

# **Convolutional Neural Networks**

## Part1

Jongbin Ryu

# Contents

- Introduction
- Convolutional Neural Networks (CNN)
- CNN Architectures

Example with an image with 4 pixels, and 3 classes (**cat/dog/ship**)



3 Classes



Cat?

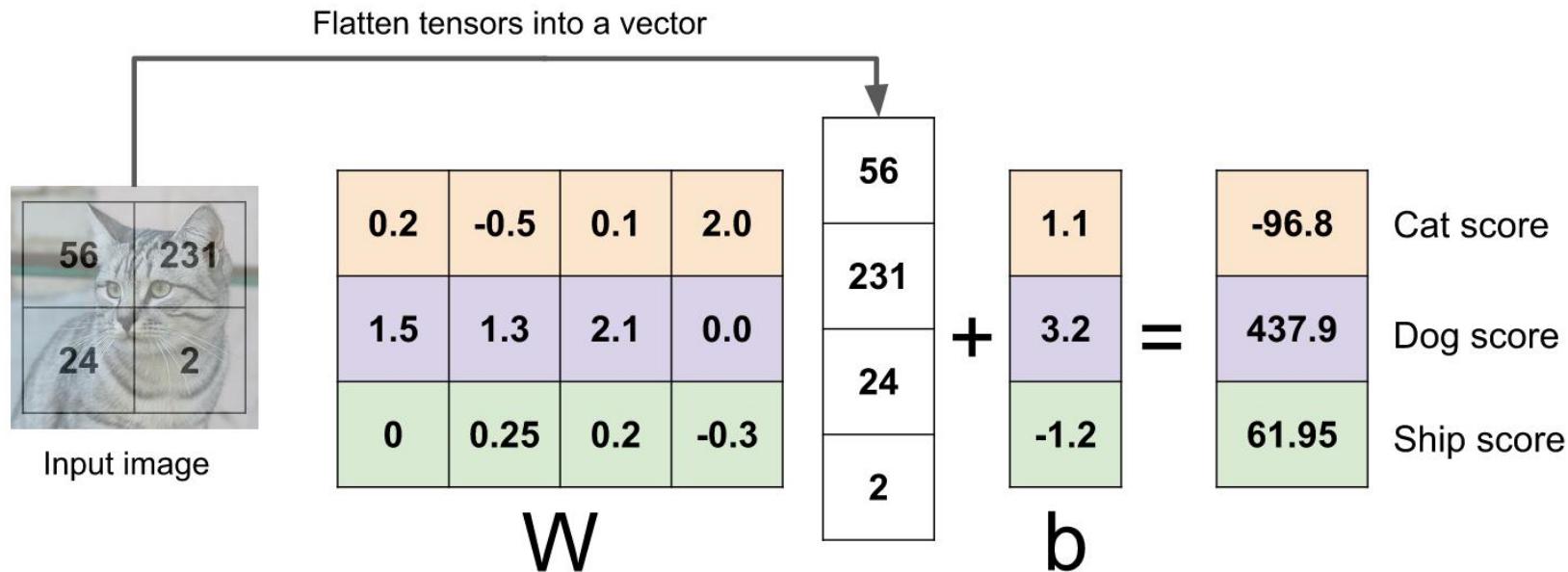


Dog?



Ship?

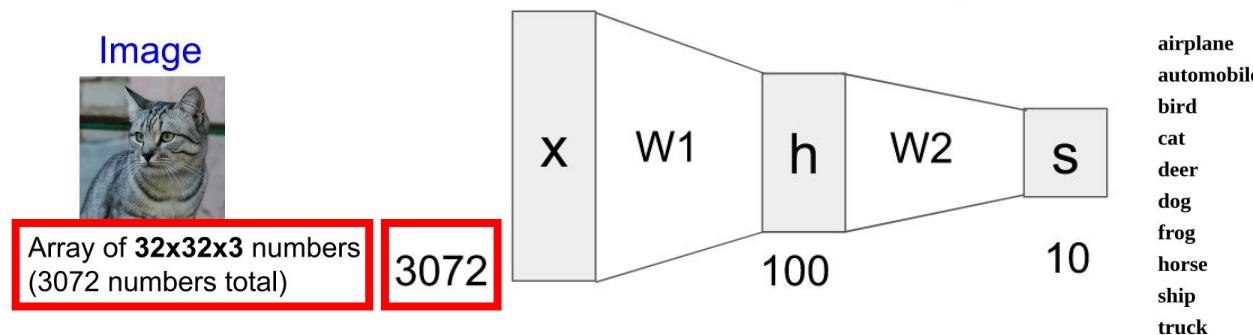
# Example with an image with 4 pixels, and 3 classes (cat/dog/ship)



# Neural networks: without the brain stuff

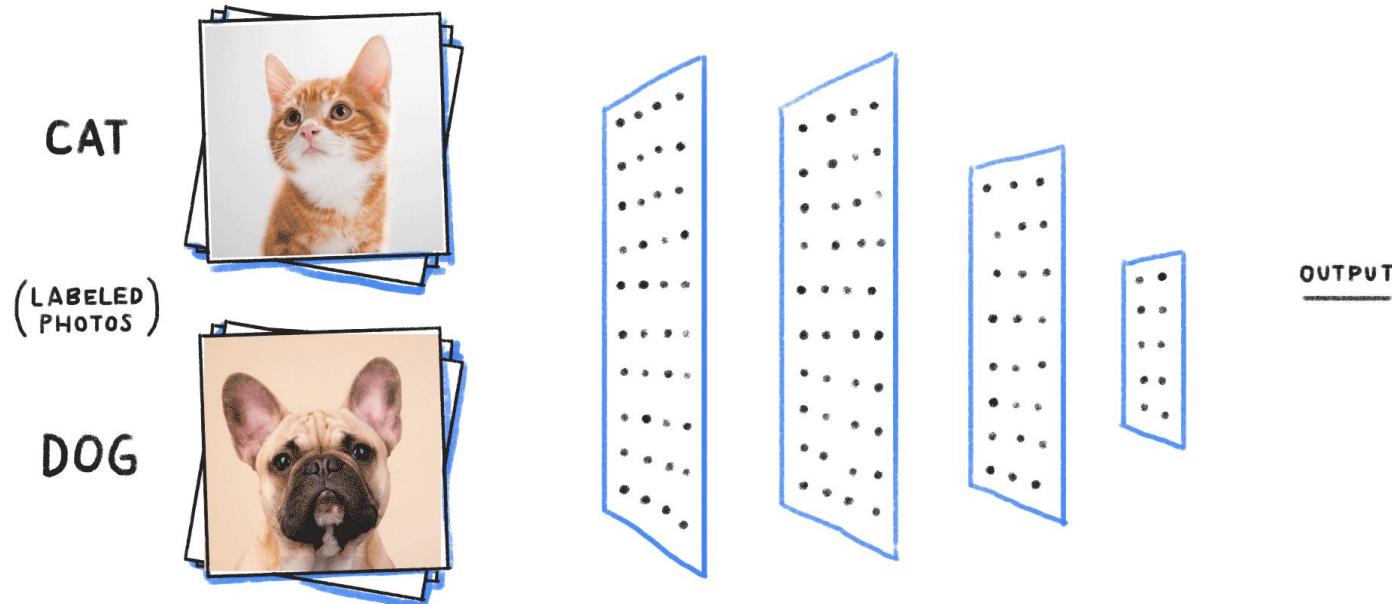
(Before) Linear score function:  $f = Wx$

(Now) 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$



$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

# Next: Convolutional Neural Networks



## A bit of history...

The **Mark I Perceptron** machine was the first implementation of the perceptron algorithm.

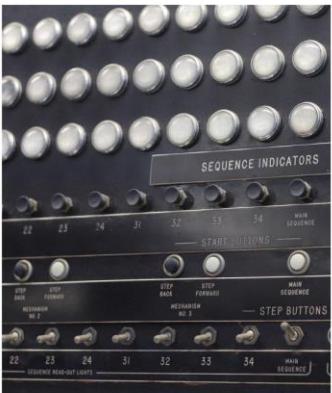
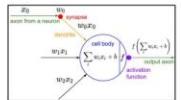
The machine was connected to a camera that used  $20 \times 20$  cadmium sulfide photocells to produce a 400-pixel image.

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

recognized letters of the alphabet

update rule:

$$w_i(t+1) = w_i(t) + \alpha(d_j - y_j(t))x_{j,i}$$



This image by Rocky Acosta is licensed under CC BY 3.0

Frank Rosenblatt, ~1957: Perceptron

## A bit of history...

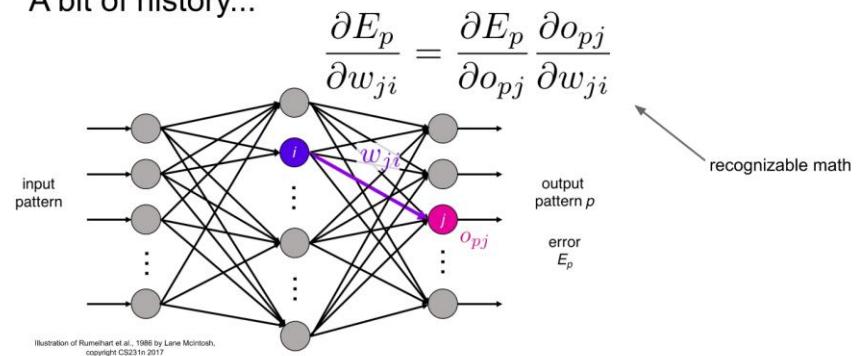
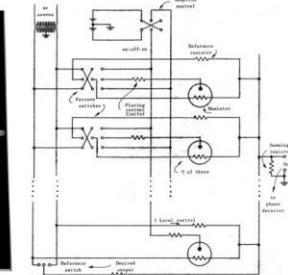
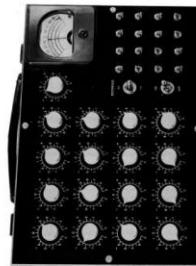
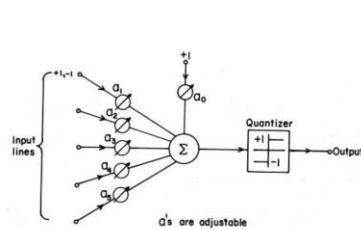


Illustration of Rumelhart et al., 1986 by Lane McIntosh, copyright CS231n 2017

Rumelhart et al., 1986: First time back-propagation became popular

## A bit of history...



These figures are reproduced from Widrow, 1960, Stanford Electronics Laboratories Technical Project with permission from Stanford University Special Collections.

Widrow and Hoff, ~1960: Adaline/Madaline

## A bit of history...

[Hinton and Salakhutdinov 2006]

Reinvigorated research in Deep Learning

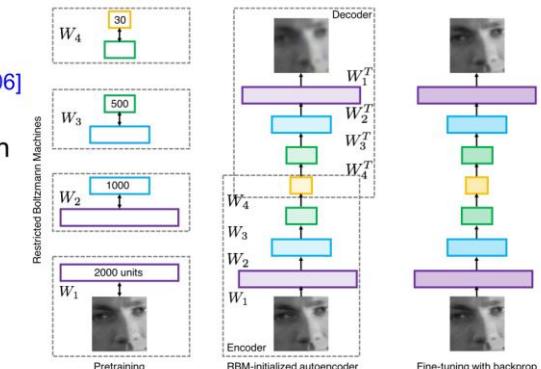


Illustration of Hinton and Salakhutdinov 2006 by Lane McIntosh, copyright CS231n 2017

# A bit of history:

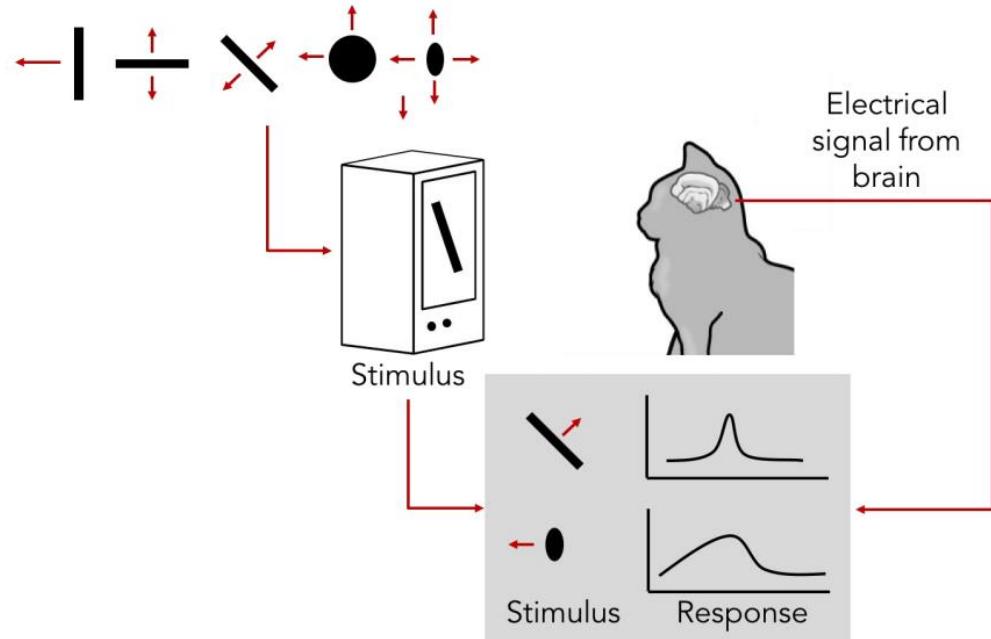
**Hubel & Wiesel,  
1959**

RECEPTIVE FIELDS OF SINGLE  
NEURONES IN  
THE CAT'S STRIATE CORTEX

**1962**

RECEPTIVE FIELDS, BINOCULAR  
INTERACTION  
AND FUNCTIONAL ARCHITECTURE IN  
THE CAT'S VISUAL CORTEX

**1968...**



Cat image by CNX OpenStax is licensed under CC BY 4.0; changes made

# Hierarchical organization

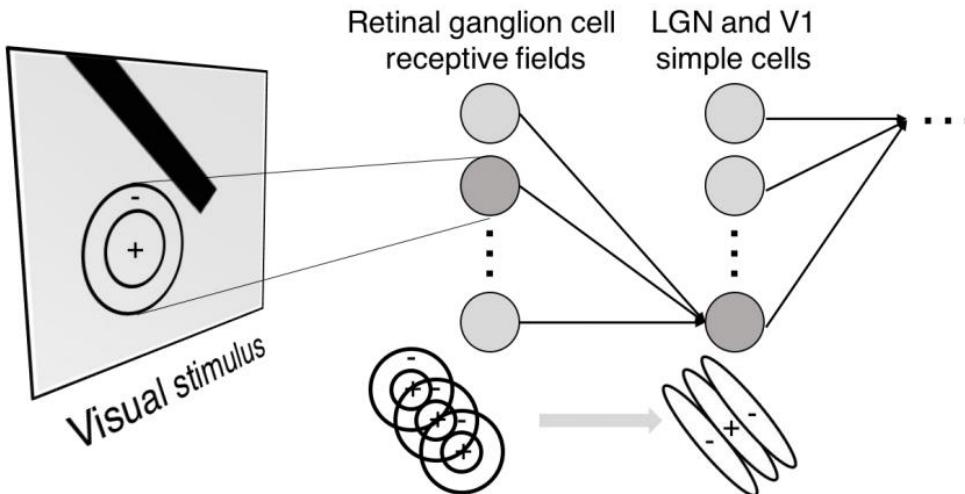


Illustration of hierarchical organization in early visual pathways by Lane McIntosh, copyright CS231n 2017

**Simple cells:**  
Response to light orientation

**Complex cells:**  
Response to light orientation and movement

**Hypercomplex cells:**  
response to movement with an end point



No response

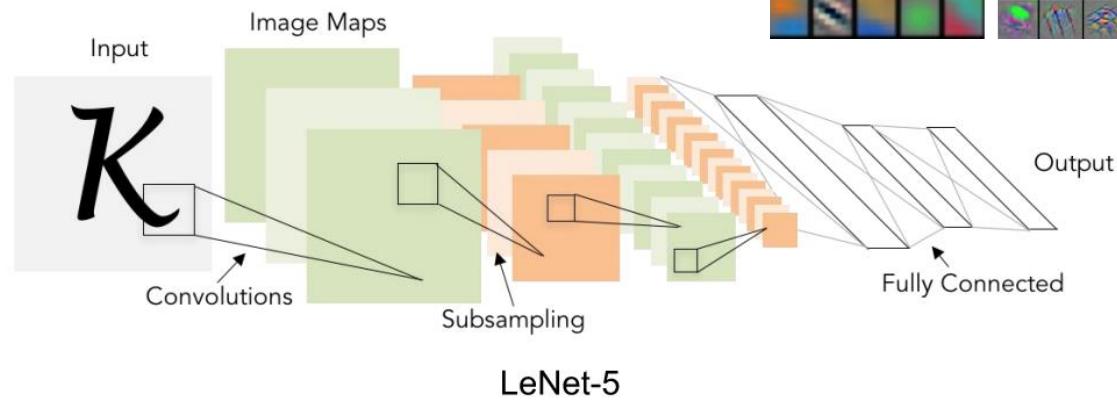
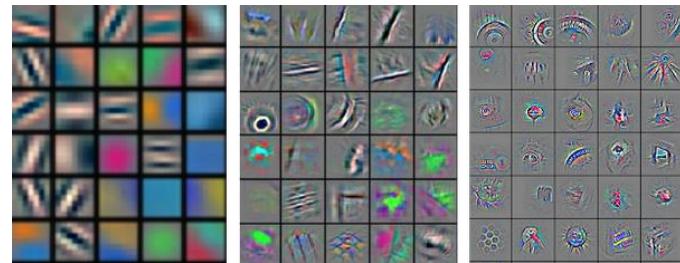


Response  
(end point)



# A bit of history: Gradient-based learning applied to document recognition

[LeCun, Bottou, Bengio, Haffner 1998]



# A bit of history:

## ImageNet Classification with Deep Convolutional Neural Networks

[Krizhevsky, Sutskever, Hinton, 2012]



*First strong result*

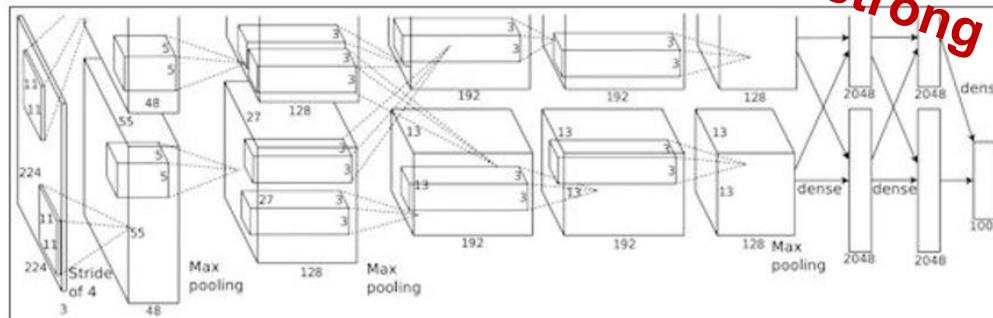
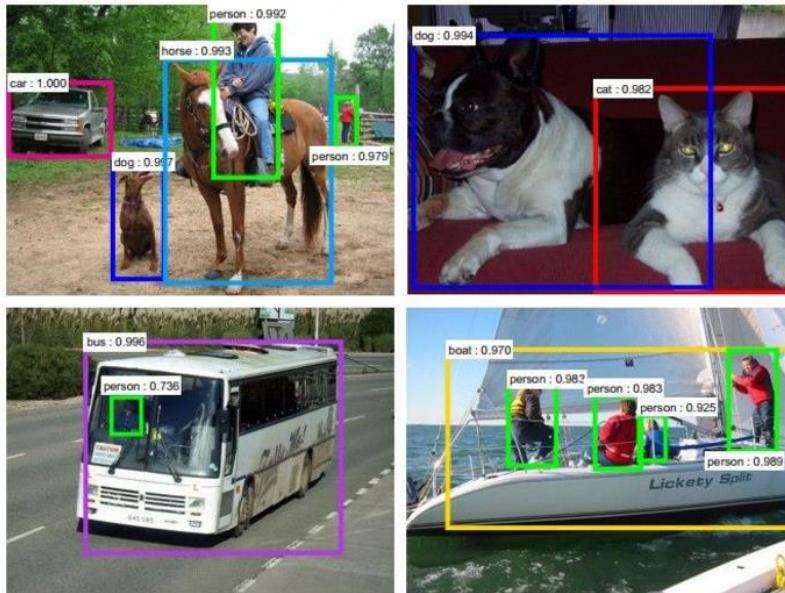


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

## “AlexNet”

# Fast-forward to today: ConvNets are everywhere

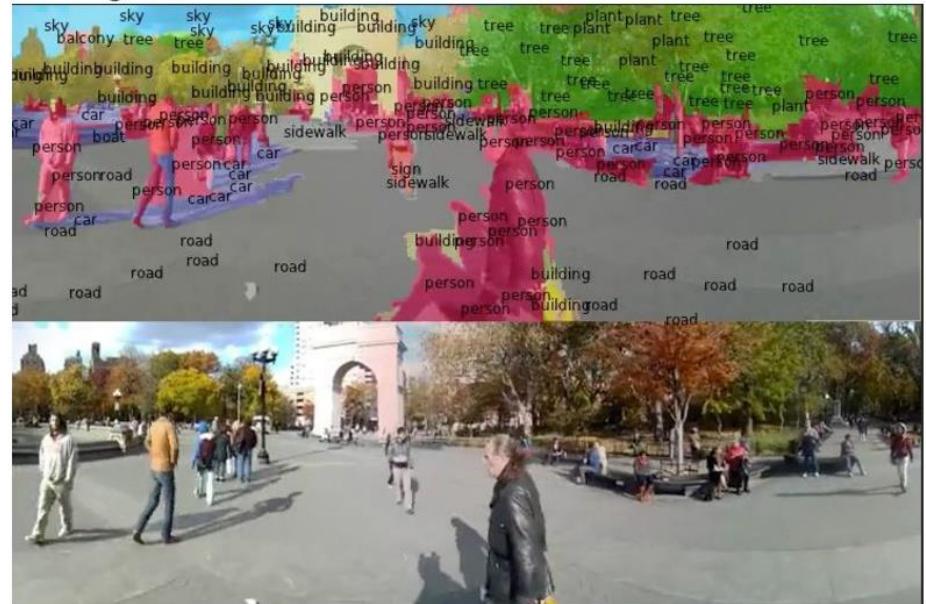
## Detection



Figures copyright Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun, 2015. Reproduced with permission.

[Faster R-CNN: Ren, He, Girshick, Sun 2015]

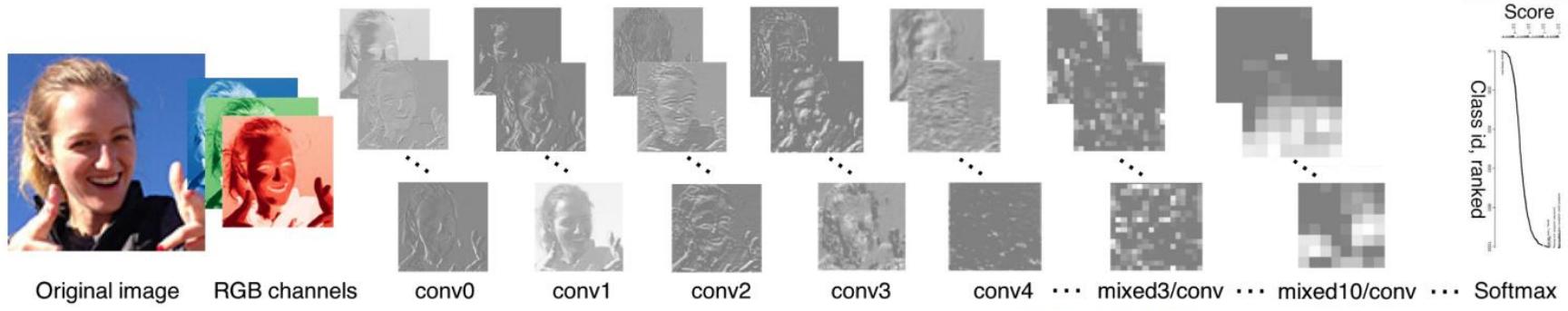
## Segmentation



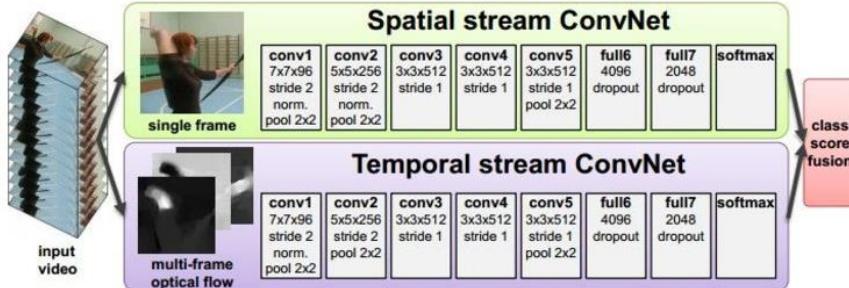
Figures copyright Clement Farabet, 2012.  
Reproduced with permission.

[Farabet et al., 2012]

# Fast-forward to today: ConvNets are everywhere



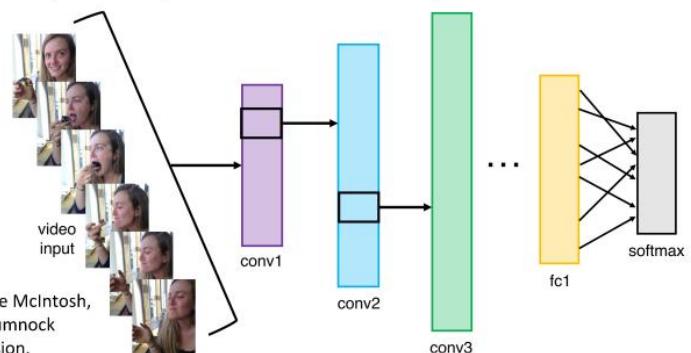
Activations of [inception-v3 architecture](#) [Szegedy et al. 2015] to image of Emma McIntosh, used with permission. Figure and architecture not from Taigman et al. 2014.



[Simonyan et al. 2014]

Figures copyright Simonyan et al., 2014.  
Reproduced with permission.

Illustration by Lane McIntosh,  
photos of Katie Cumnock  
used with permission.

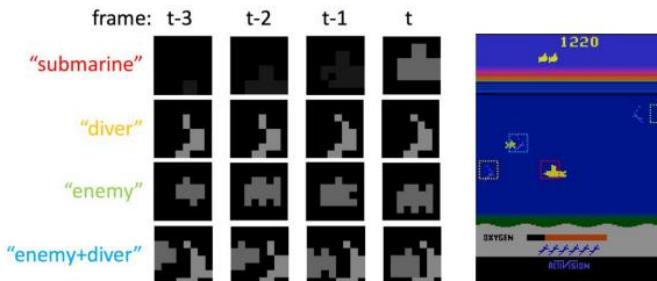


# Fast-forward to today: ConvNets are everywhere



Images are examples of pose estimation, not actually from Toshev & Szegedy 2014. Copyright Lane McIntosh.

[Toshev, Szegedy 2014]



[Guo et al. 2014]

Figures copyright Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard Lewis, and Xiaoshi Wang, 2014. Reproduced with permission.

