetuned from AlexNet



Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

# 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

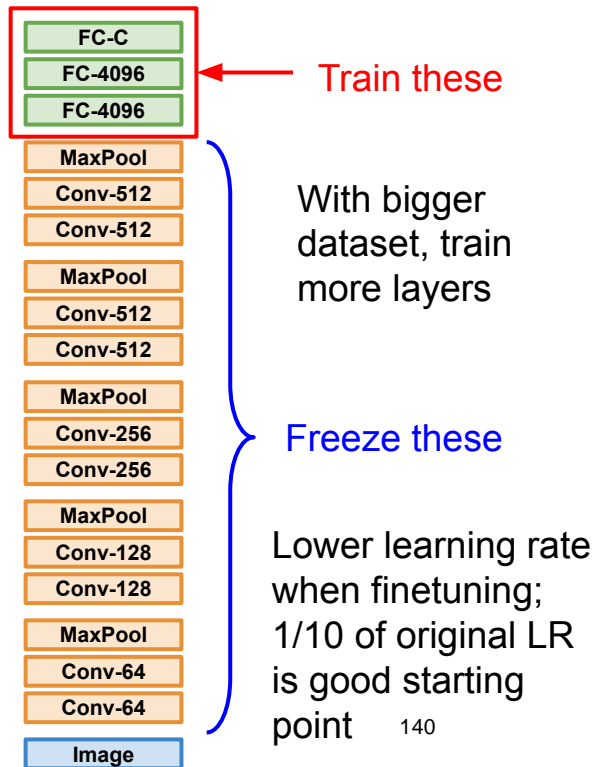
## 1. Train on Imagenet



## 2. Small Dataset (C classes)



## 3. Bigger dataset





More specific

More generic

	<b>very similar dataset</b>	<b>very different dataset</b>
<b>very little data</b>	?	?
<b>quite a lot of data</b>	?	?



More specific

More generic

	<b>very similar dataset</b>	<b>very different dataset</b>
<b>very little data</b>	Use Linear Classifier on top layer	?
<b>quite a lot of data</b>	Finetune a few layers	?



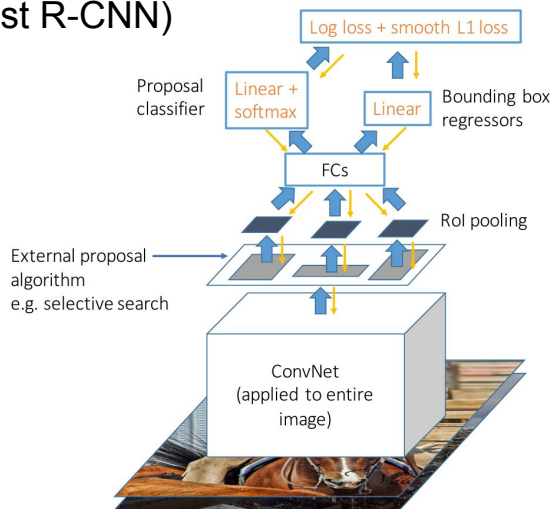
More specific

More generic

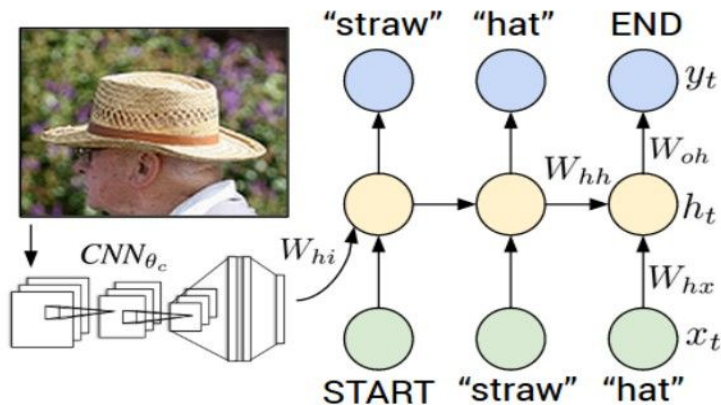
	<b>very similar dataset</b>	<b>very different dataset</b>
<b>very little data</b>	Use Linear Classifier on top layer	You're in trouble... Try linear classifier from different stages
<b>quite a lot of data</b>	Finetune a few layers	Finetune a larger number of layers

