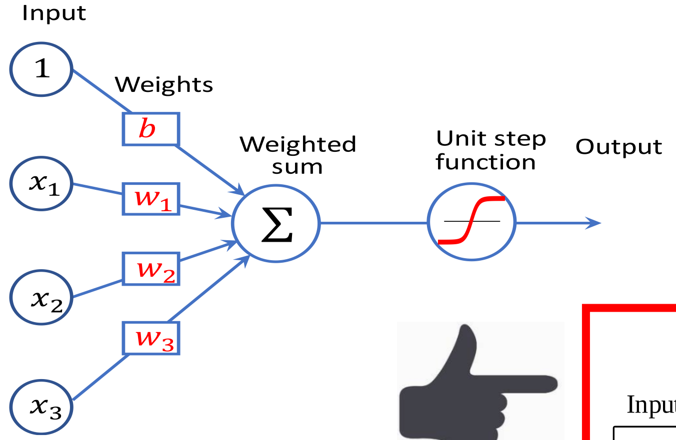
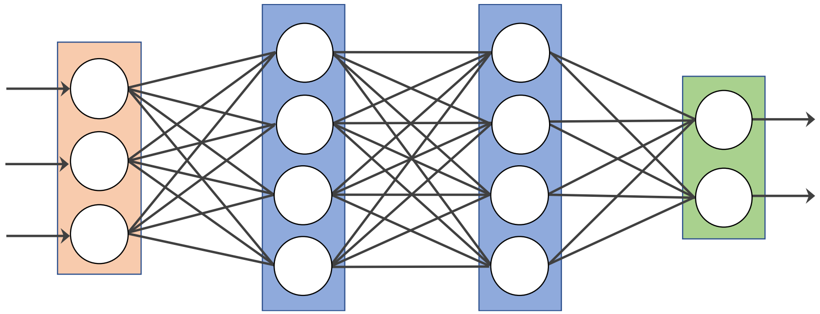
In this Course

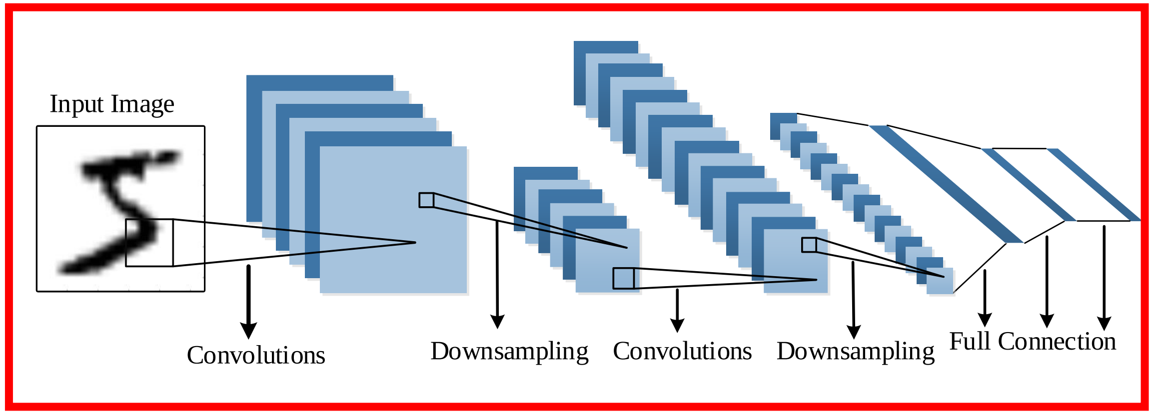
1. DL basics, linear regression, logistic regression etc.



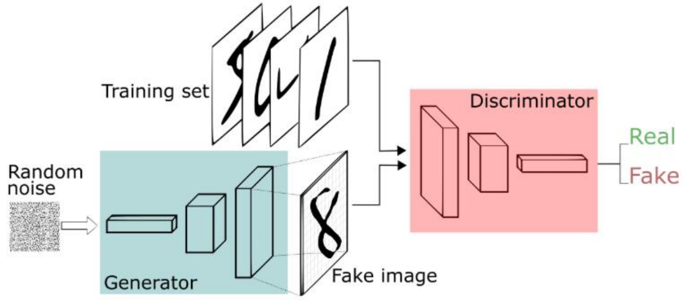
2. Multilayer neural networks, backpropagation



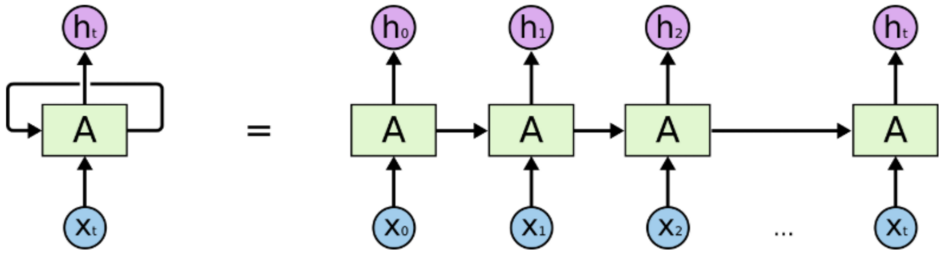
3. Convolutional Neural Networks and Applications



4. Generative Adversarial Networks

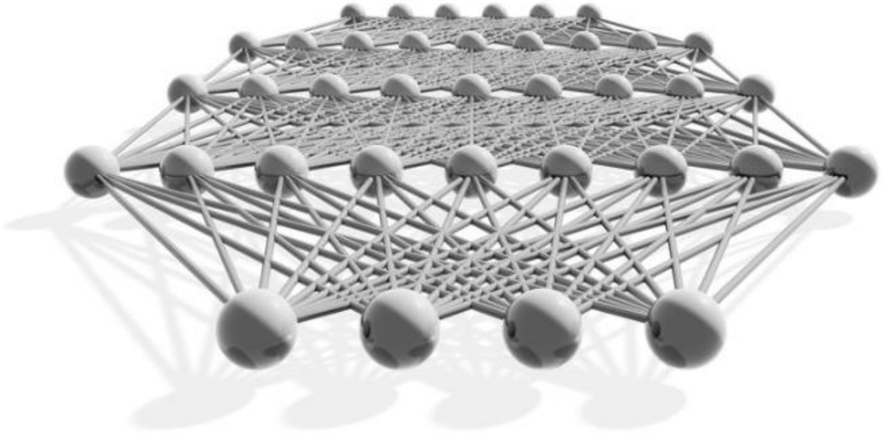


5. Recurrent networks and applications



Last Lecture

- Neural Networks
- Multilayer Neural Networks
- Backpropagation



Lecture 4

Convolutional Neural Networks

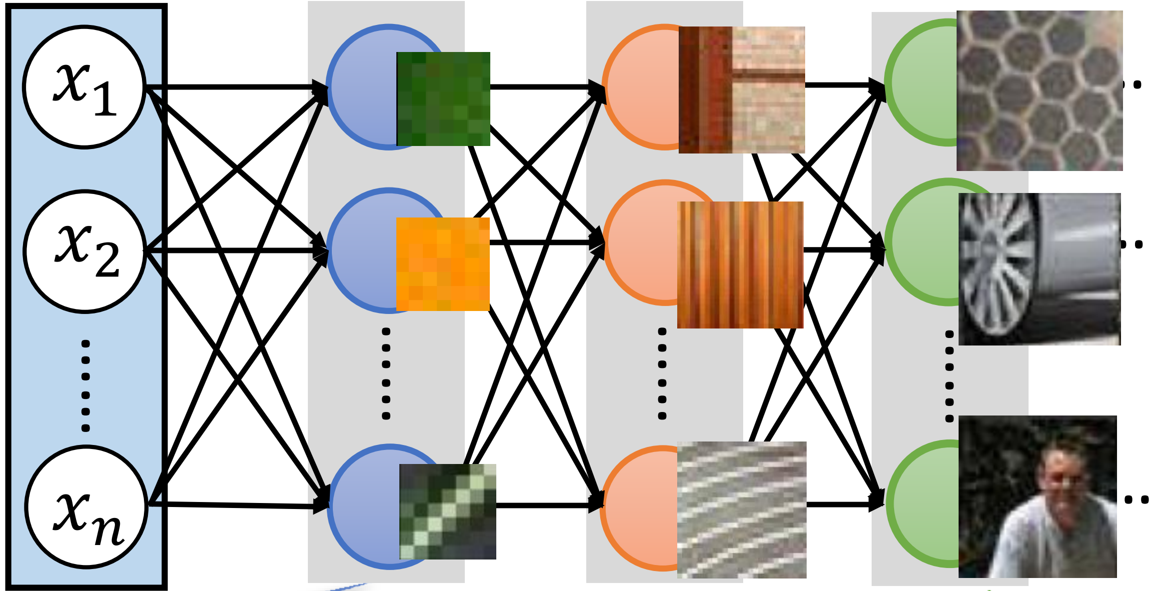
- CNN Basics
- Typical CNN Architectures

Many slides adapted from Hungyi Lee

Why CNN for Image?

100*100*3

Represented as pixels



The most basic classifiers

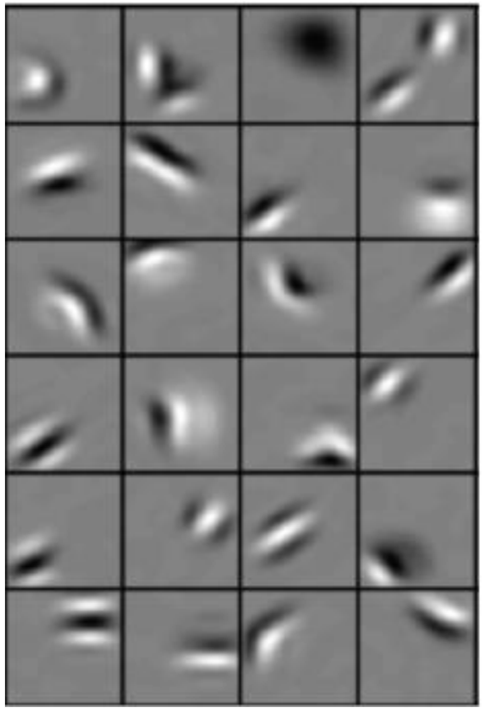
Use 1st layer as module to build classifiers

Use 2nd layer as module .....

Can the network be simplified by considering the properties of images?

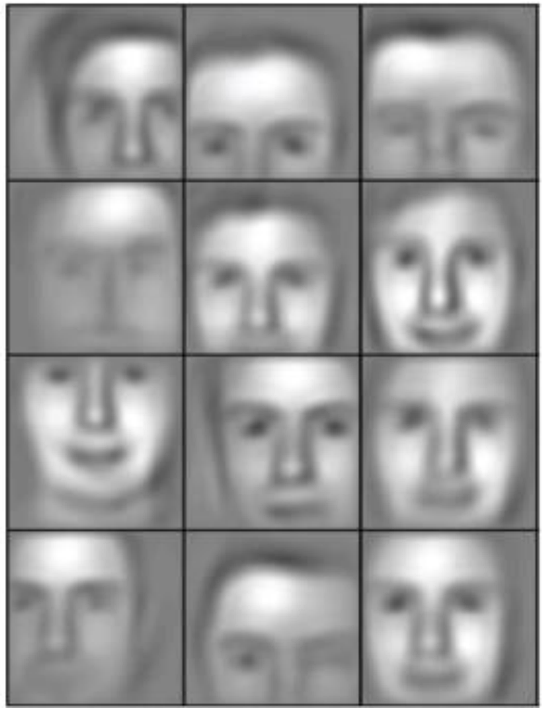
Why CNN for Image

- Hierarchical structure of objects.
    - Objects consist of object parts.
    - Object parts consist of simple, local patterns.



Low level features

Mid level features



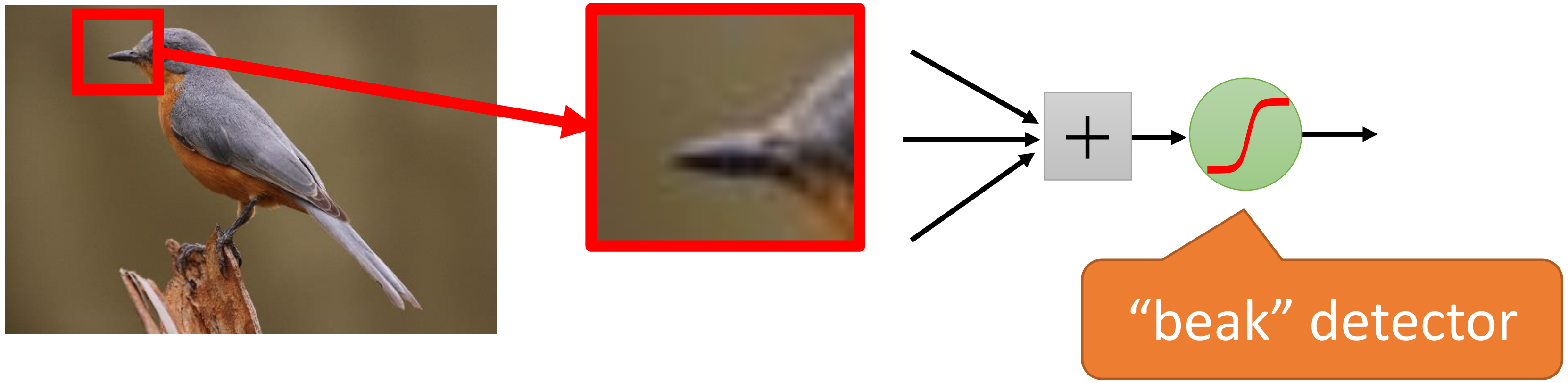
High level features

Why CNN for Image

- Some patterns are much smaller than the whole image

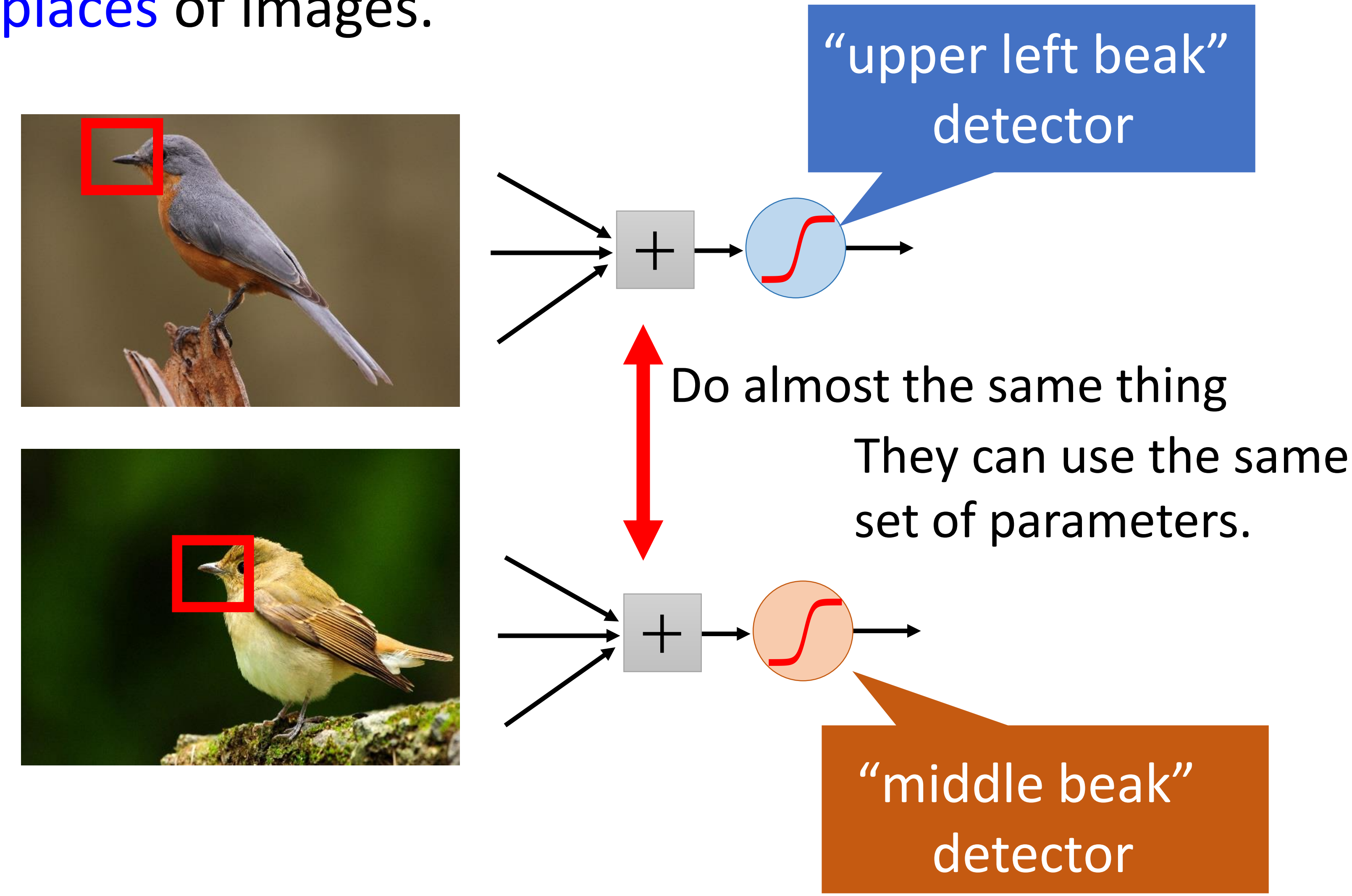
A neuron does not have to see the whole image to discover the pattern.

Connecting to small region with less parameters



Why CNN for Image

- Same objects or same patterns appear at different places of images.



Why CNN for Image

- Subsampling/scaling the pixels will not change the object category

bird

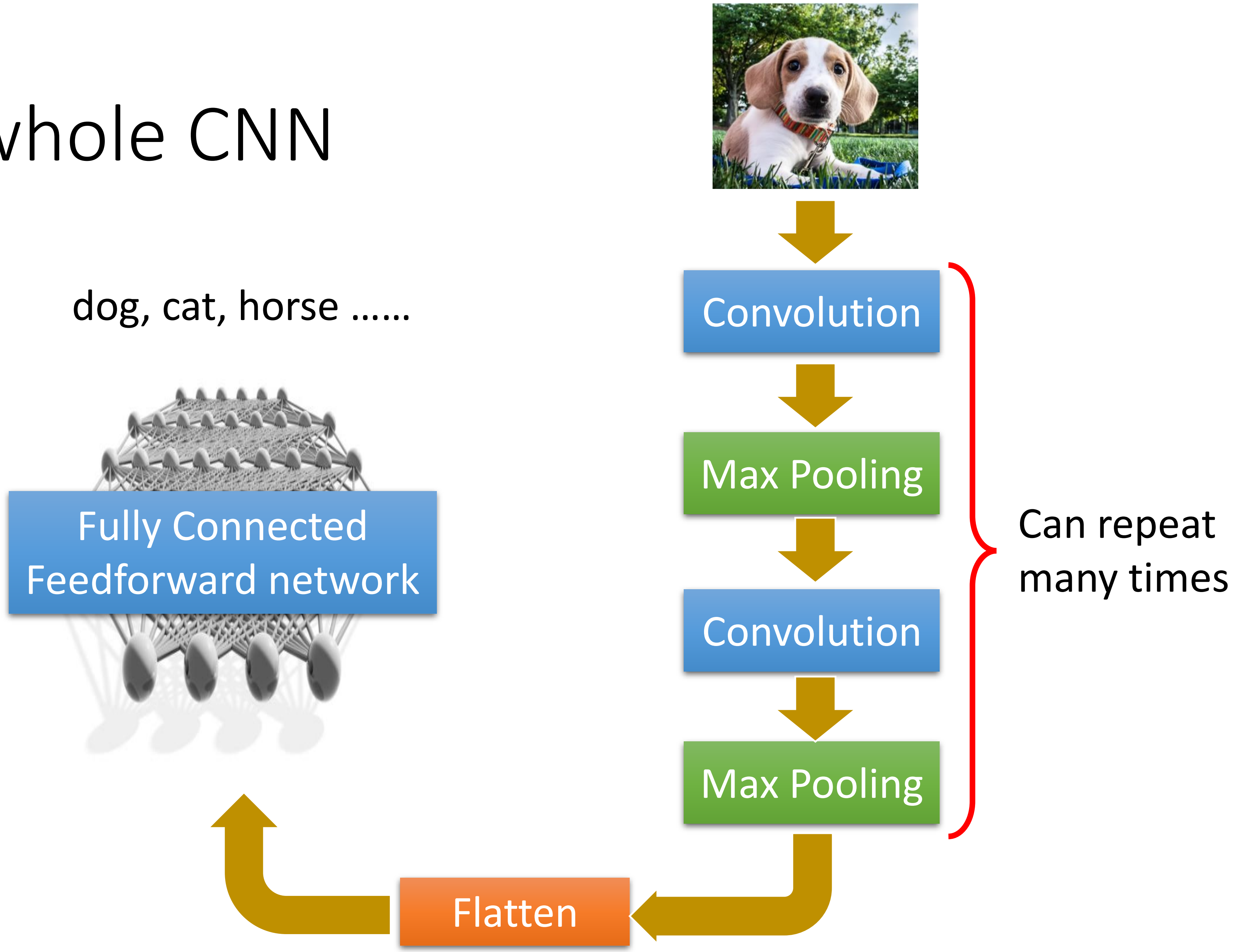
subsampling

bird

We can subsample the pixels to make image smaller

➡ Less parameters for the network to process the image

The whole CNN



The whole CNN

Property 1

- Some patterns are much smaller than the whole image

Property 2

- The same patterns appear in different regions.

Property 3

- Subsampling the pixels will not change the object

Convolution

Max Pooling

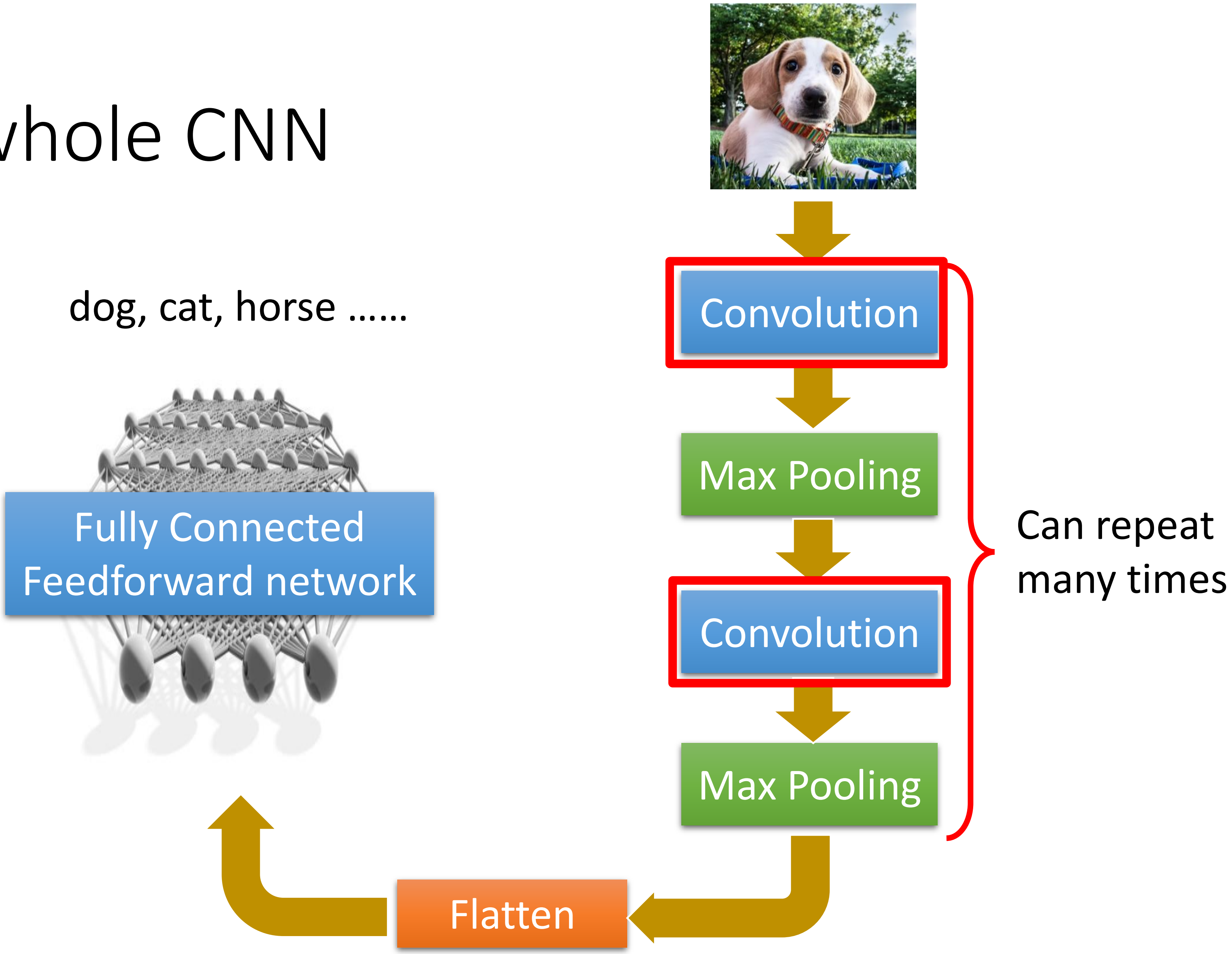
Convolution

Max Pooling

Can repeat many times

Flatten

The whole CNN



CNN – Convolution

Those are the network parameters to be learned.