## No errors



*A white teddy bear sitting in the grass*



*A man riding a wave on top of a surfboard*

## Minor errors



*A man in a baseball uniform throwing a ball*



*A cat sitting on a suitcase on the floor*

## Somewhat related



*A woman is holding a cat in her hand*



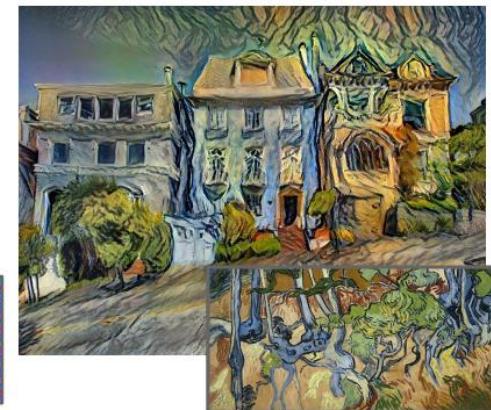
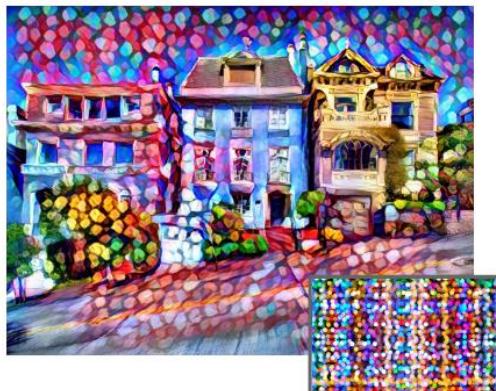
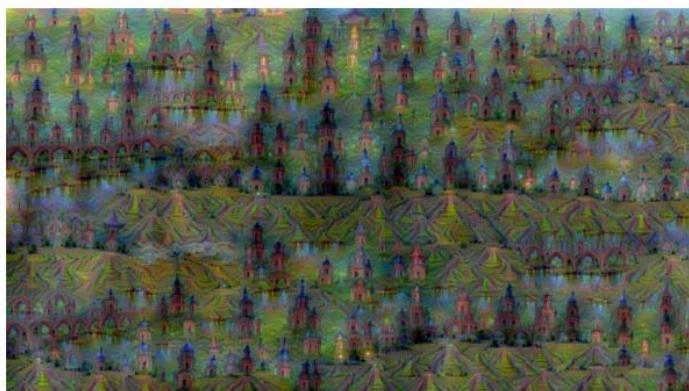
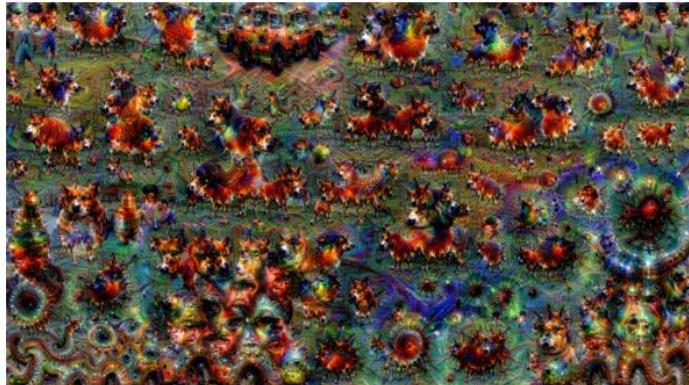
*A woman standing on a beach holding a surfboard*

# Image Captioning

[Vinyals et al., 2015]  
[Karpathy and Fei-Fei, 2015]

All images are CC0 Public domain:  
<https://pixabay.com/en/luggage-antique-cat-1643010/>  
<https://pixabay.com/en/teddy-plush-bears-cute-teddy-bear-1623436/>  
<https://pixabay.com/en/surf-wave-summer-sport-literal-1668716/>  
<https://pixabay.com/en/woman-female-model-portrait-adult-983967/>  
<https://pixabay.com/en/handstand-lake-meditation-496008/>  
<https://pixabay.com/en/baseball-player-shortstop-infield-1045263/>

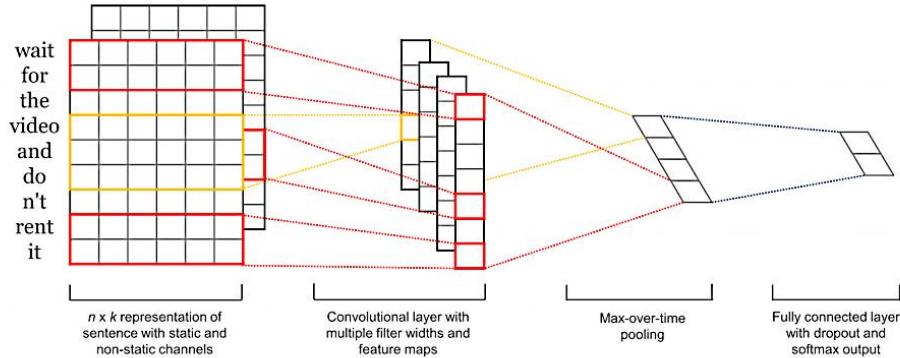
Captions generated by Justin Johnson using [Neuraltalk2](#)



Figures copyright Justin Johnson, 2015. Reproduced with permission. Generated using the Inceptionism approach from a [blog post](#) by Google Research.

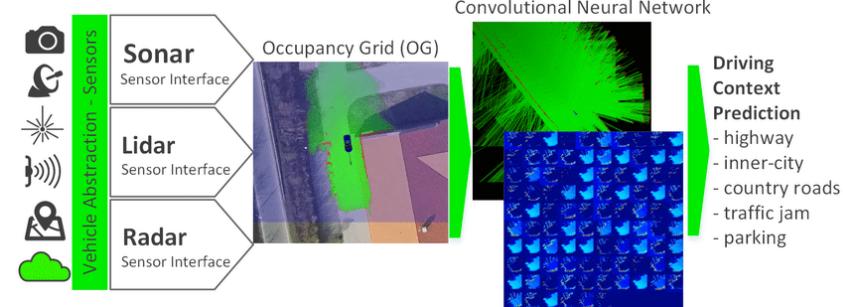
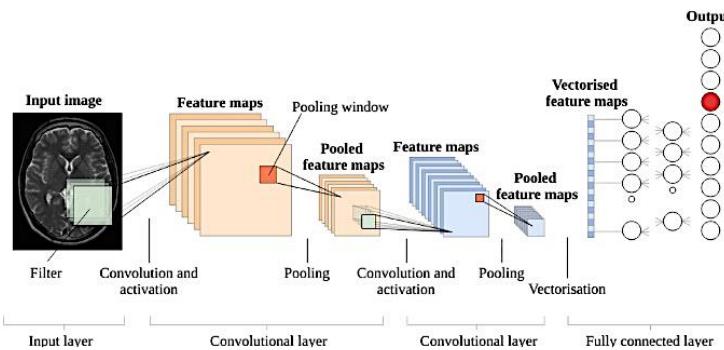
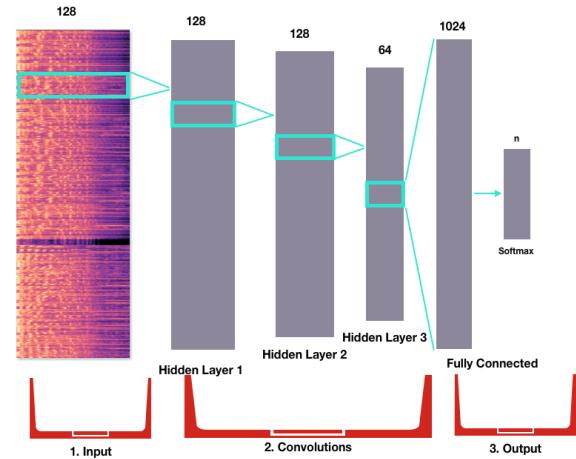
[Original image](#) is CC0 public domain  
[Starry Night](#) and [Tree Roots](#) by Van Gogh are in the public domain  
[Bokeh image](#) is in the public domain  
Stylized images copyright Justin Johnson, 2017;  
reproduced with permission

Gatys et al, "Image Style Transfer using Convolutional Neural Networks", CVPR 2016  
Gatys et al, "Controlling Perceptual Factors in Neural Style Transfer", CVPR 2017



Cited by 8664

Kim, Yoon. "Convolutional Neural Networks for Sentence Classification." *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2014.

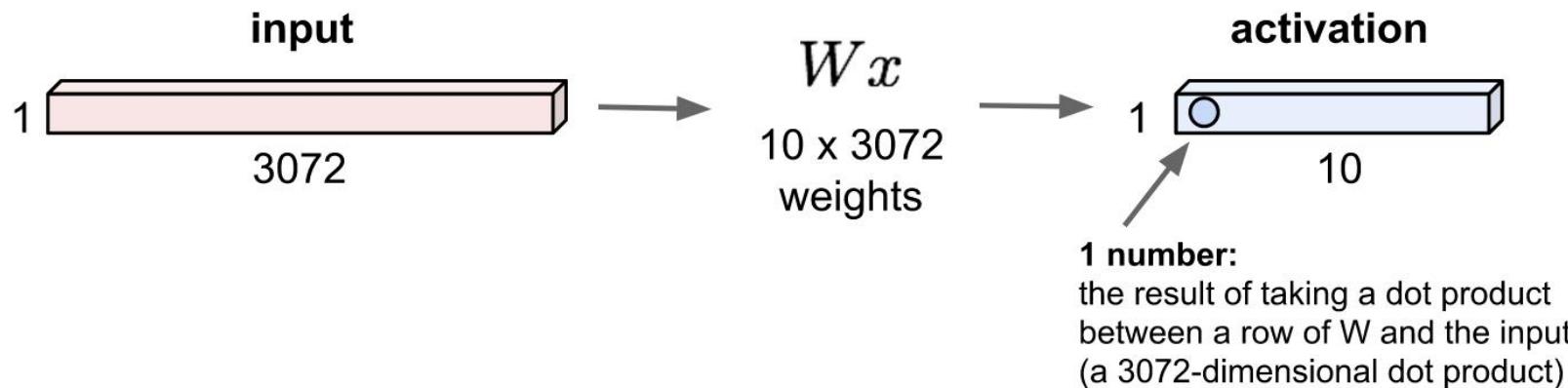


# Contents

- Introduction
- Convolutional Neural Networks (CNN)
- CNN Architectures

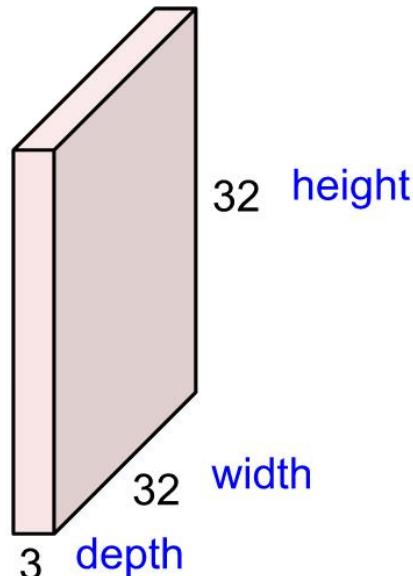
# Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1

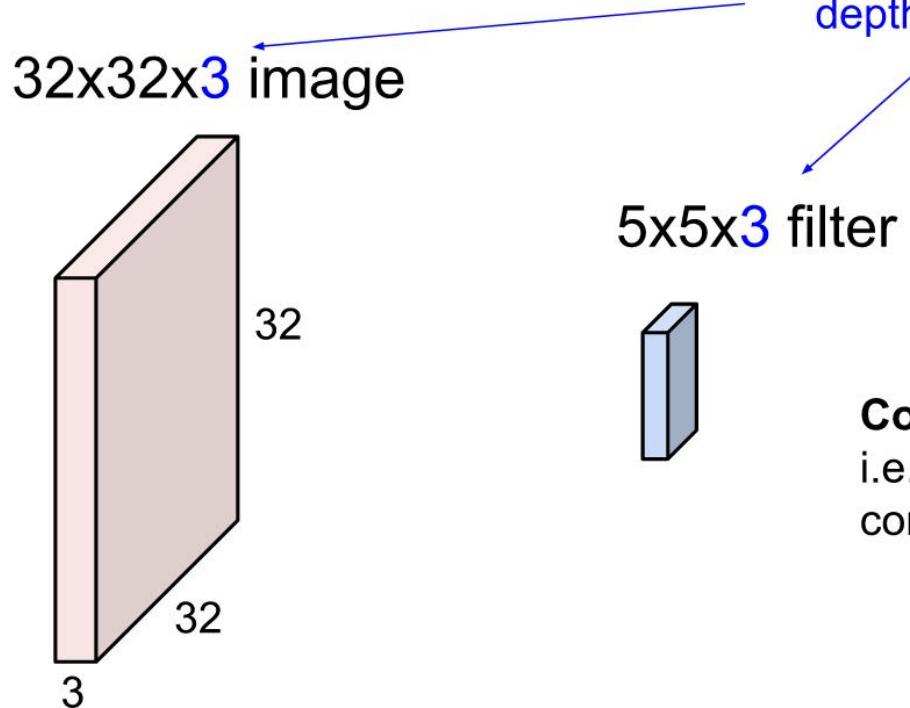


# Convolution Layer

32x32x3 image -> preserve spatial structure



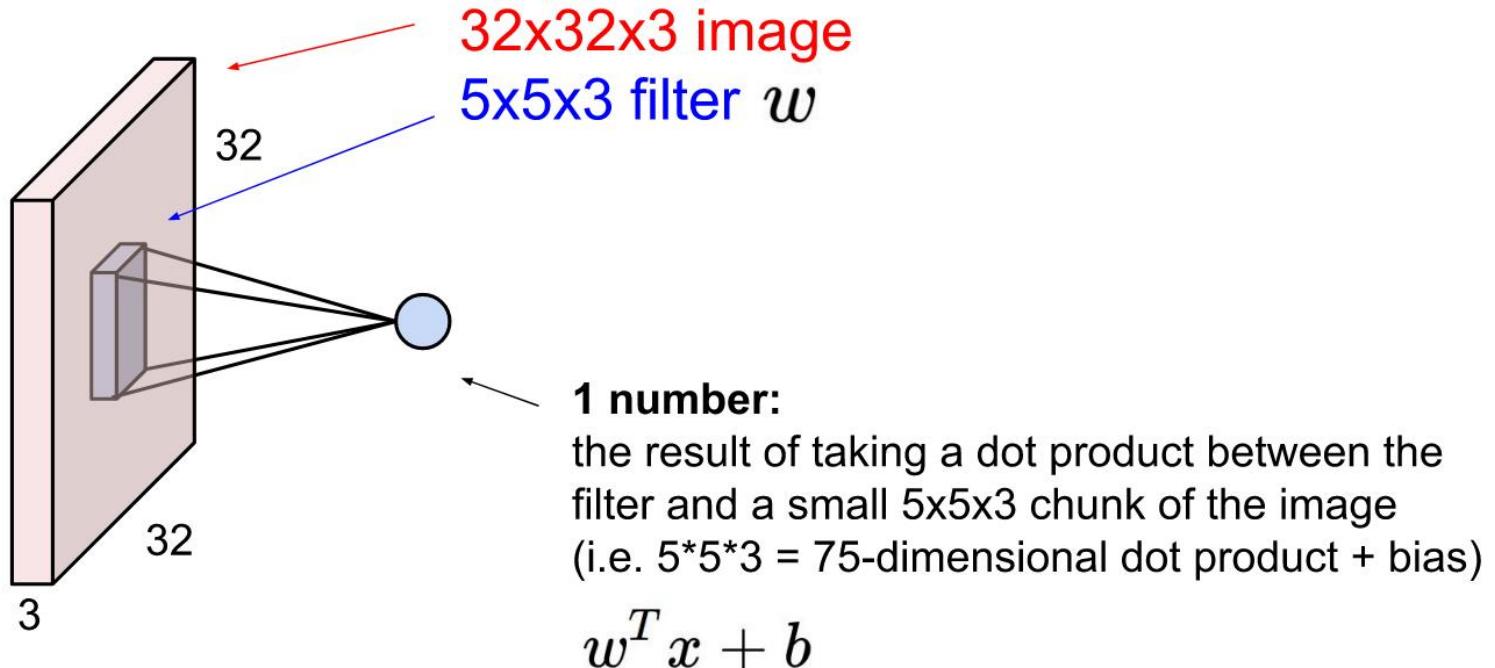
# Convolution Layer



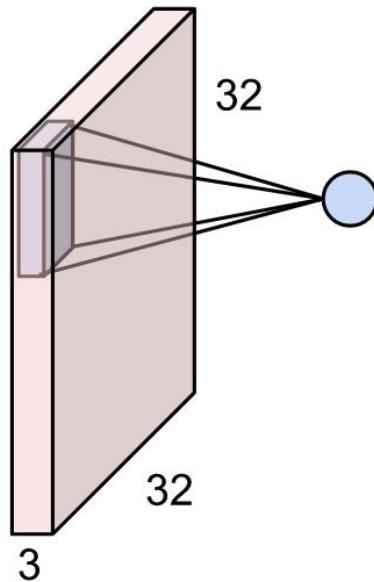
Filters always extend the full depth of the input volume

**Convolve** the filter with the image  
i.e. “slide over the image spatially,  
computing dot products”

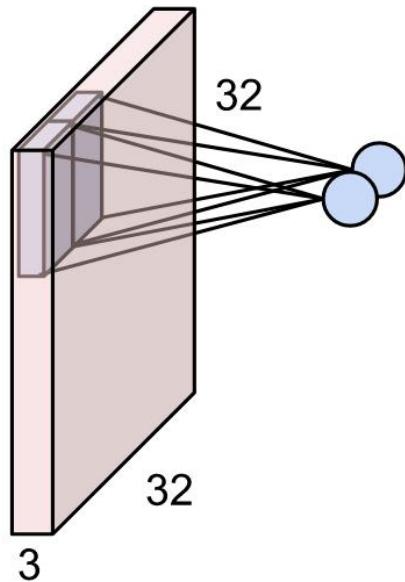
# Convolution Layer



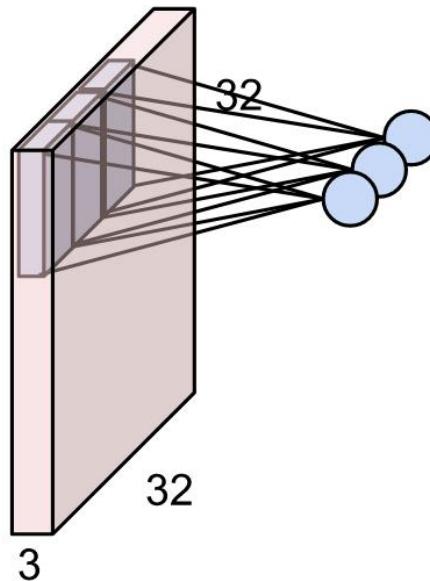
# Convolution Layer



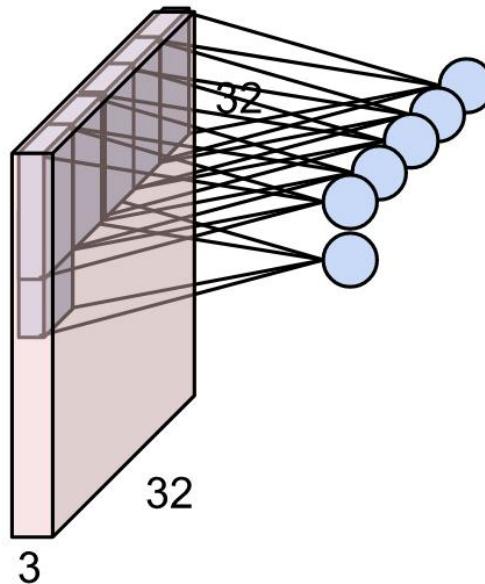
# Convolution Layer



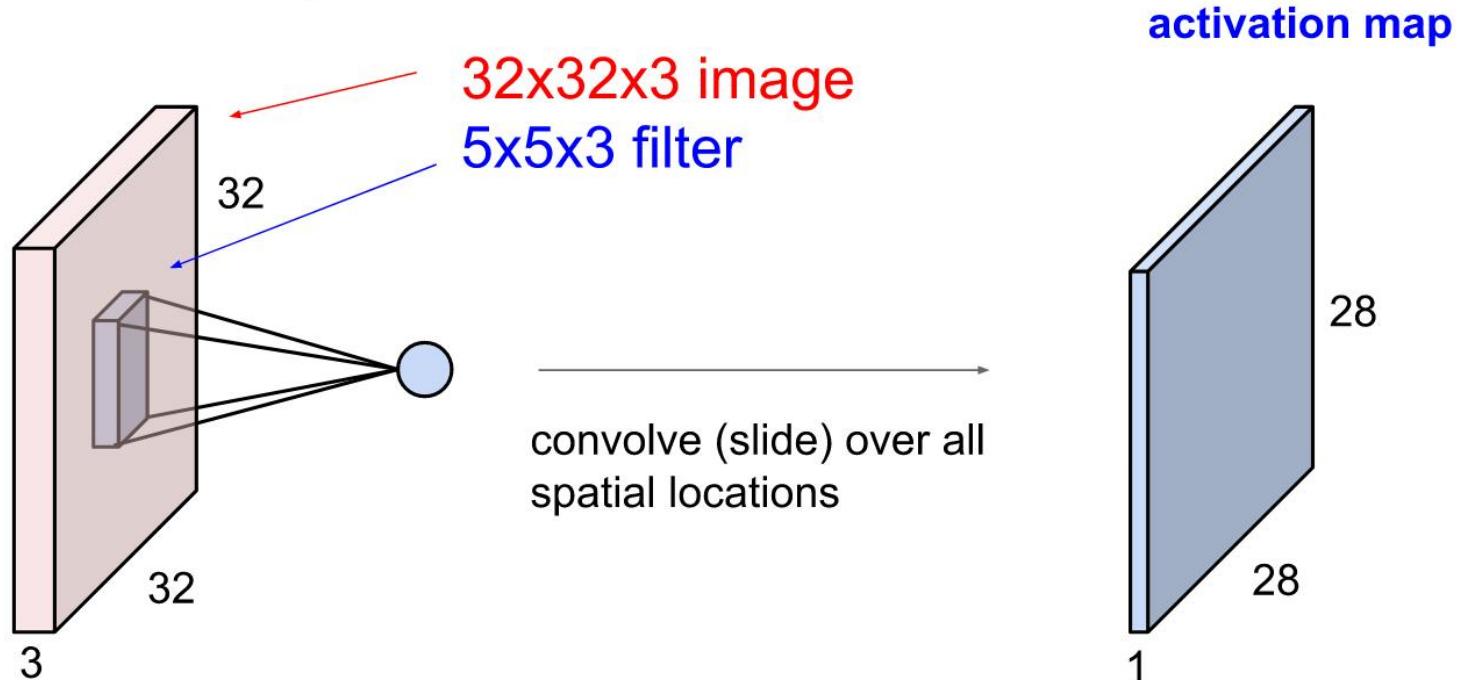
# Convolution Layer



# Convolution Layer

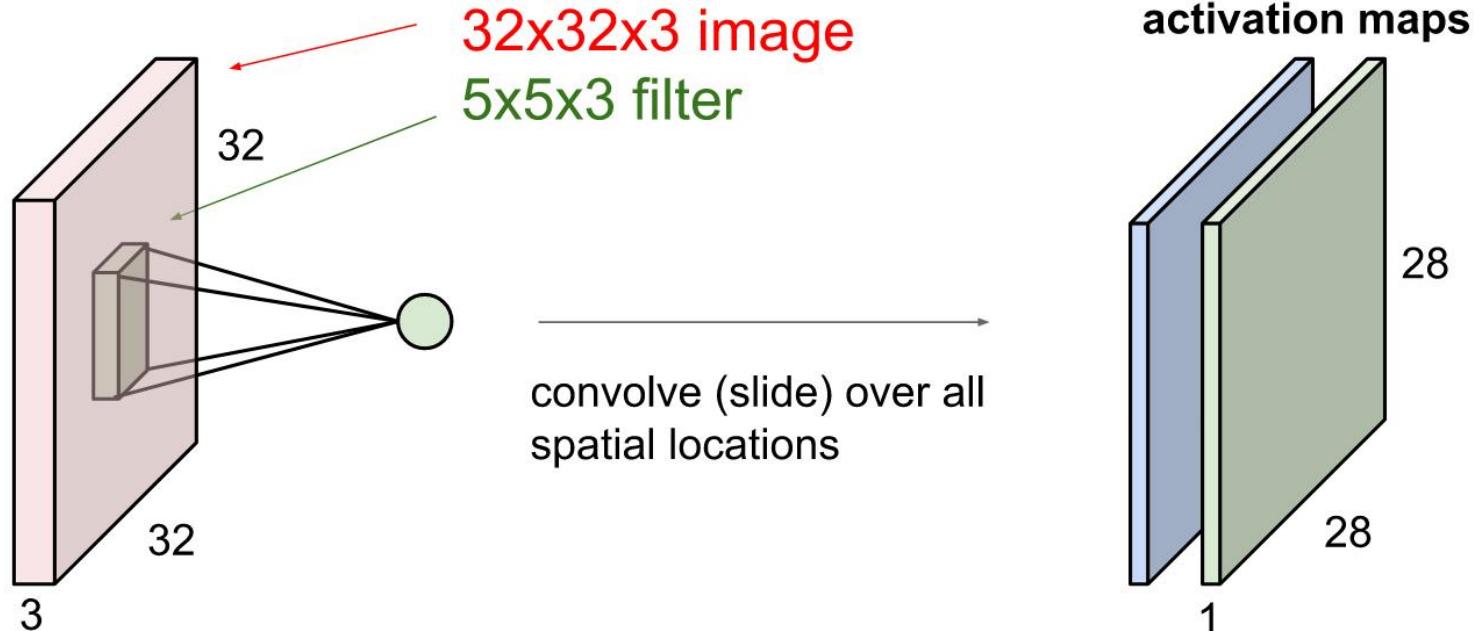


# Convolution Layer

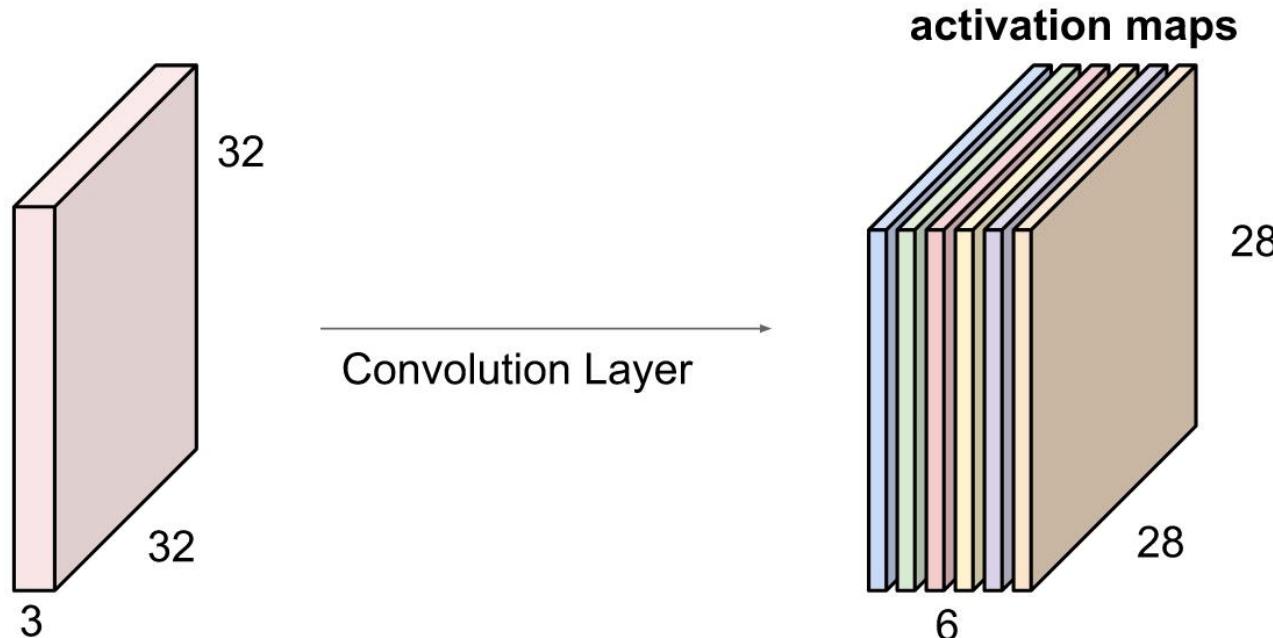


# Convolution Layer

consider a second, green filter

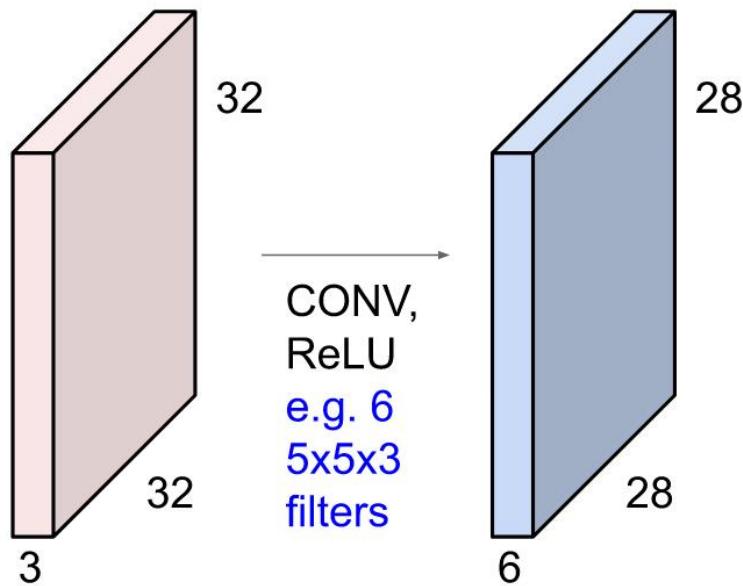


For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

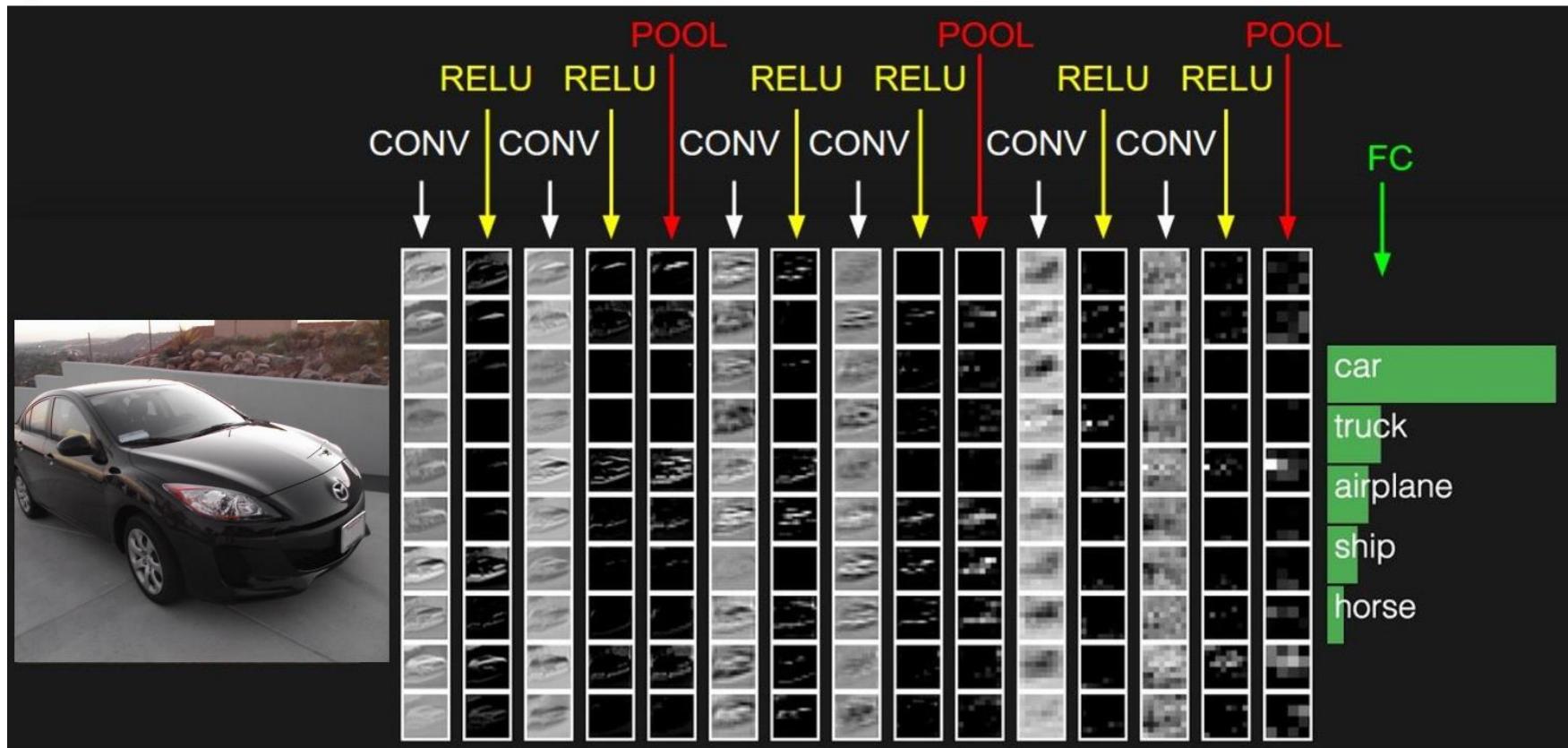


We stack these up to get a “new image” of size 28x28x6!