# Transfer learning with CNNs is pervasive... (it's the norm, not an exception)

## Object Detection (Fast R-CNN)



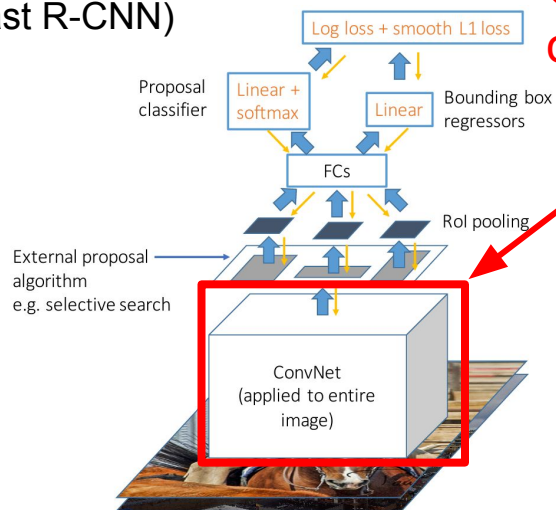
## Image Captioning: CNN + RNN





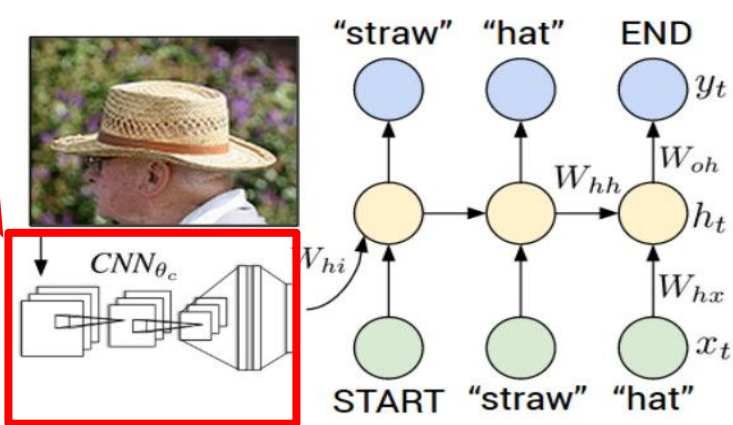
# Transfer learning with CNNs is pervasive... (it's the norm, not an exception)

## Object Detection (Fast R-CNN)



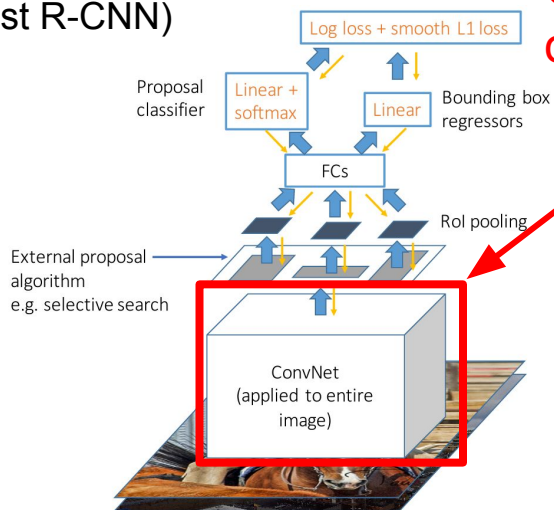
**CNN pretrained  
on ImageNet**

## Image Captioning: CNN + RNN



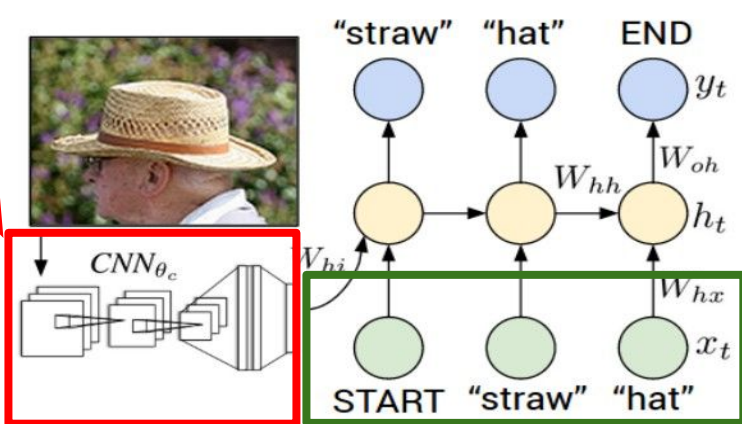
# Transfer learning with CNNs is pervasive... (it's the norm, not an exception)