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 ×6 image

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

Matrix

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

Matrix

⋮

Property 1 Each filter detects a small pattern (3 × 3).

CNN – Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

3 -1

6×6 image

CNN – Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

If stride=2

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6×6 image

3 -3

We set stride=1 below

CNN – Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6×6 image

3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

Property 2

CNN – Convolution

-1	1	-1
-1	1	-1
-1	1	-1

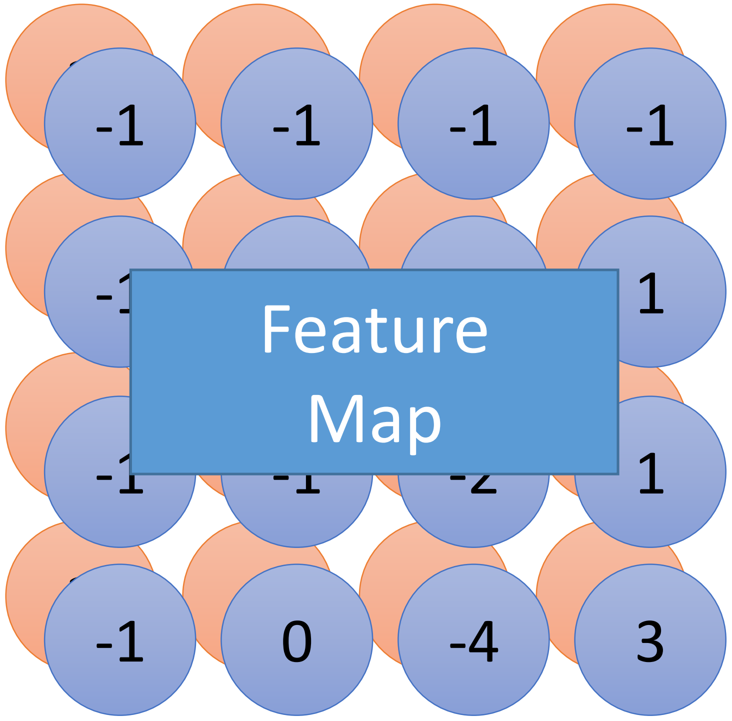
Filter 2

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6×6 image

Do the same process for every filter



4×4 image

CNN – Color Image

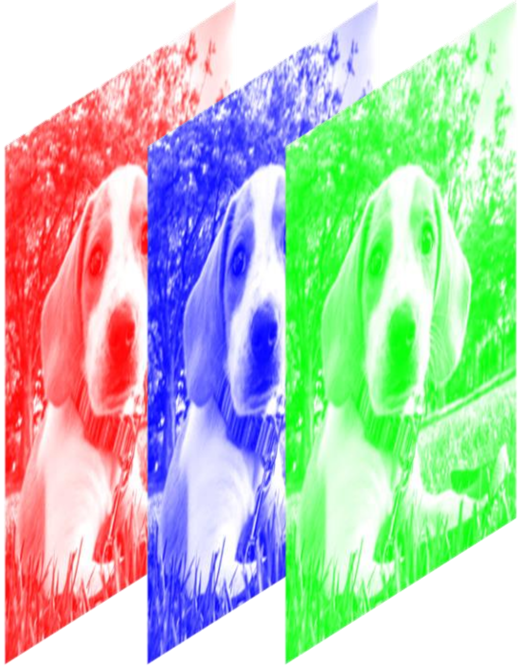
1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

-1	1	-1
-1	1	-1
-1	1	-1

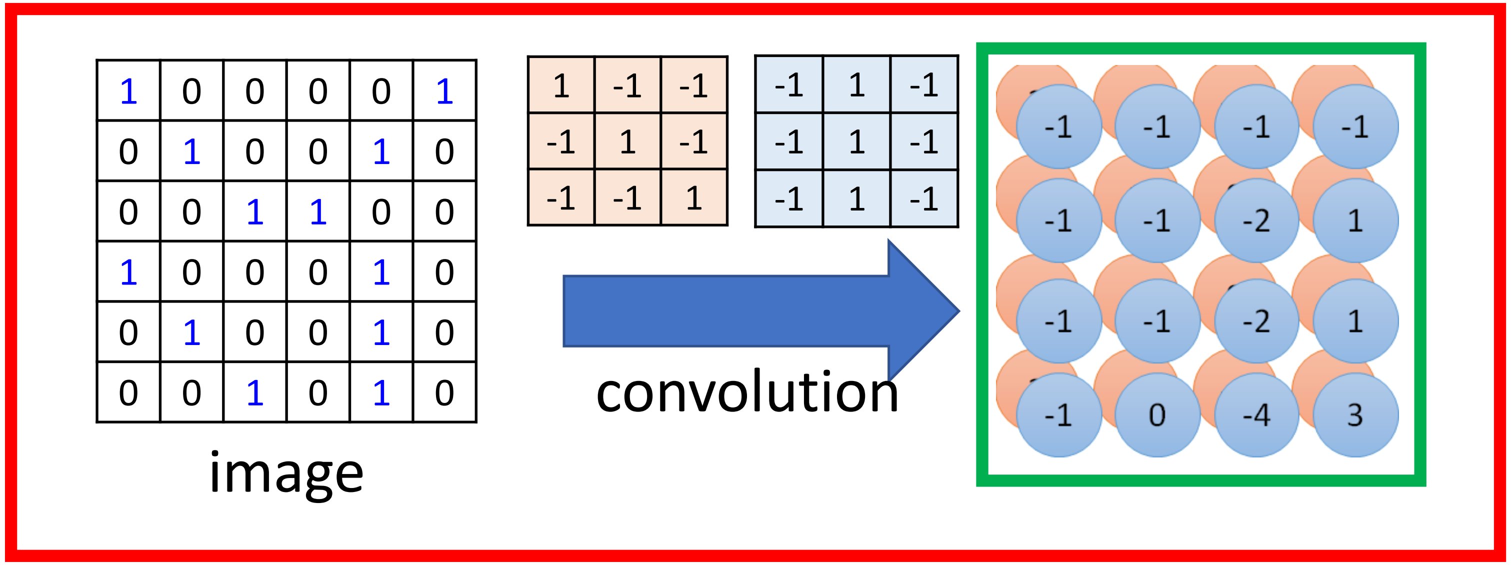
Filter 2

Color image

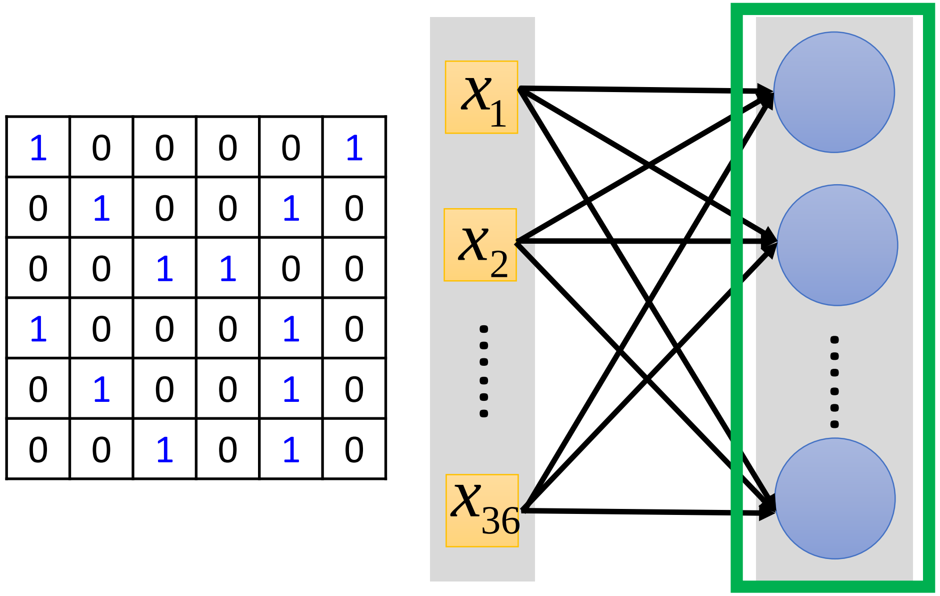


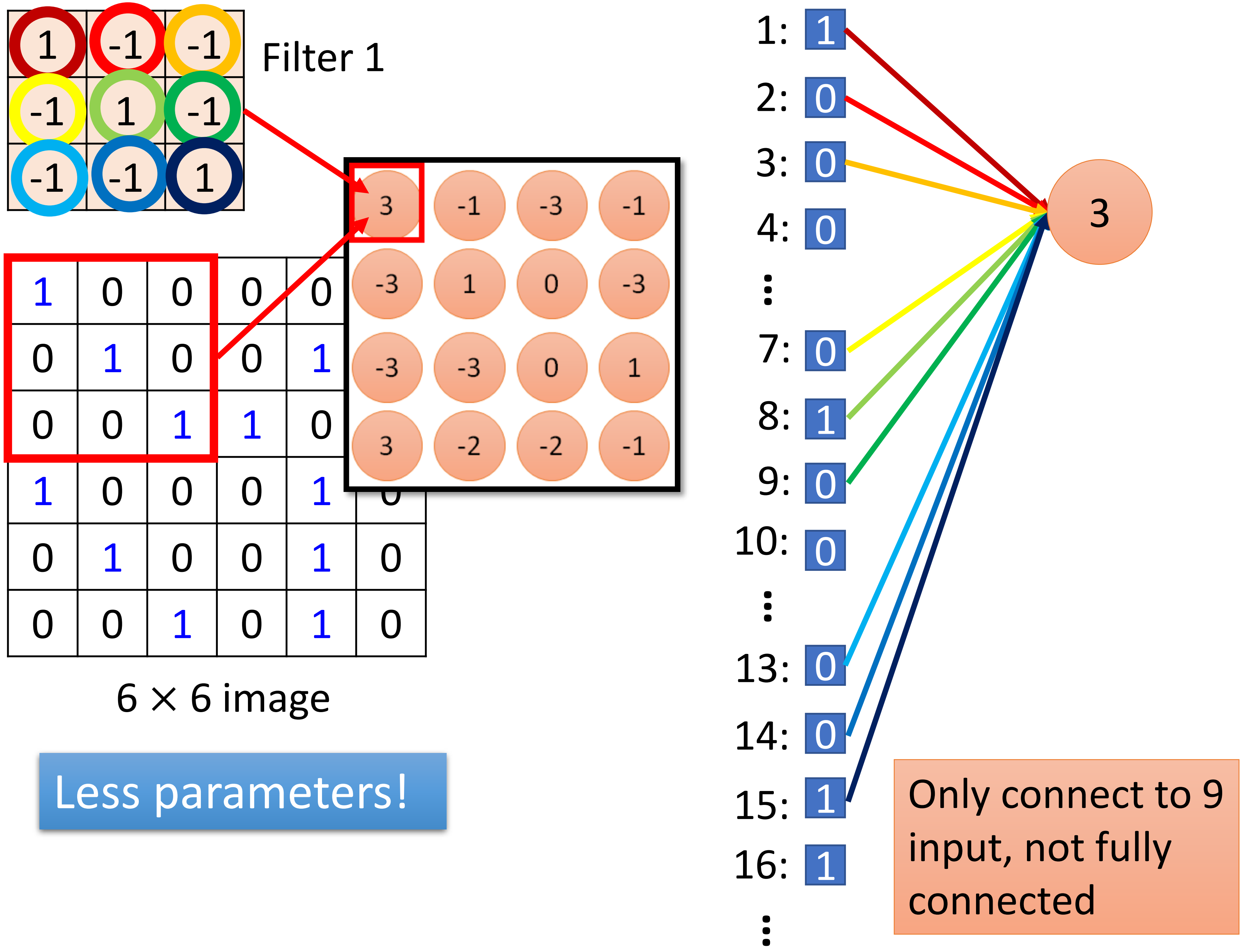
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

Convolution v.s. Fully Connected

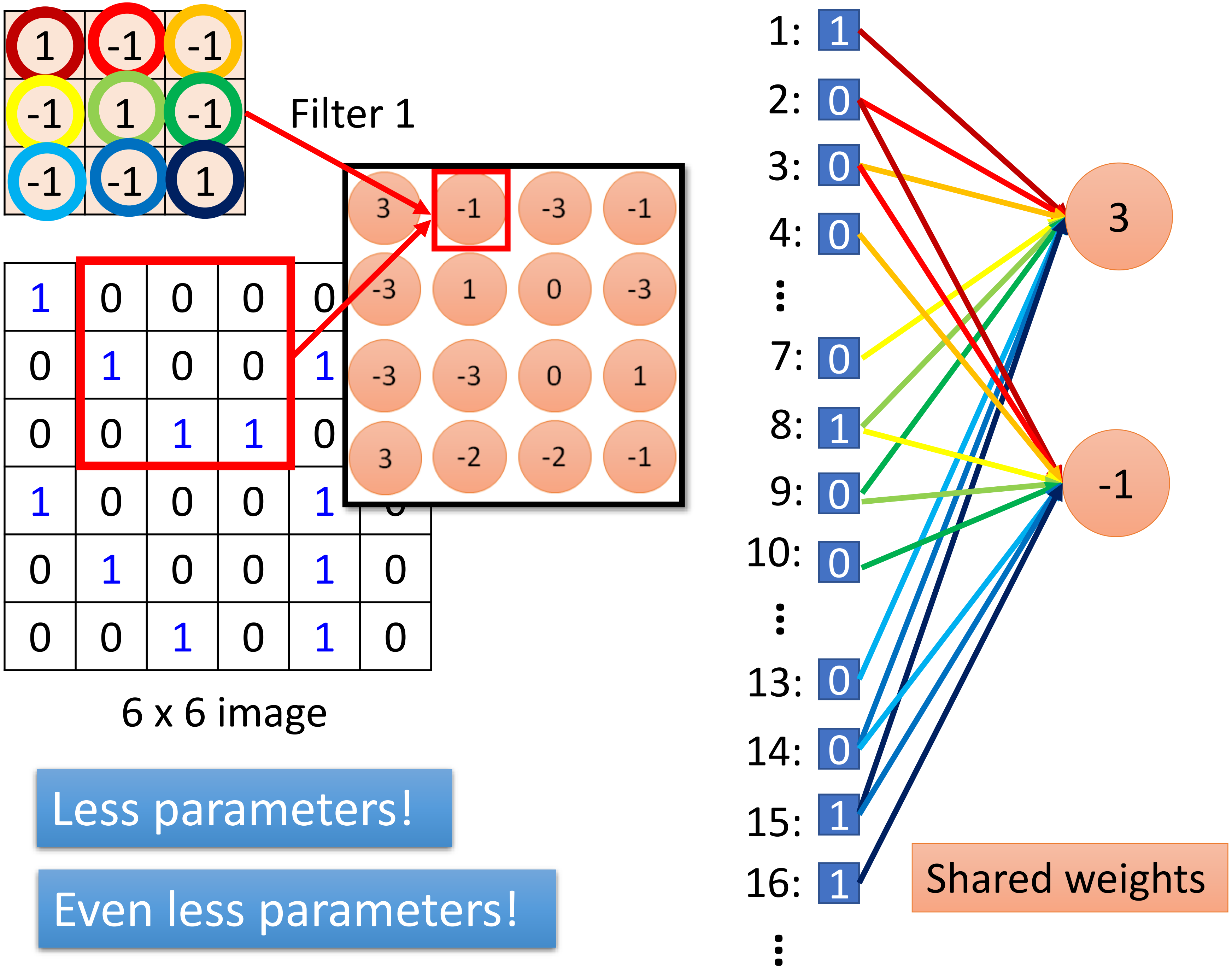


Fully Connected



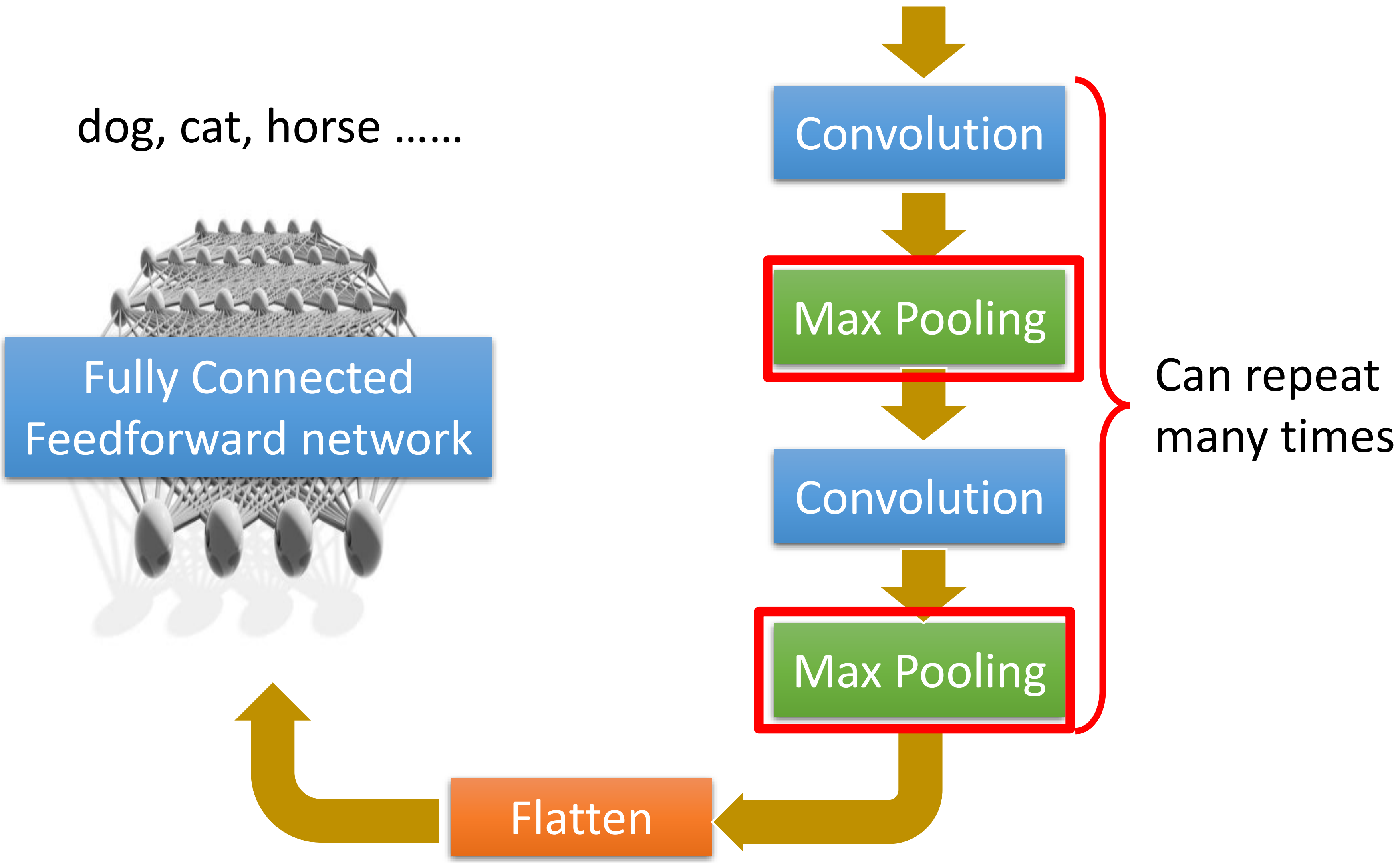


1 -1 -1
-1 1 -1
-1 -1 1
Filter 1
1 0 0 0 0 1
0 1 0 0 1 0
0 0 1 1 0 0
1 0 0 0 1 0
0 1 0 0 1 0
0 0 1 0 1 0
6 × 6 image
3 -1 -3 -1
-3 1 0 -3
-3 -3 0 1
3 -2 -2 -1
1: 1
2: 0
3: 0
4: 0
7: 0
8: 1
9: 0
10: 0
13: 0
14: 0
15: 1
16: 1
3
Less parameters!
Only connect to 9 input, not fully connected



1 -1 -1
-1 1 -1
-1 -1 1
Filter 1
3 -1 -3 -1
-3 1 0 -3
-3 -3 0 1
3 -2 -2 -1
1 0 0 0 0 1
0 1 0 0 1 0
0 0 1 1 0 0
1 0 0 0 1 0
0 1 0 0 1 0
0 0 1 0 1 0
6 x 6 image
Less parameters!
Even less parameters!
1: 1
2: 0
3: 0
4: 0
⋮
7: 0
8: 1
9: 0
10: 0
⋮
13: 0
14: 0
15: 1
16: 1
⋮
3
-1
Shared weights