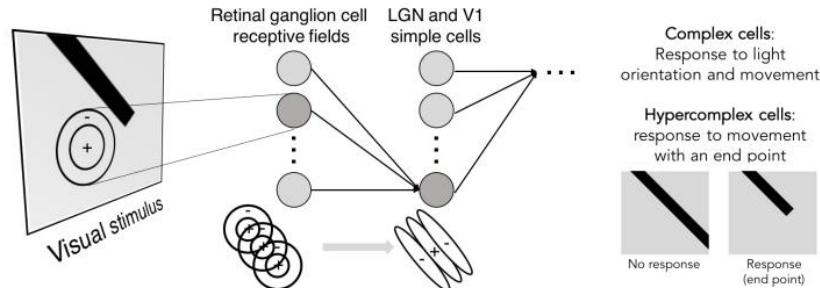
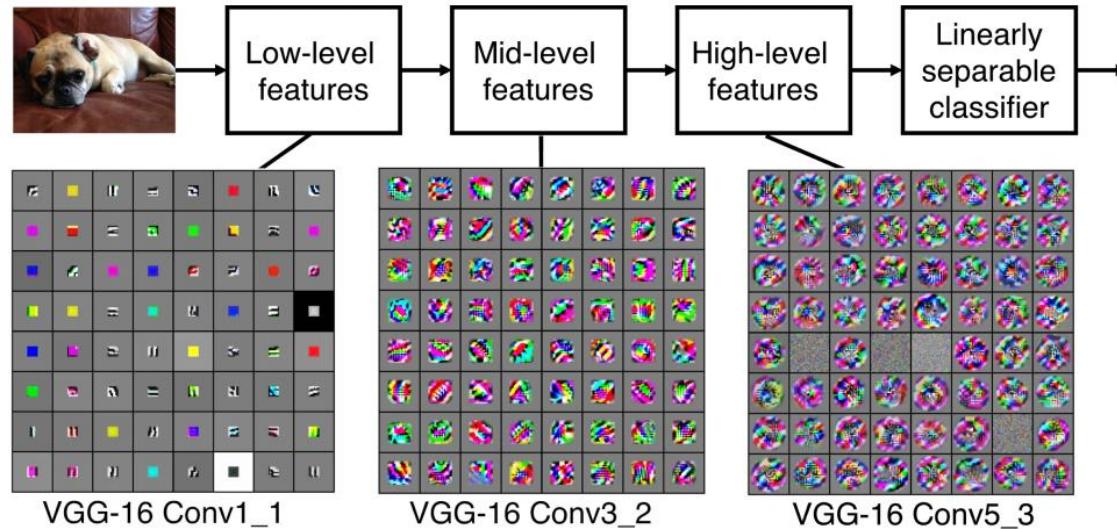
**Preview:** ConvNet is a sequence of Convolution Layers, interspersed with activation functions



preview:

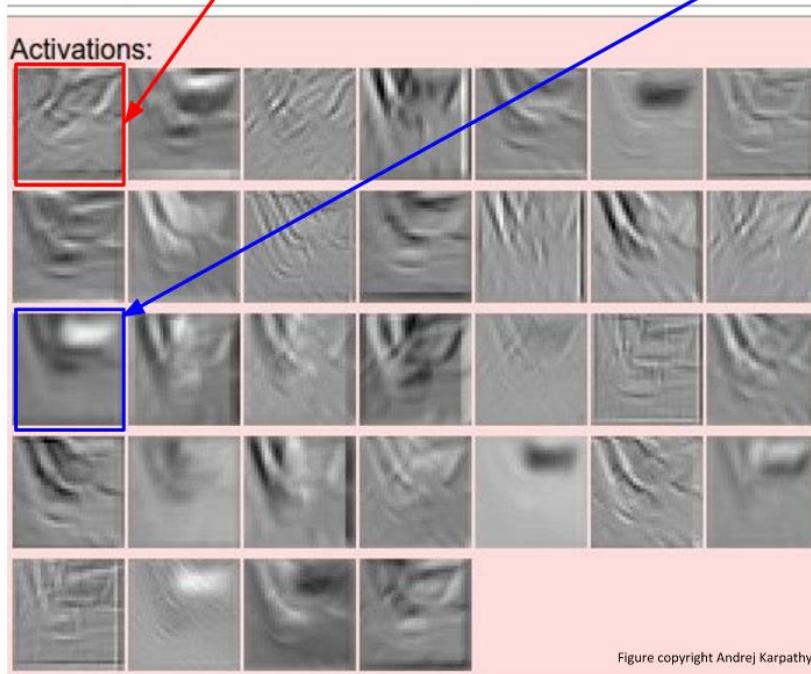


## Preview





one filter =>  
one activation map



example 5x5 filters  
(32 total)

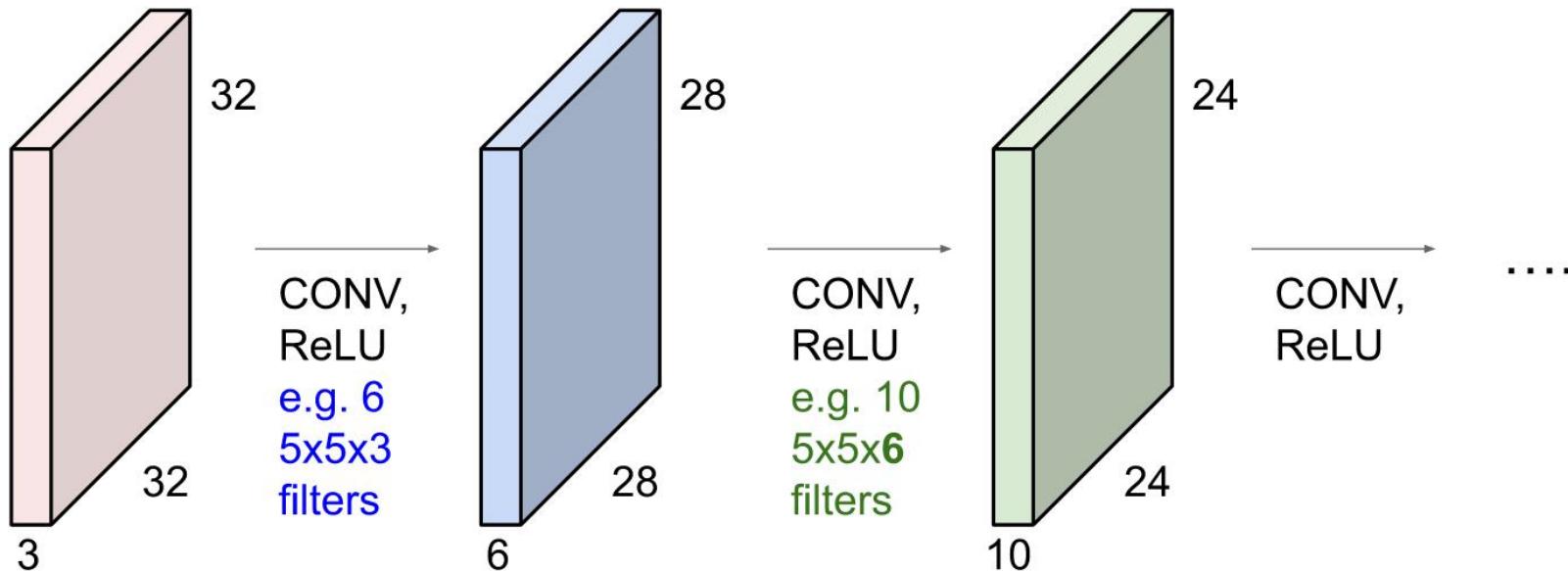
We call the layer convolutional  
because it is related to convolution  
of two signals:

$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1, n_2] \cdot g[x - n_1, y - n_2]$$

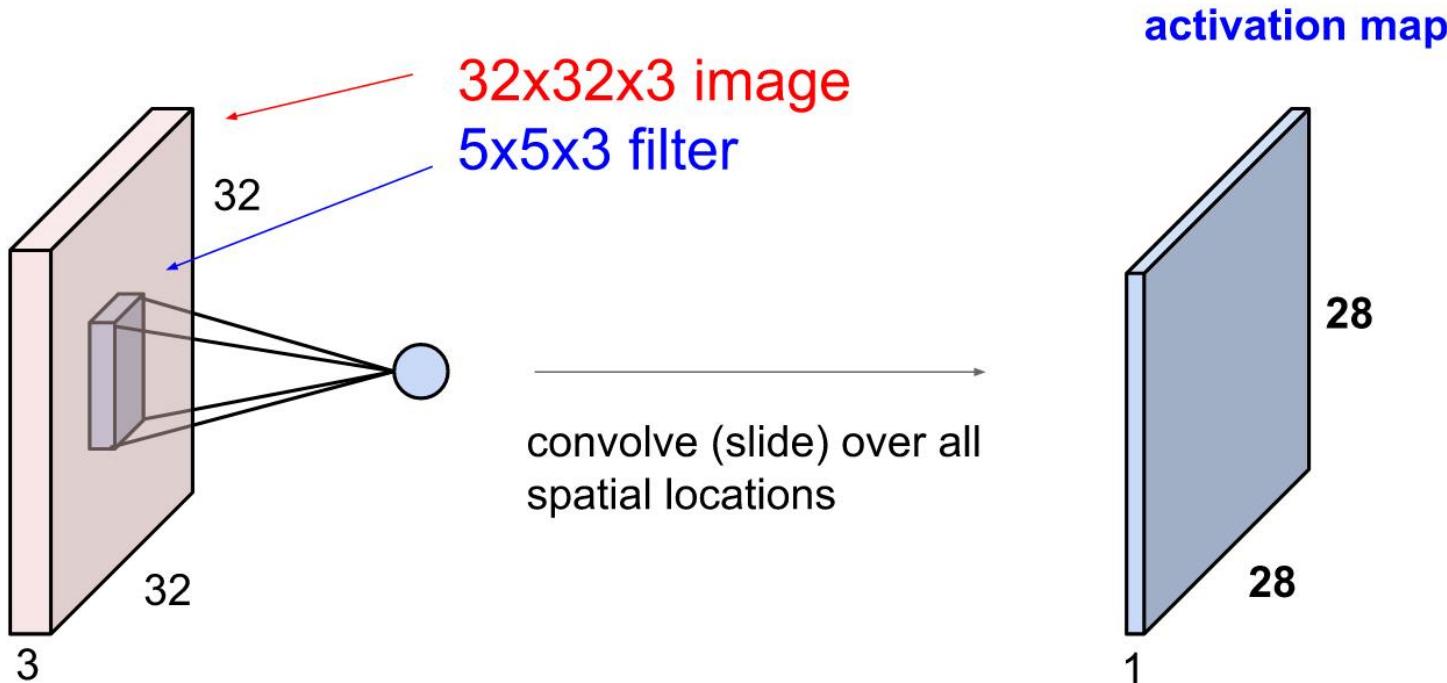
elementwise multiplication and sum of  
a filter and the signal (image)

## Remember back to...

E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially!  
(32 -> 28 -> 24 ...). Shrinking too fast is not good, doesn't work well.

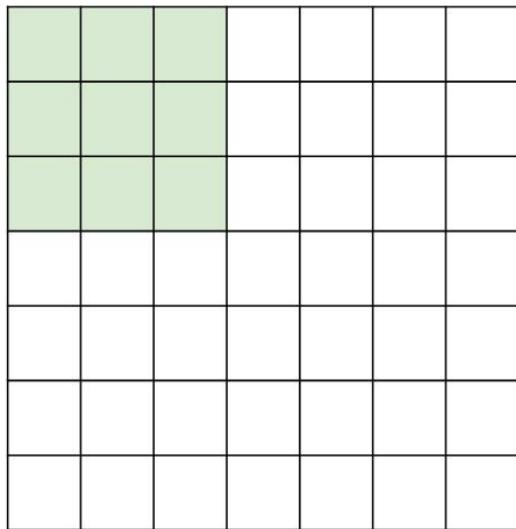


## A closer look at spatial dimensions:



## A closer look at spatial dimensions:

7

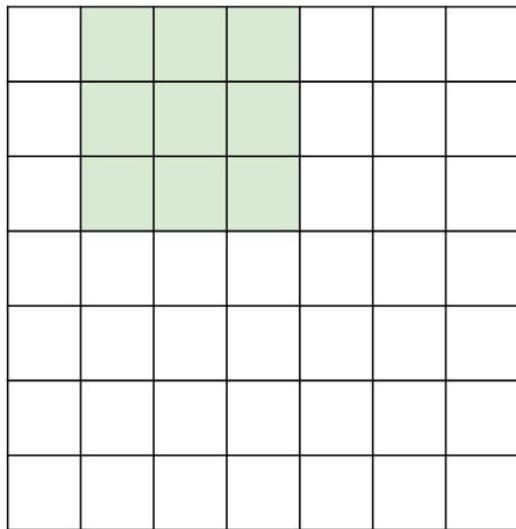


7x7 input (spatially)  
assume 3x3 filter

7

## A closer look at spatial dimensions:

7

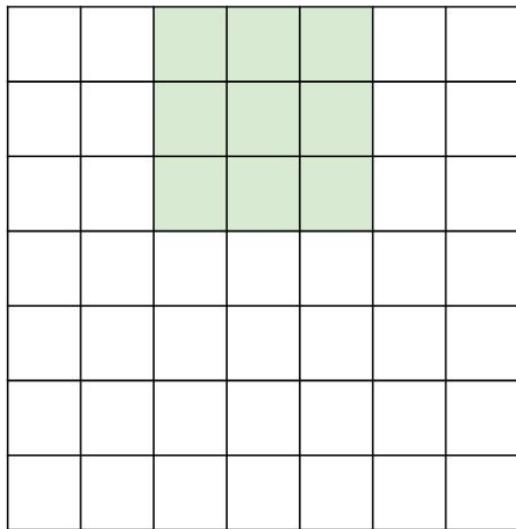


7x7 input (spatially)  
assume 3x3 filter

7

## A closer look at spatial dimensions:

7

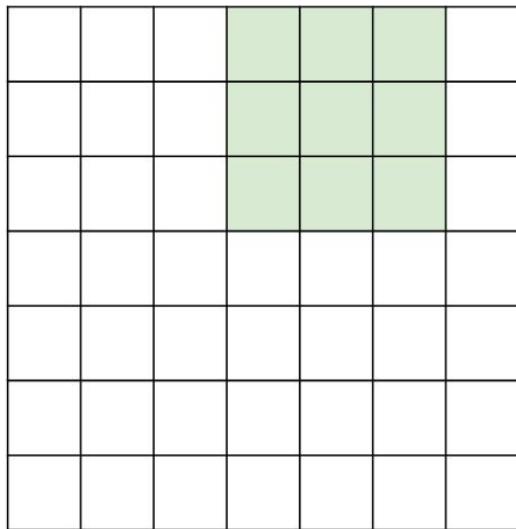


7x7 input (spatially)  
assume 3x3 filter

7

A closer look at spatial dimensions:

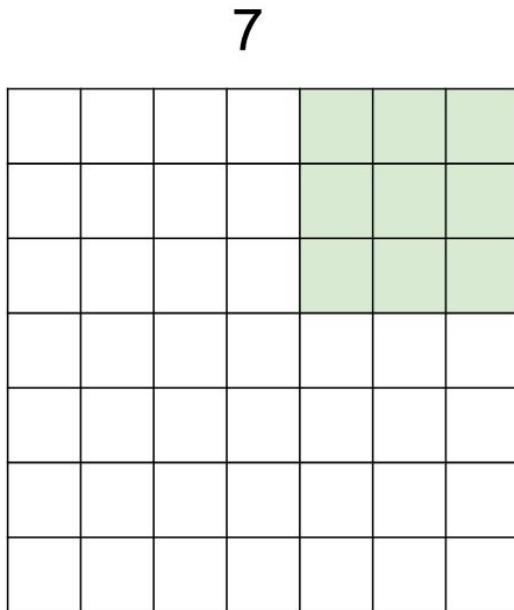
7



7x7 input (spatially)  
assume 3x3 filter

7

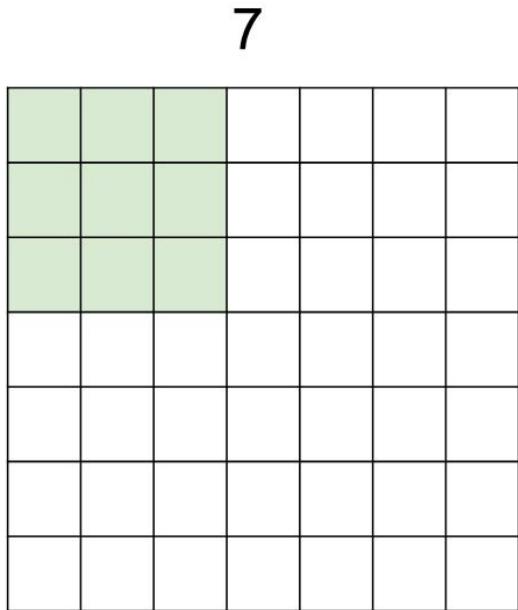
## A closer look at spatial dimensions:



7x7 input (spatially)  
assume 3x3 filter

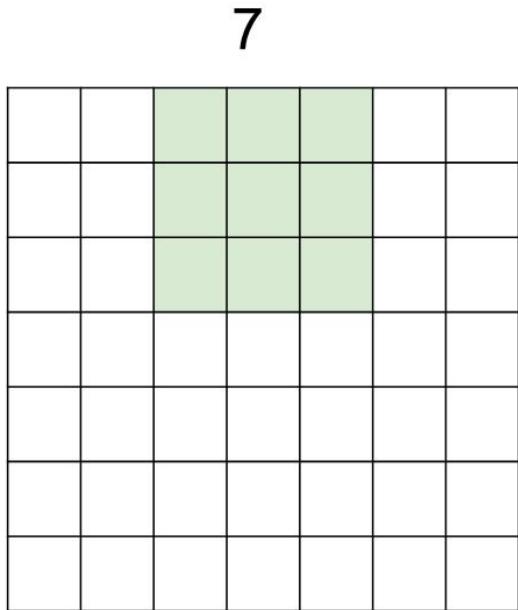
**=> 5x5 output**

A closer look at spatial dimensions:



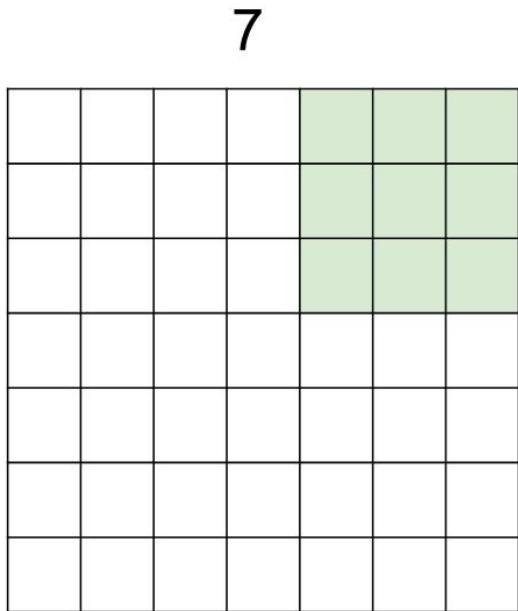
7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 2**

A closer look at spatial dimensions:



7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 2**

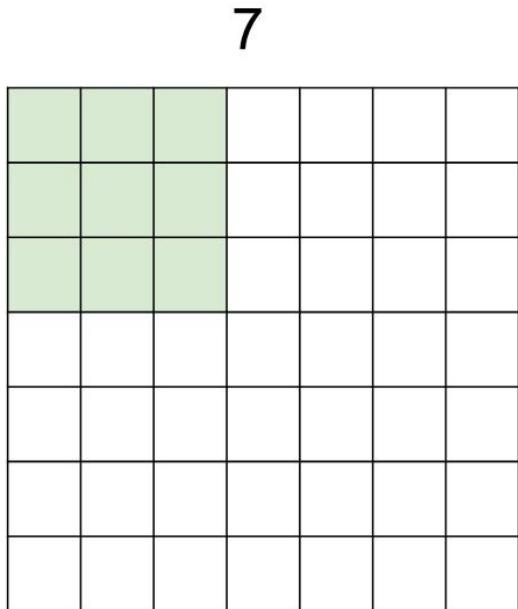
A closer look at spatial dimensions:



7

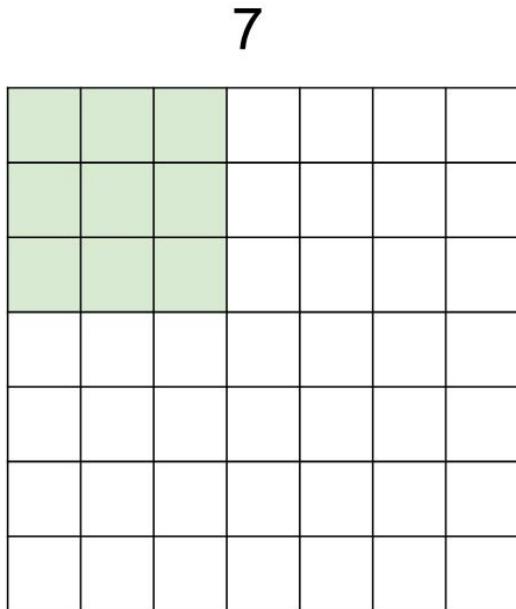
7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 2**  
**=> 3x3 output!**

A closer look at spatial dimensions:



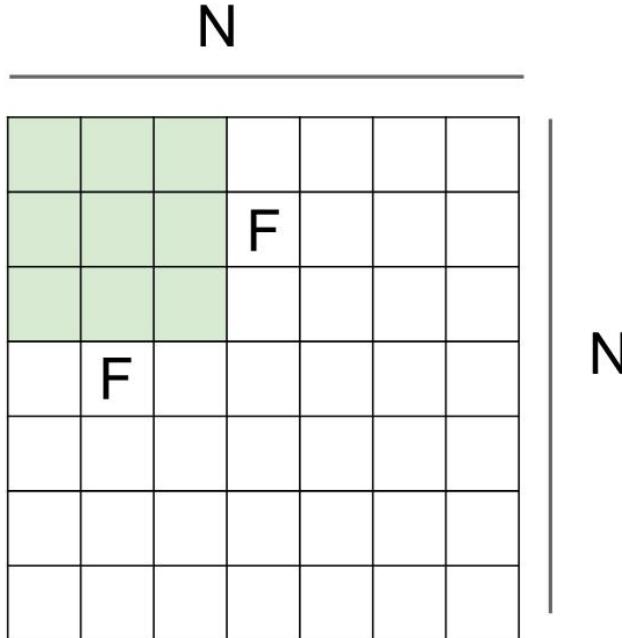
7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 3?**

## A closer look at spatial dimensions:



7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 3?**

**doesn't fit!**  
cannot apply 3x3 filter on  
7x7 input with stride 3.



Output size:  
 **$(N - F) / \text{stride} + 1$**

e.g.  $N = 7$ ,  $F = 3$ :  
stride 1 =>  $(7 - 3)/1 + 1 = 5$   
stride 2 =>  $(7 - 3)/2 + 1 = 3$   
stride 3 =>  $(7 - 3)/3 + 1 = 2.33$  :\

# In practice: Common to zero pad the border

|   |   |   |   |   |   |  |  |
|---|---|---|---|---|---|--|--|
| 0 | 0 | 0 | 0 | 0 | 0 |  |  |
| 0 |   |   |   |   |   |  |  |
| 0 |   |   |   |   |   |  |  |
| 0 |   |   |   |   |   |  |  |
| 0 |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |
|   |   |   |   |   |   |  |  |

e.g. input 7x7

**3x3 filter, applied with stride 1**

**pad with 1 pixel border => what is the output?**

(recall:)

$$(N - F) / \text{stride} + 1$$

# In practice: Common to zero pad the border

|   |   |   |   |   |   |  |  |  |
|---|---|---|---|---|---|--|--|--|
| 0 | 0 | 0 | 0 | 0 | 0 |  |  |  |
| 0 |   |   |   |   |   |  |  |  |
| 0 |   |   |   |   |   |  |  |  |
| 0 |   |   |   |   |   |  |  |  |
| 0 |   |   |   |   |   |  |  |  |
|   |   |   |   |   |   |  |  |  |
|   |   |   |   |   |   |  |  |  |
|   |   |   |   |   |   |  |  |  |
|   |   |   |   |   |   |  |  |  |
|   |   |   |   |   |   |  |  |  |
|   |   |   |   |   |   |  |  |  |
|   |   |   |   |   |   |  |  |  |
|   |   |   |   |   |   |  |  |  |
|   |   |   |   |   |   |  |  |  |
|   |   |   |   |   |   |  |  |  |
|   |   |   |   |   |   |  |  |  |
|   |   |   |   |   |   |  |  |  |

e.g. input 7x7

**3x3 filter, applied with stride 1**

**pad with 1 pixel border => what is the output?**

**7x7 output!**

(recall:)

$$(N + 2P - F) / \text{stride} + 1$$

# In practice: Common to zero pad the border

|   |   |   |   |   |   |  |  |  |
|---|---|---|---|---|---|--|--|--|
| 0 | 0 | 0 | 0 | 0 | 0 |  |  |  |
| 0 |   |   |   |   |   |  |  |  |
| 0 |   |   |   |   |   |  |  |  |
| 0 |   |   |   |   |   |  |  |  |
| 0 |   |   |   |   |   |  |  |  |
|   |   |   |   |   |   |  |  |  |
|   |   |   |   |   |   |  |  |  |
|   |   |   |   |   |   |  |  |  |
|   |   |   |   |   |   |  |  |  |

e.g. input 7x7

**3x3 filter, applied with stride 1**

**pad with 1 pixel border => what is the output?**

**7x7 output!**

in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with  $(F-1)/2$ . (will preserve size spatially)

e.g.  $F = 3 \Rightarrow$  zero pad with 1

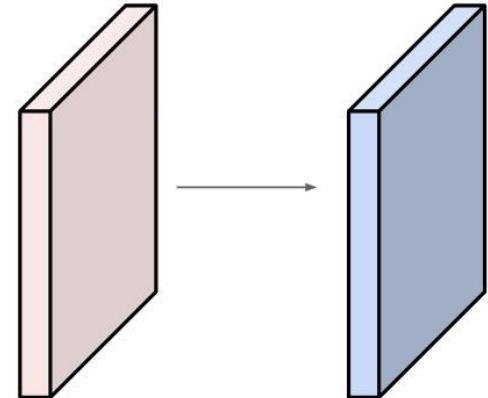
$F = 5 \Rightarrow$  zero pad with 2

$F = 7 \Rightarrow$  zero pad with 3

Examples time:

Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

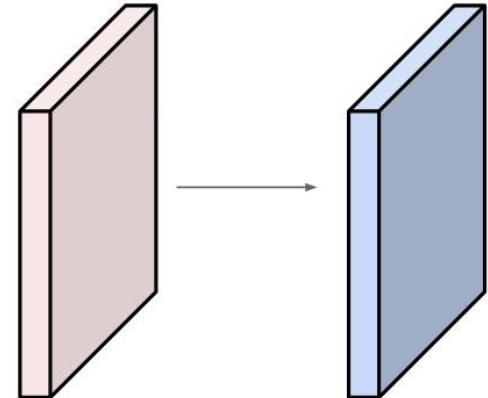


Output volume size: ?