## Object Detection (Fast R-CNN)



**CNN pretrained  
on ImageNet**

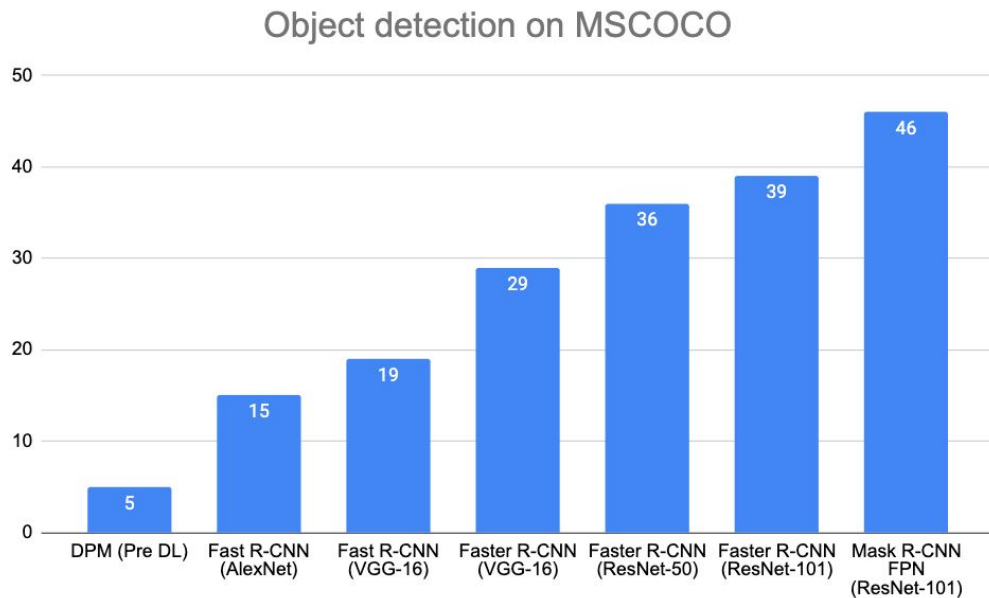
## Image Captioning: CNN + RNN



**Word vectors pretrained  
with word2vec**

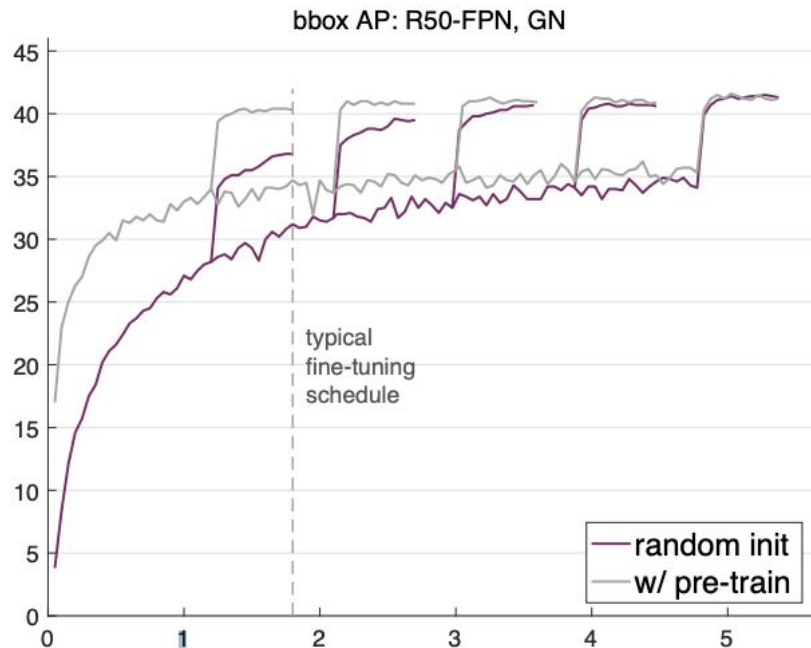
# Transfer learning with CNNs -

## Architecture matters



# Transfer learning with CNNs is pervasive...

## But recent results show it might not always be necessary!



Training from scratch can work just as well as training from a pretrained ImageNet model for object detection

But it takes 2-3x as long to train.

They also find that collecting more data is better than finetuning on a related task

# 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

TensorFlow: <https://github.com/tensorflow/models>

PyTorch: <https://github.com/pytorch/vision>

Next time: Training Neural Networks