The whole CNN



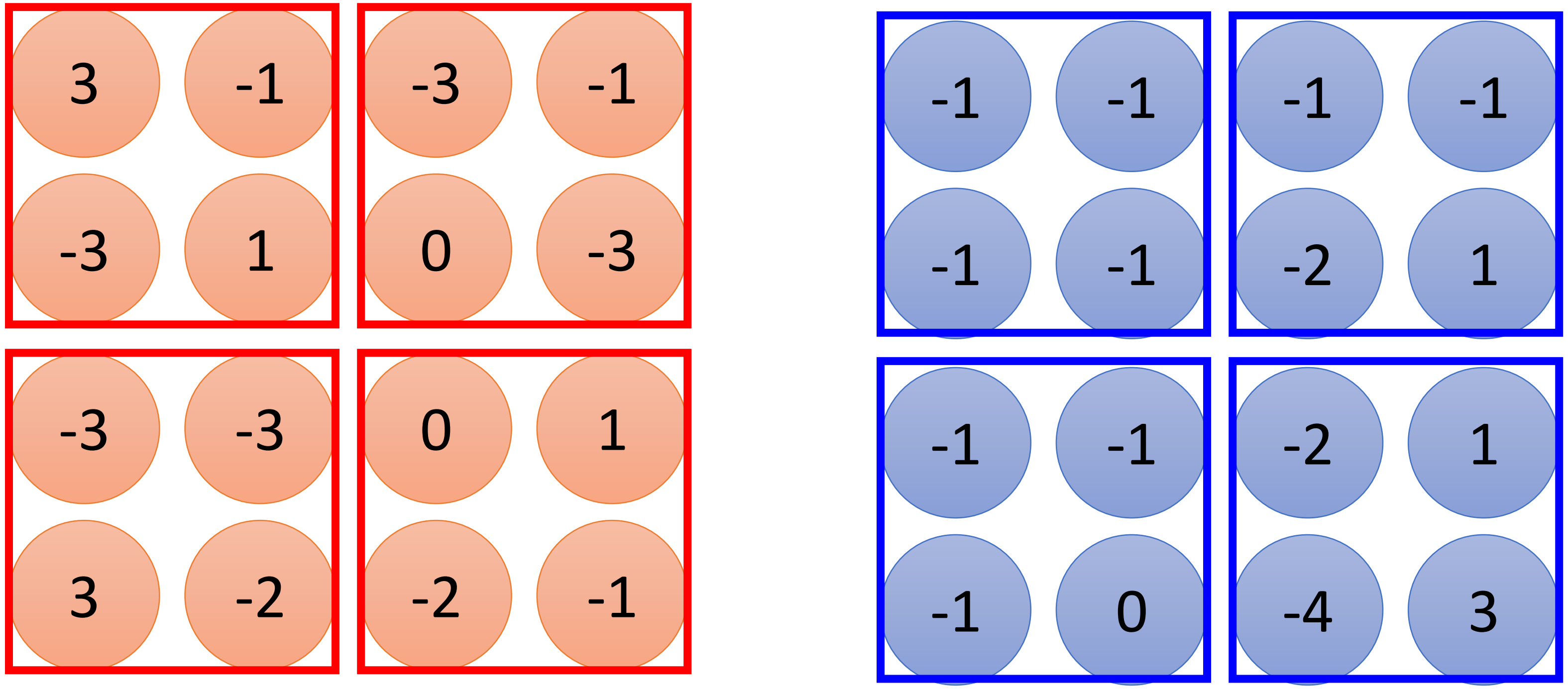
CNN – Max Pooling

Filter 1

1	-1	-1
-1	1	-1
-1	-1	1

Filter 2

-1	1	-1
-1	1	-1
-1	1	-1

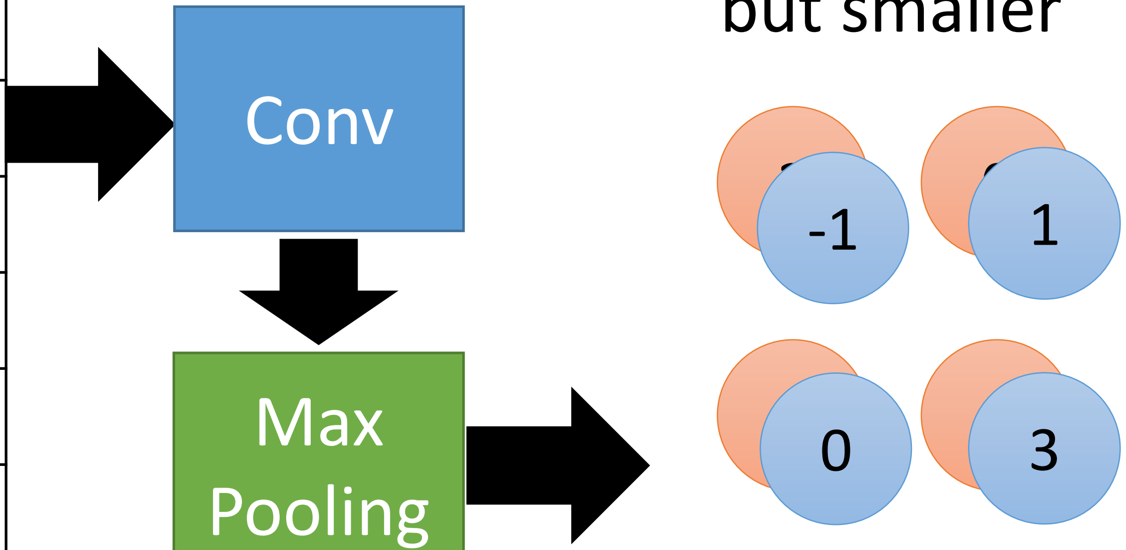


CNN – Max Pooling

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6×6 image

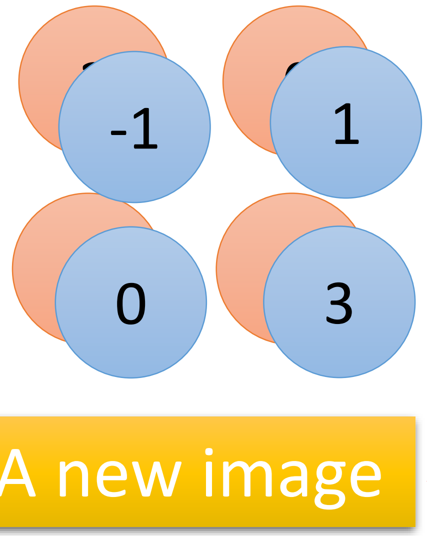
New image but smaller

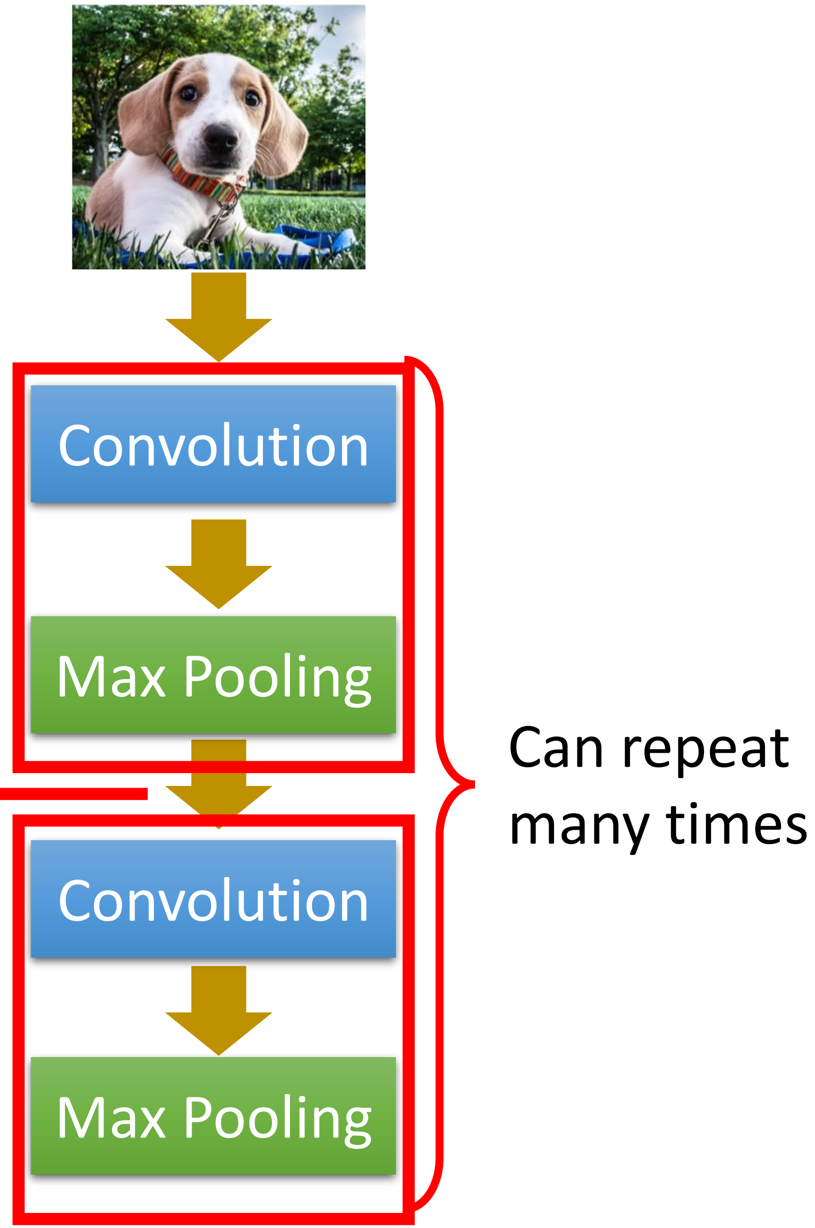


2×2 image

Each filter is a channel

The whole CNN





Smaller than the original image

The number of the channel is the number of filters

The whole CNN

Convolution

Max Pooling

A new image

Convolution

Max Pooling

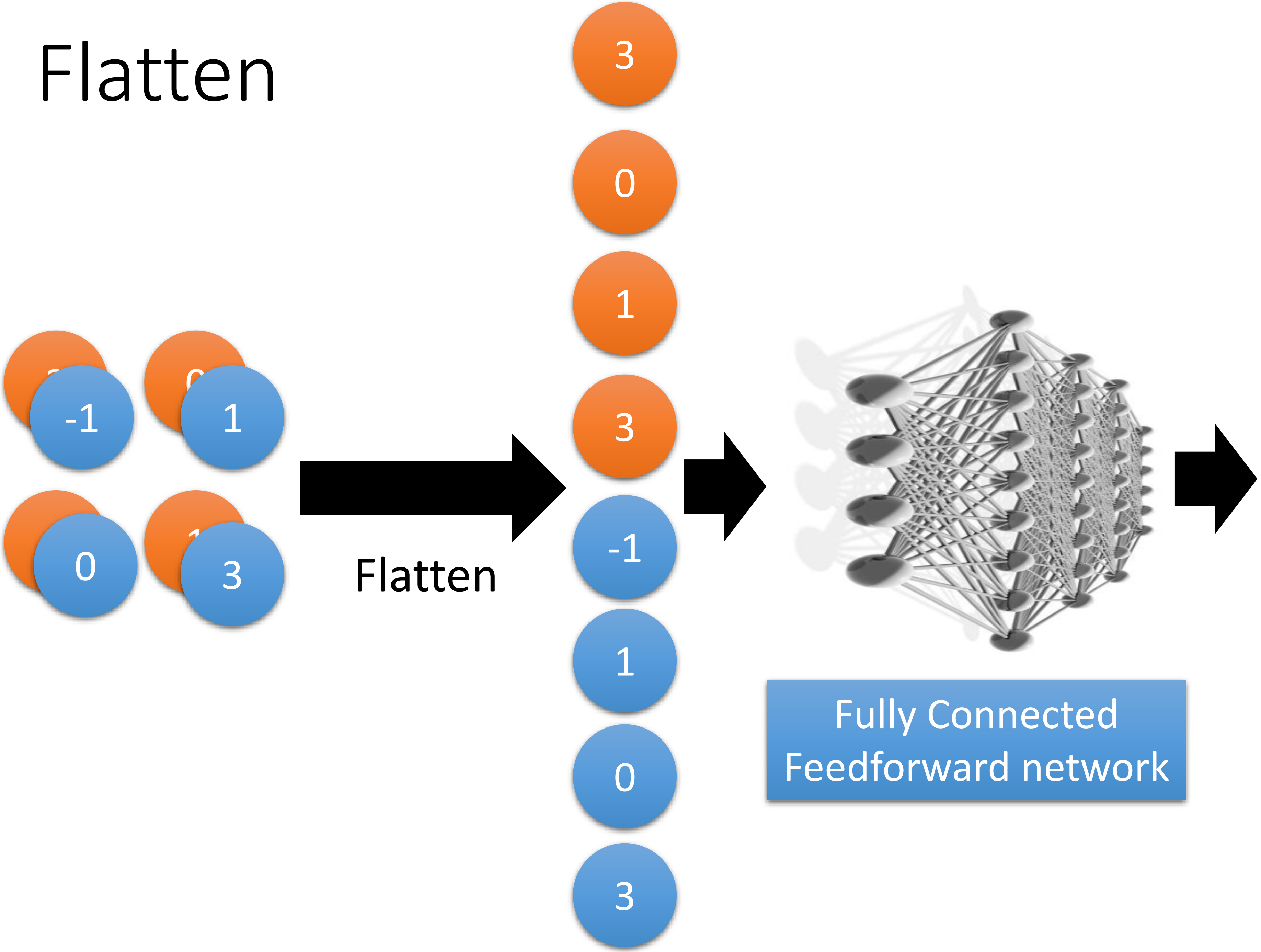
A new image

Flatten

Fully Connected Feedforward network

dog, cat, horse ……

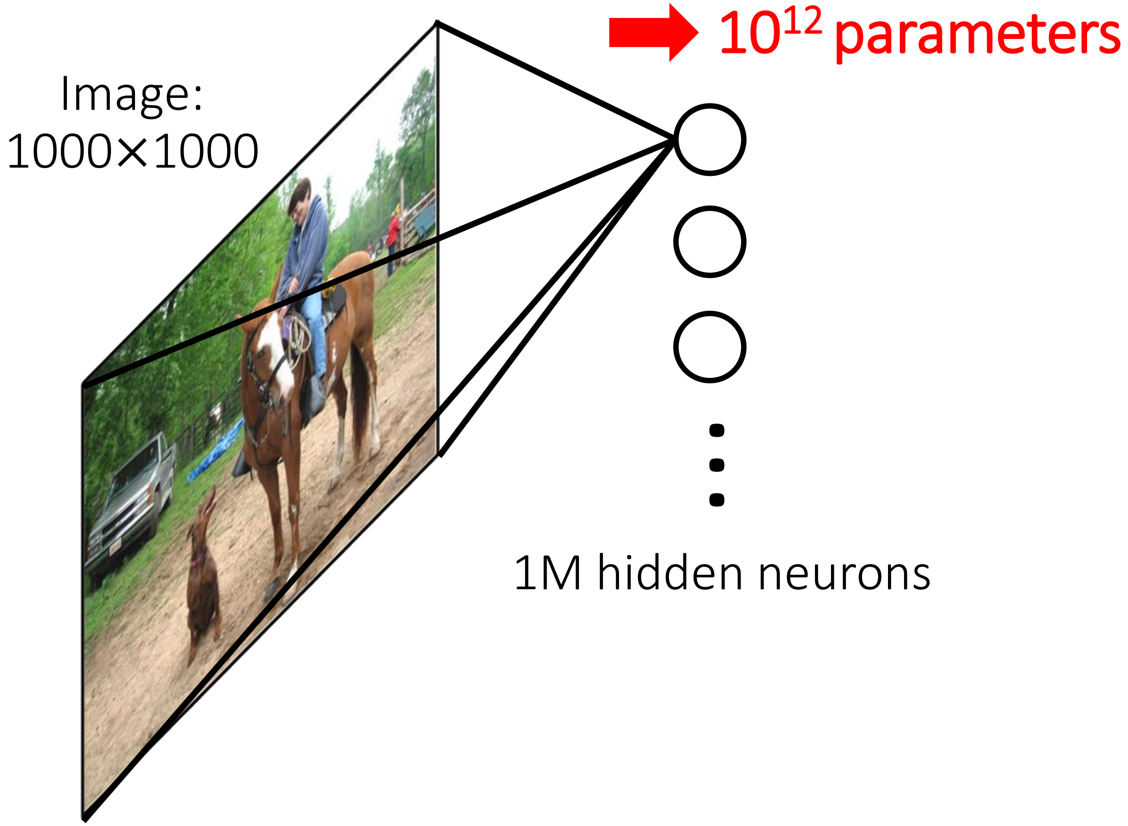
Flatten



-1
1
0
3
Flatten
3
0
1
3
-1
1
0
3
Fully Connected
Feedforward network

CNN-Summary

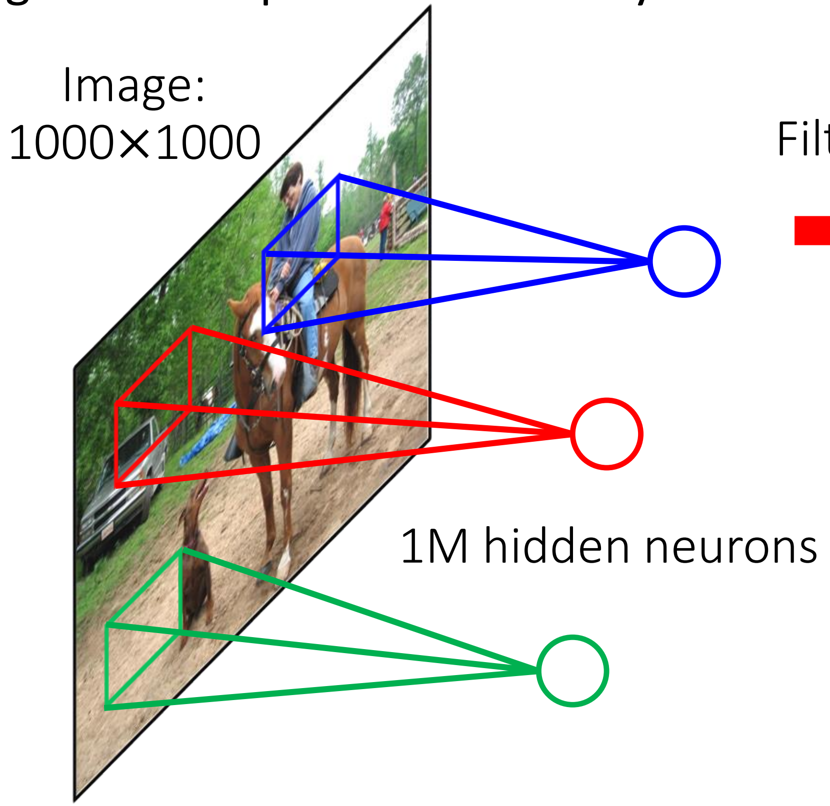
- Problems of fully connected neural networks
  - Every output unit interacts with every input unit (pixel)
  - The number of weights grows largely with the size of the input image
  - Pixels in distance are less correlated



CNN-Summary

◻ Locally connected neural networks

- Sparse connectivity: a hidden unit is only connected to a local patch (weights connected to the patch are called filter or kernel)
- Inspired by biological systems, where a cell is sensitive to a small subregion of the input space, called a receptive field. Many cells are tiled to cover the entire visual field.
- The design of such sparse connectivity is based on domain knowledge.



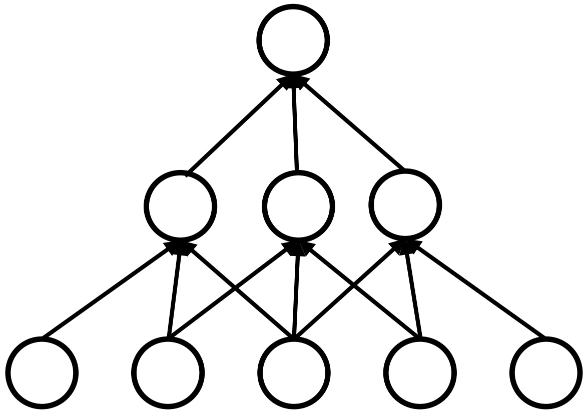
➡ 10^8 parameters

Significantly less than 10^{12} parameters

CNN-Summary

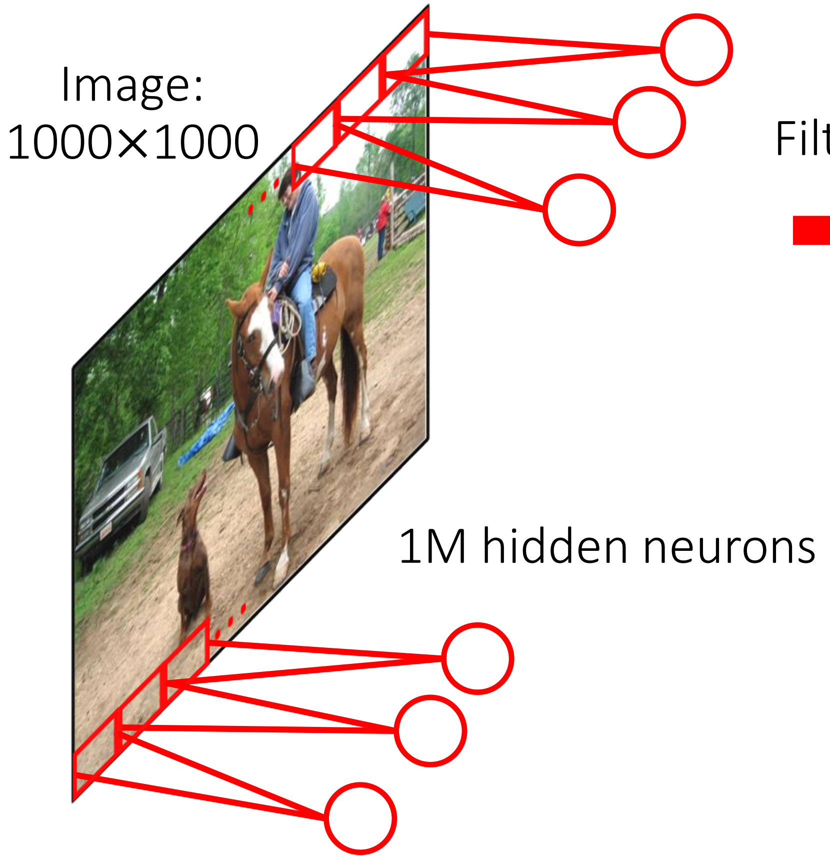
▫ Locally connected neural networks

- The learned filter is a spatially local pattern
- A hidden node at a higher layer has a larger receptive field in the input
- Stacking many such layers leads to "filters" which become increasingly "global"



CNN-Summary ◻ Share weights

- Translation invariance: capture statistics in local patches and they are independent of locations. (Similar edges appear at different locations)
- Hidden nodes at different locations share the same weights. It greatly reduces the number of parameters to learn.
- We may only locally share weights or not share weights at top layers.



Filter Size: 10x10

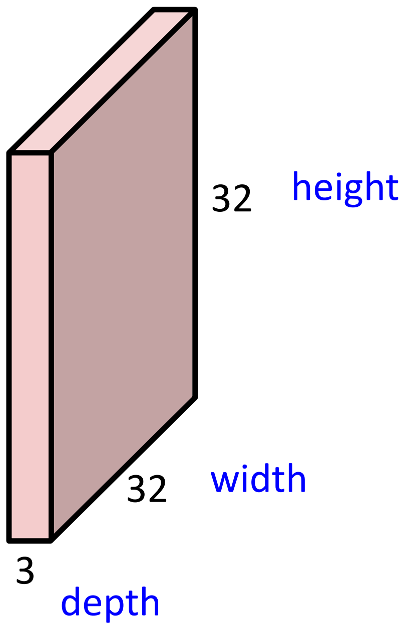
➡ 10^2 parameters

Significantly less than 10^{12} parameters

CNN: Practices

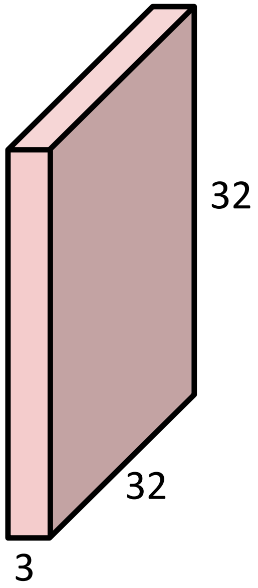
Convolution Layer

$32 \times 32 \times 3$ image

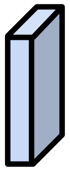


Convolution Layer

$32 \times 32 \times 3$ image



5x5x3 filter

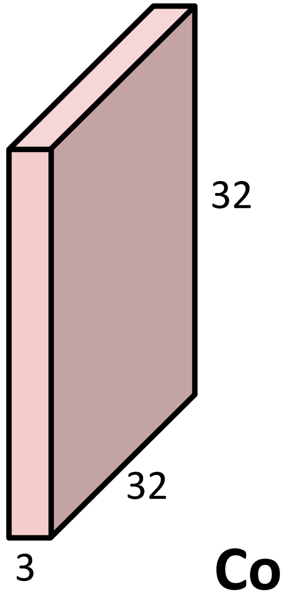


Convolve the filter with the image
i.e. “slide over the image spatially, computing dot products”

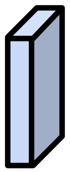
Convolution Layer

Filters always extend the full depth of the input volume

- 32x32x3 image



5x5x3 filter

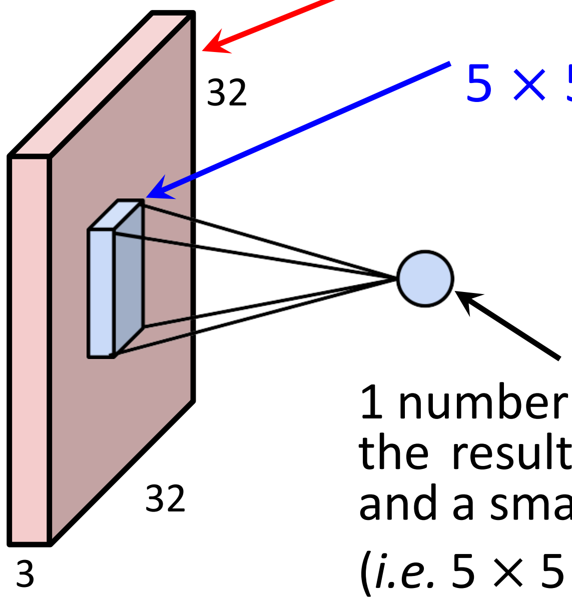


Convolve the filter with the image *i.e.* “slide over the image spatially, computing dot products”