Examples time:

Input volume: **32x32x3**

**10 5x5** filters with stride **1**, pad **2**



Output volume size:

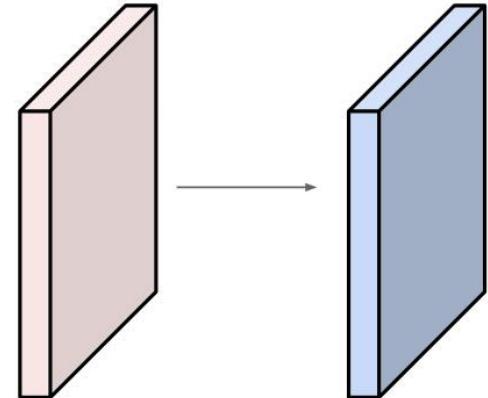
$(32+2*2-5)/1+1 = 32$  spatially, so

**32x32x10**

Examples time:

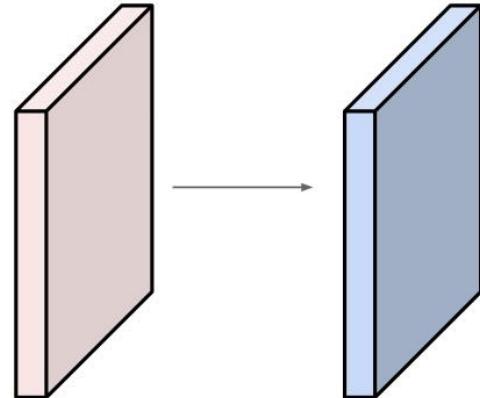
Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2



Number of parameters in this layer?

Examples time:



Input volume: **32x32x3**

**10 5x5** filters with stride 1, pad 2

Number of parameters in this layer?

each filter has  $5*5*3 + 1 = 76$  params

(+1 for bias)

=> **76\*10 = 760**

# Convolution layer: summary

Let's assume input is  $W_1 \times H_1 \times C$

Conv layer needs 4 hyperparameters:

- Number of filters **K**
- The filter size **F**
- The stride **S**
- The zero padding **P**

This will produce an output of  $W_2 \times H_2 \times K$

where:

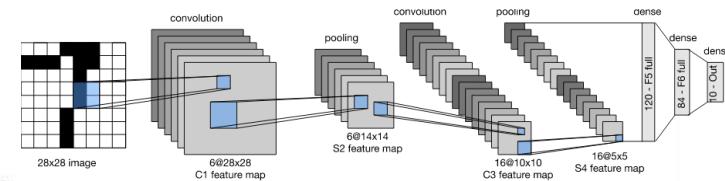
- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$

Number of parameters:  $F^2CK$  and  $K$  biases

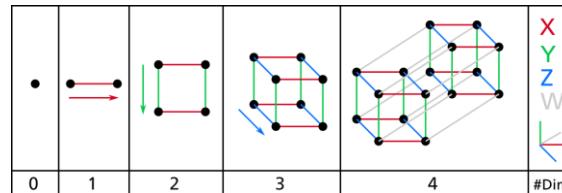
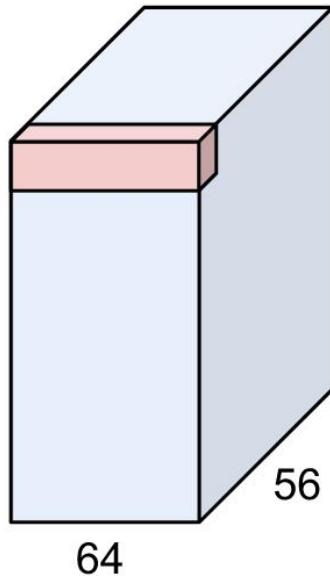
Common settings:

$K = (\text{powers of 2, e.g. } 32, 64, 128, 512)$

- $F = 3, S = 1, P = 1$
- $F = 5, S = 1, P = 2$
- $F = 1, S = 1, P = 0 ?$



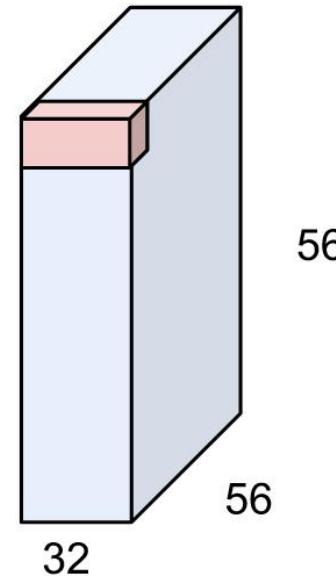
(btw, 1x1 convolution layers make perfect sense)



1x1 CONV  
with 32 filters

→

(each filter has size  
 $1 \times 1 \times 64$ , and performs a  
64-dimensional dot  
product)



# Example: CONV layer in PyTorch

## Conv2d

```
CLASS torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1,  
padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')
```

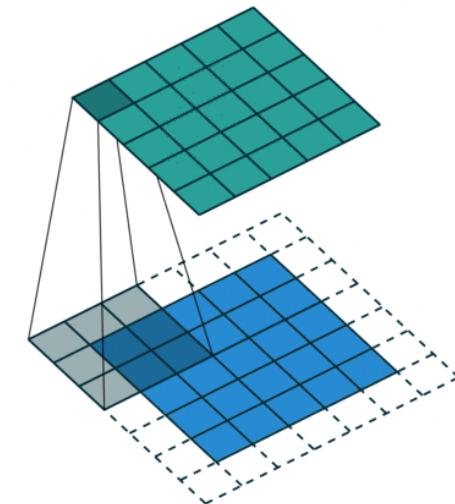
[SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size  $(N, C_{\text{in}}, H, W)$  and output  $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$  can be precisely described as:

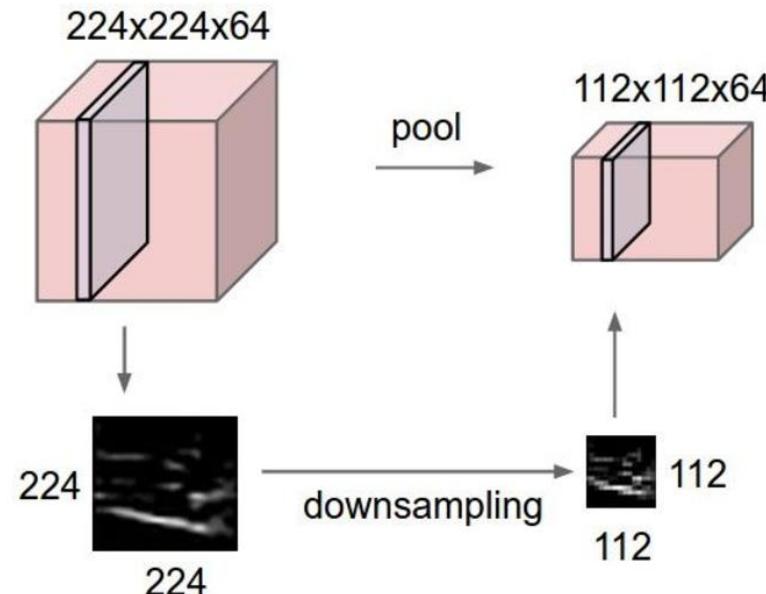
$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where  $\star$  is the valid 2D **cross-correlation** operator,  $N$  is a batch size,  $C$  denotes a number of channels,  $H$  is a height of input planes in pixels, and  $W$  is width in pixels.



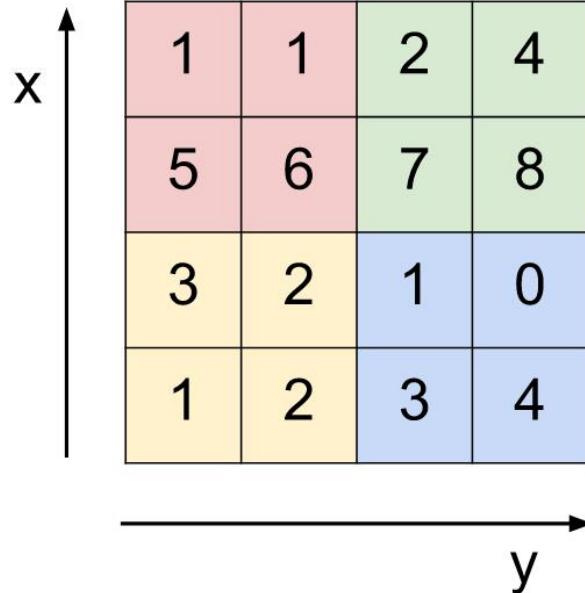
# Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:
- makes the representation become approximately invariant to small translations of the input



# MAX POOLING

Single depth slice



max pool with 2x2 filters  
and stride 2

A 2x2 grid representing the output of max pooling. It contains the maximum values from the overlapping 2x2 receptive fields in the input:

|   |   |
|---|---|
| 6 | 8 |
| 3 | 4 |

## Pooling layer: summary

Let's assume input is  $W_1 \times H_1 \times C$

Conv layer needs 2 hyperparameters:

- The spatial extent **F**
- The stride **S**

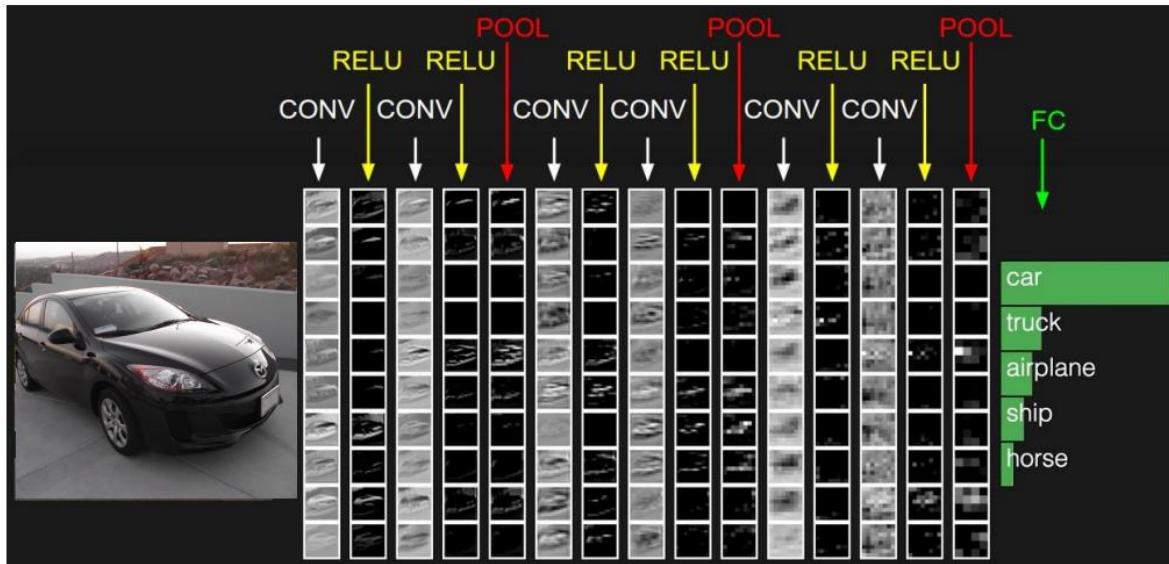
This will produce an output of  $W_2 \times H_2 \times C$  where:

- $W_2 = (W_1 - F)/S + 1$
- $H_2 = (H_1 - F)/S + 1$

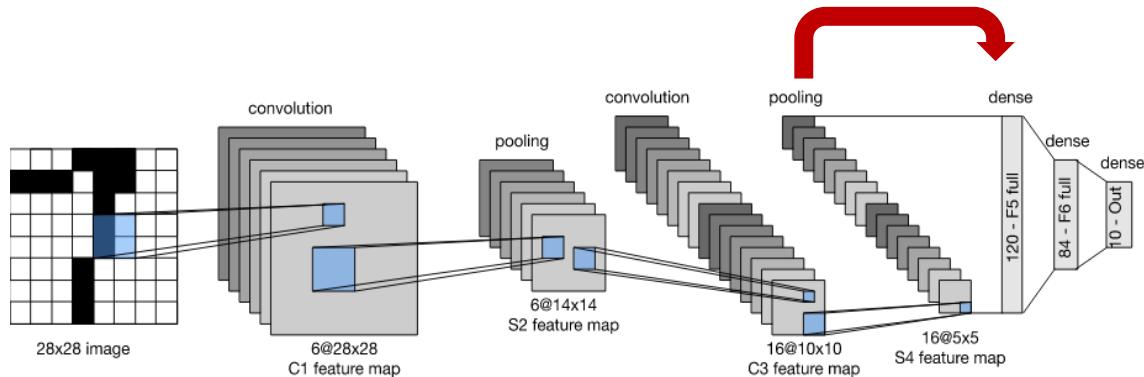
Number of parameters: 0

# Fully Connected Layer (FC layer)

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks

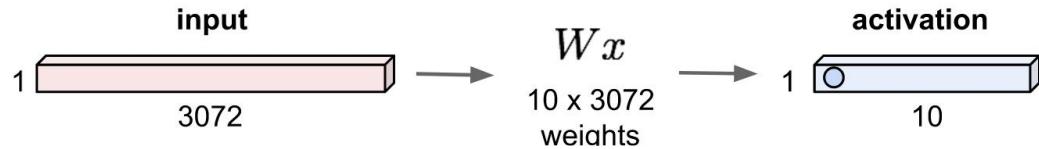


# Reminder: Fully Connected Layer



32x32x3 image  $\rightarrow$  stretch to 3072 x 1

Each neuron  
looks at the full  
input volume

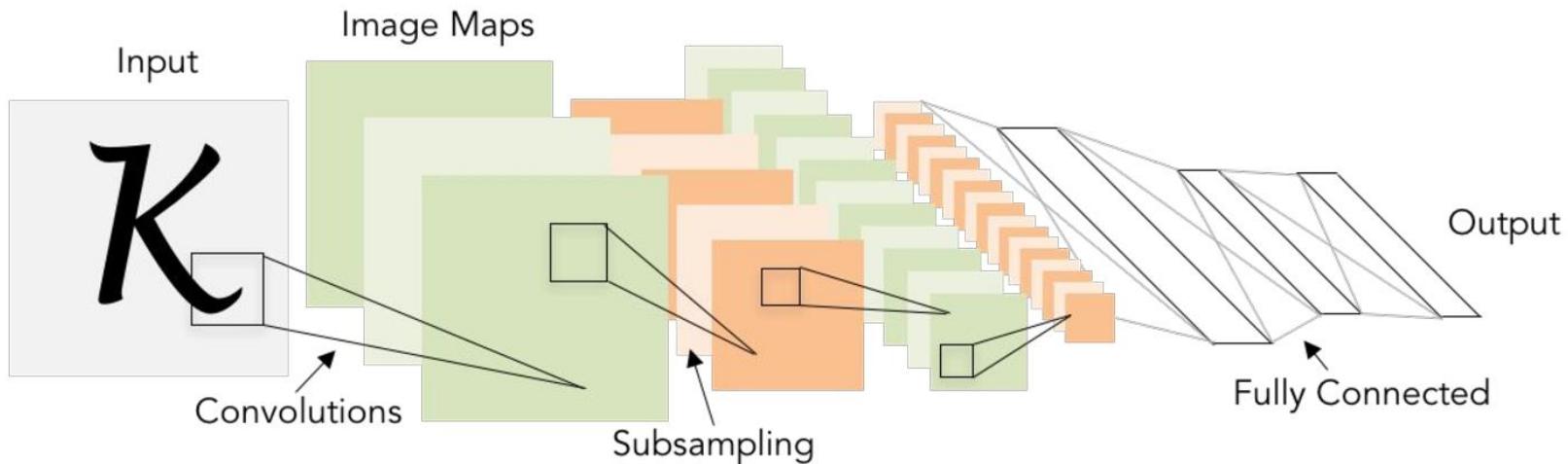


# Contents

- Introduction
- Convolutional Neural Networks (CNN)
- **CNN Architectures**

# Review: LeNet-5

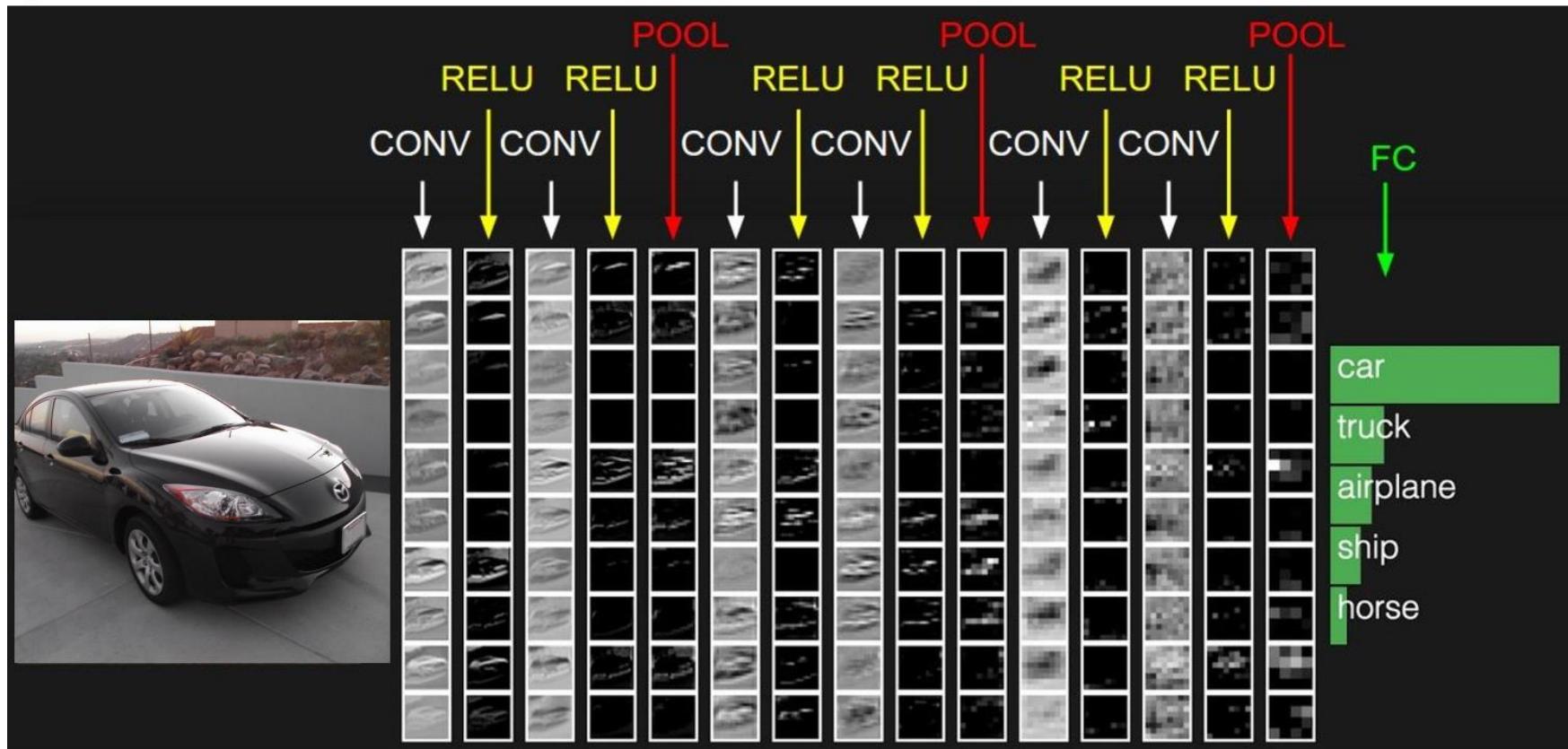
[LeCun et al., 1998]



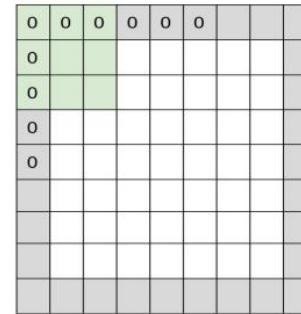
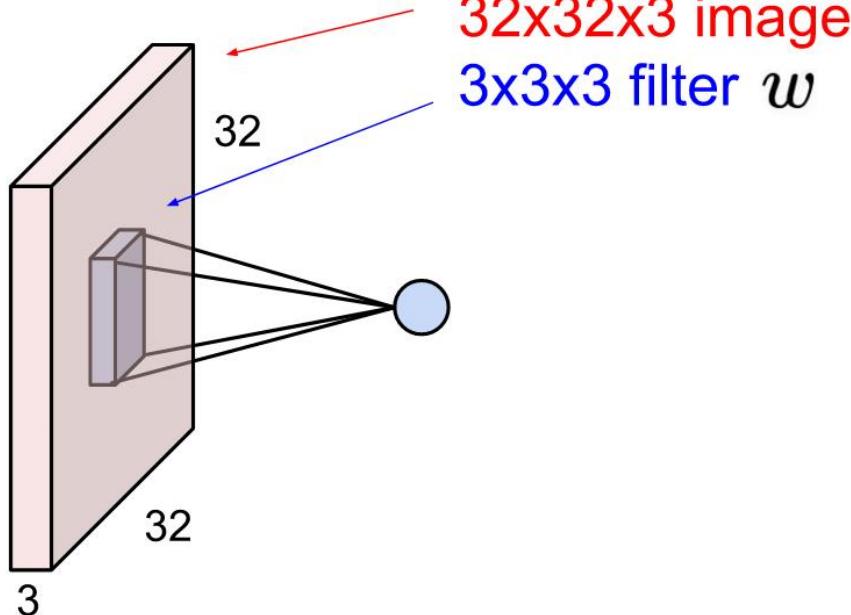
Conv filters were 5x5, applied at stride 1

Subsampling (Pooling) layers were 2x2 applied at stride 2  
i.e. architecture is [CONV-POOL-CONV-POOL-FC-FC]

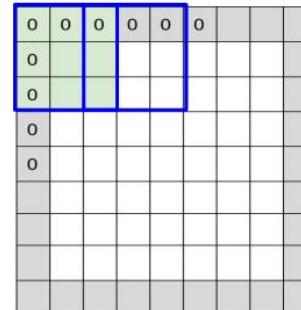
preview:



# Review: Convolution



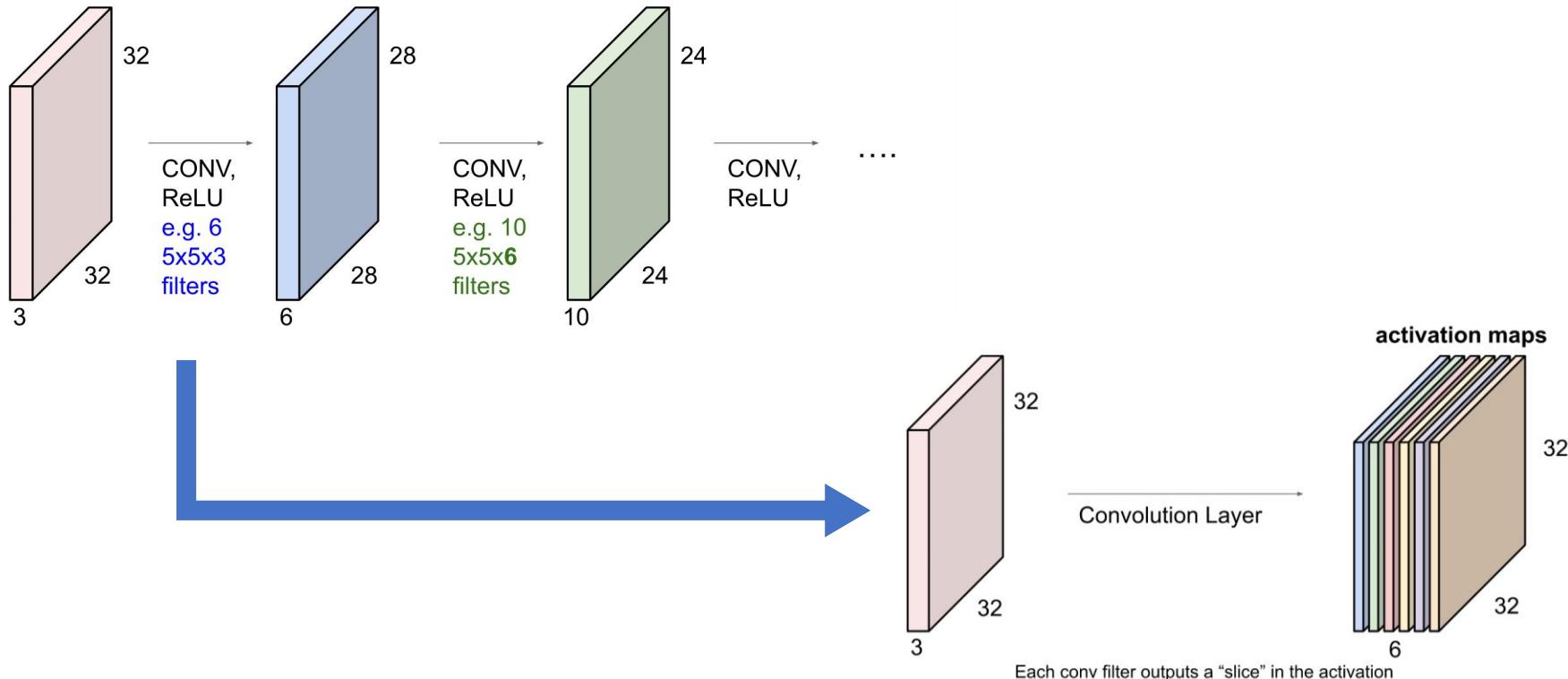
**Padding:**  
Preserve  
input spatial  
dimensions in  
output activations



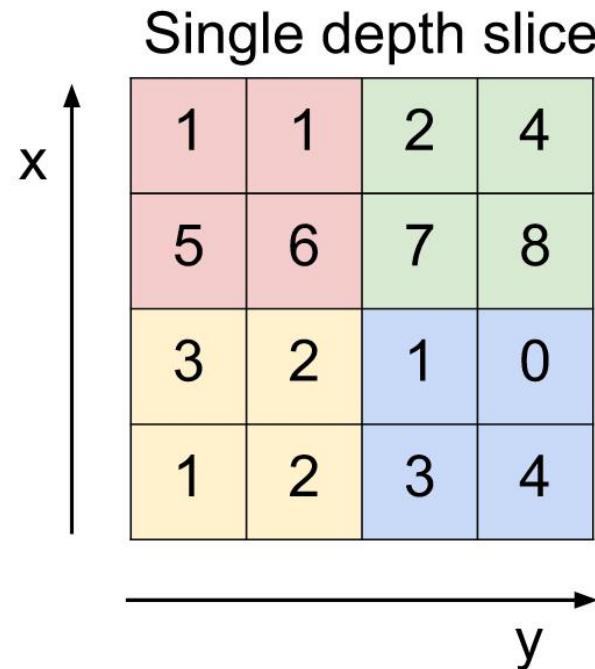
**Stride:**  
Downsample  
output activations

## Remember back to...

E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially!  
(32 -> 28 -> 24 ...). Shrinking too fast is not good, doesn't work well.



# Review: Pooling

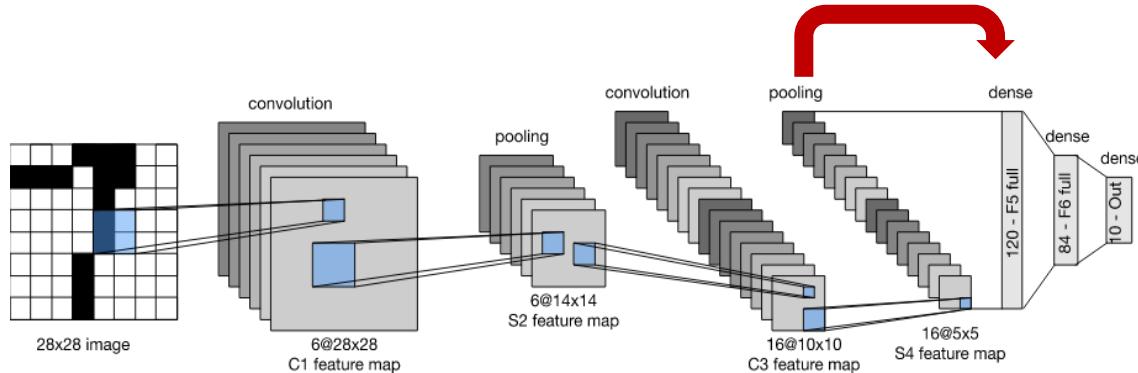


max pool with 2x2 filters  
and stride 2

A 2x2 matrix resulting from max pooling with 2x2 filters and stride 2. The top-left cell contains 6 (from the max of 1, 1, 5, 6), the top-right cell contains 8 (from the max of 2, 4, 7, 8), the bottom-left cell contains 3 (from the max of 3, 2, 1, 0), and the bottom-right cell contains 4 (from the max of 1, 2, 3, 4).

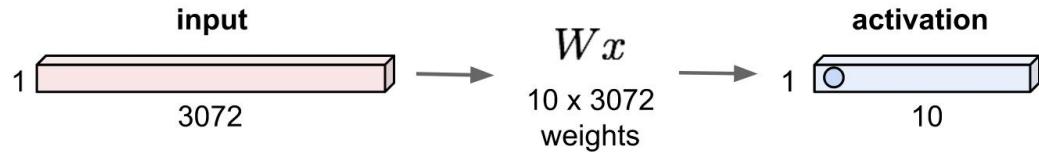
|   |   |
|---|---|
| 6 | 8 |
| 3 | 4 |

# Fully Connected Layer



32x32x3 image  $\rightarrow$  stretch to 3072 x 1

Each neuron  
looks at the full  
input volume



# Today: CNN Architectures

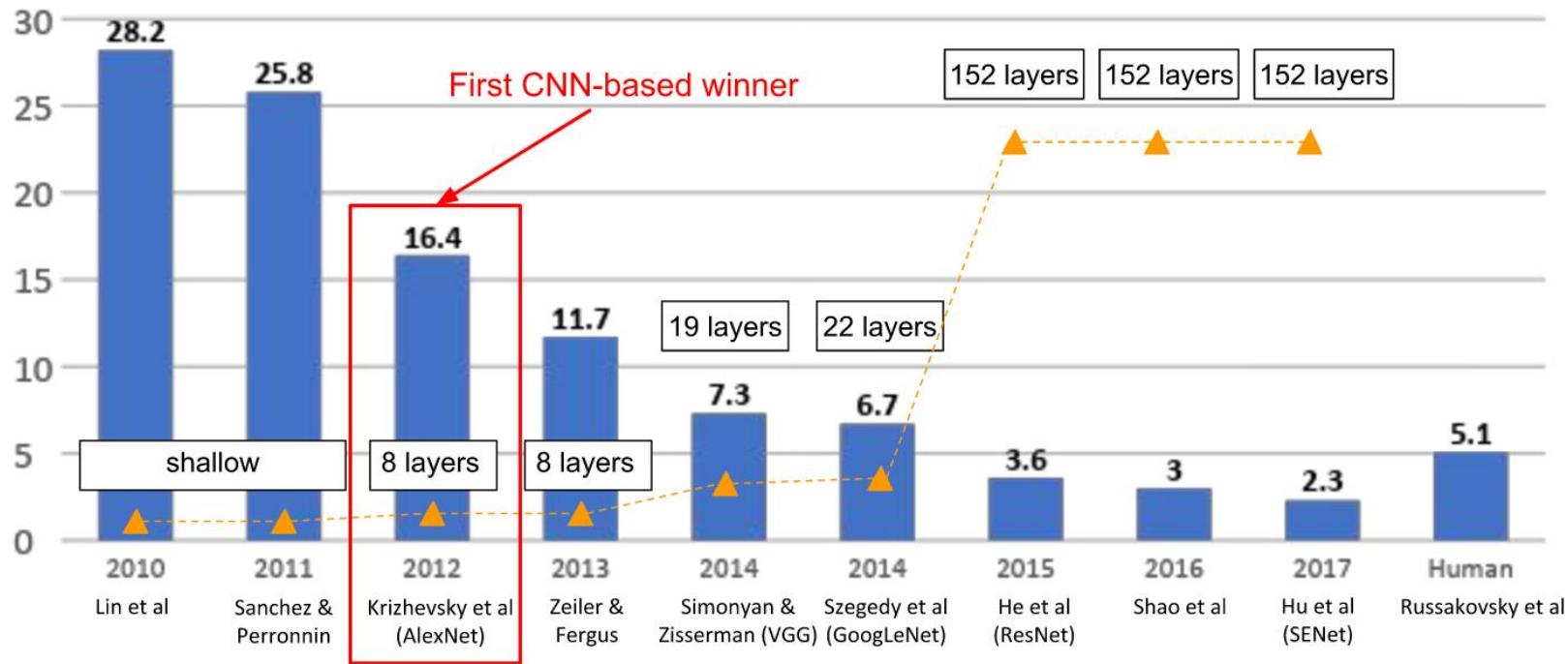
## Case Studies

- AlexNet
- VGG
- GoogLeNet
- ResNet

## Also....

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

# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



# Case Study: AlexNet

[Krizhevsky et al. 2012]

## Architecture:

CONV1

MAX POOL1

NORM1

CONV2

MAX POOL2

NORM2

CONV3

CONV4

CONV5

Max POOL3

FC6

FC7

FC8

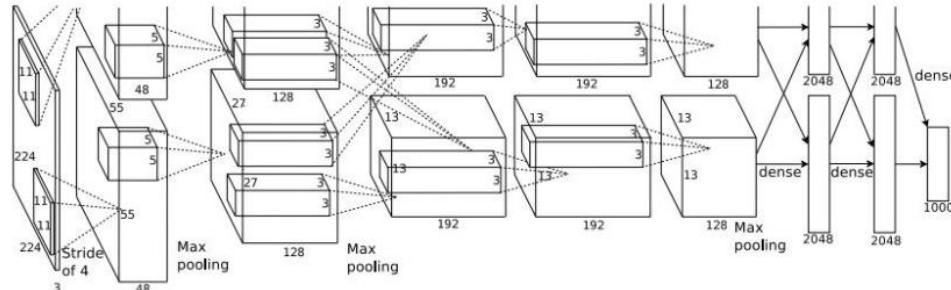
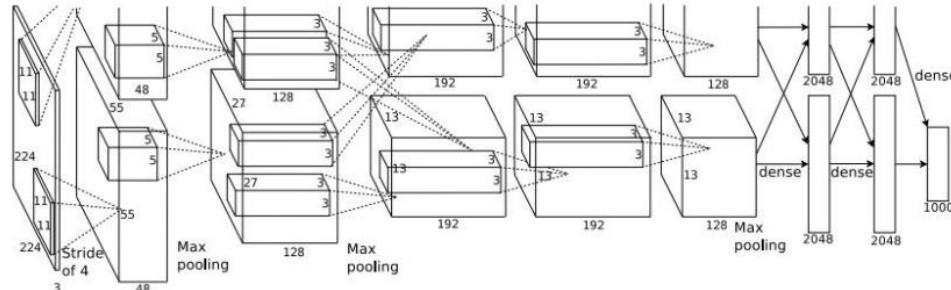


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

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

=>

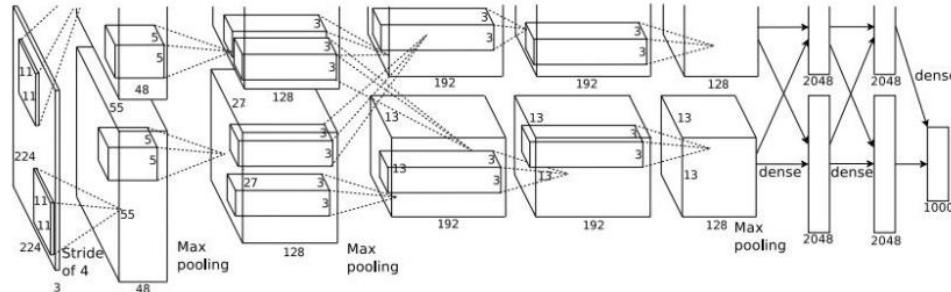
Q: what is the output volume size? Hint:  $(227-11)/4+1 = 55$

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

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

# Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

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

=>

Output volume [55x55x96]

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

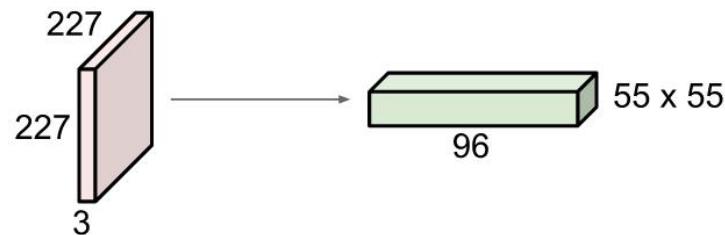
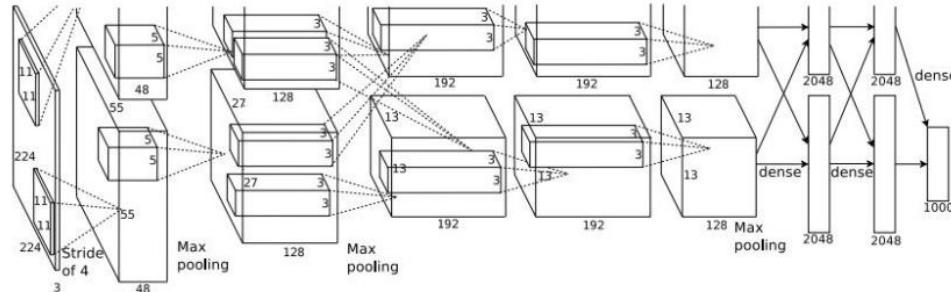


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# 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?

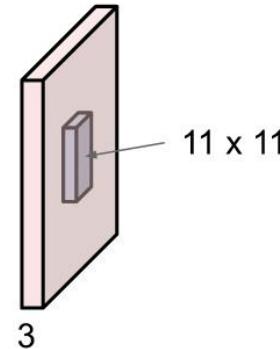
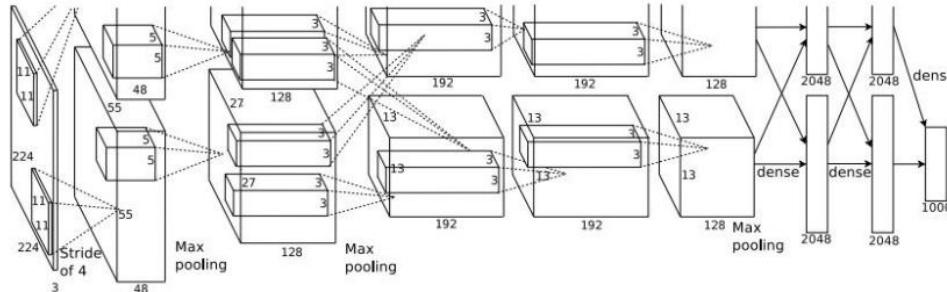


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

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

=>

Output volume **[55x55x96]**

Parameters:  $(11 \times 11 \times 3) \times 96 = 35K + \text{bias}$

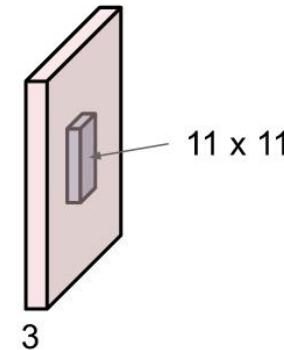
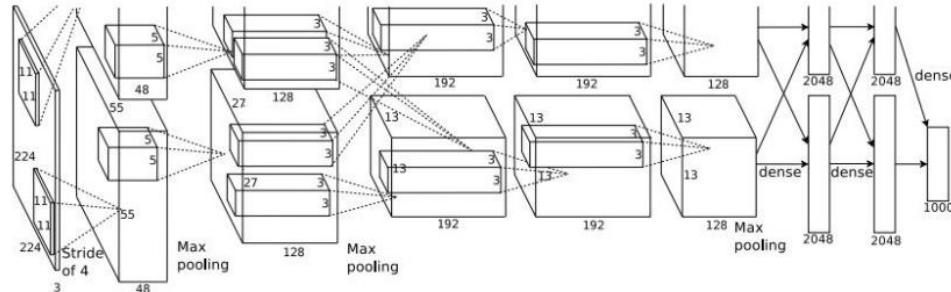


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

After CONV1: 55x55x96

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

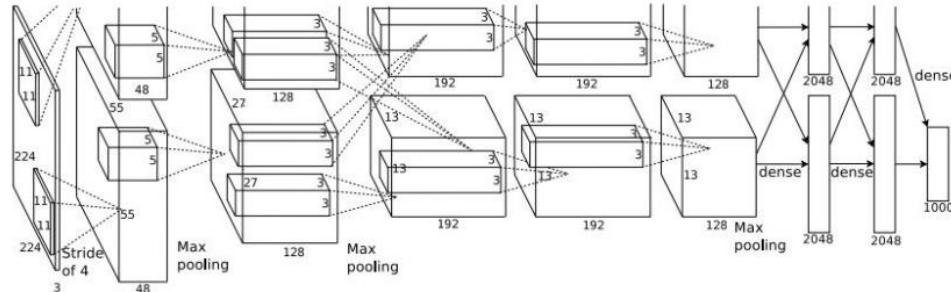
**Second layer (POOL1):** 3x3 filters applied at stride 2

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

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

# 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

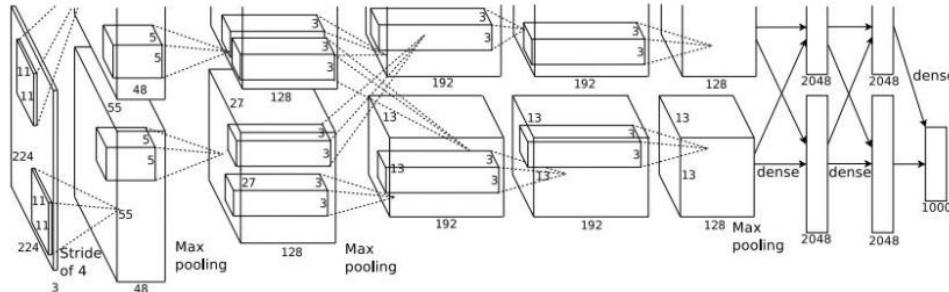
Output volume: 27x27x96

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

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

# 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!

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

# Case Study: AlexNet

[Krizhevsky et al. 2012]

Input: 227x227x3 images

After CONV1: 55x55x96

After POOL1: 27x27x96

...

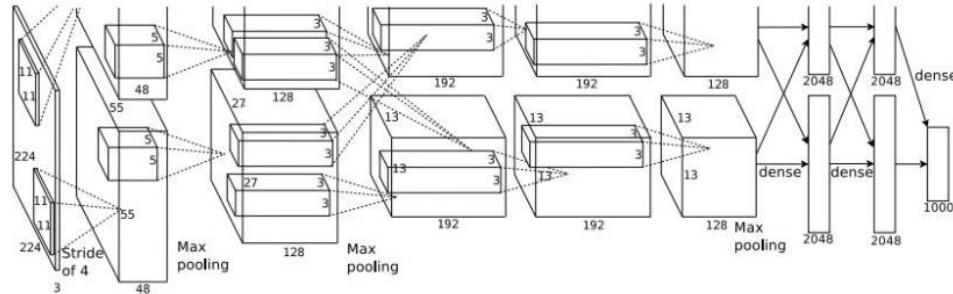


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# 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)

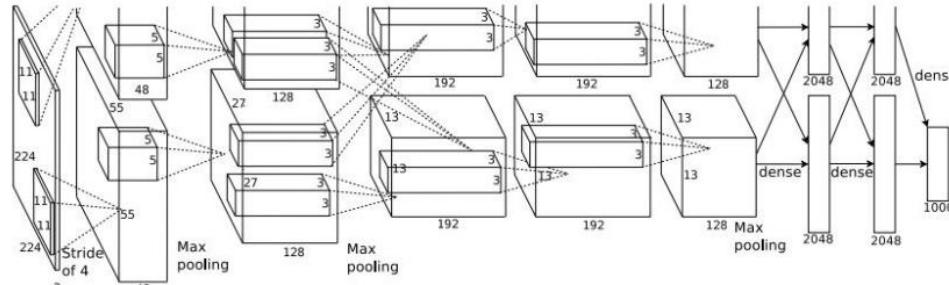


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# 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

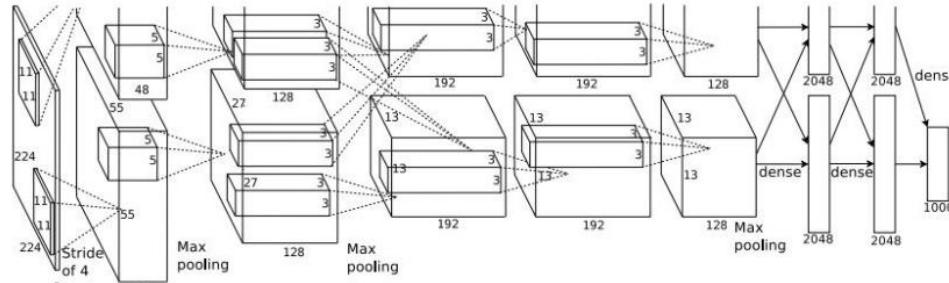
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1

[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 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%

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

# 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

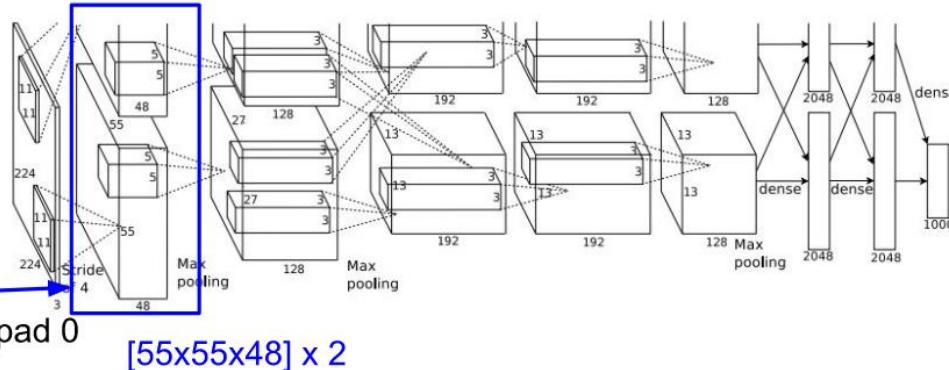
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1

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

[4096] FC6: 4096 neurons

[4096] FC7: 4096 neurons

[1000] FC8: 1000 neurons (class scores)



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.

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

# 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

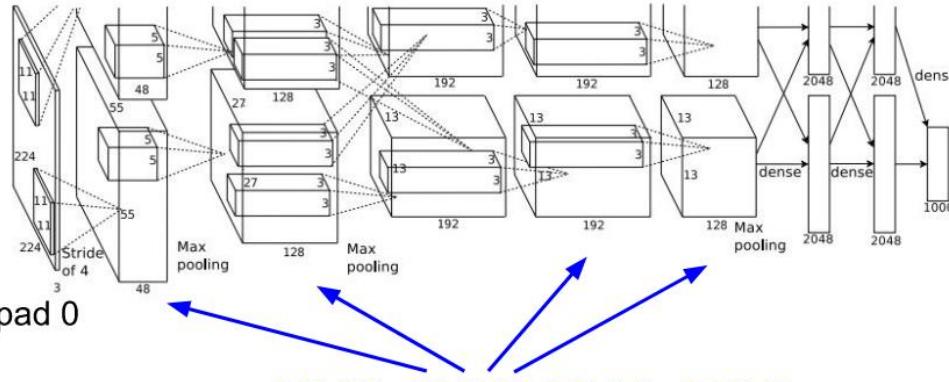
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1

[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

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

# 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

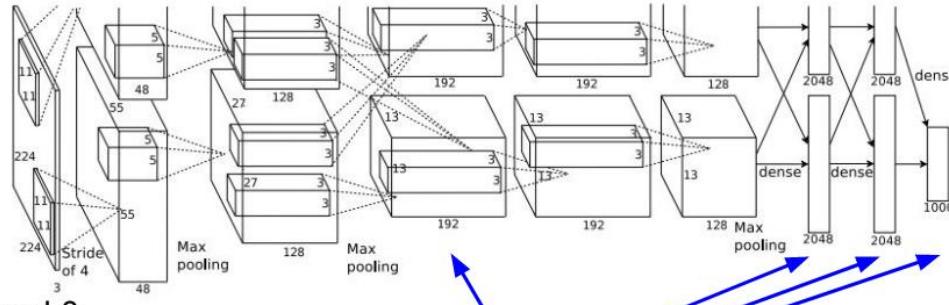
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1

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

[4096] FC6: 4096 neurons

[4096] FC7: 4096 neurons

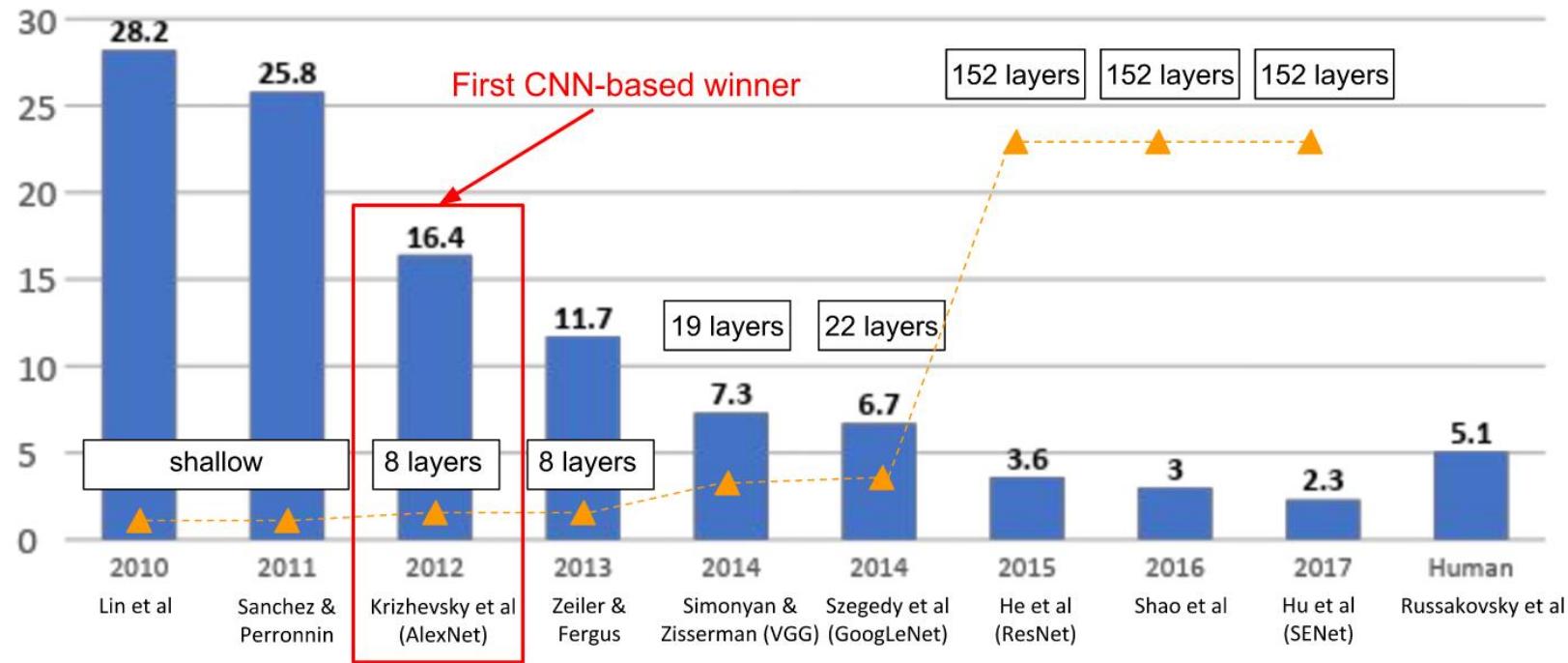
[1000] FC8: 1000 neurons (class scores)



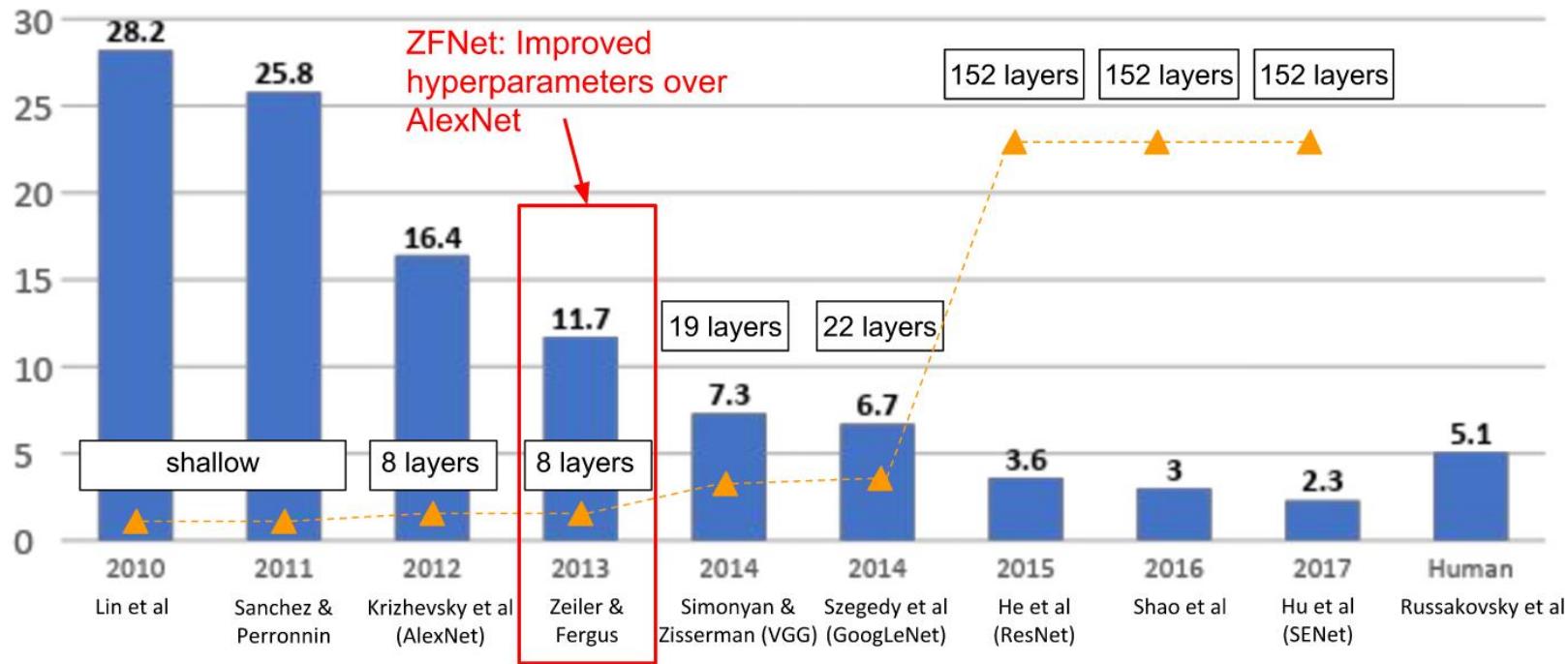
CONV3, FC6, FC7, FC8:  
Connections with all feature maps in  
preceding layer, communication  
across GPUs