Convolution Layer

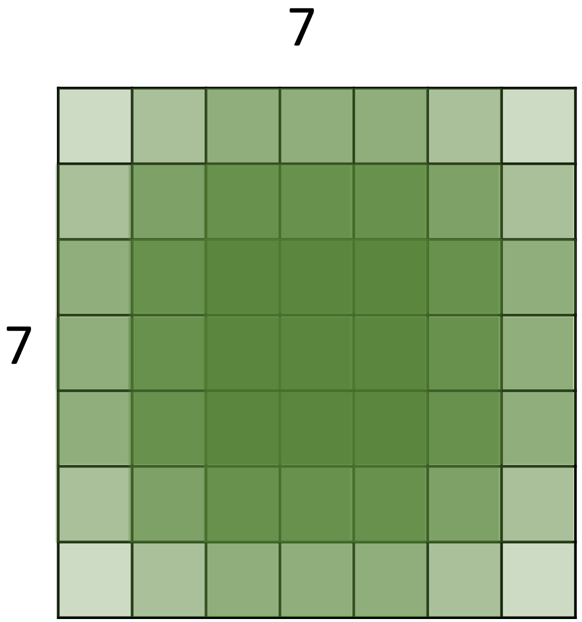
$32 \times 32 \times 3$ image

$5 \times 5 \times 3$ filter $\boldsymbol{w}$



1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(*i.e.* $5 \times 5 \times 3 = 75$-dimensional dot product + bias)

$$\boldsymbol{w}^T \boldsymbol{x} + b$$



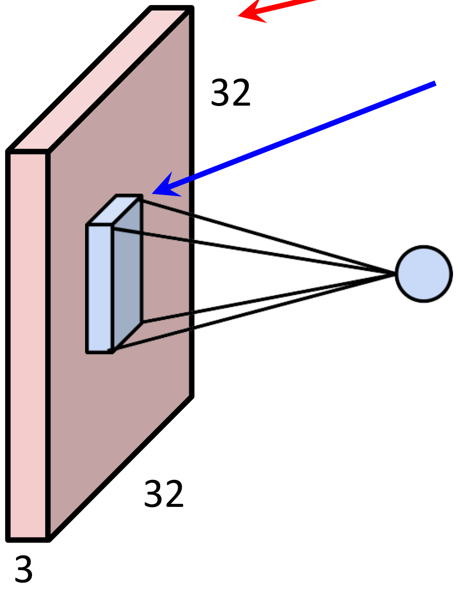
7x7 input (spatially)
assume 3x3 filter

⇒ **5x5 output**

Convolution Layer

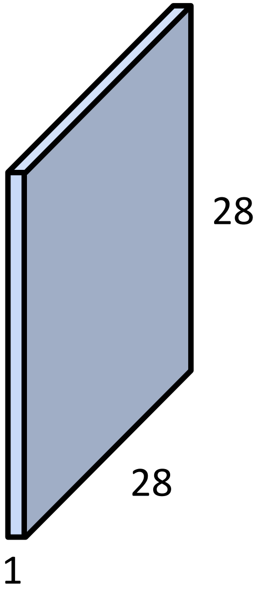
32x32x3 image

5x5x3 filter



convolve (slide) over all spatial locations

activation map

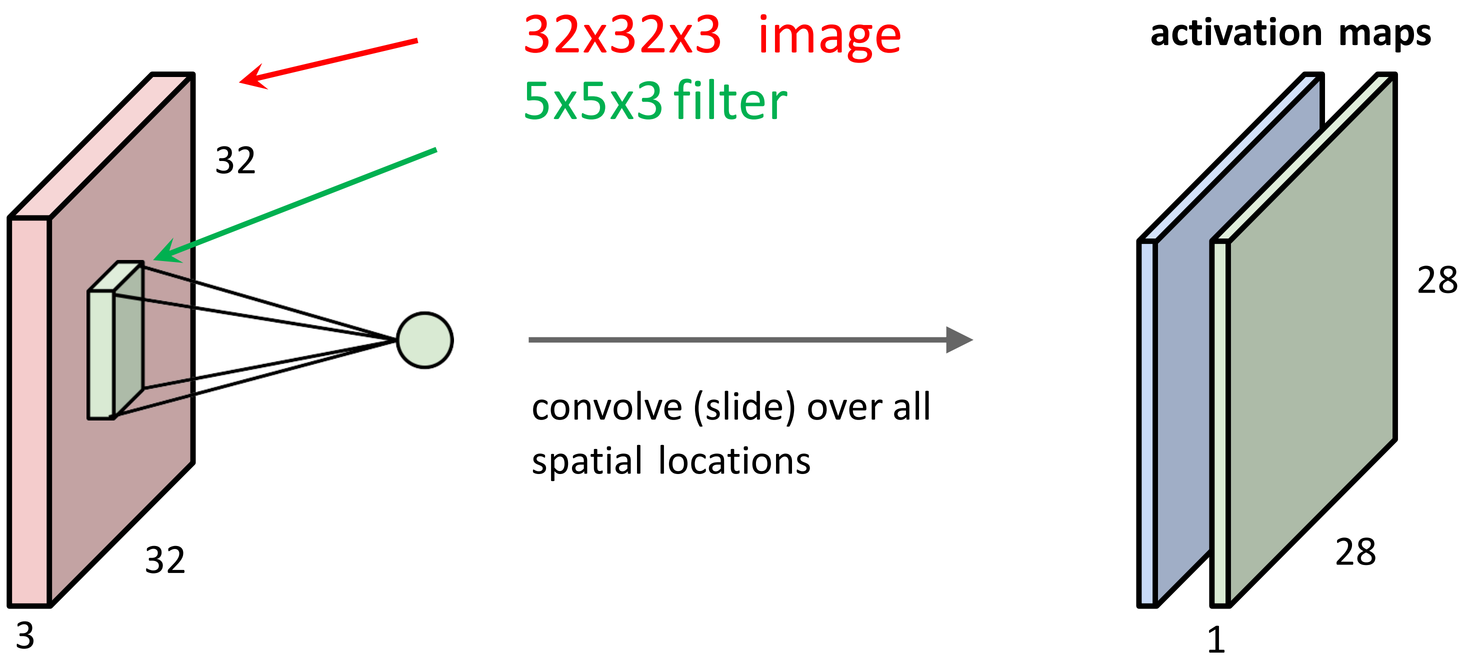


$$\frac{\text{ImageSize} - \text{FilterSize}}{\text{Stride}} + 1$$

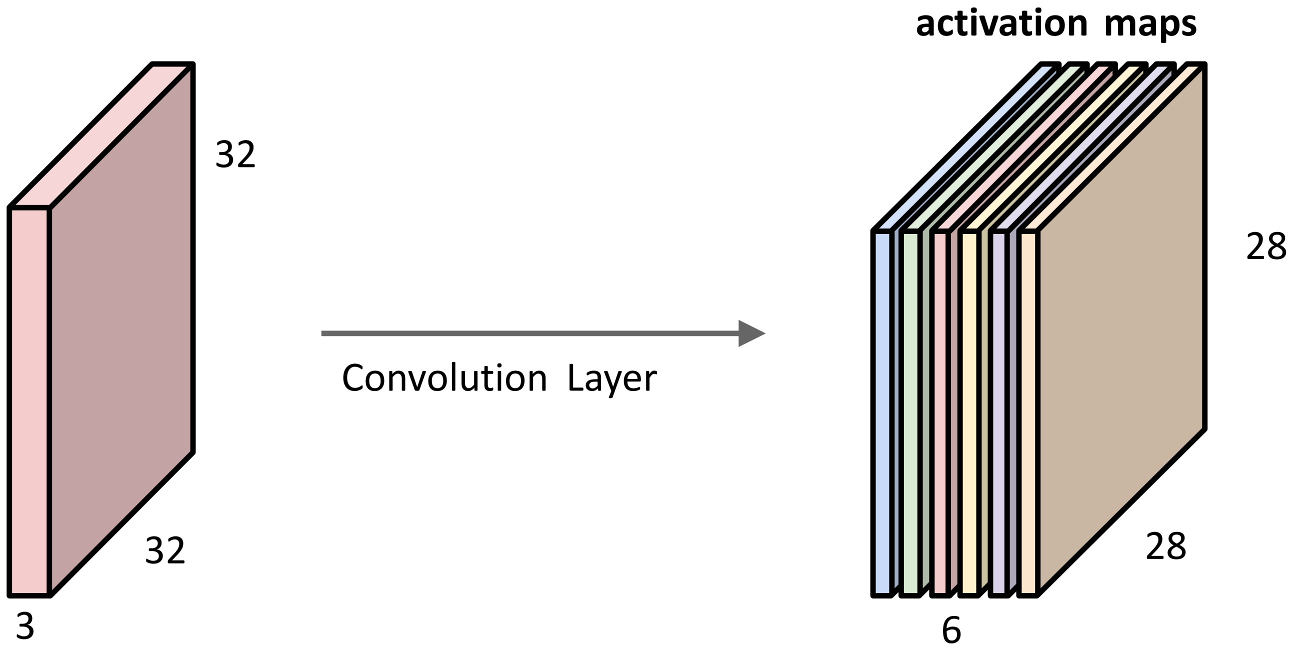
$$\frac{32 - 5}{1} + 1 = 28$$

Convolution Layer

consider a second, green filter

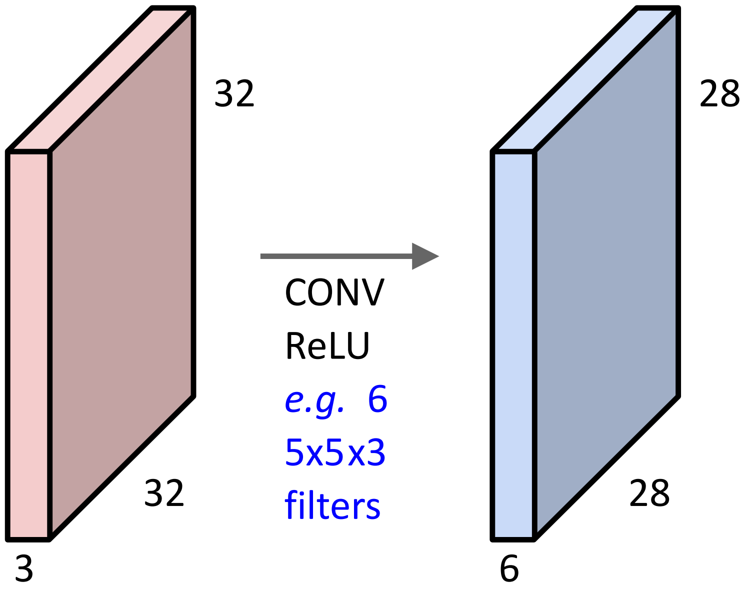


For example, if we had 6 filters of size 5x5x3 , we'll get 6 separate activation maps:

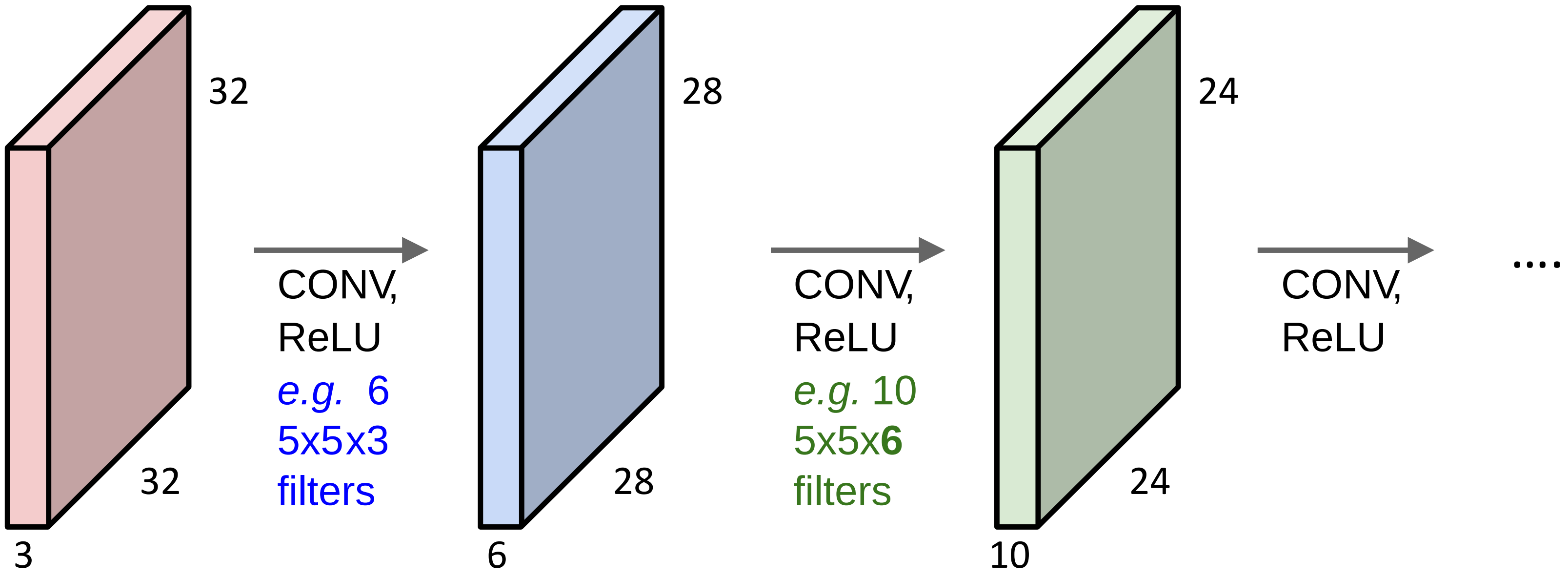


We stack these up to get a "new image" of size 28x28x6!

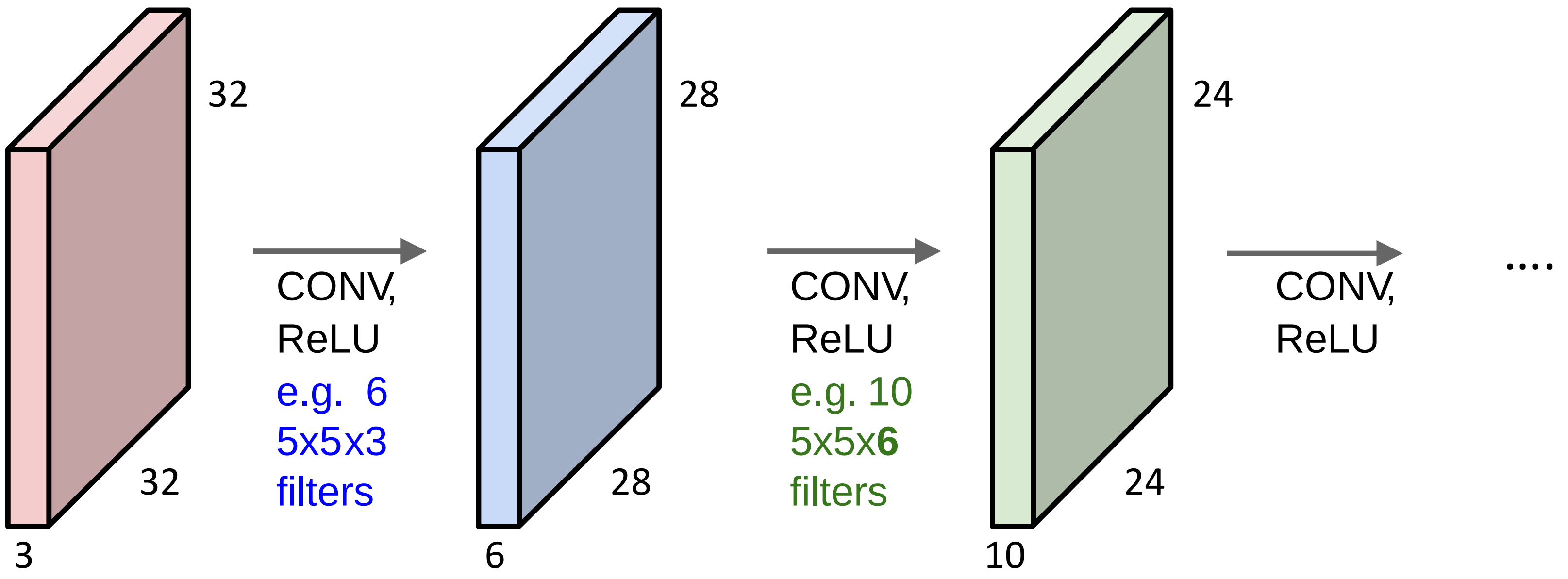
Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions.



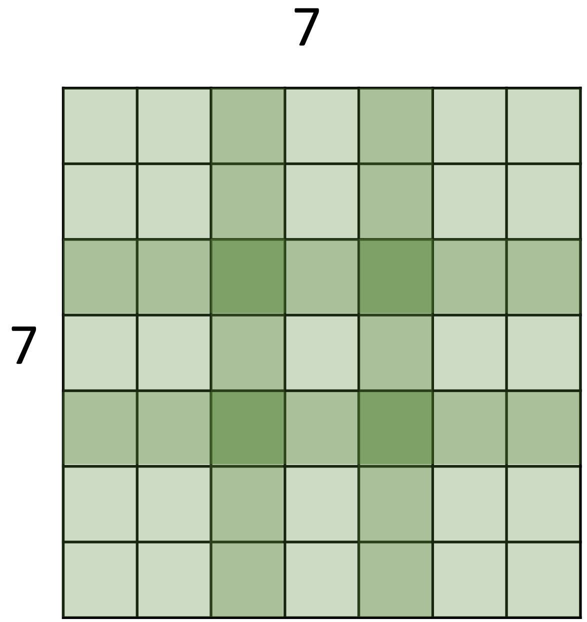
Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions.



32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially! Shrinking too fast is not good, doesn't work well.



A closer look at spatial dimensions:



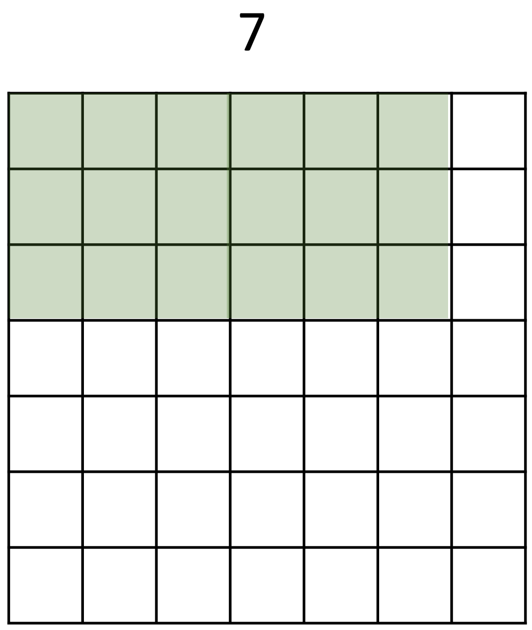
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

$\Rightarrow$ 3×3 output

$$\frac{\text{ImageSize} - \text{FilterSize}}{\text{Stride}} + 1$$

$$\frac{7-3}{2} + 1 = 3$$

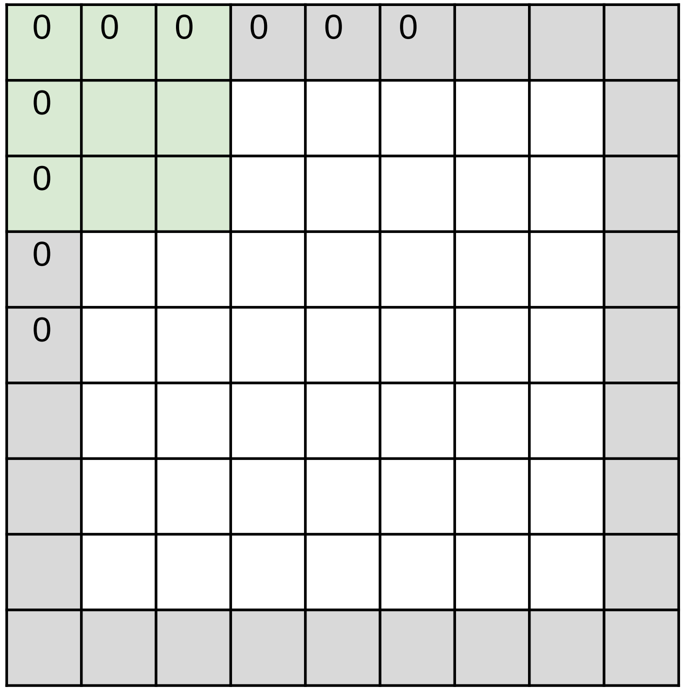
A closer look at spatial dimensions:



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

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

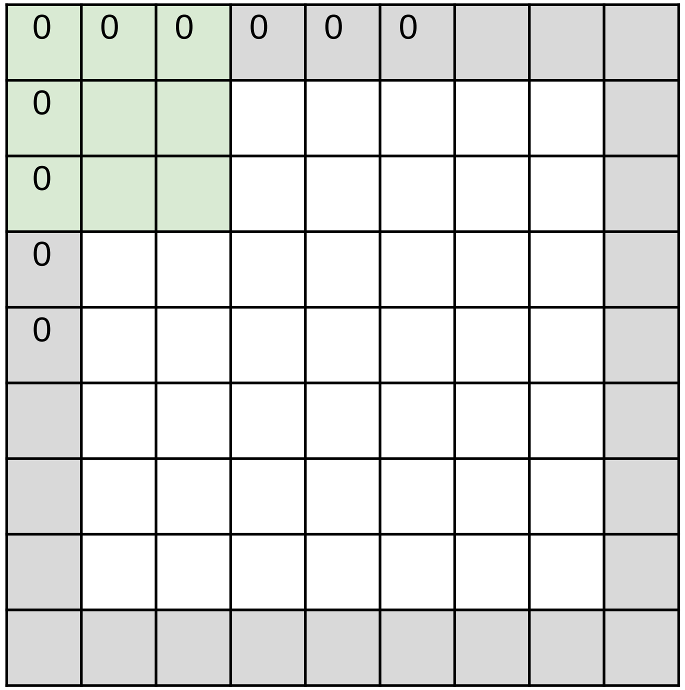
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border $\Rightarrow$ what is the output?

7x7 output!

In practice: Common to zero pad the border

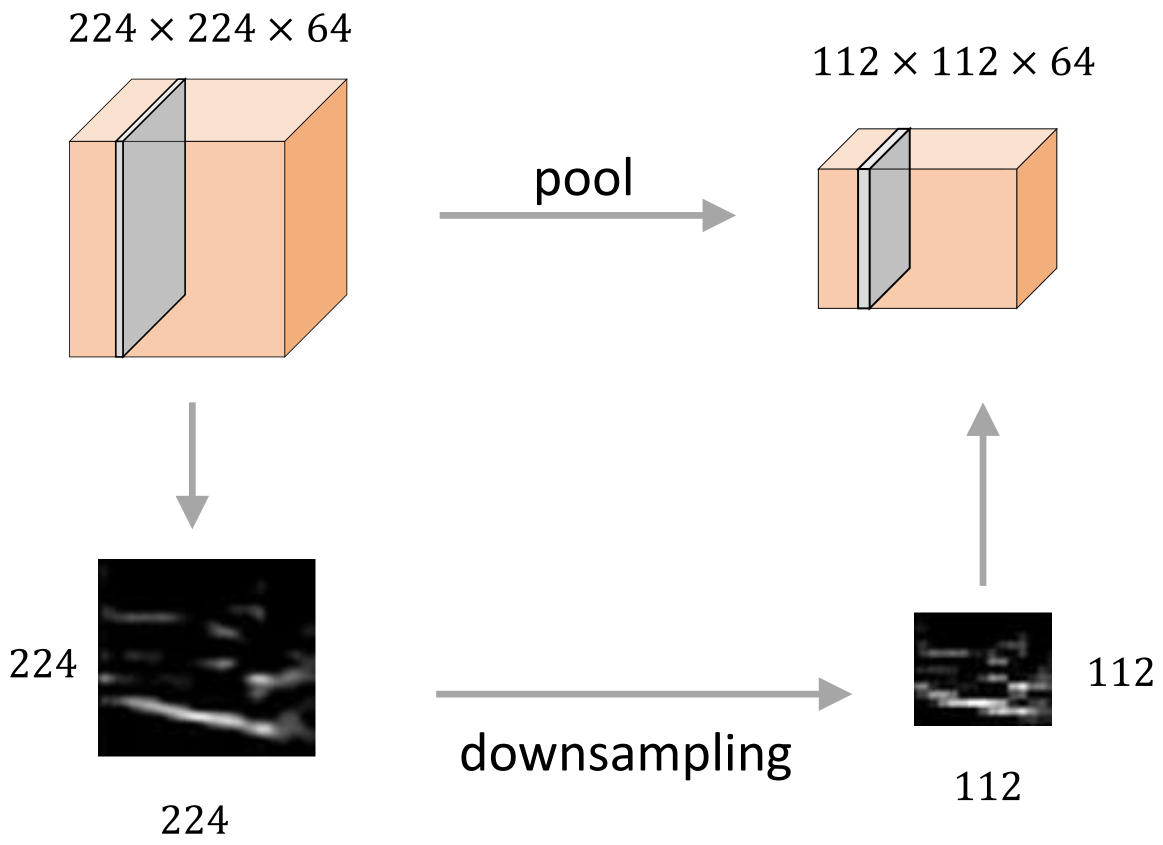


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border ⇒ what is the output?

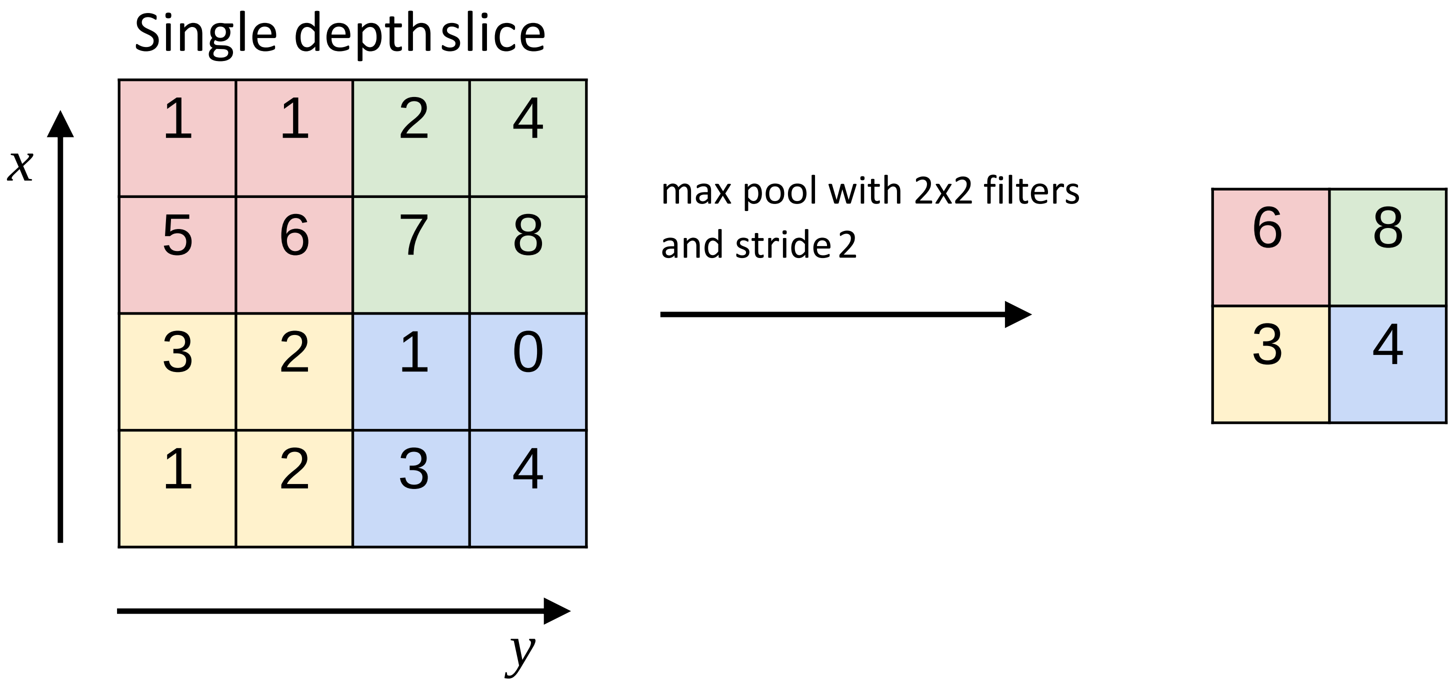
7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently



MAX Pooling


Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

Lecture 4

Convolutional Neural Networks

- CNN Basics
- Typical CNN Architectures

Typical CNN Architectures

Case Studies

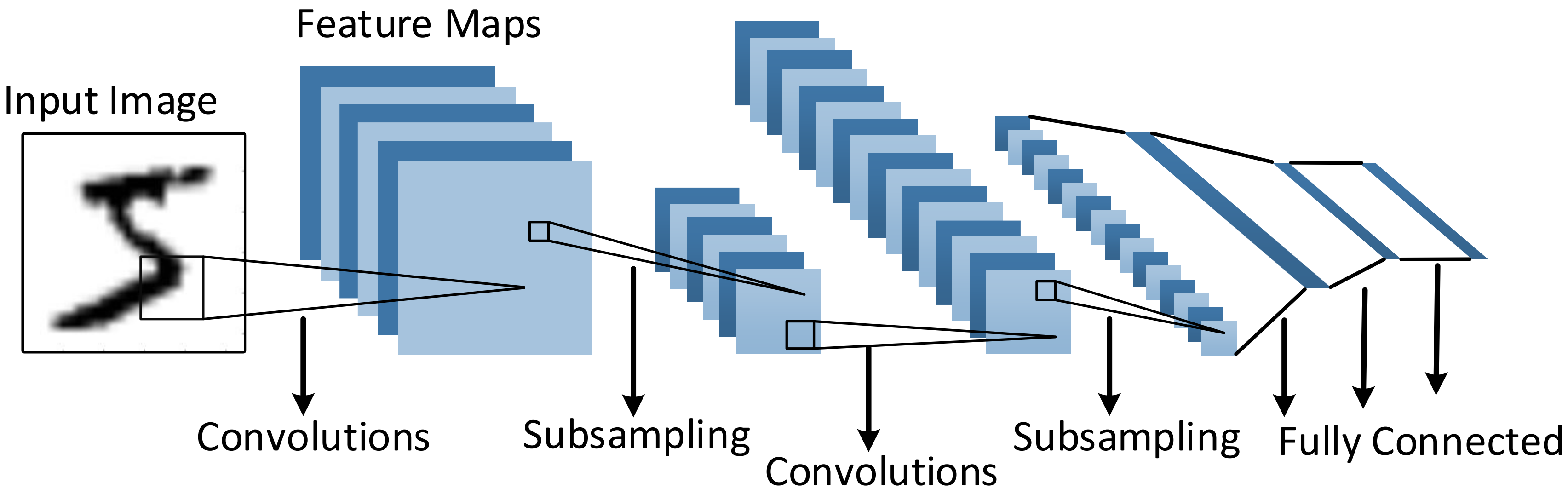
- AlexNet
- VGG
- GoogLeNet
- ResNet

Briefly talk about …

- NIN (Network in Network)
- Wide ResNet
- ResNeXT
- Stochastic Depth
- DenseNet
- SENet
- FractalNet
- SqueezeNet

Review: LeNet

[LeCun *et al.* 1998]



Conv filters were 5×5, applied at stride 1

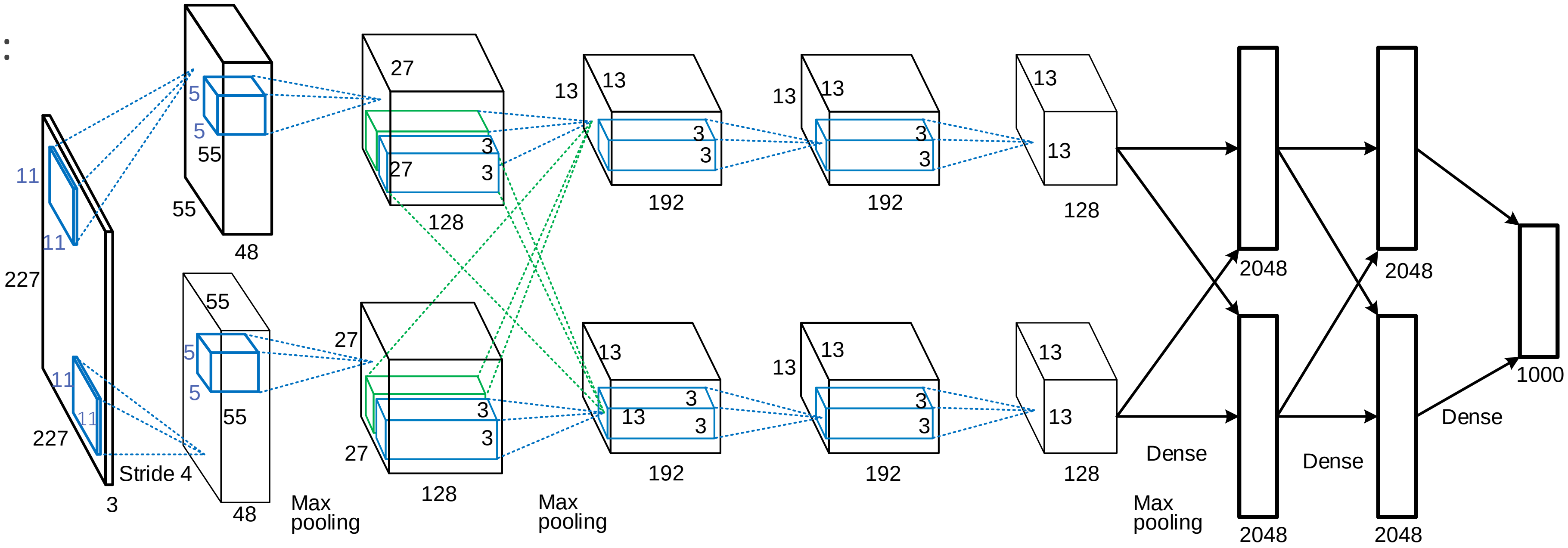
Subsampling (Pooling) layers were 2 × 2 applied at stride 2

i.e. architecture is [CONV→POOL → CONV → POOL → FC → FC]

Case Study: AlexNet

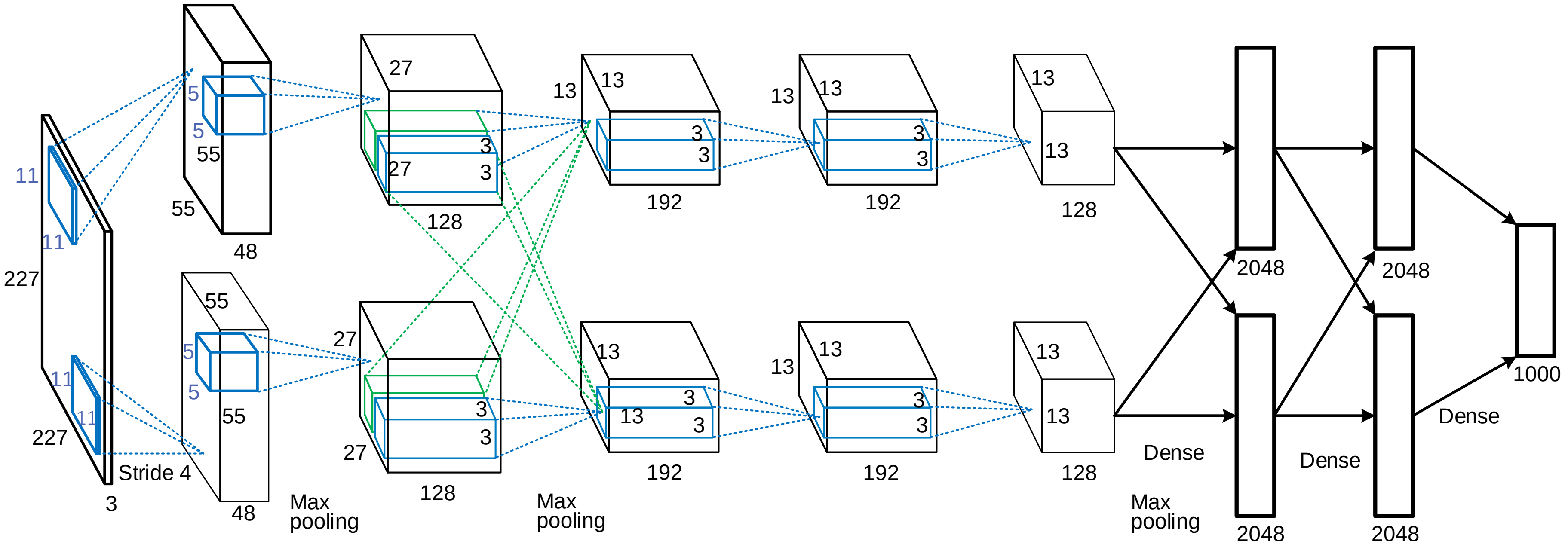
[Krizhevsky *et al.* 2012]

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: 227×227 × 3 images

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

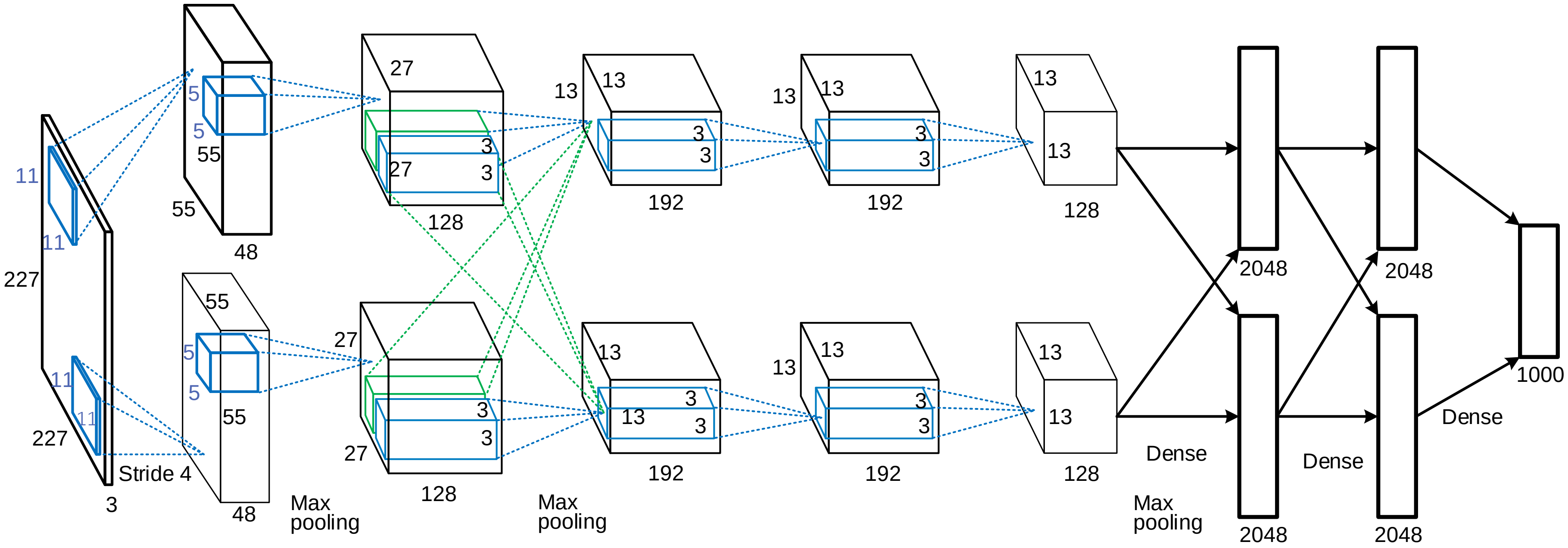
Q: what is the output volume size?

$$\frac{\text{ImageSize} - \text{FilterSize}}{\text{Stride}} + 1$$

(227-11)/4+1 = 55

Case Study: AlexNet

[Krizhevsky *et al.* 2012]



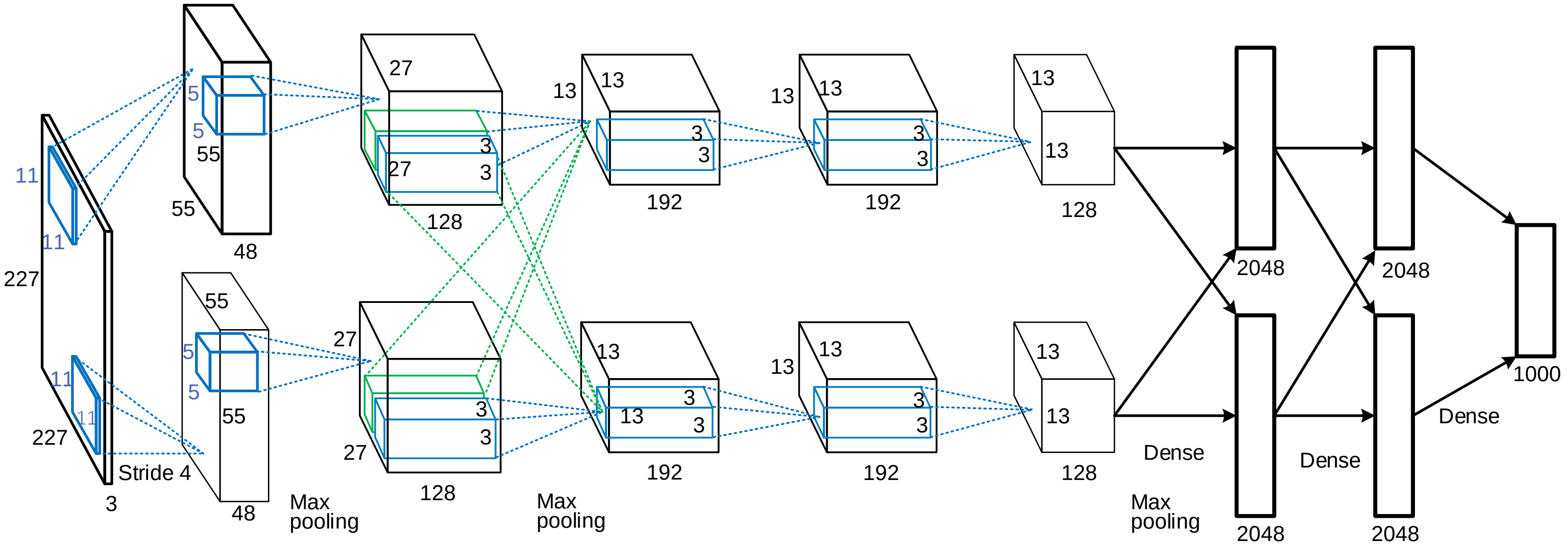
Input: 227×227×3 images

First layer (CONV1): 96 11×11 filters applied at stride 4
Output volume **[55×55×96]**

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

Case Study: AlexNet

[Krizhevsky *et al.* 2012]

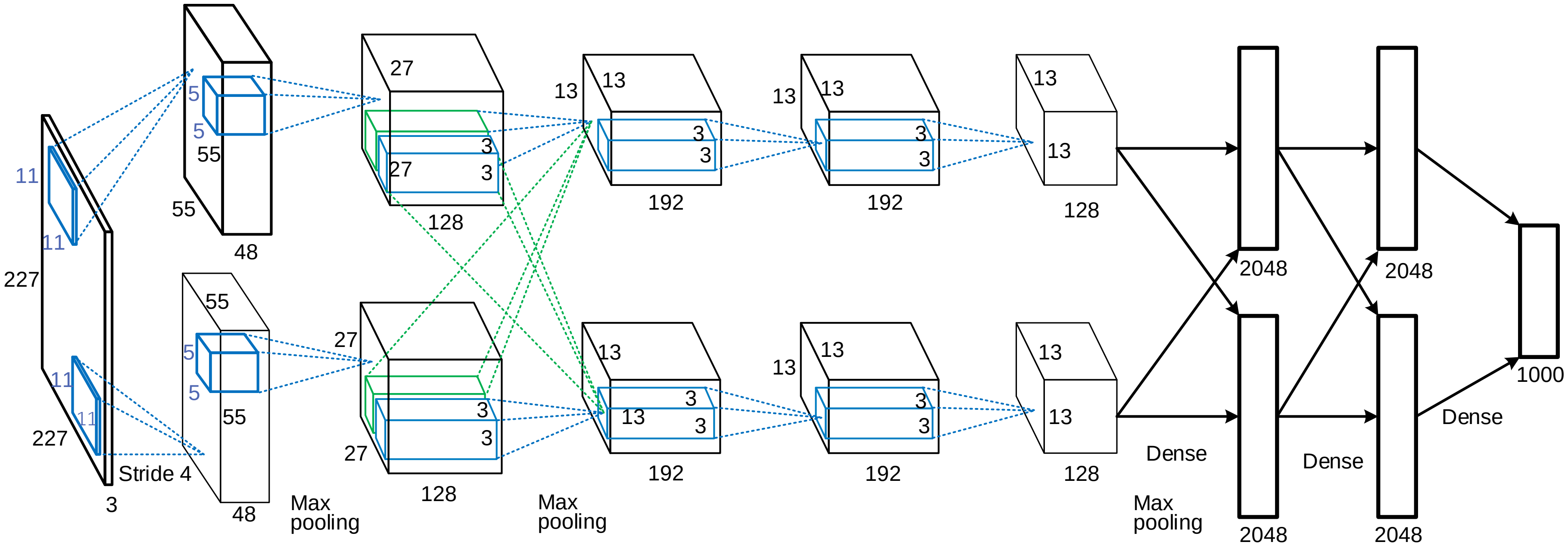


Input: 227×227×3 images

First layer (CONV1): 96 11×11 filters applied at stride 4
Output volume **[55×55×96]**
Parameters: (11*11*3)*96 = **35K**

Case Study: AlexNet

[Krizhevsky *et al.* 2012]



Input: 227×227×3 images

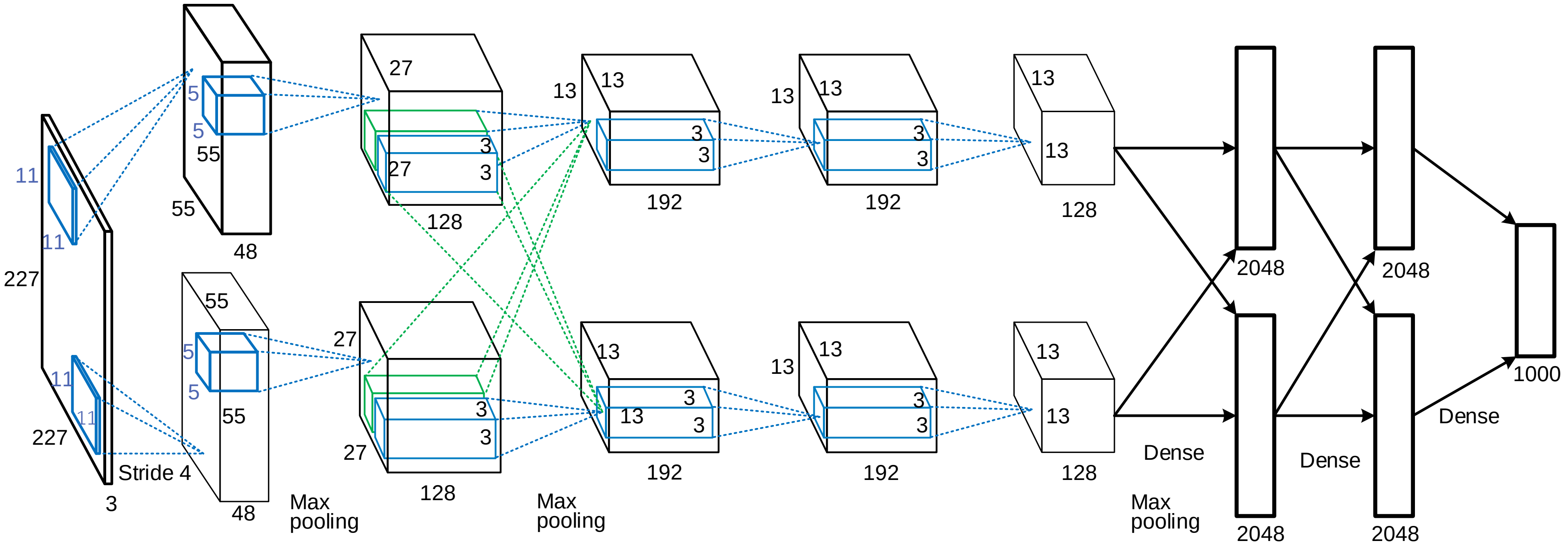
After CONV1: 55×55×96

Second layer (POOL1): 3×3 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: 227×227×3 images

After CONV1: 55×55×96

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

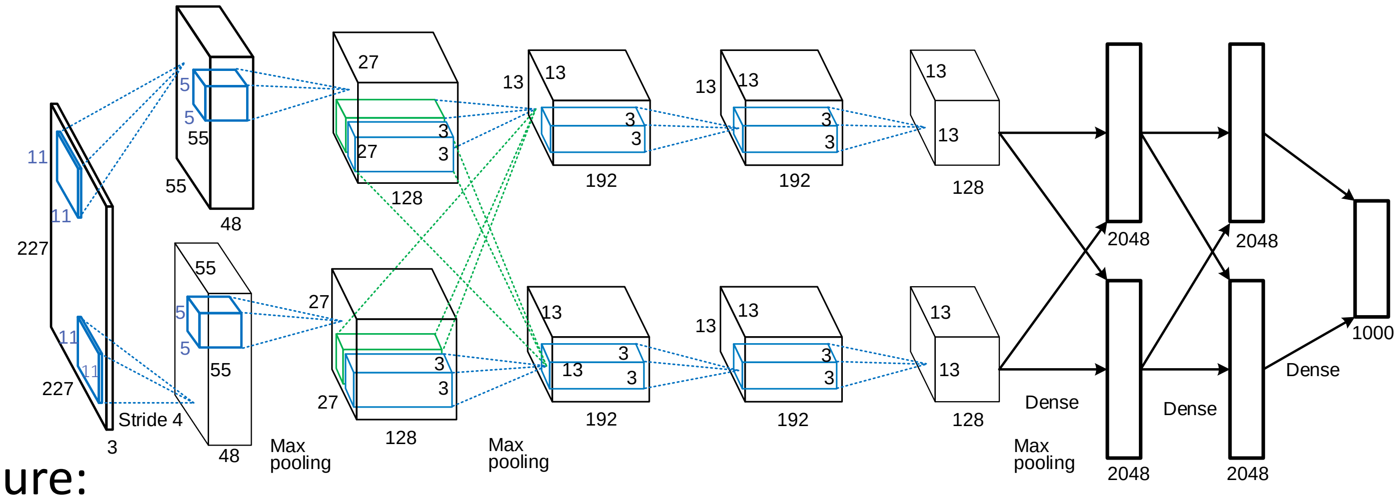
Output volume: 27×27×96

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

Parameters: 0!