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

# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

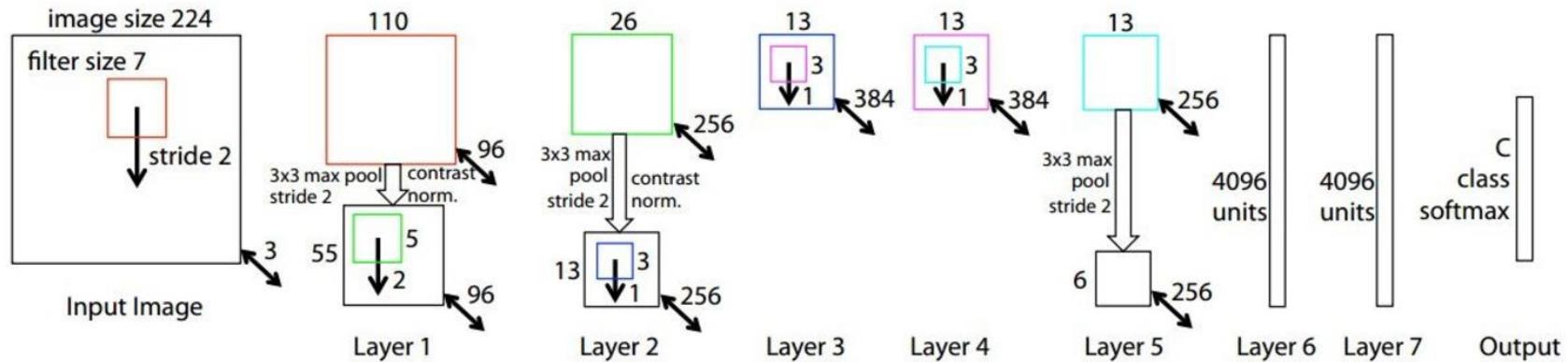


# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



# ZFNet

[Zeiler and Fergus, 2013]



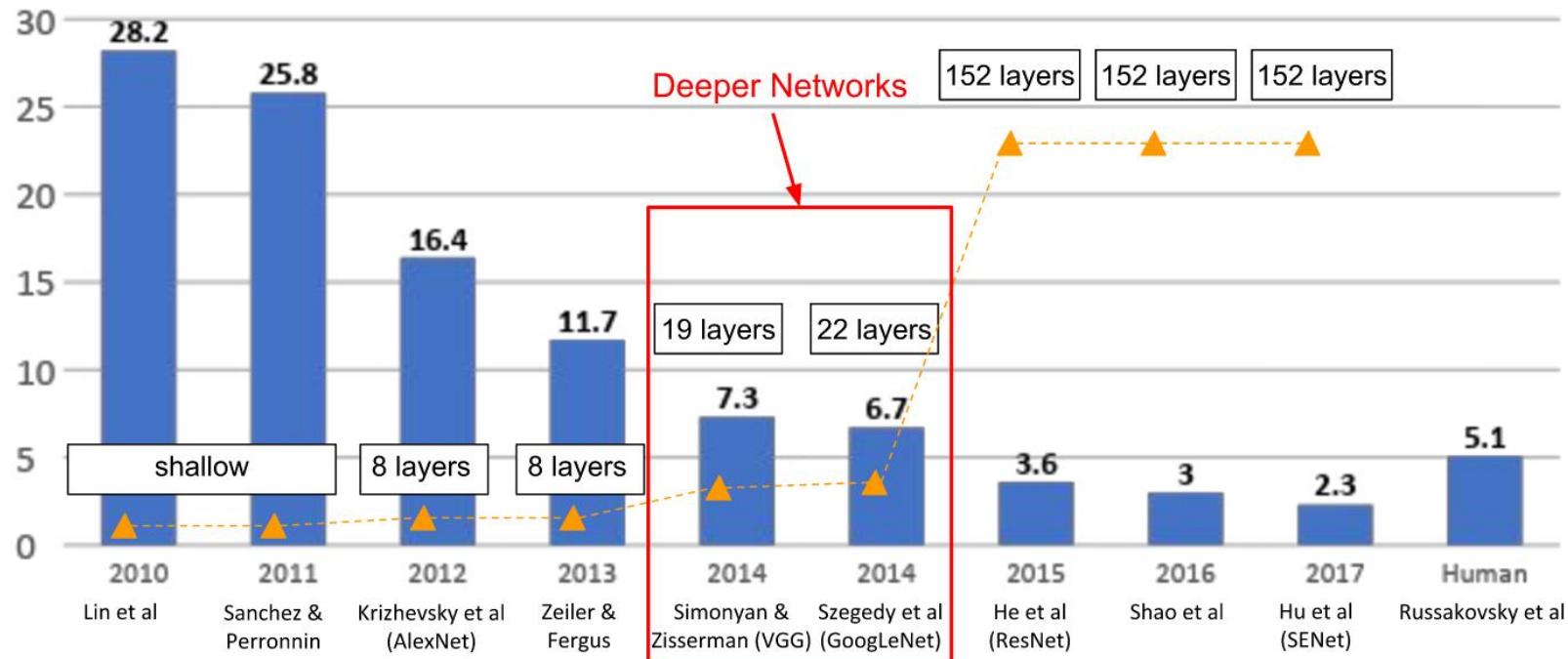
AlexNet but:

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%  $\rightarrow$  11.7%

# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

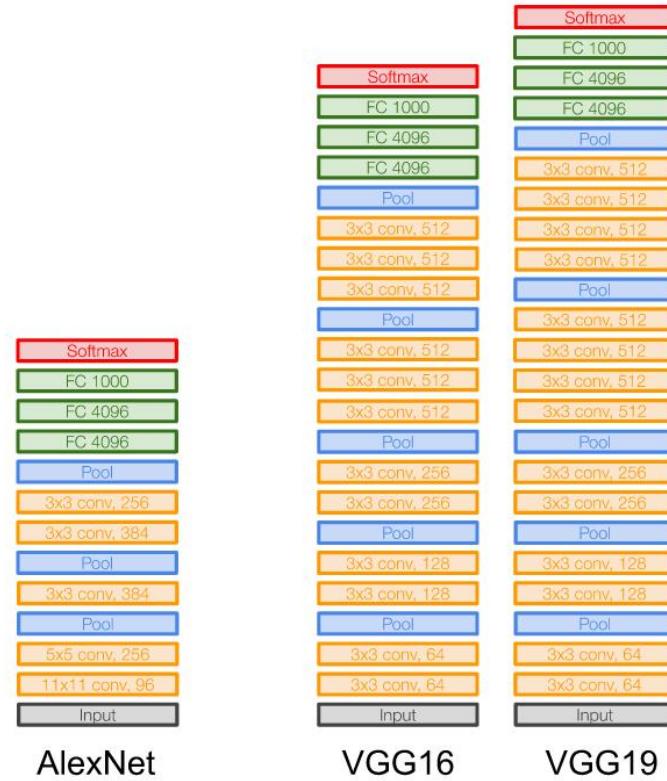
8 layers (AlexNet)

-> 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)

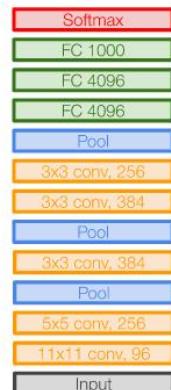
-> 7.3% top 5 error in ILSVRC'14



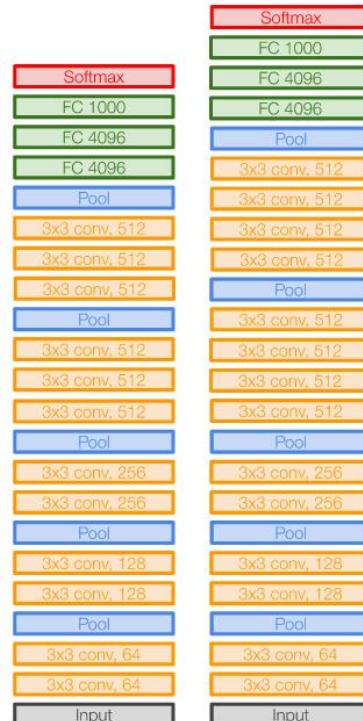
# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

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



AlexNet



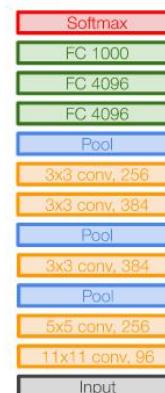
# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

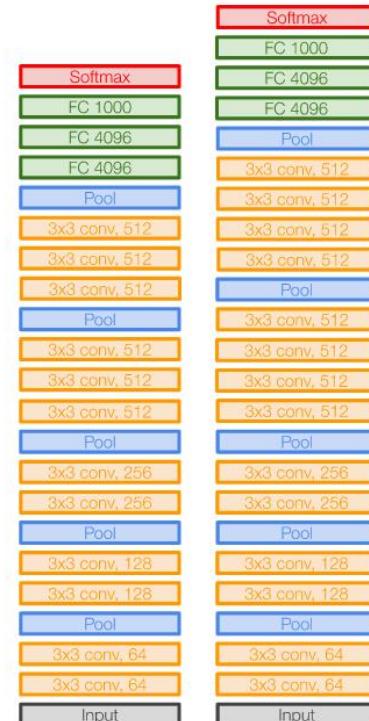
Q: Why use smaller filters? (3x3 conv)

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



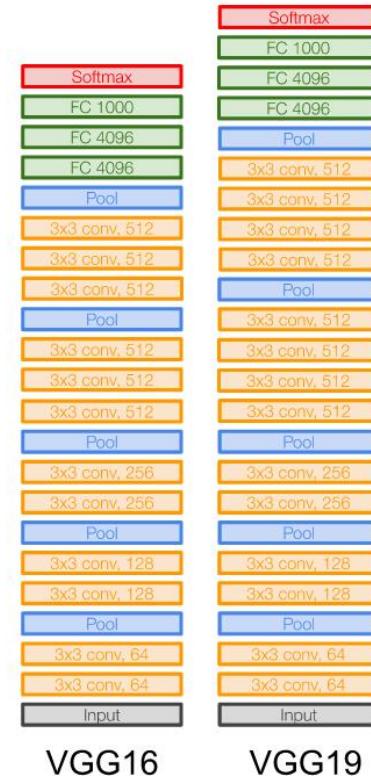
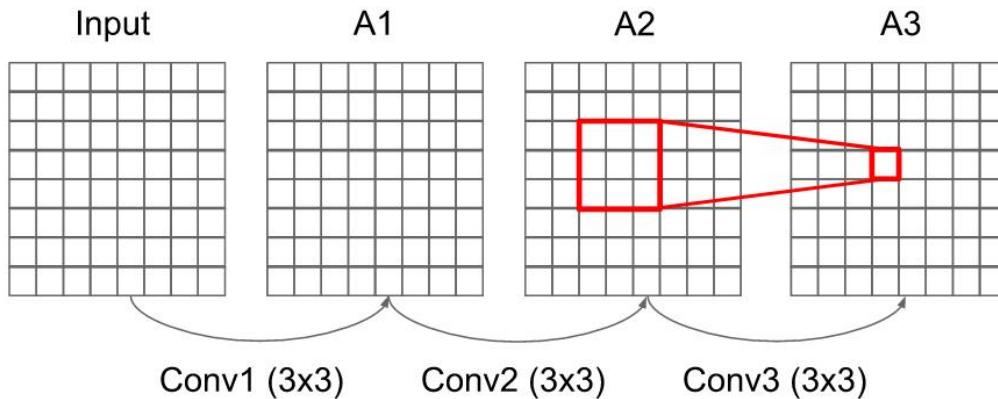
VGG16

VGG19

# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

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



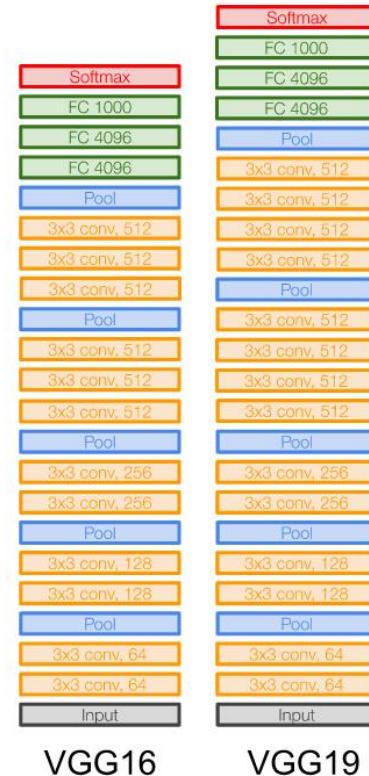
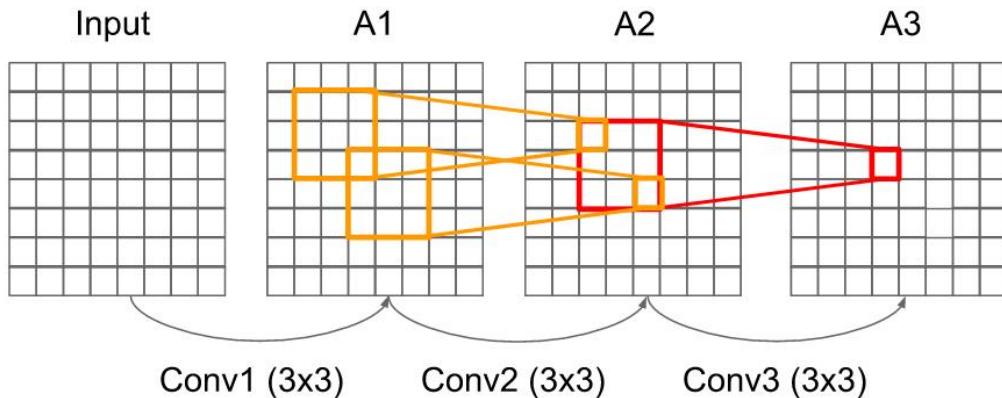
VGG16

VGG19

# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

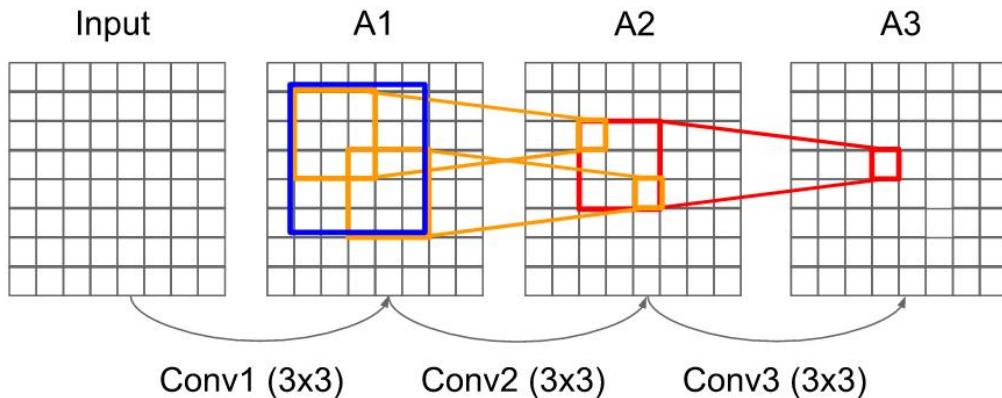
Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?



# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

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

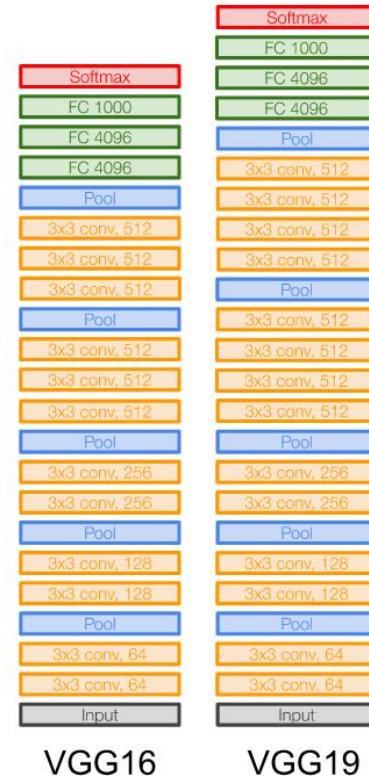
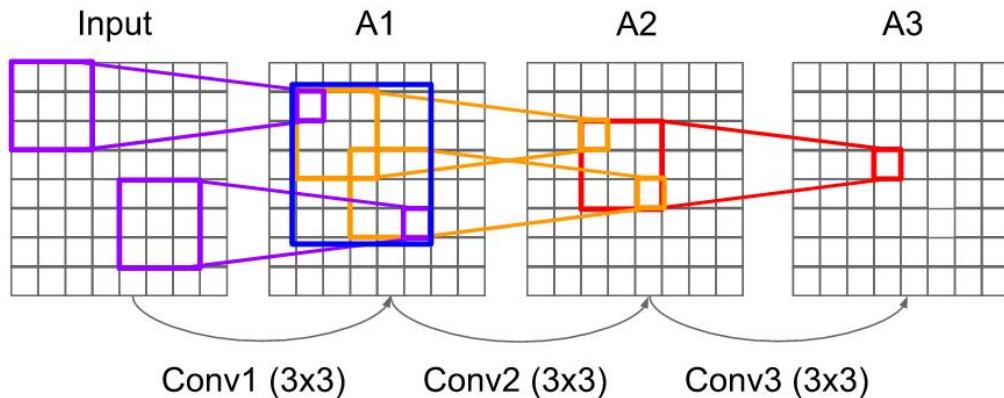


VGG16      VGG19

# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

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



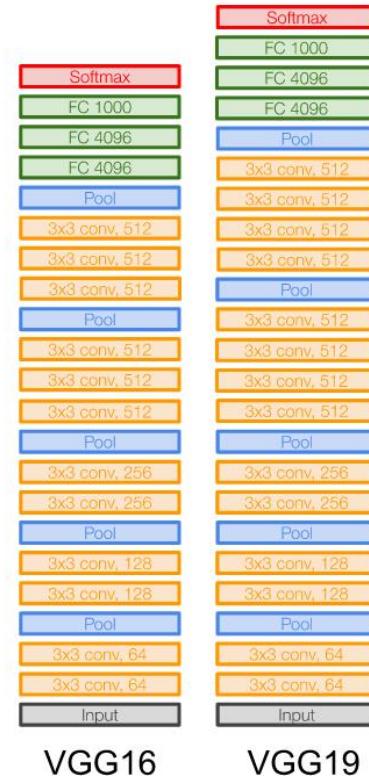
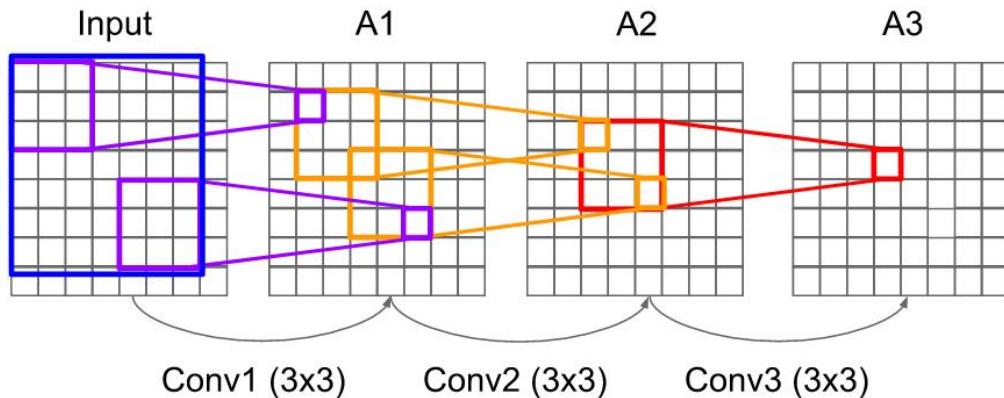
VGG16

VGG19

# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

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



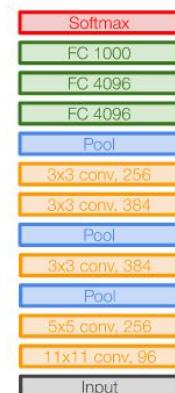
# Case Study: VGGNet

[Simonyan and Zisserman, 2014]

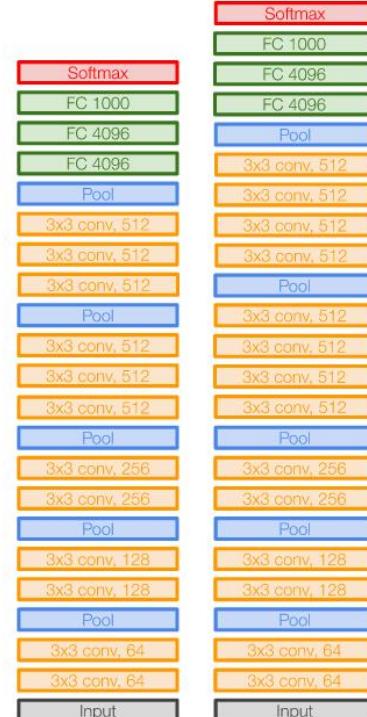
Q: Why use smaller filters? (3x3 conv)

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

[7x7]



AlexNet



VGG16

VGG19

# Case Study: VGGNet

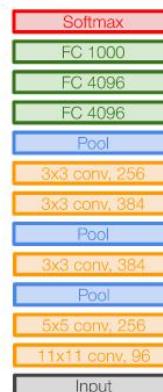
[Simonyan and Zisserman, 2014]

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

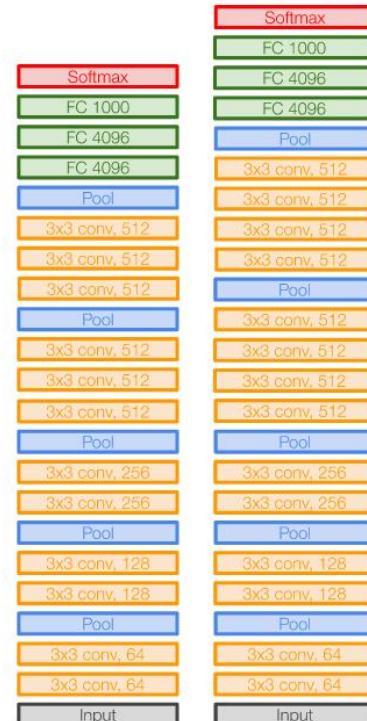
Stack of three 3x3 conv (stride 1) layers  
has same **effective receptive field** as  
one 7x7 conv layer

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

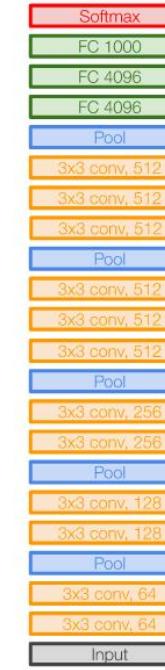
CONV3-512: [14x14x512] memory:  $14 \times 14 \times 512 = 100K$  params:  $(3 \times 3 \times 512) \times 512 = 2,359,296$

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

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

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

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



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

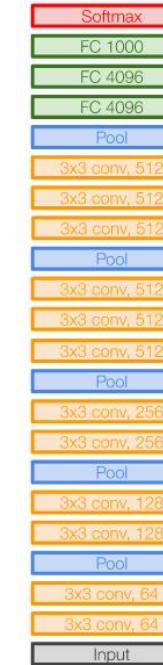
CONV3-512: [14x14x512] memory: 14\*14\*512=100K params:  $(3*3*512)*512 = 2,359,296$

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



VGG16

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

**TOTAL** params: 138M parameters

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Note:

Most memory is in early CONV

Most params are in late FC

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

TOTAL params: 138M parameters

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

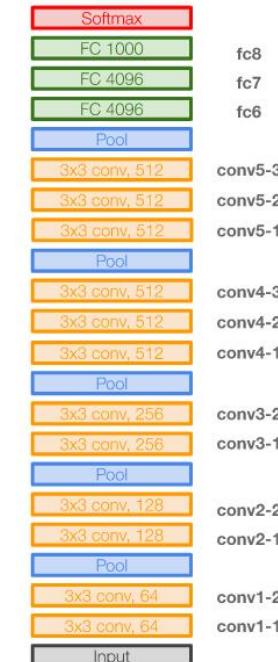
FC: [1x1x4096] memory: 4096 params:  $7 \times 7 \times 512 \times 4096 = 102,760,448$

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

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

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

**TOTAL** params: 138M parameters



VGG16

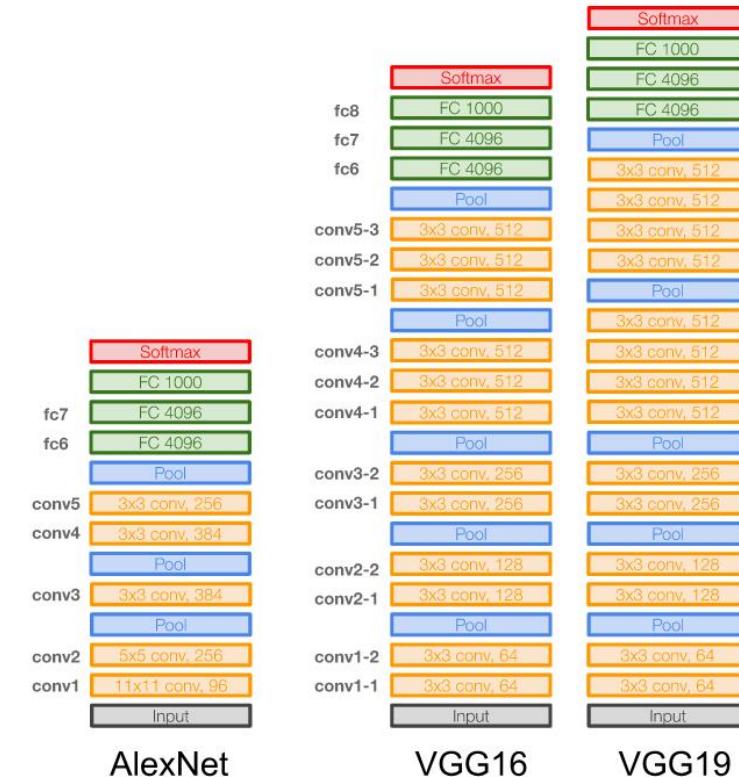
Common names

# Case Study: VGGNet

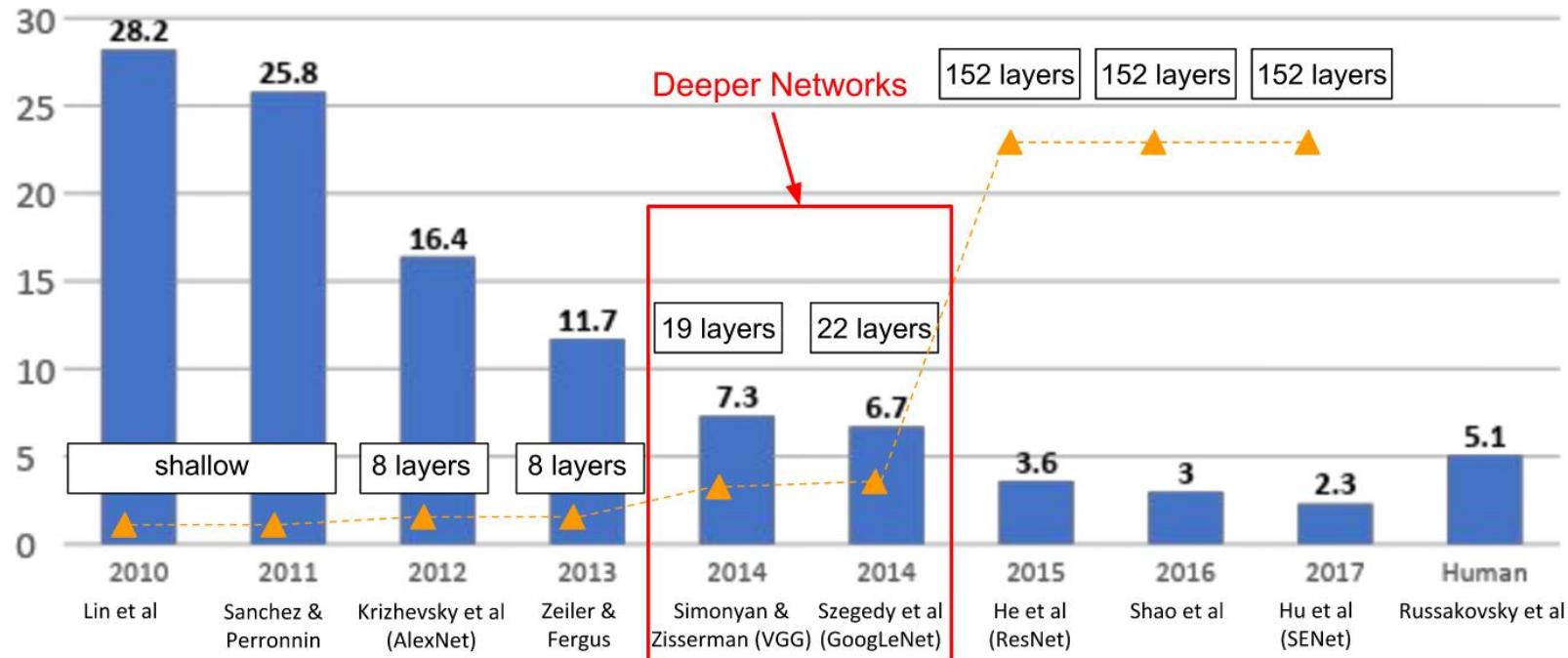
[Simonyan and Zisserman, 2014]