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)

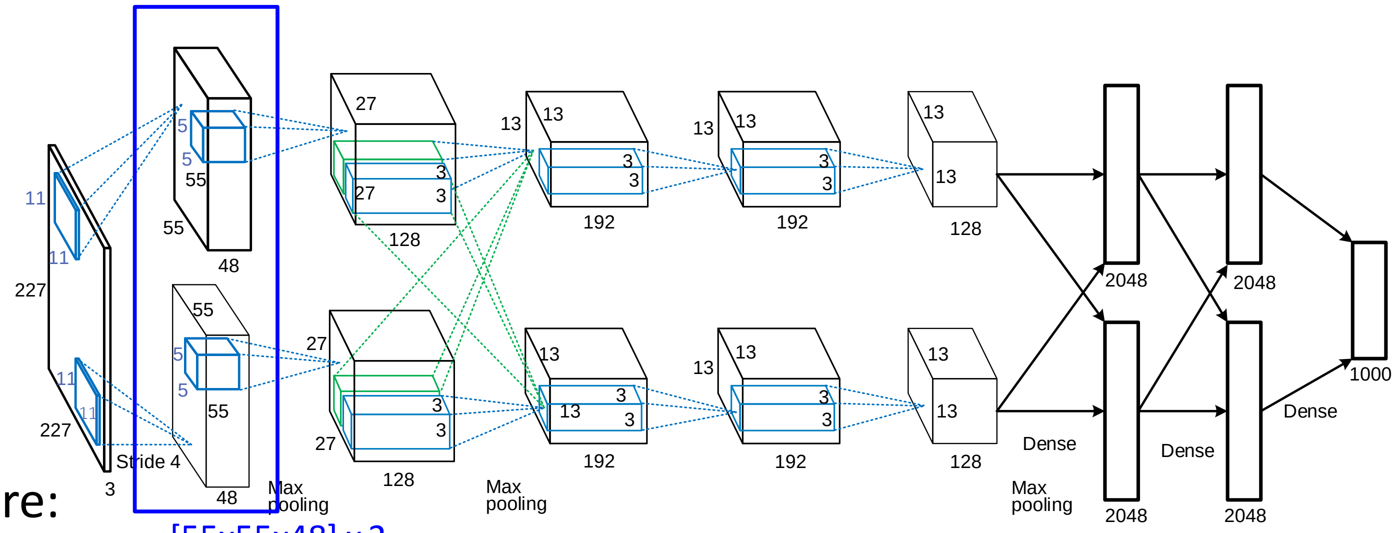
Details:

- 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 0.01, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 0.0005

7 CNN ensemble: 18.2%→ 15.4%

Case Study: AlexNet

[Krizhevsky *et al.* 2012]



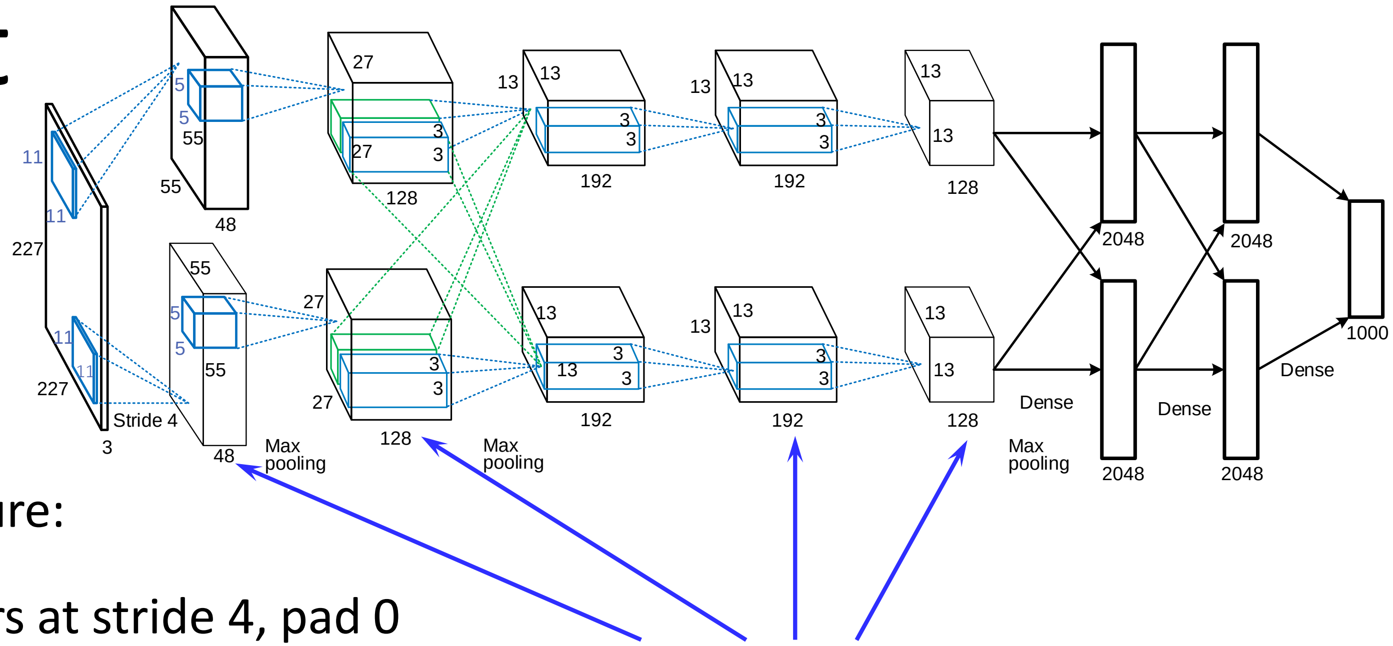
[55x55x48] x 2

Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

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

Case Study: AlexNet

[Krizhevsky *et al.* 2012]

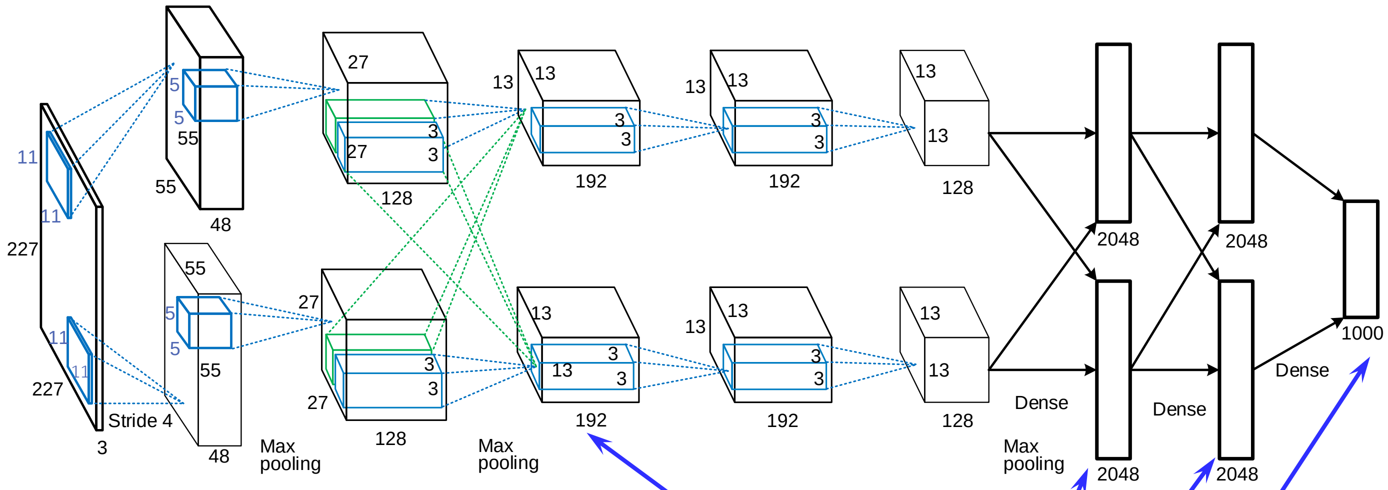


Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] 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

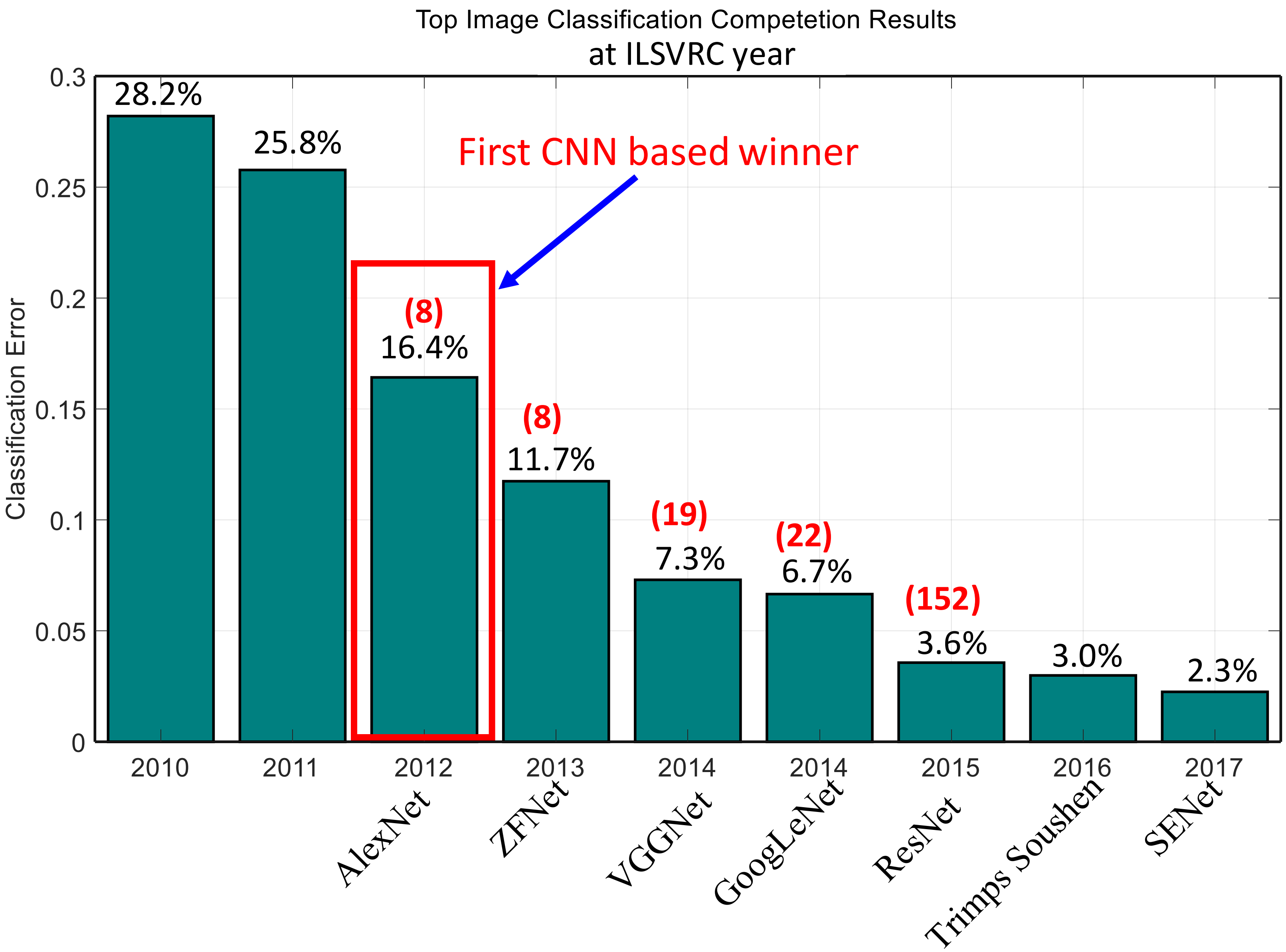
Case Study: AlexNet

[Krizhevsky *et al.* 2012]

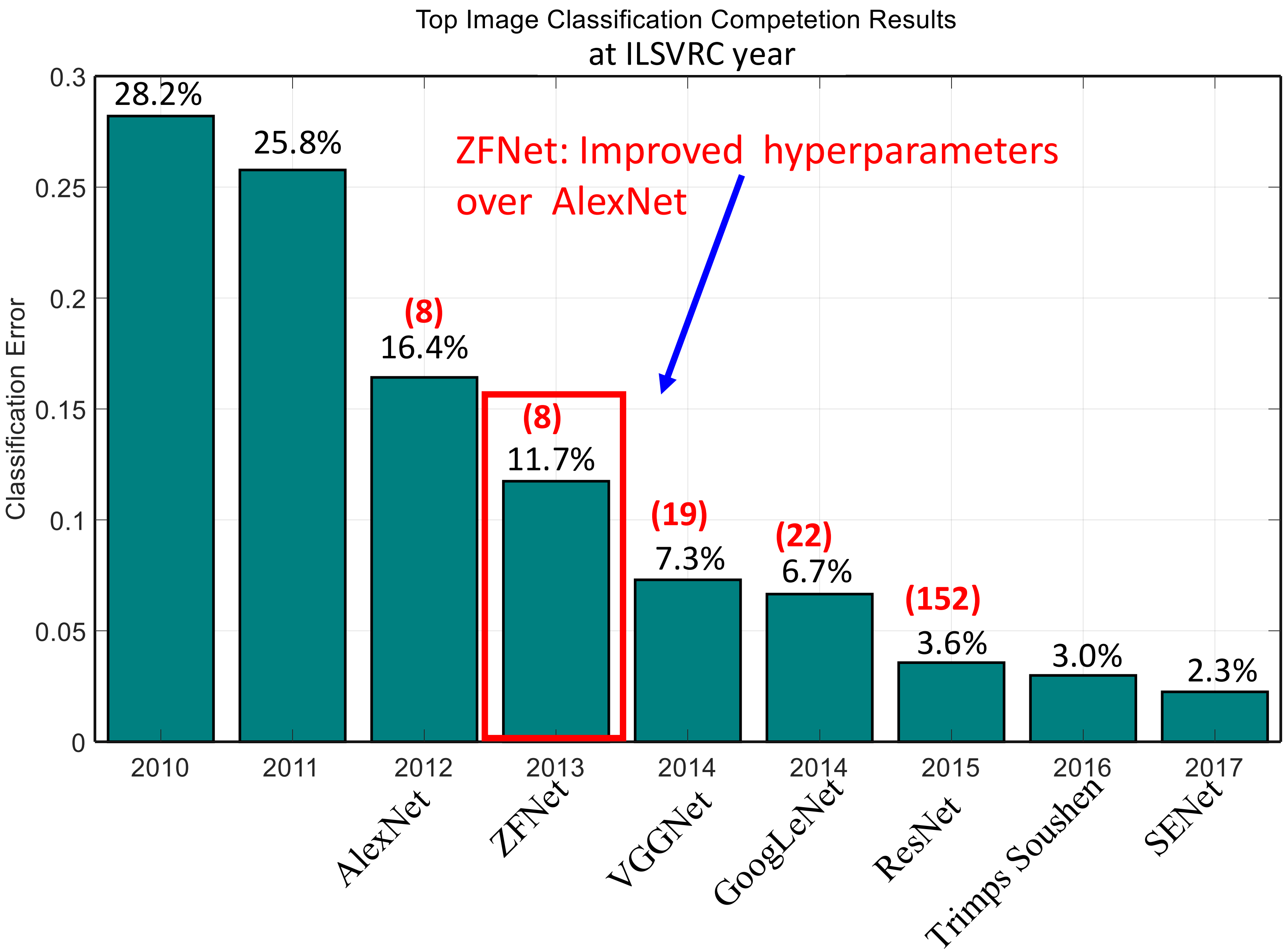


Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] 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



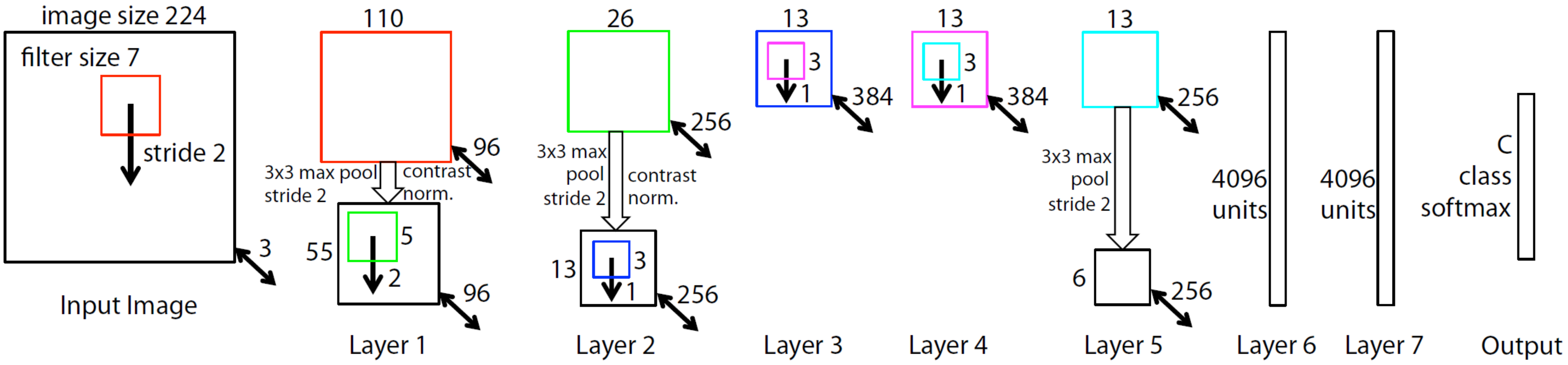
Li Liu et al., Deep Learning for Generic Object Detection: A Survey, IJCV, 2019.



Li Liu et al., Deep Learning for Generic Object Detection: A Survey, IJCV, 2019.

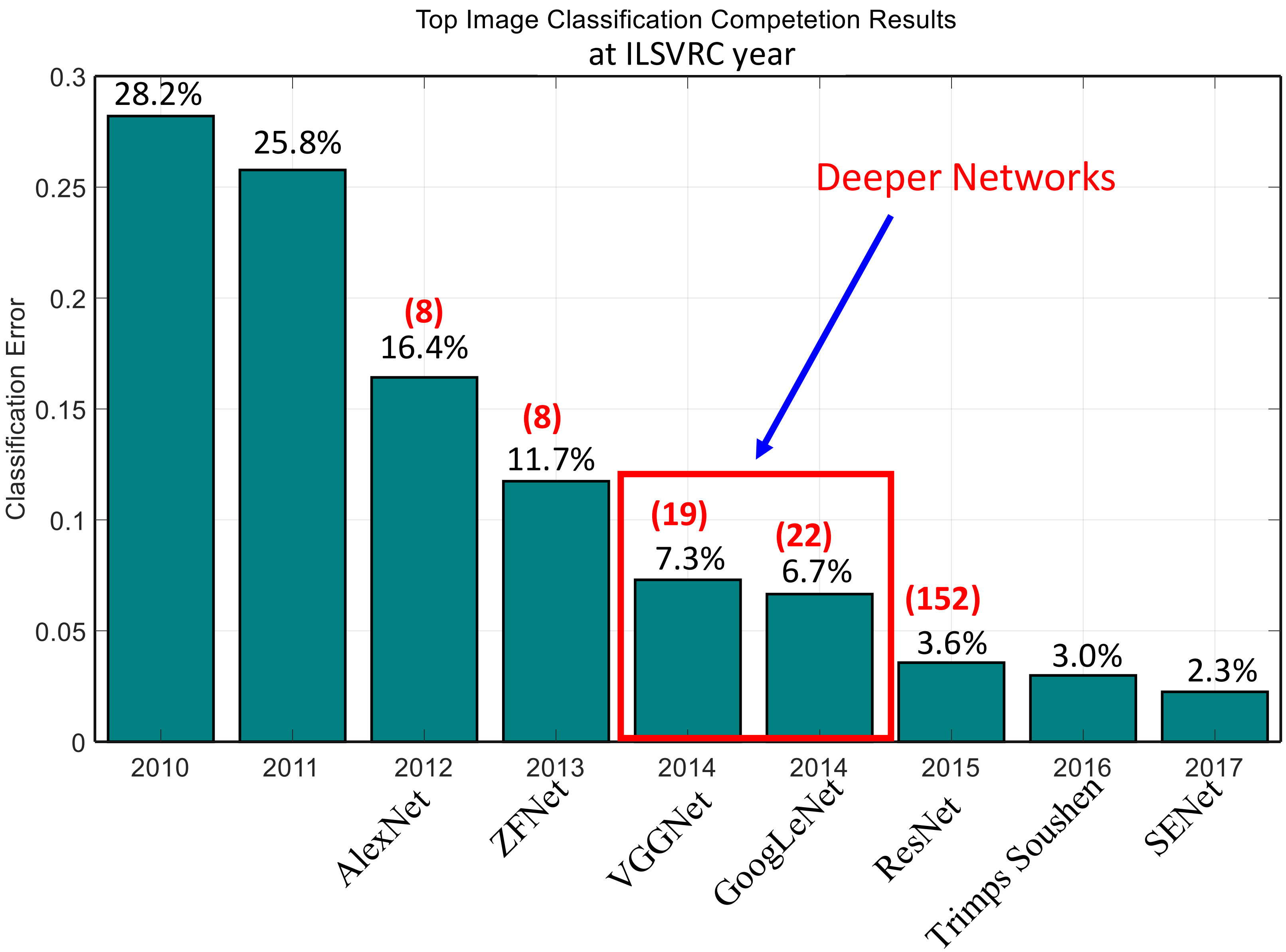
ZFNet

[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% →11.7%



Li Liu et al., Deep Learning for Generic Object Detection: A Survey, IJCV, 2019.

Case Study: VGGNet

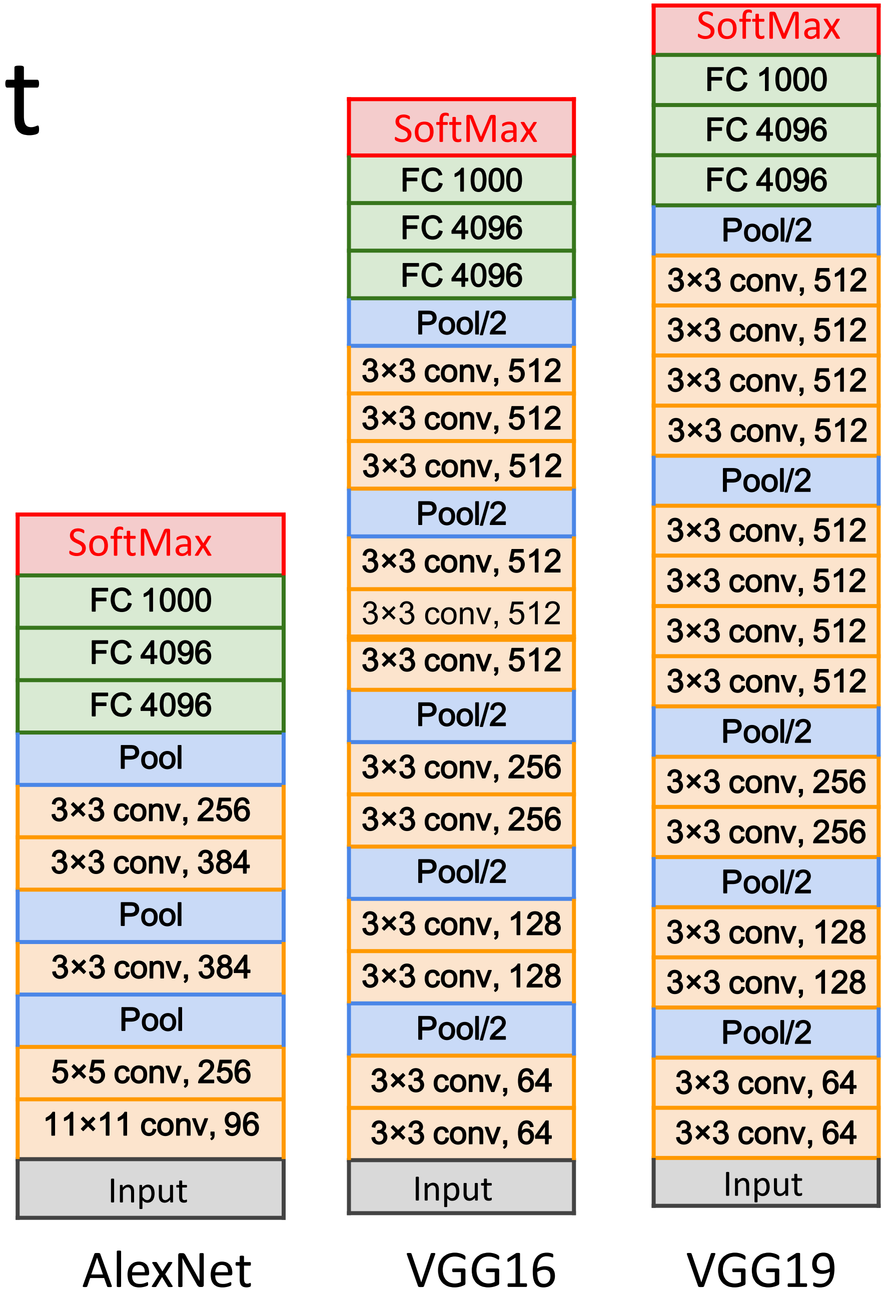
[Simonyan and Zisserman, 2014]

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]

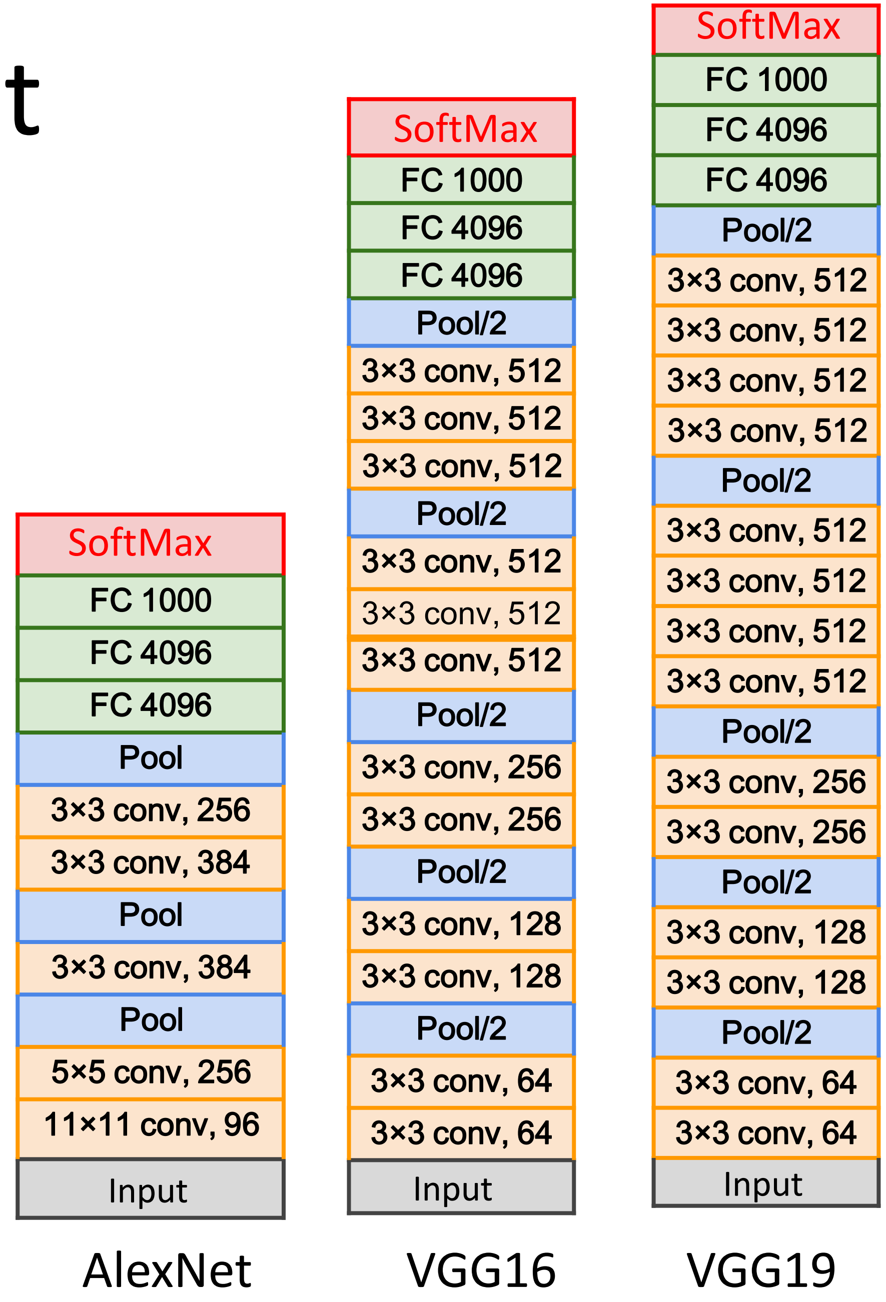
Q: Why use smaller filters? (3x3 conv)

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

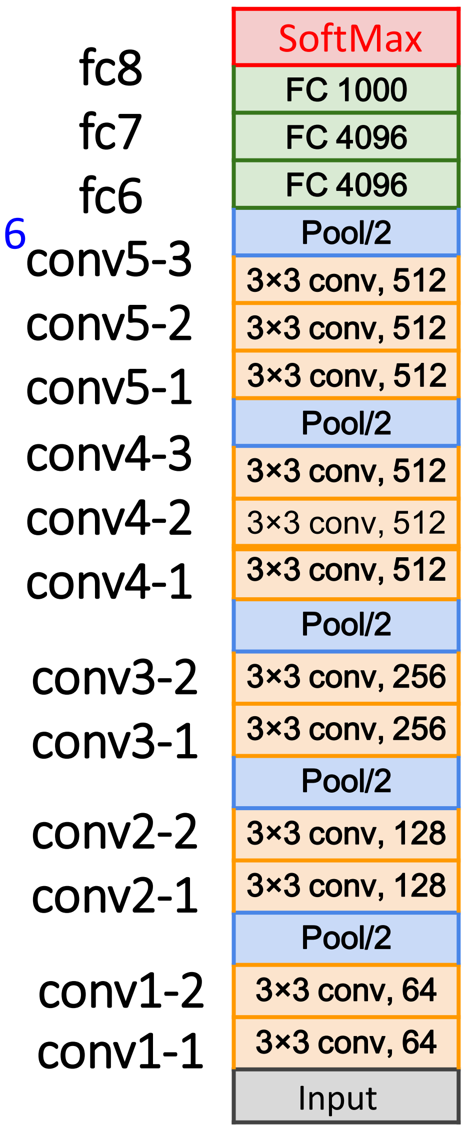
Q: What is the effective receptive field of three 3×3 conv (stride 1) layers?
[7×7]

But deeper, more nonlinearities

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



INPUT: [224x224x3] memory: 224*224*3=150K params: 0
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! about*2 for bwd)
TOTAL params: 138M parameters

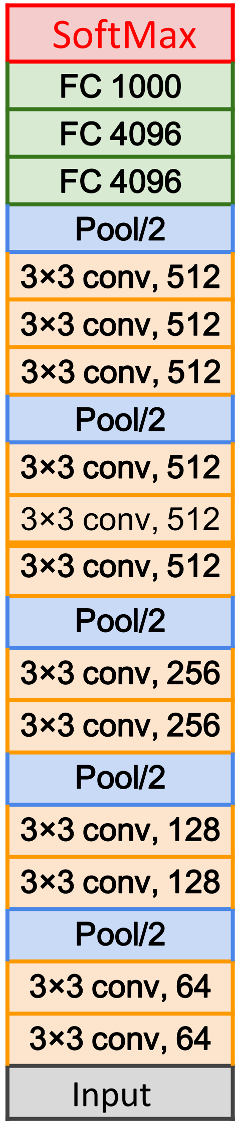
VGG16

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
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

Most memory is in early CONV

Most params are in late FC

TOTAL memory: 24M * 4 bytes ~= 96MB / image (only forward! about*2 for bwd)
TOTAL params: 138M parameters



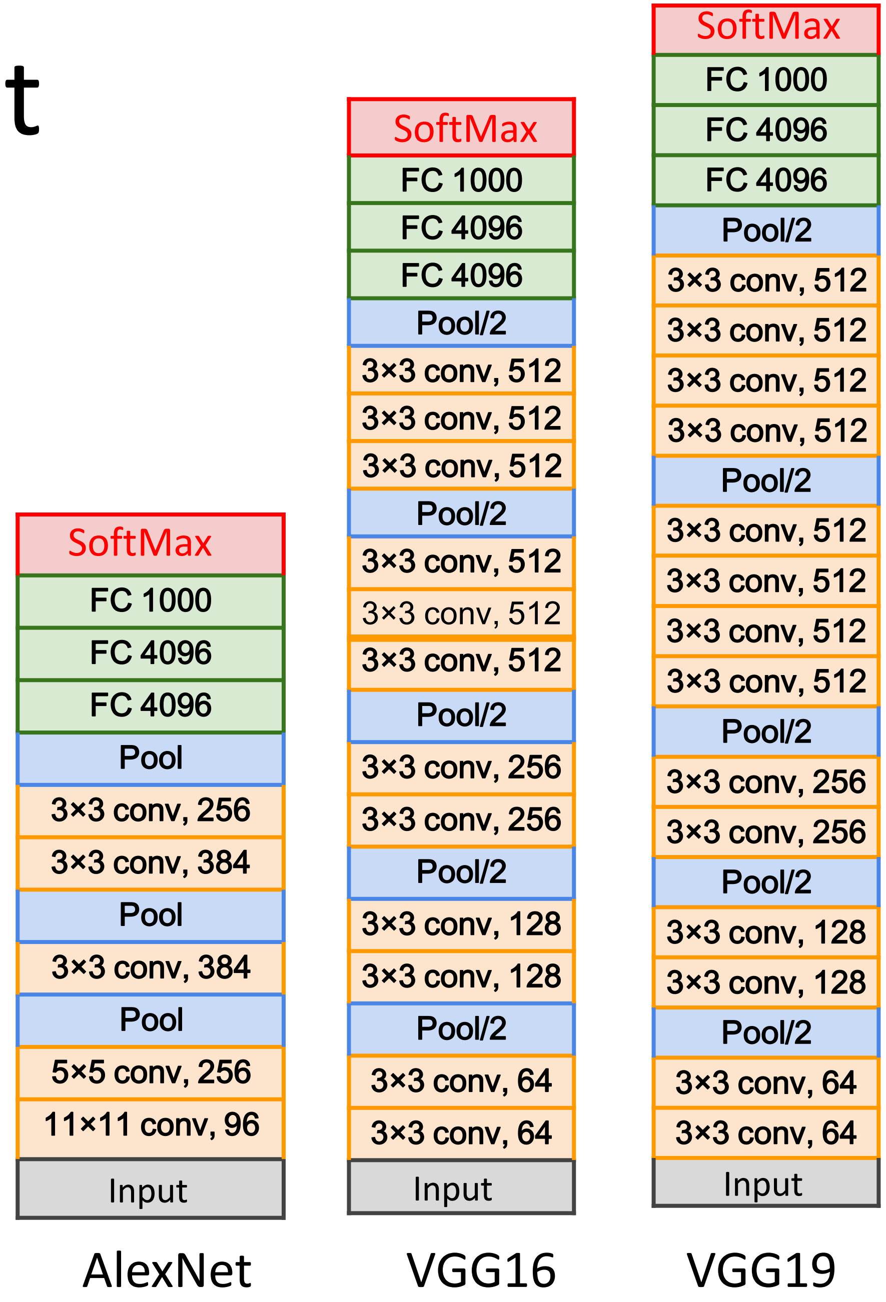
VGG16

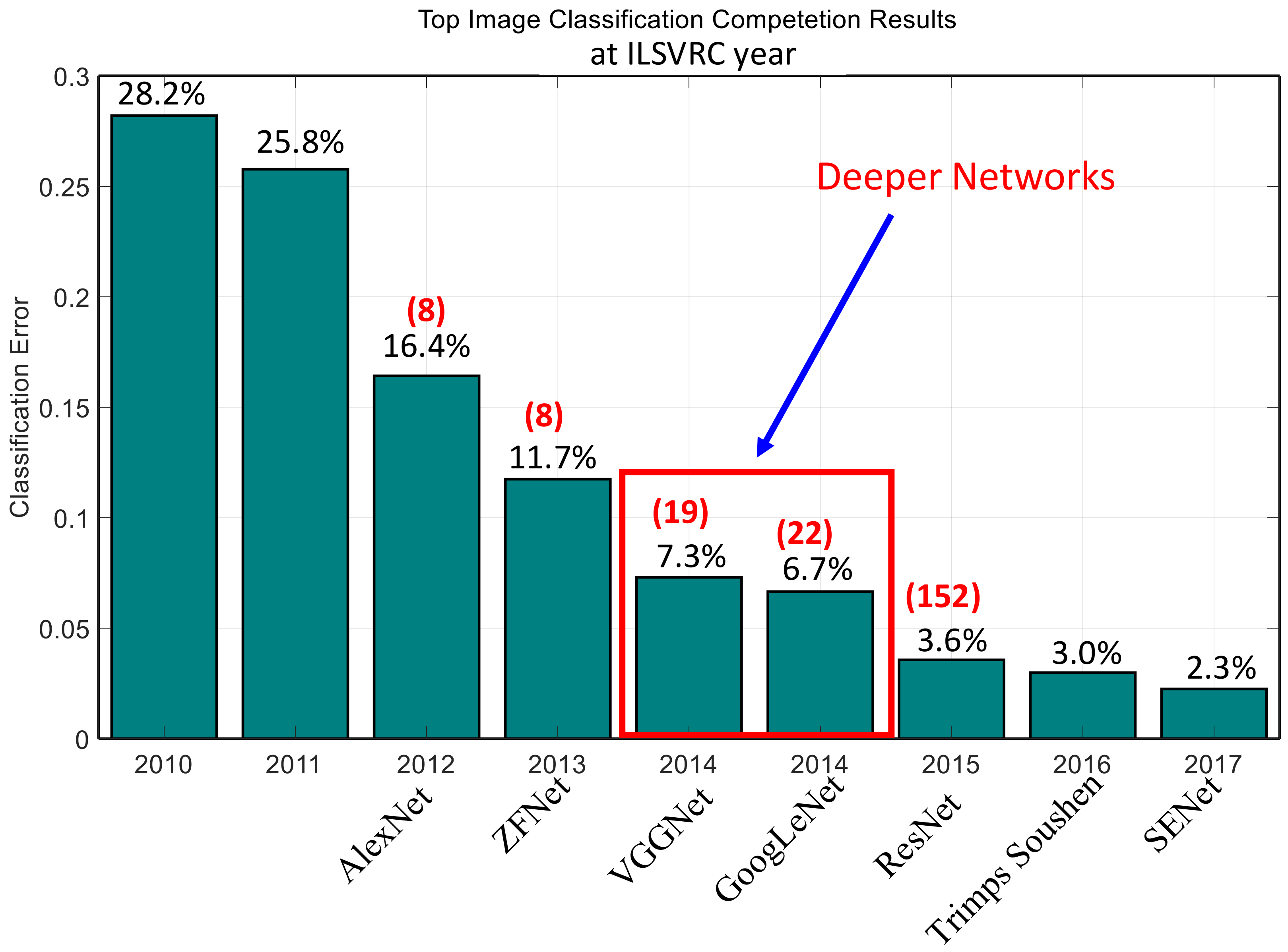
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Details:

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



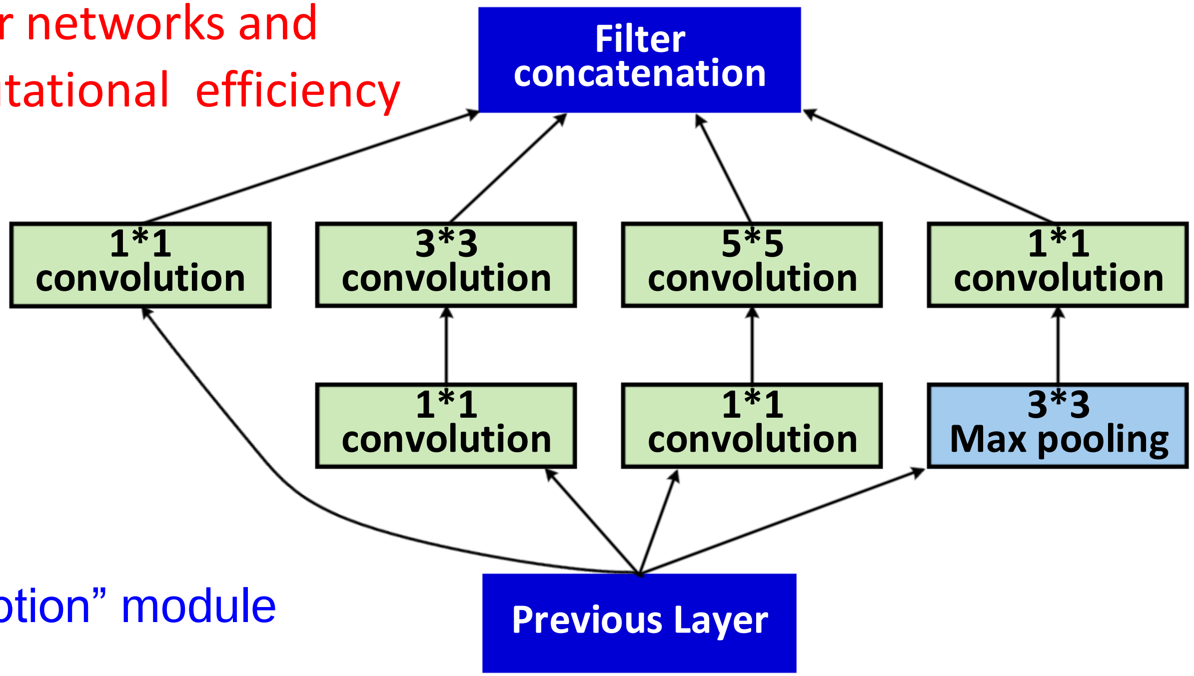


Li Liu et al., Deep Learning for Generic Object Detection: A Survey, IJCV, 2019.

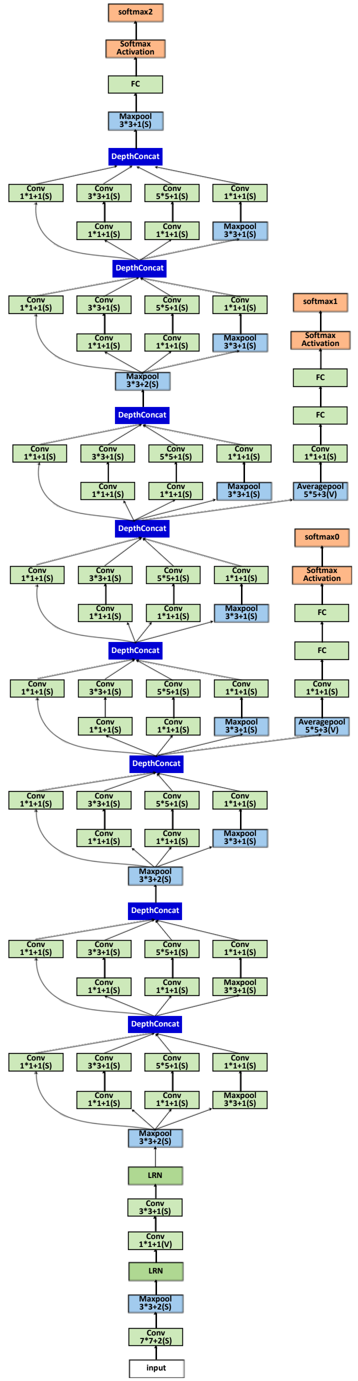
Case Study: GoogLeNet

[Szegedy et al., 2014]

Deeper networks and computational efficiency



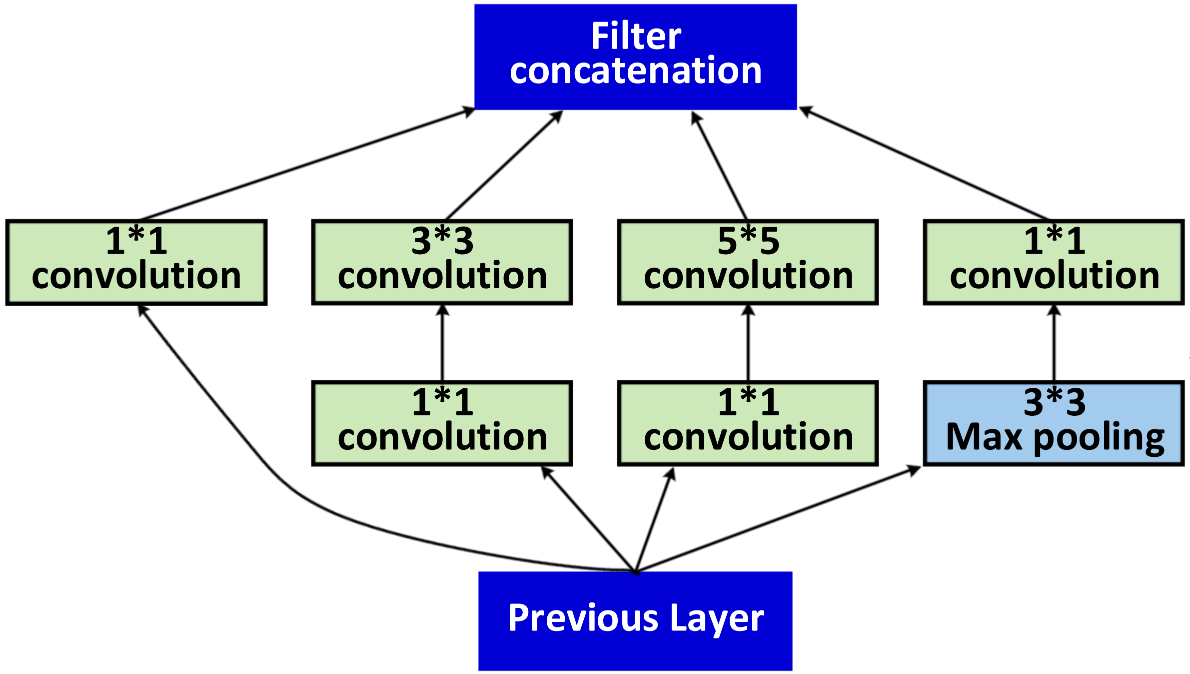
Inception module



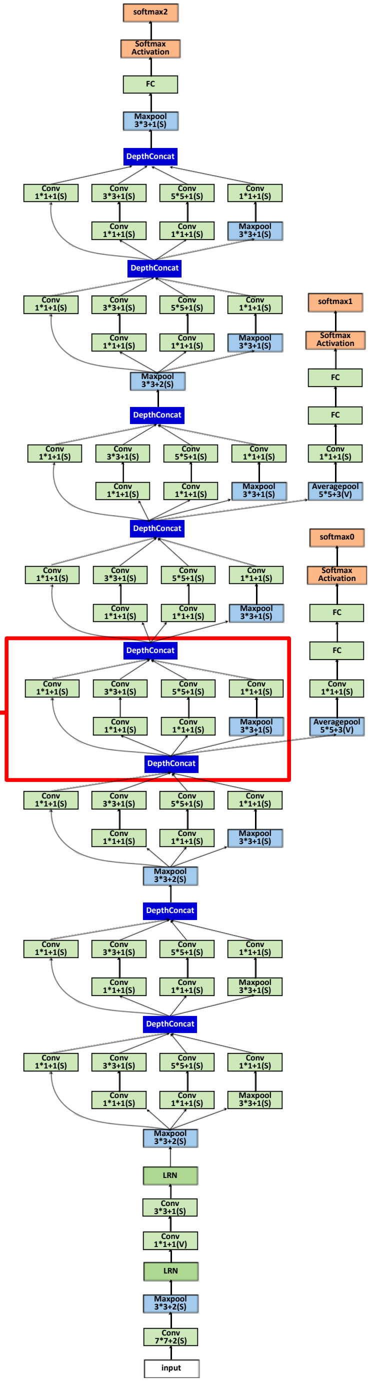
- 22 layers
- Efficient "Inception" module
- No FC layers
- Only 5 million parameters!
  - 12× less than AlexNet
- ILSVRC'14 classification winner (6.7% top 5 error)

Case Study: GoogLeNet

[Szegedy et al., 2014]



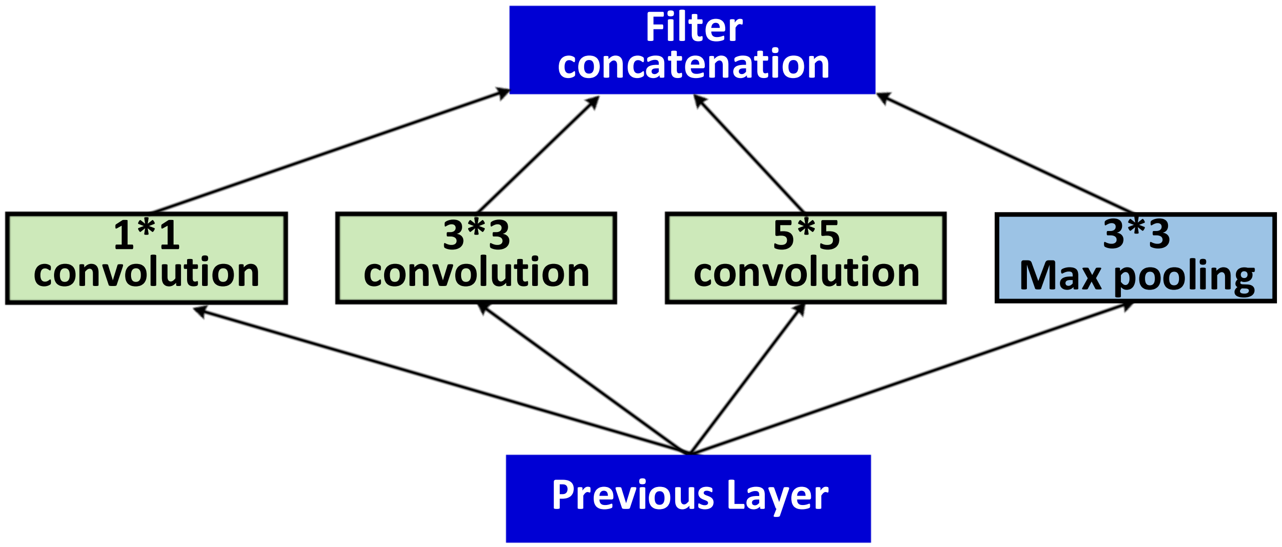
Inception module



"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 depthwise.

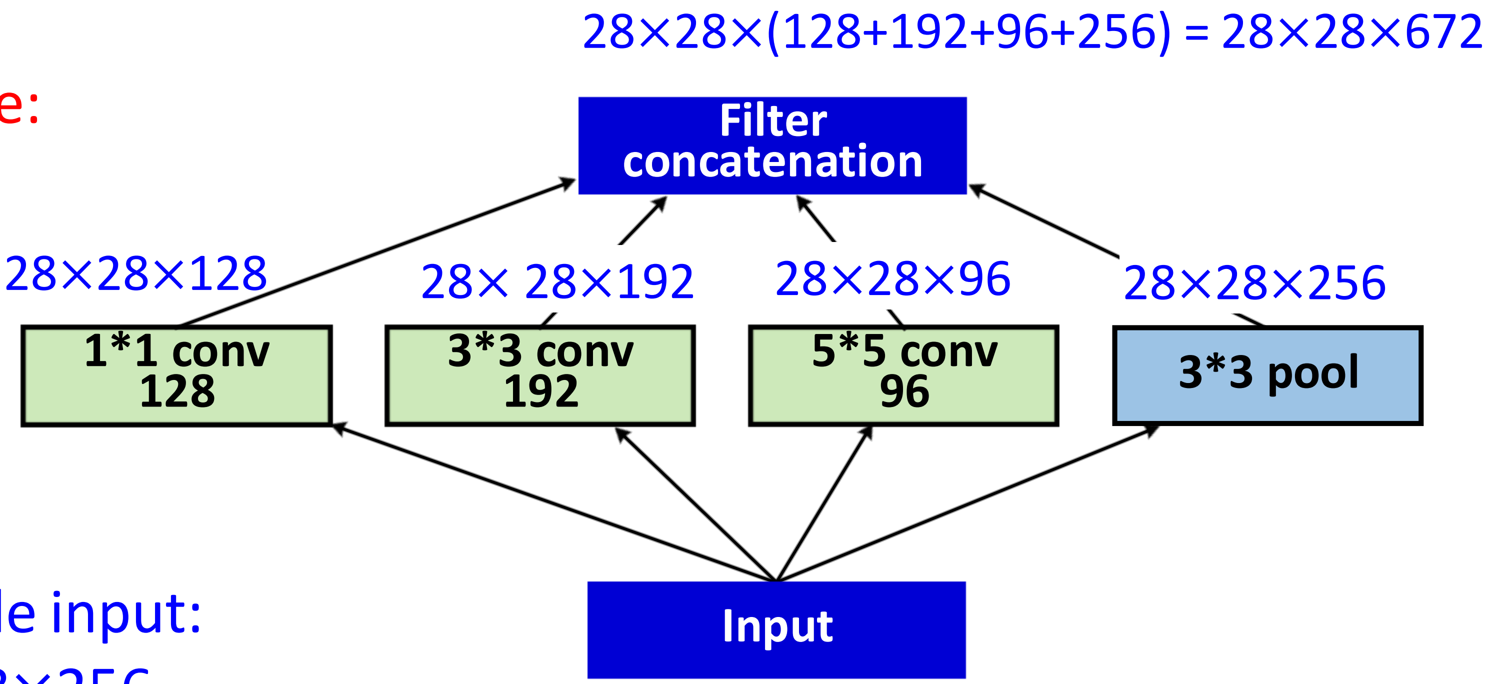
Q: What is the problem with this? [Hint: Computational complexity]

Case Study: GoogLeNet

[Szegedy et al., 2014]

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

Example:



Module input:
28×28×256

Naive Inception module

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

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

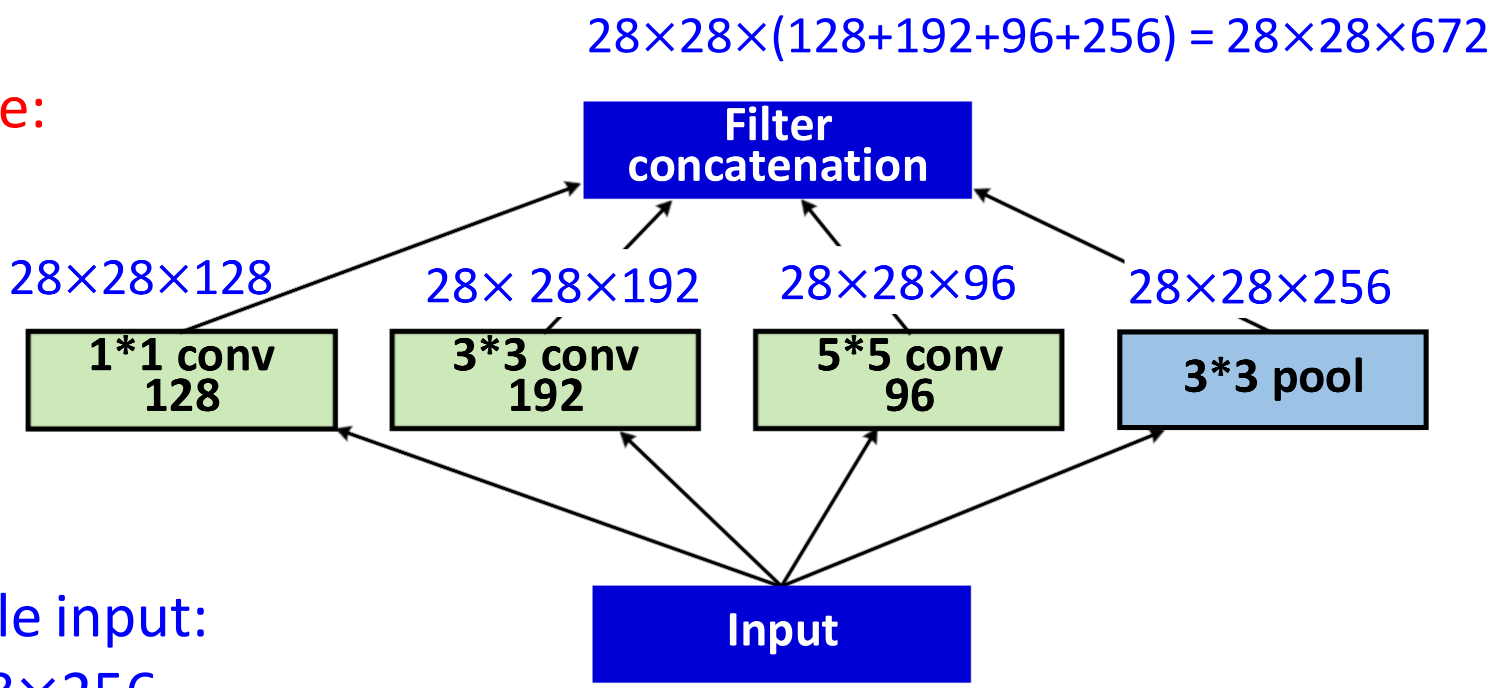
Q3: What is output size after filter concatenation?