## 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



# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

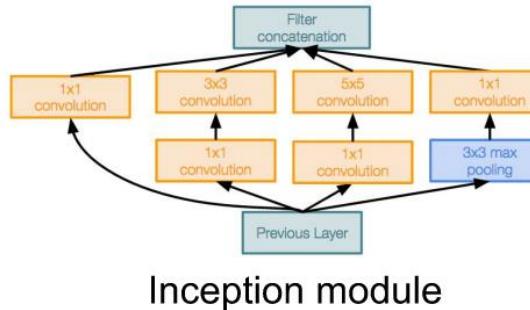


# Case Study: GoogLeNet

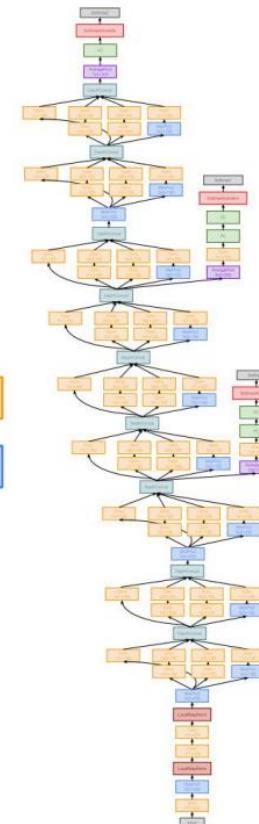
[Szegedy et al., 2014]

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



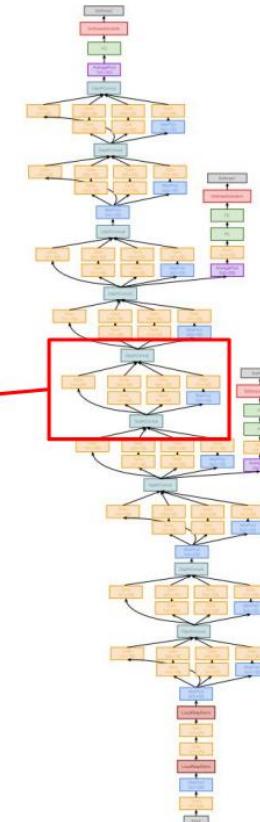
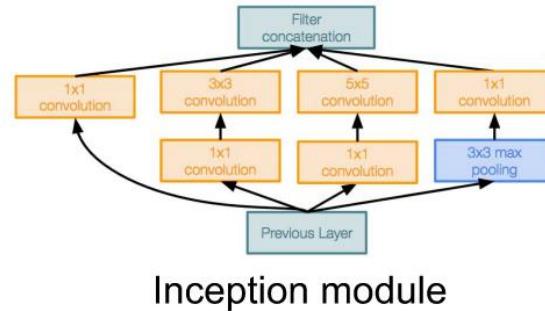
Inception module



# Case Study: GoogLeNet

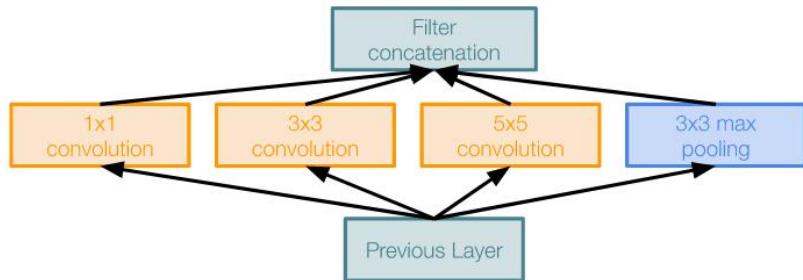
[Szegedy et al., 2014]

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



# Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

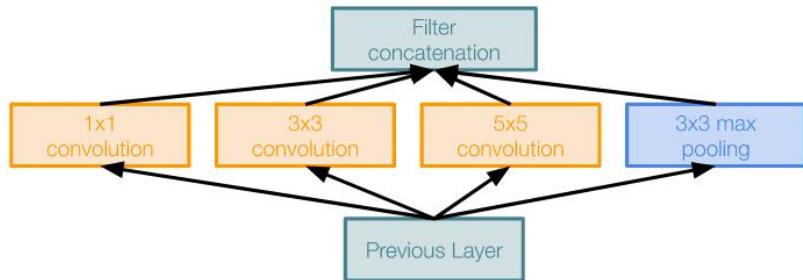
Apply parallel filter operations on the input from previous layer:

- 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]

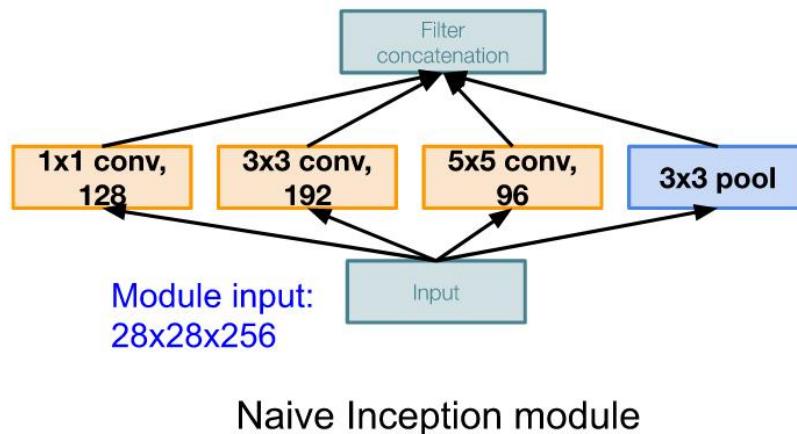
# Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

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

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



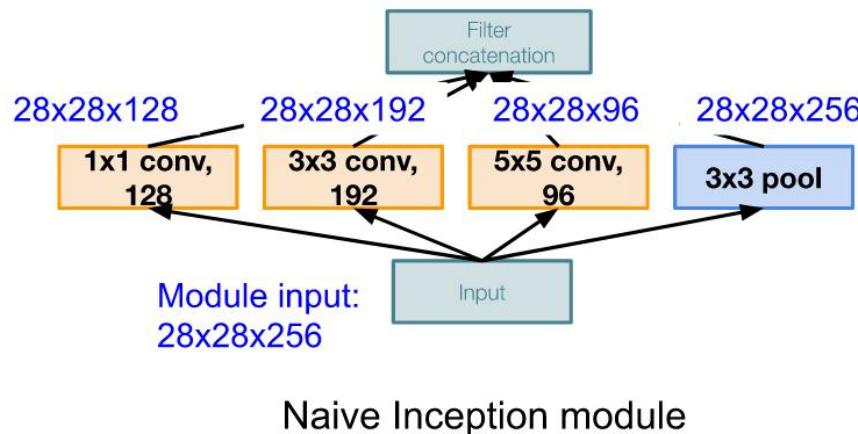
Naive Inception module

# Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

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



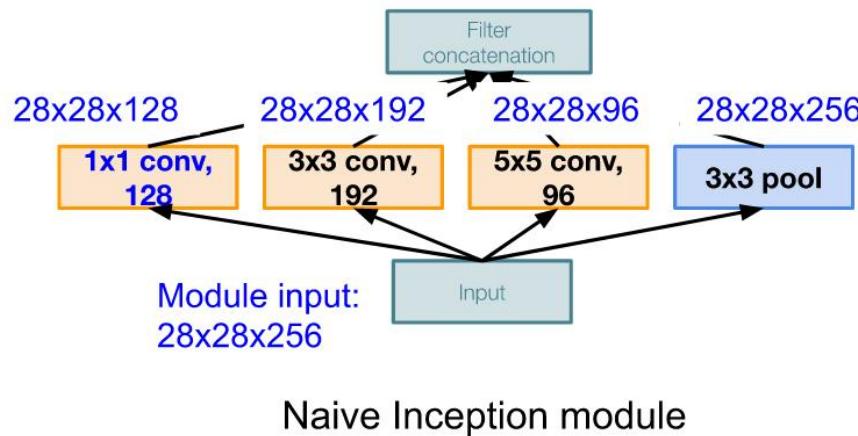
Q: What is the problem with this?  
[Hint: Computational complexity]

# Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q2: What is output size after  
filter concatenation?



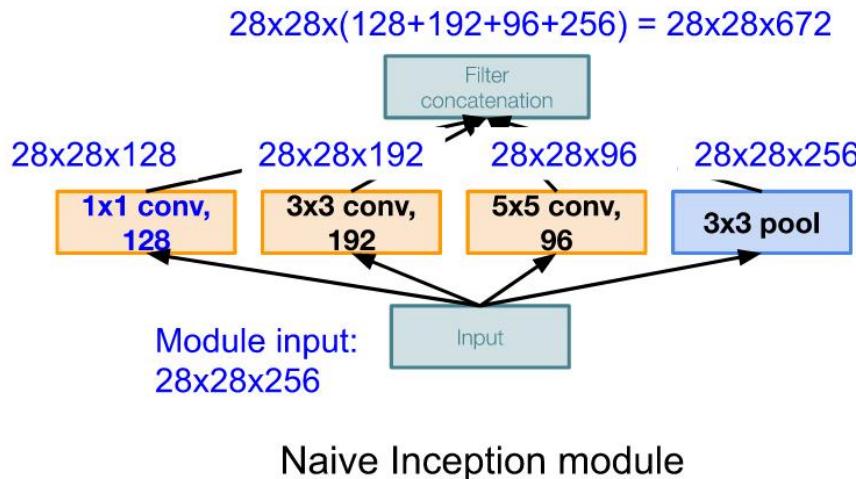
Q: What is the problem with this?  
[Hint: Computational complexity]

# Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q2: What is output size after filter concatenation?



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

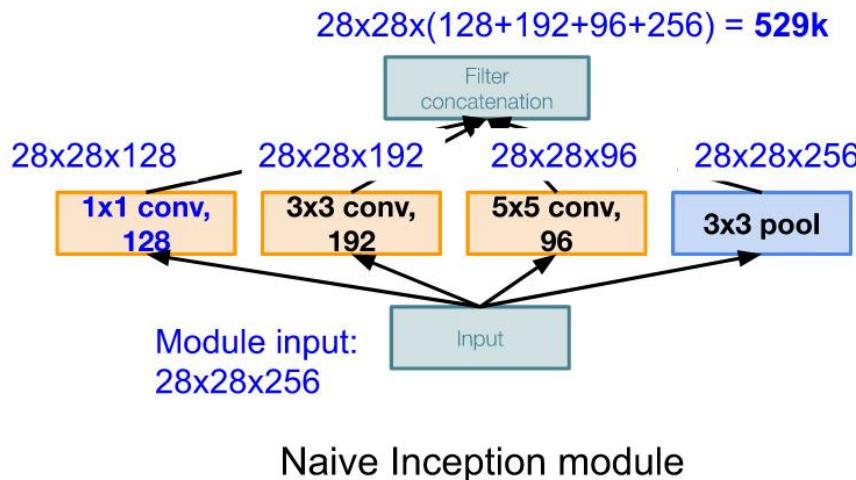
Pooling layer also preserves feature depth, which means total depth after concatenation can only grow at every layer!

# Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q2: What is output size after filter concatenation?

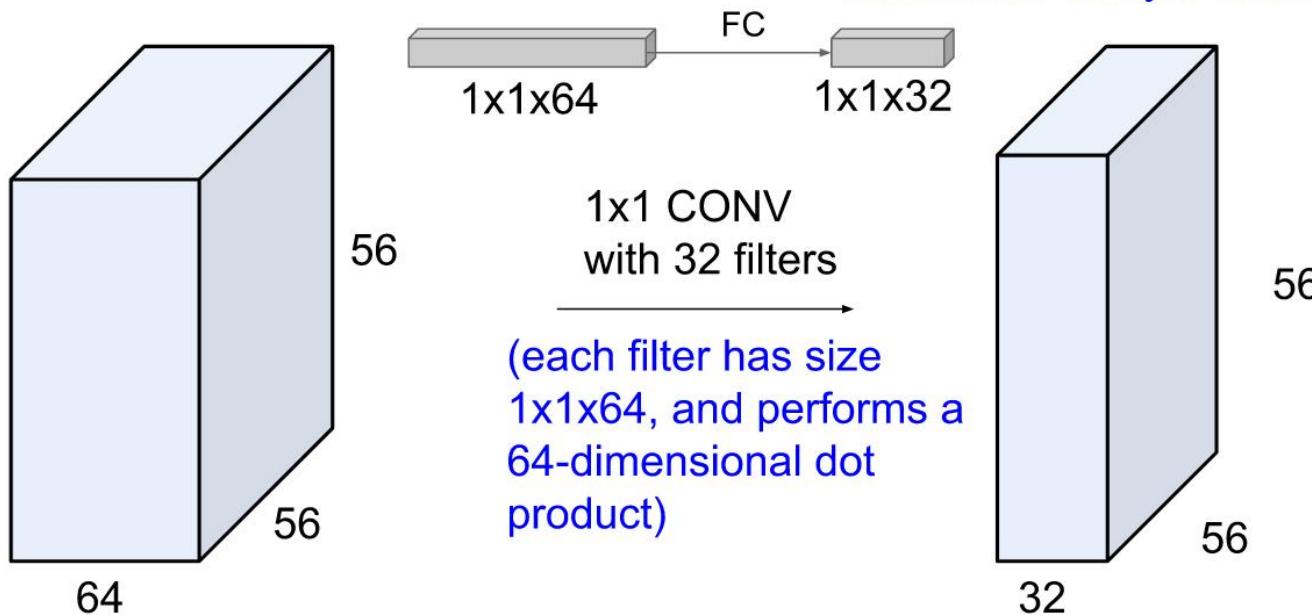


Q: What is the problem with this?  
[Hint: Computational complexity]

Solution: “bottleneck” layers that use 1x1 convolutions to reduce feature channel size

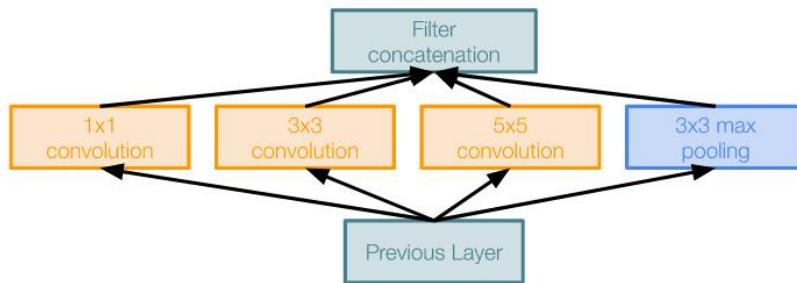
# Review: 1x1 convolutions

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



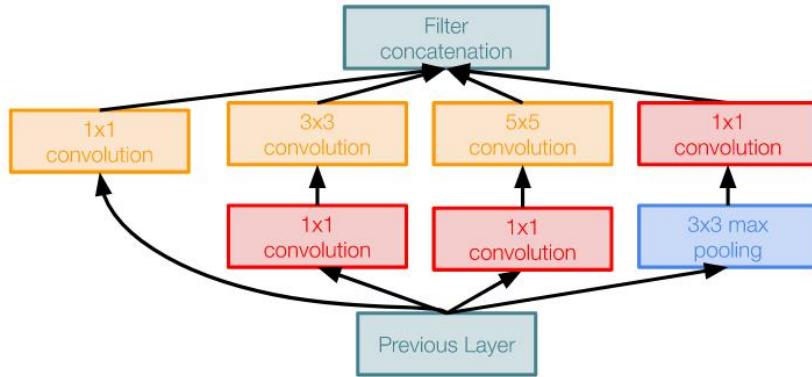
# Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

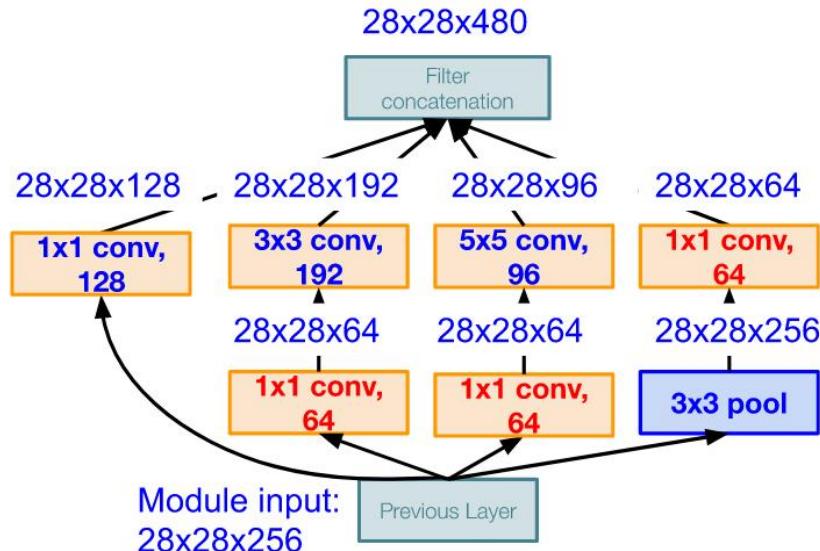
1x1 conv “bottleneck”  
layers



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 “1x1 conv, 64 filter” bottlenecks:

## Conv Ops:

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

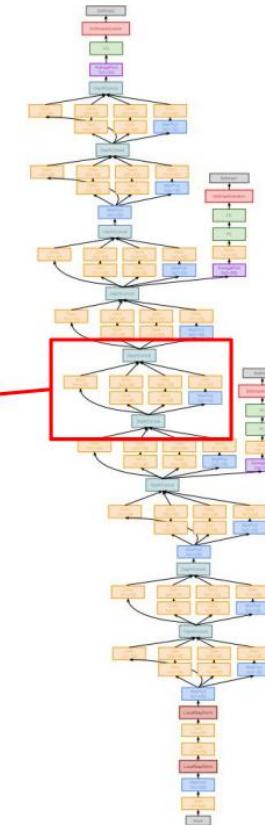
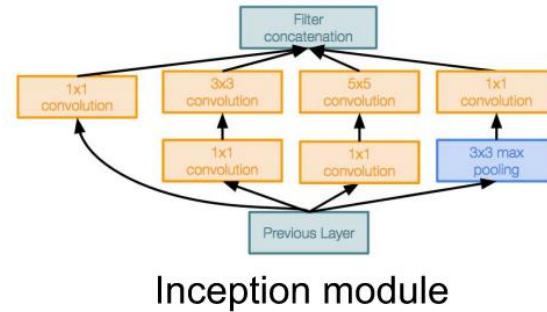
**Total: 358M ops**

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

# Case Study: GoogLeNet

[Szegedy et al., 2014]

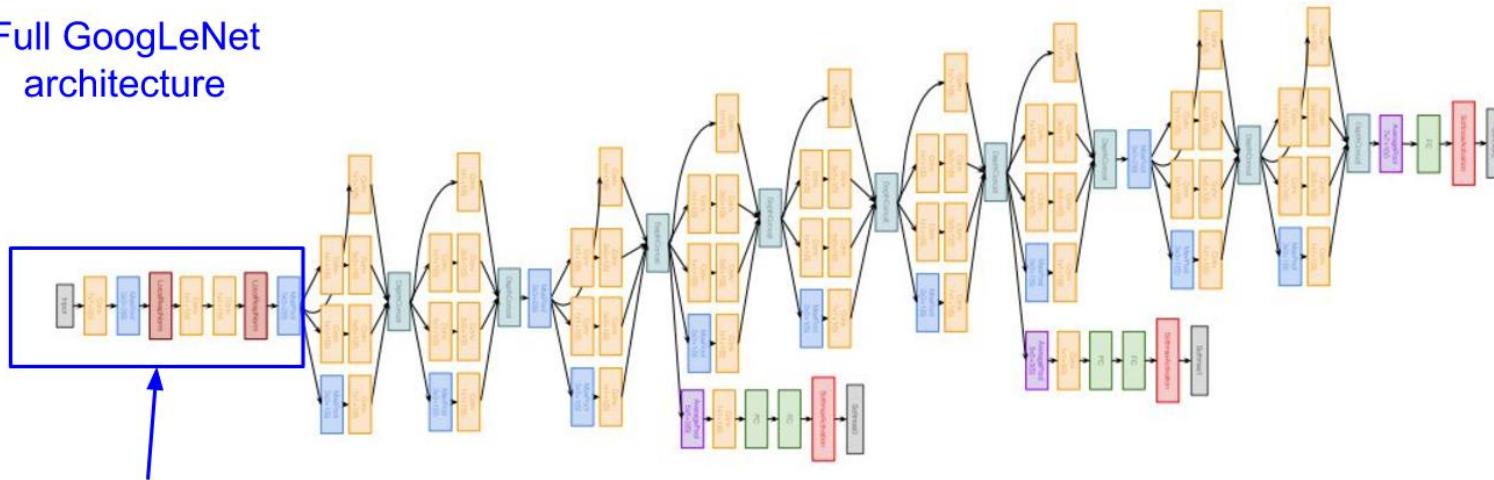
Stack Inception modules  
with dimension reduction  
on top of each other



# Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet  
architecture

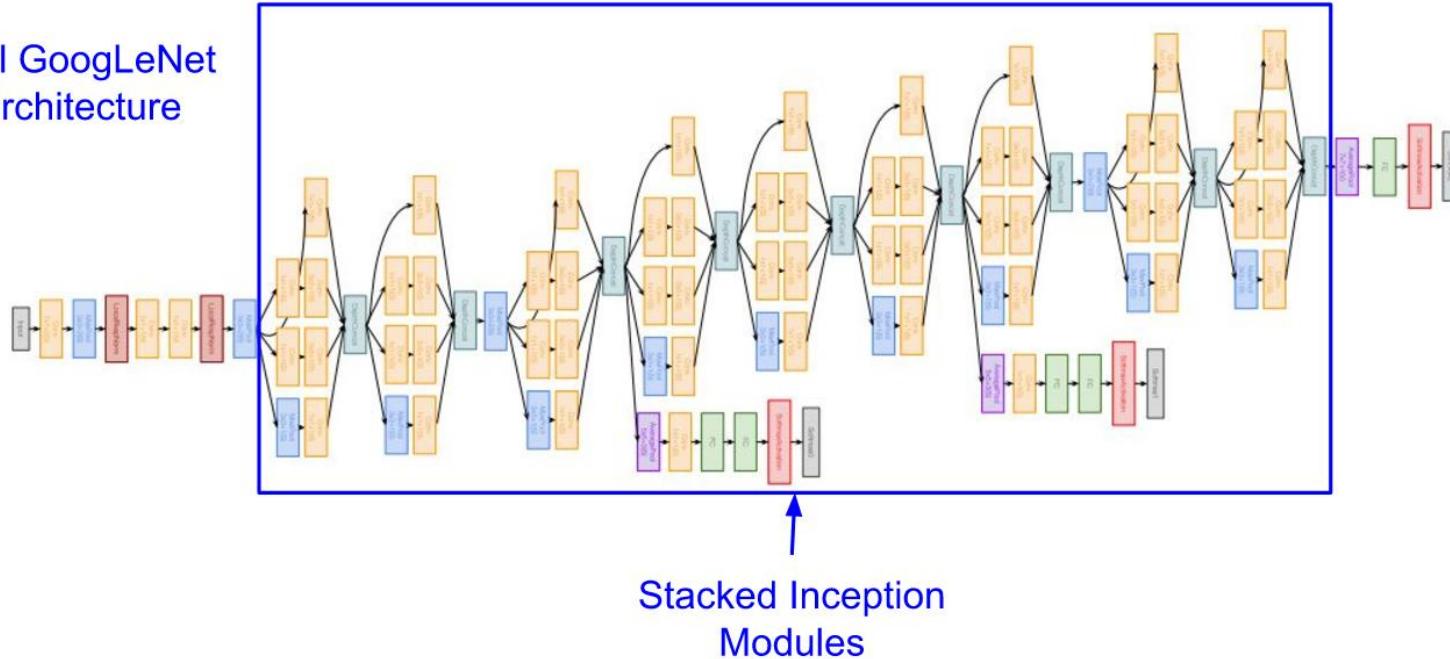


Stem Network:  
Conv-Pool-  
2x Conv-Pool

# Case Study: GoogLeNet

[Szegedy et al., 2014]

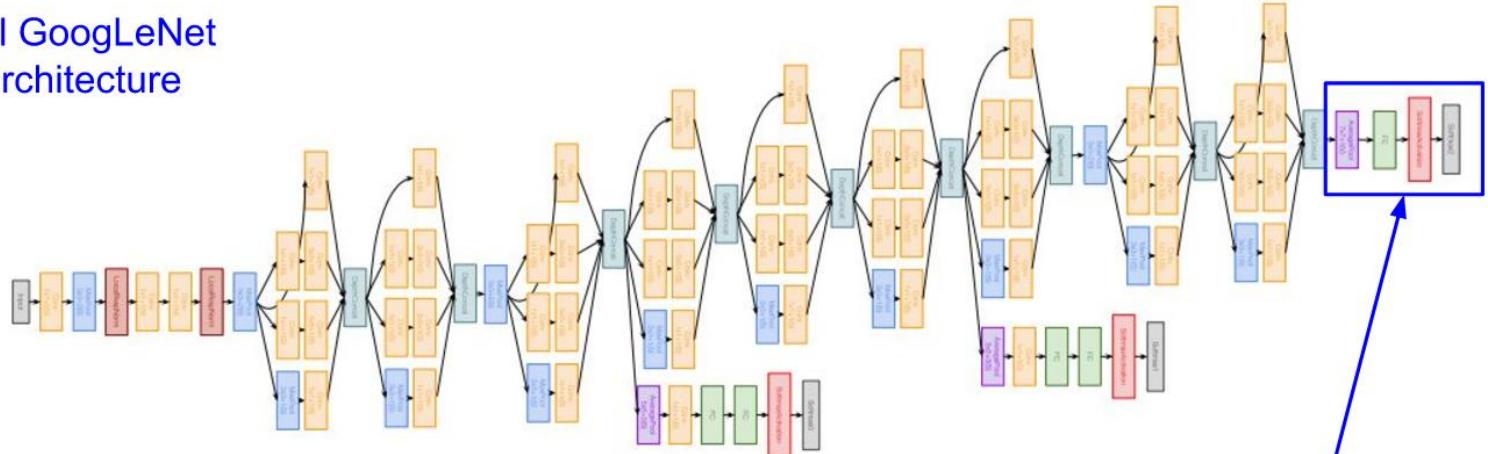
Full GoogLeNet  
architecture



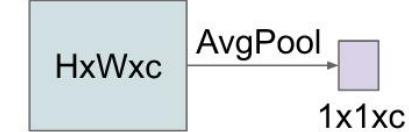
# Case Study: GoogLeNet

[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!

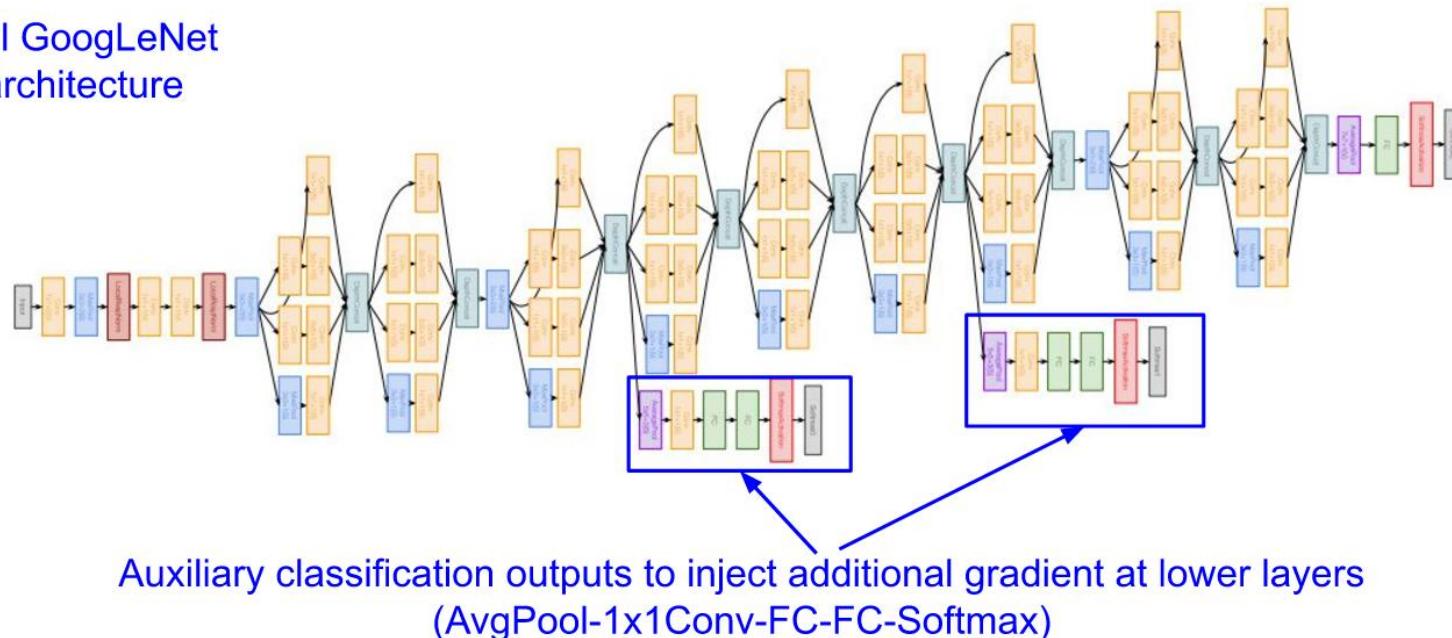


Classifier output

# Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet  
architecture

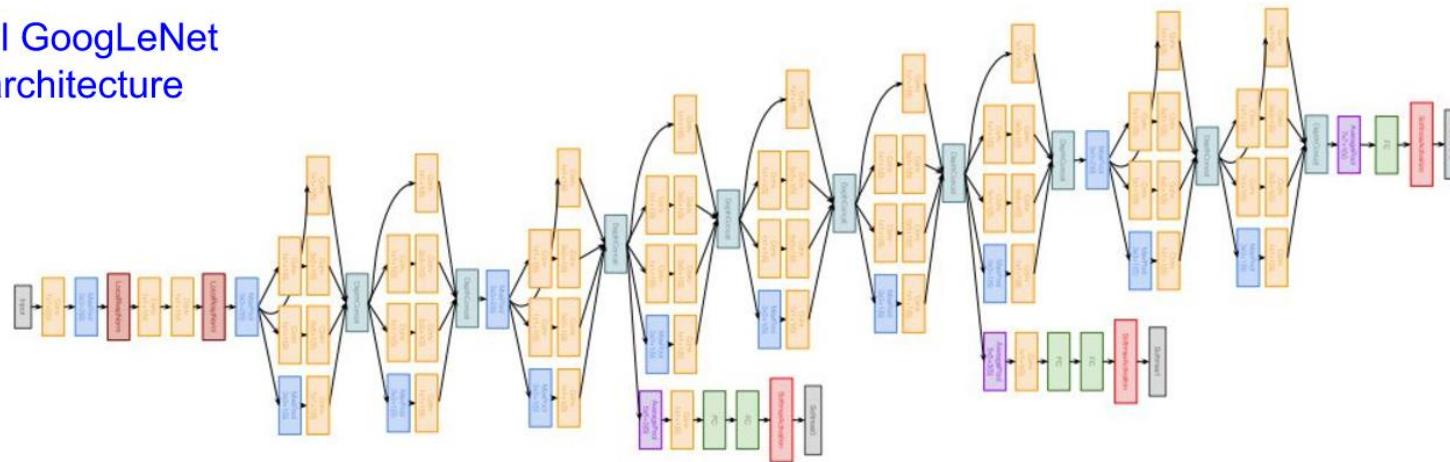


Auxiliary classification outputs to inject additional gradient at lower layers  
(AvgPool-1x1Conv-FC-FC-Softmax)

# Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet  
architecture



22 total layers with weights

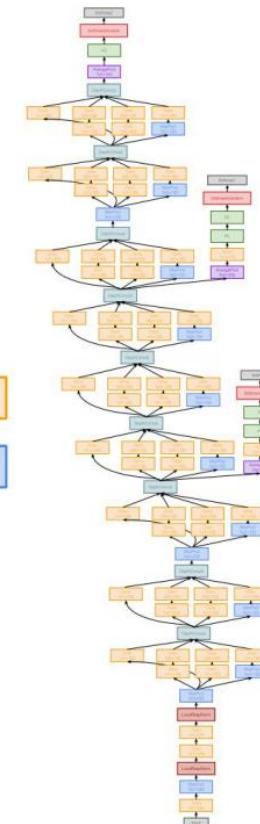
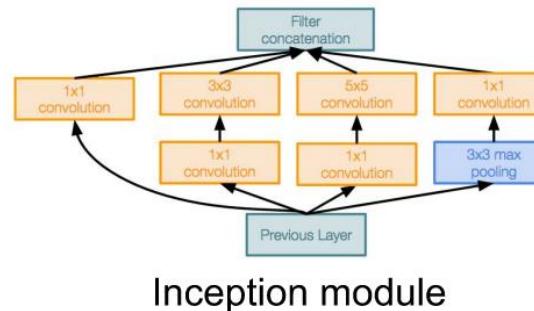
(parallel layers count as 1 layer => 2 layers per Inception module. Don't count auxiliary output layers)

# Case Study: GoogLeNet

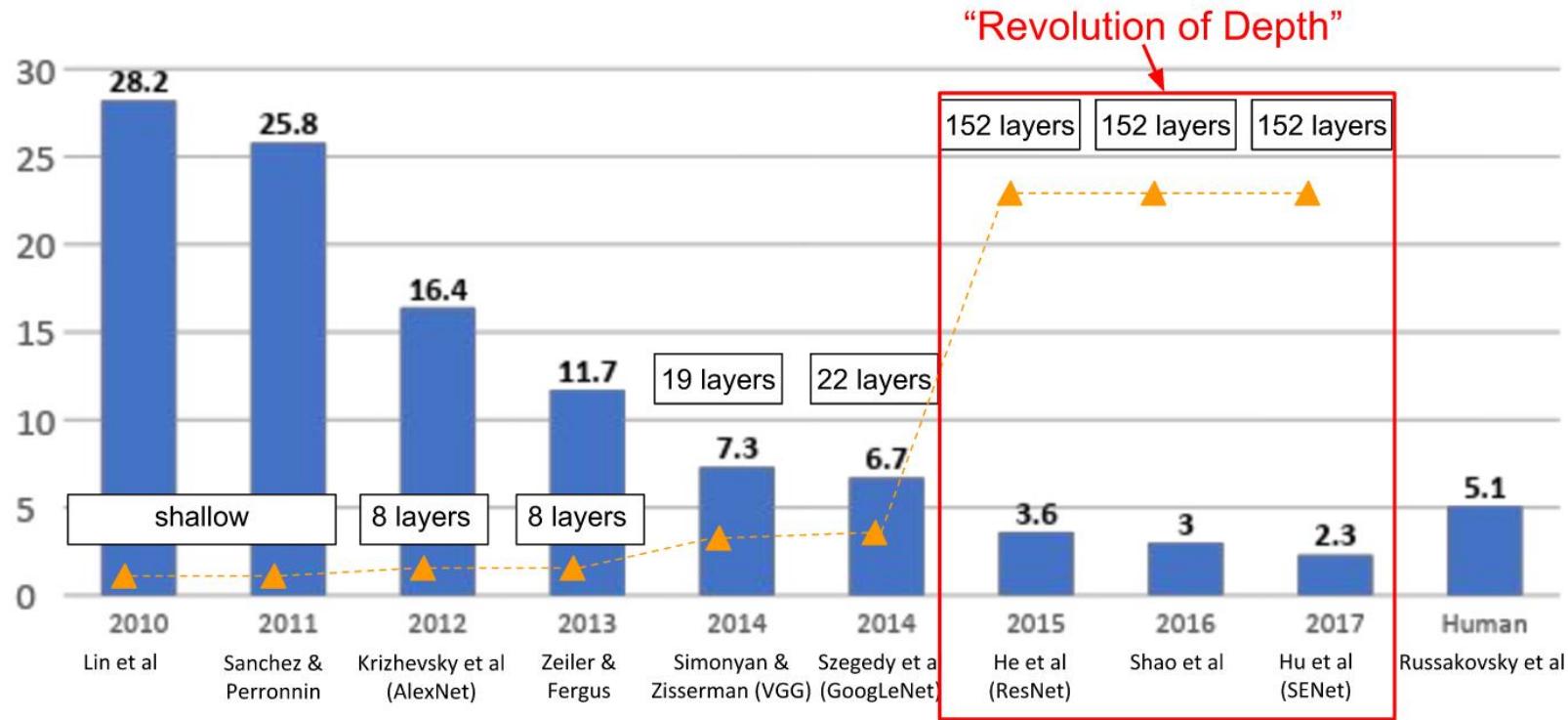
[Szegedy et al., 2014]

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

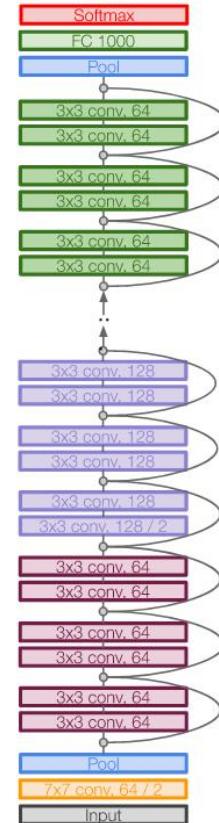
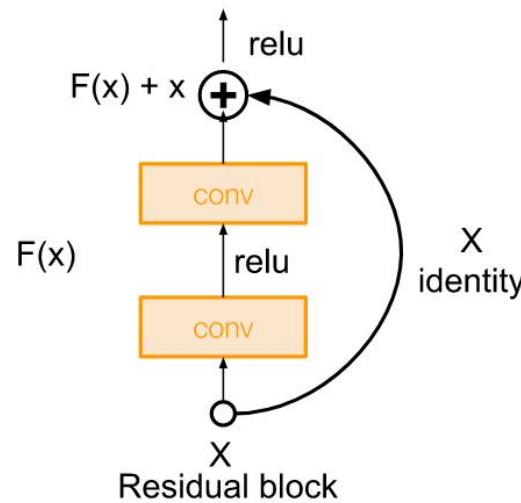


# Case Study: ResNet

[He et al., 2015]

Very deep networks using residual connections

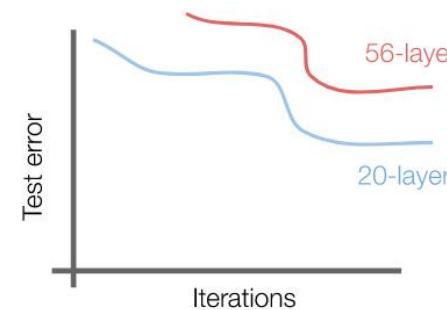
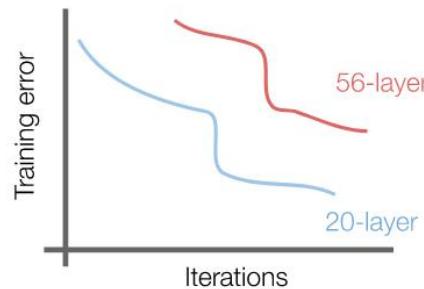
- 152-layer model for ImageNet
- ILSVRC'15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC'15 and COCO'15!



# Case Study: ResNet

[He et al., 2015]

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



56-layer model performs worse on both training and test 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

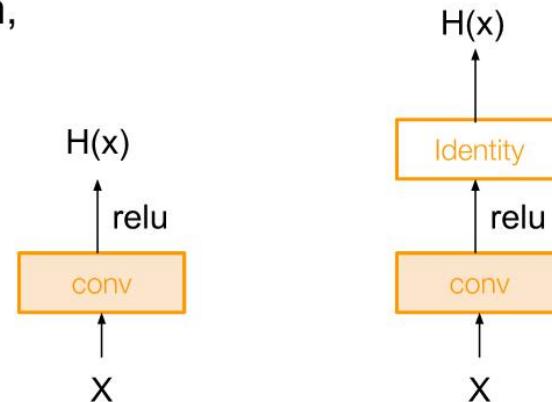
[He et al., 2015]

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?

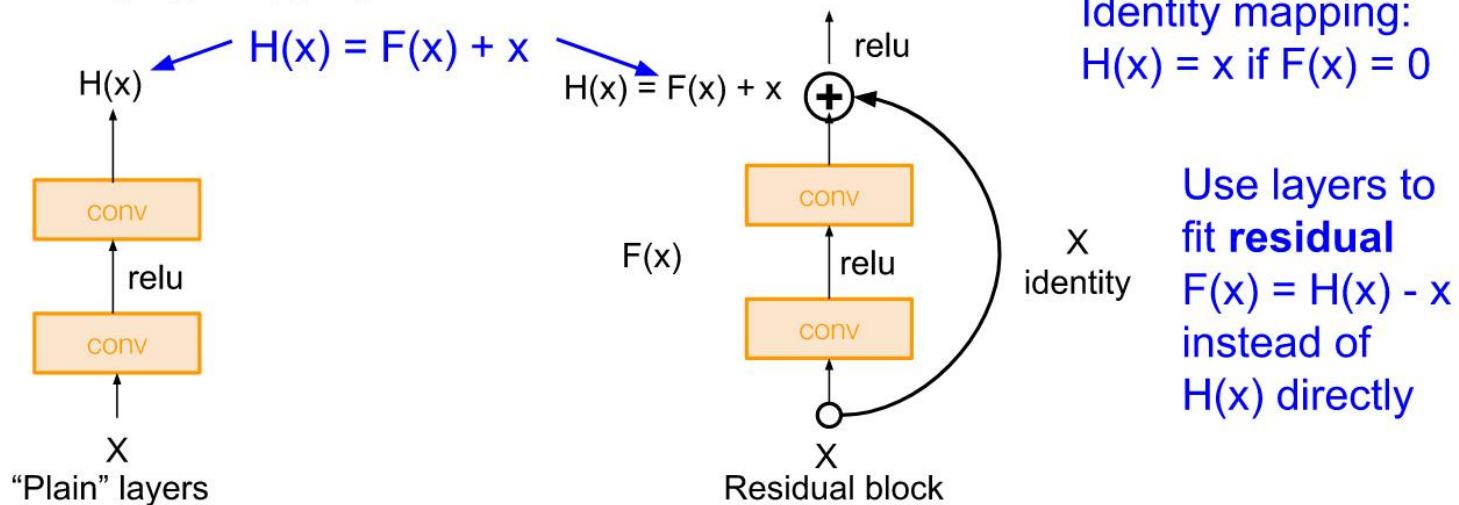
A solution by construction is copying the learned layers from the shallower model and setting additional layers to identity mapping.



# 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

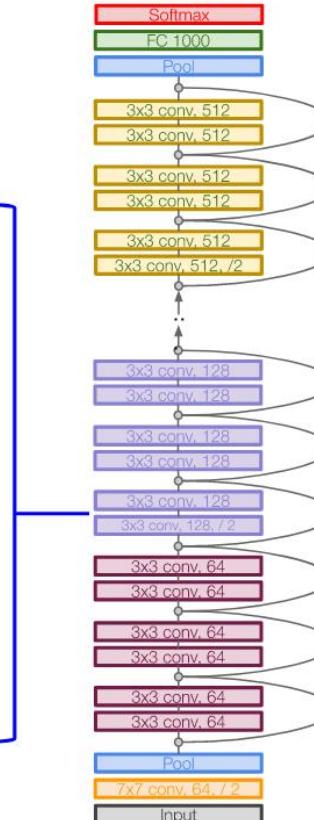
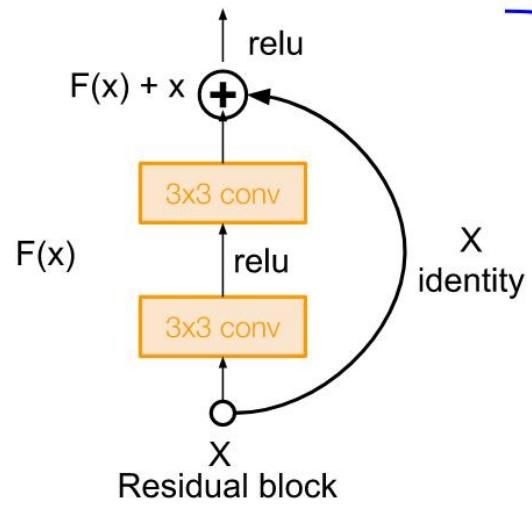


# Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

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

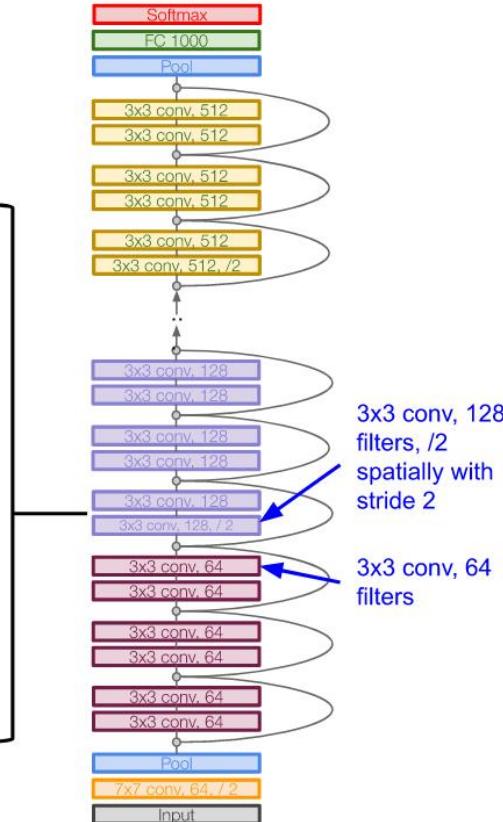
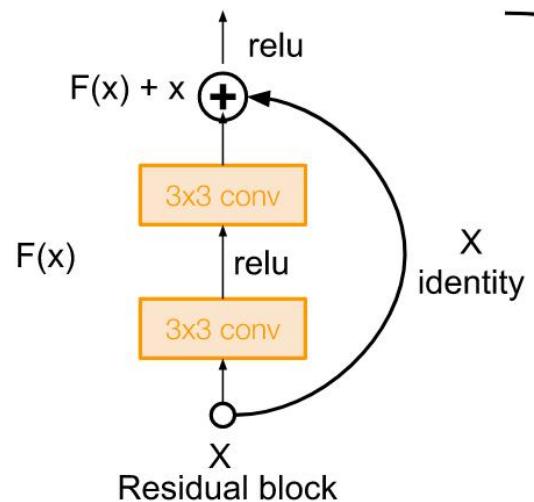


# Case Study: ResNet

[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.

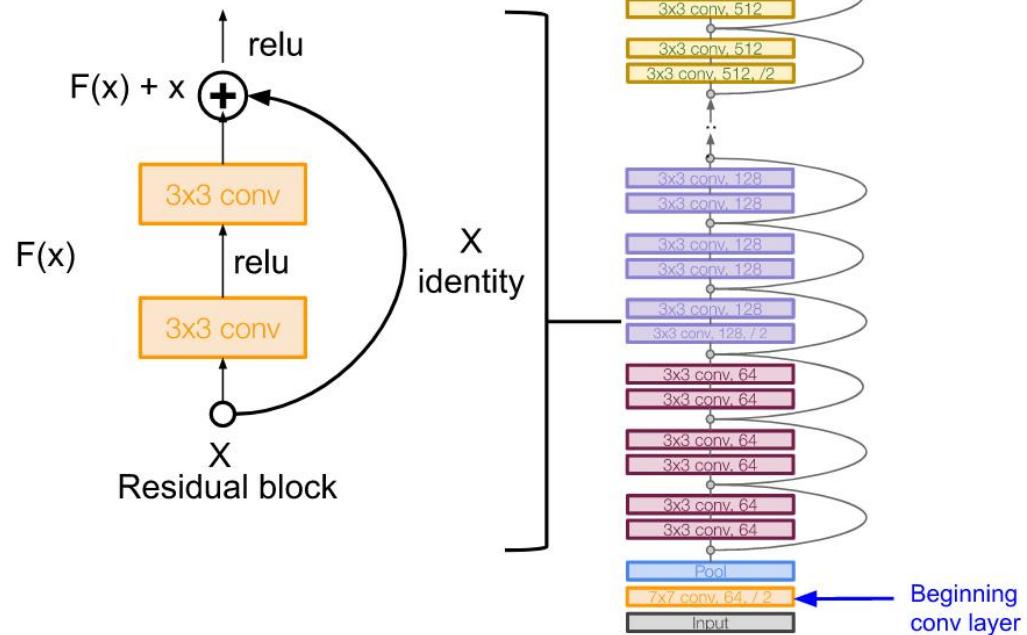


# Case Study: ResNet

[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)

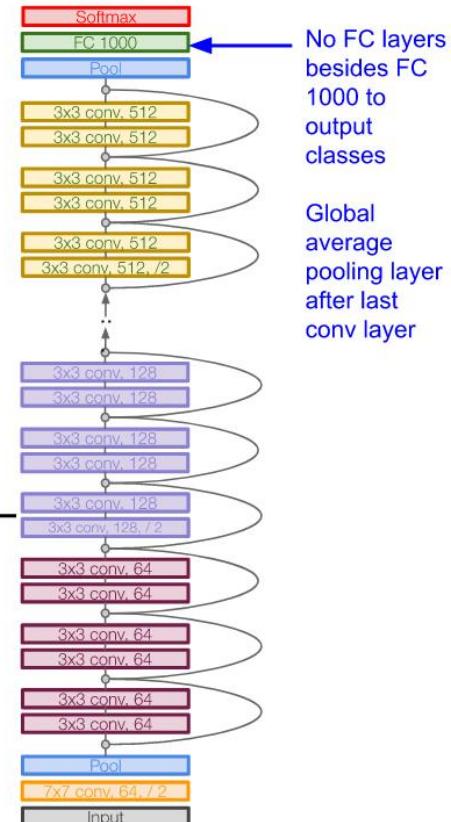
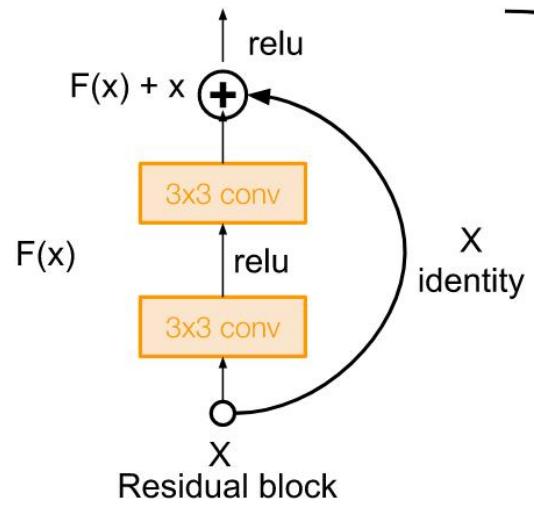


# Case Study: ResNet

[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)



# Review: VGGNet

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

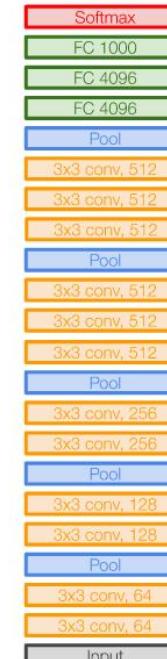
FC: [1x1x4096]      memory:  $4096$     params:  $7 \times 7 \times 512 \times 4096 = 102,760,448$

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

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

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

**TOTAL** params: 138M parameters

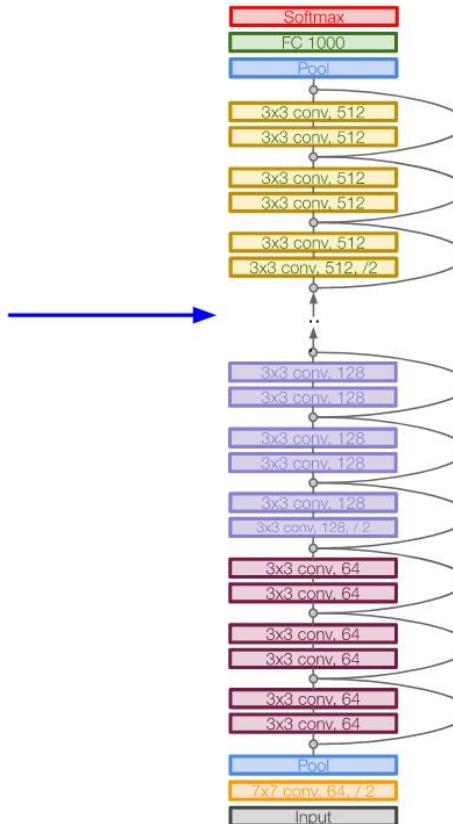


VGG16

# Case Study: ResNet

[He et al., 2015]

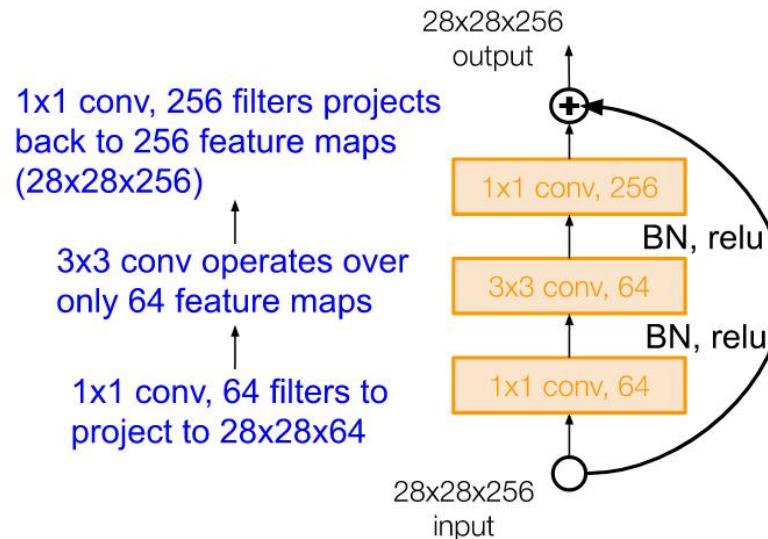
Total depths of 18, 34, 50,  
101, or 152 layers for  
ImageNet



# 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]

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

ILSVRC 2015 classification winner (3.6% top 5 error) -- better than “human performance”! (Russakovsky 2014)