Case Study: GoogLeNet

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

[Szegedy et al., 2014]

Example:



Module input:
28×28×256

Naive Inception module

Conv Ops:
[1×1 conv, 128] 28*28*128*1*1*256
[3×3 conv, 192] 28*28*192*3*3*256
[5×5 conv, 96] 28*28*96*5*5*256
Total: 854M ops

Very expensive compute

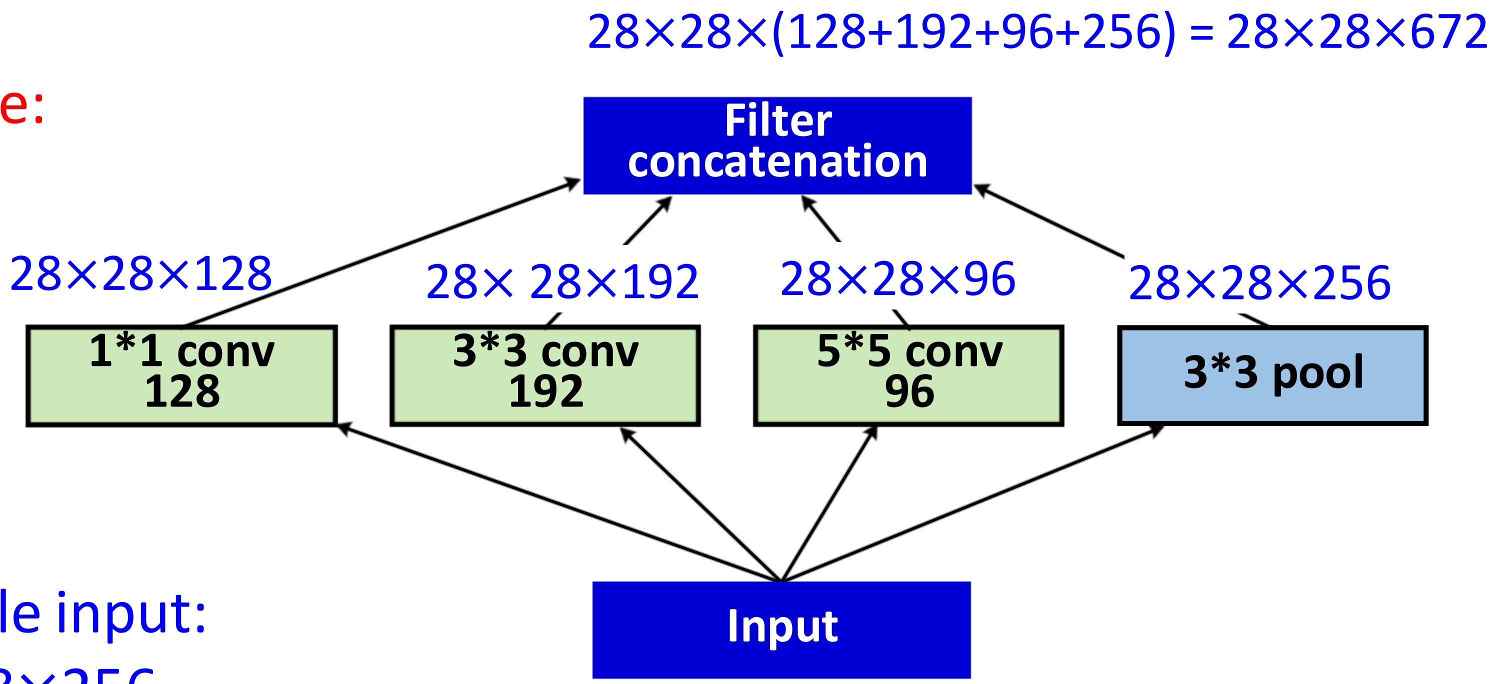
Pooling layer also preserves feature depth, which means total depth after concatenation can only grow at every layer!

Case Study: GoogLeNet

[Szegedy et al., 2014]

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

Example:

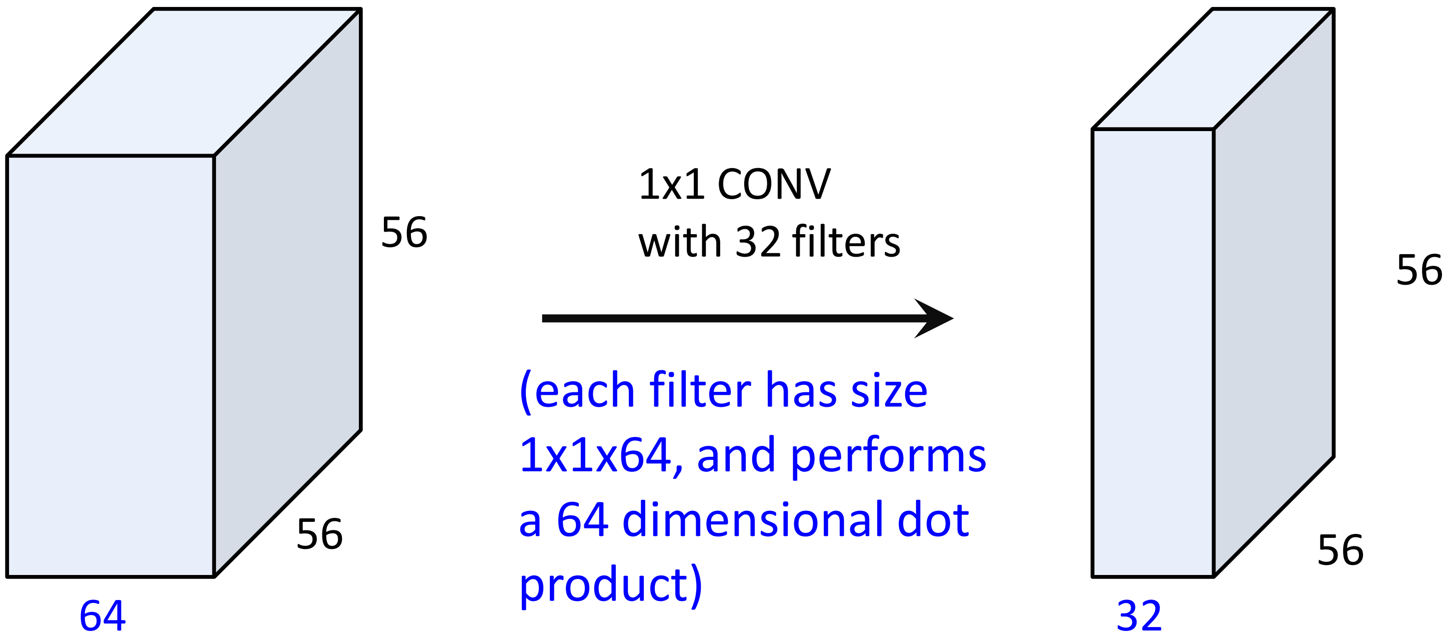


Module input: 28×28×256

Naive Inception module

Solution: “bottleneck” layers that use 1×1 convolutions to reduce feature depth.

Reminder: 1x1 convolutions

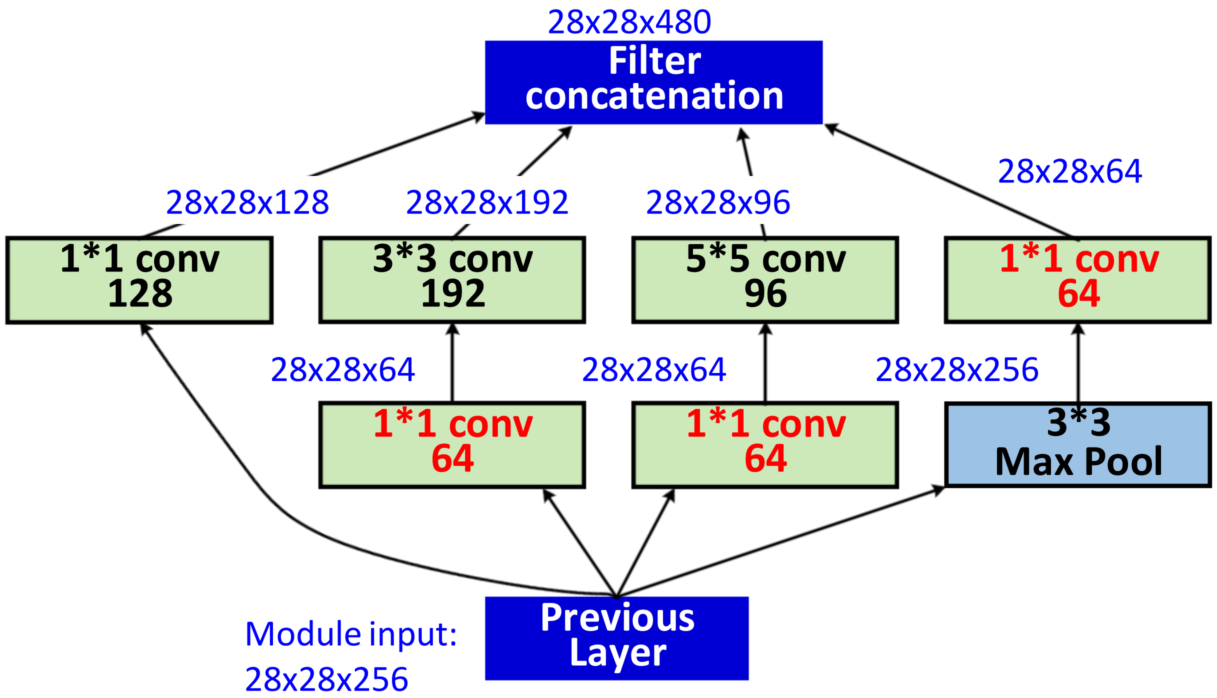


Preserves spatial dimensions, reduces depth!

Projects depth to lower dimension (combination of feature maps)

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

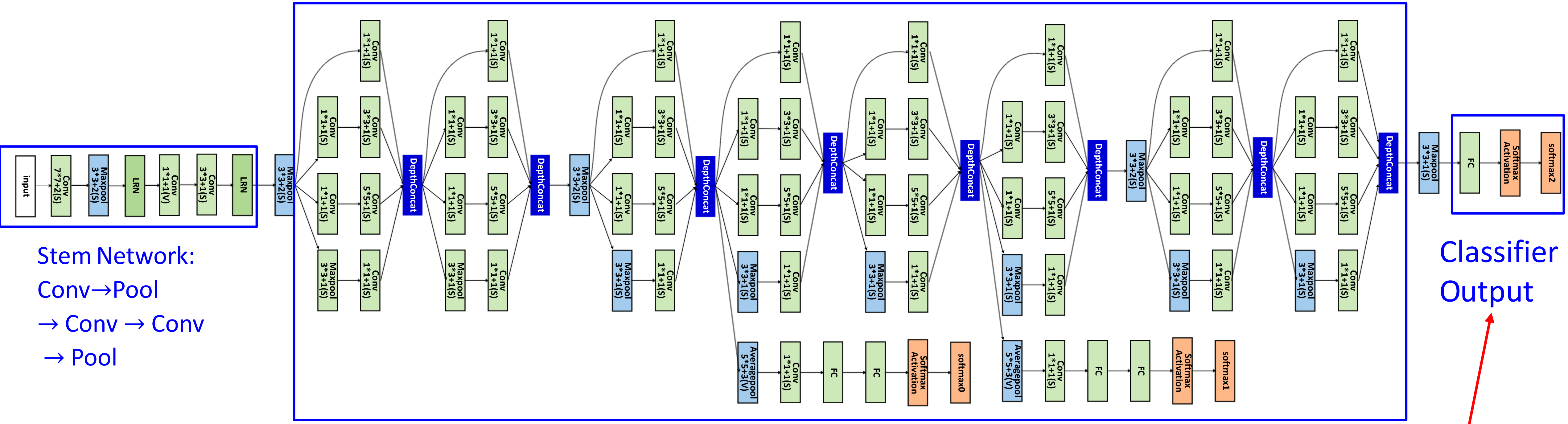
Total: 358M ops

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

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

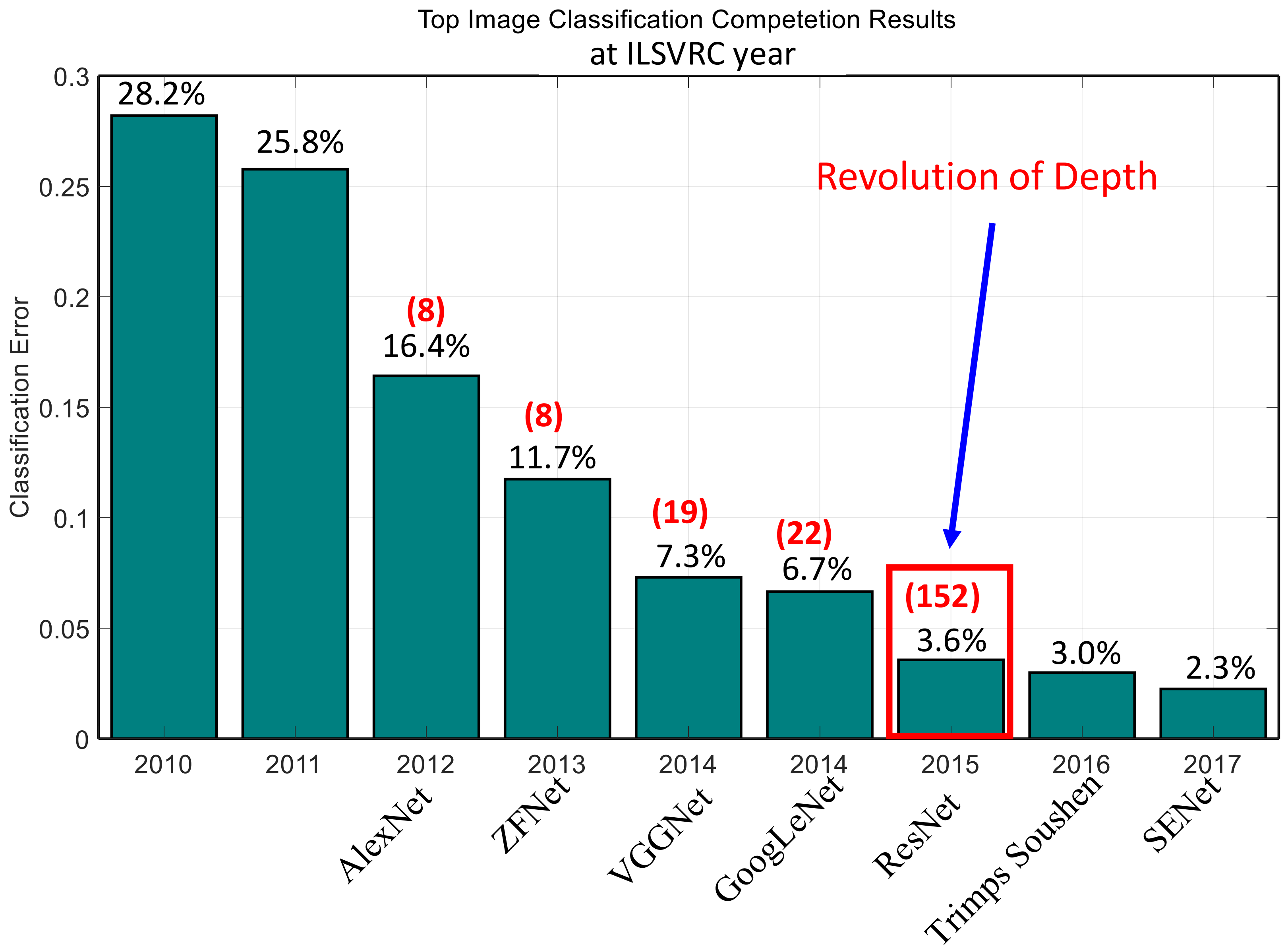


Stem Network:
Conv→Pool
→ Conv → Conv
→ Pool

Classifier Output

Stacked Inception Modules

(removed expensive FC layers!)



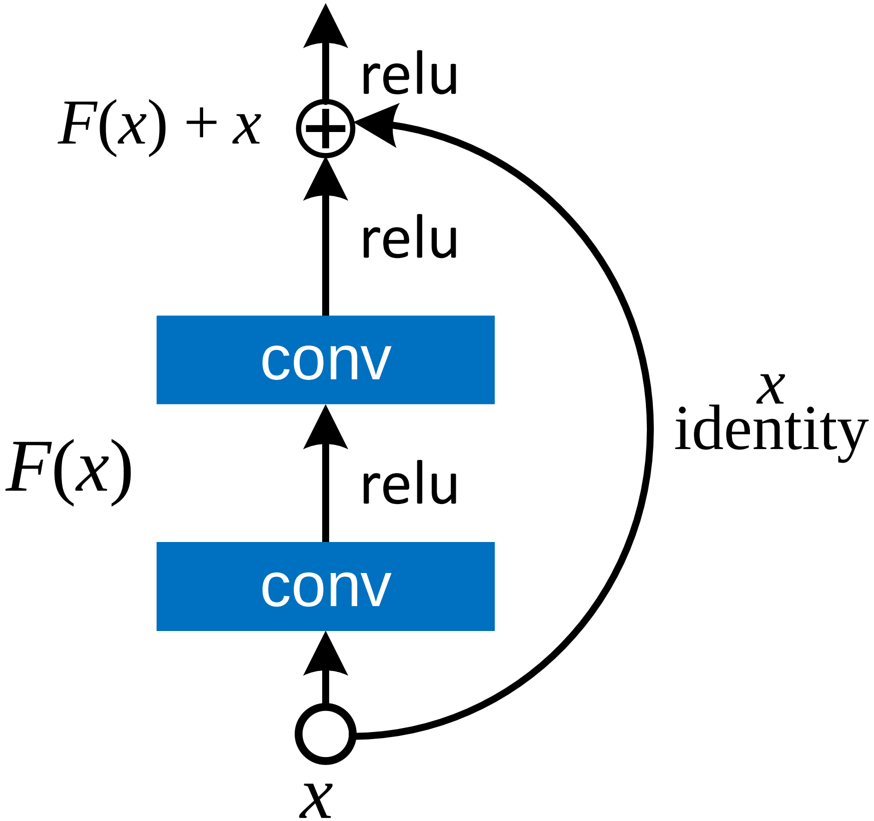
Li Liu et al., Deep Learning for Generic Object Detection: A Survey, IJCV, 2019.

Case Study: ResNet

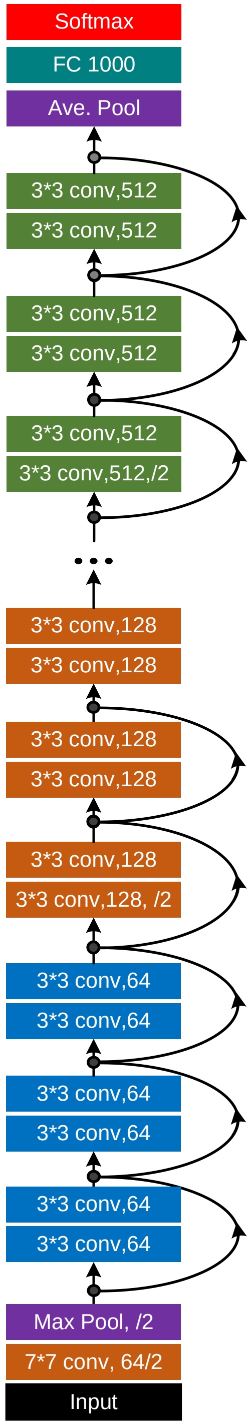
[He et al., 2015]

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!



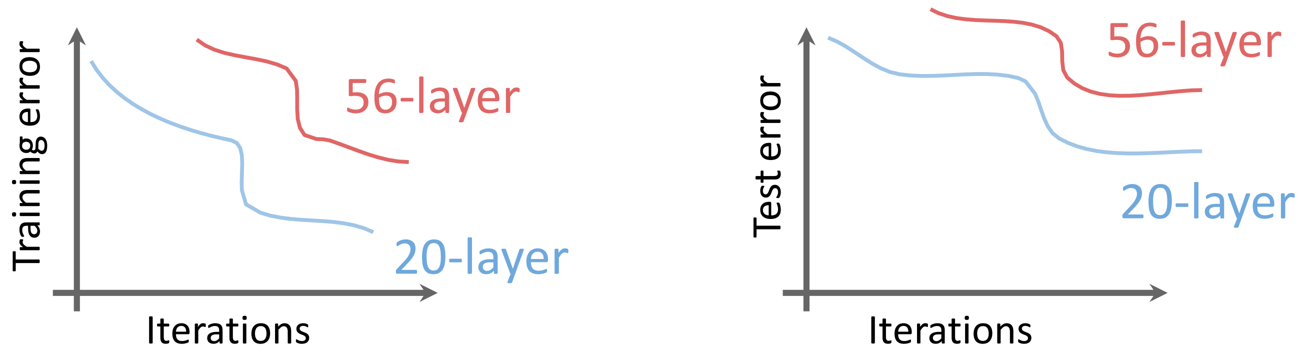
Residual block



Case Study: ResNet

[He et al., 2015]

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



Q: What's strange about these training and test curves?
[Hint: look at the order of the curves]

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]

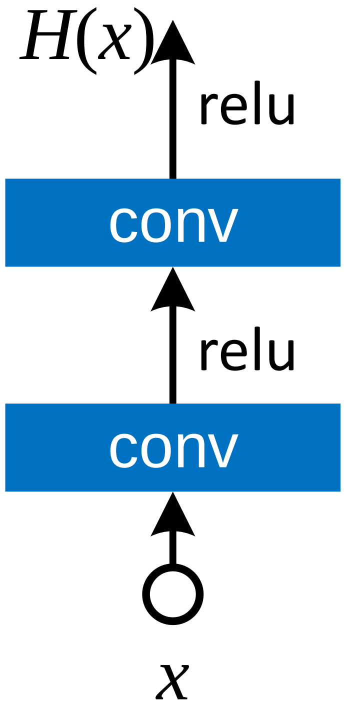
Hypothesis: the problem is an *optimization* problem, deeper models are harder to optimize

- The deeper model should be able to perform at least as well as the shallower model.
- A solution by construction is copying the learned layers from the shallower model and setting additional layers to identity mapping.

Case Study: ResNet

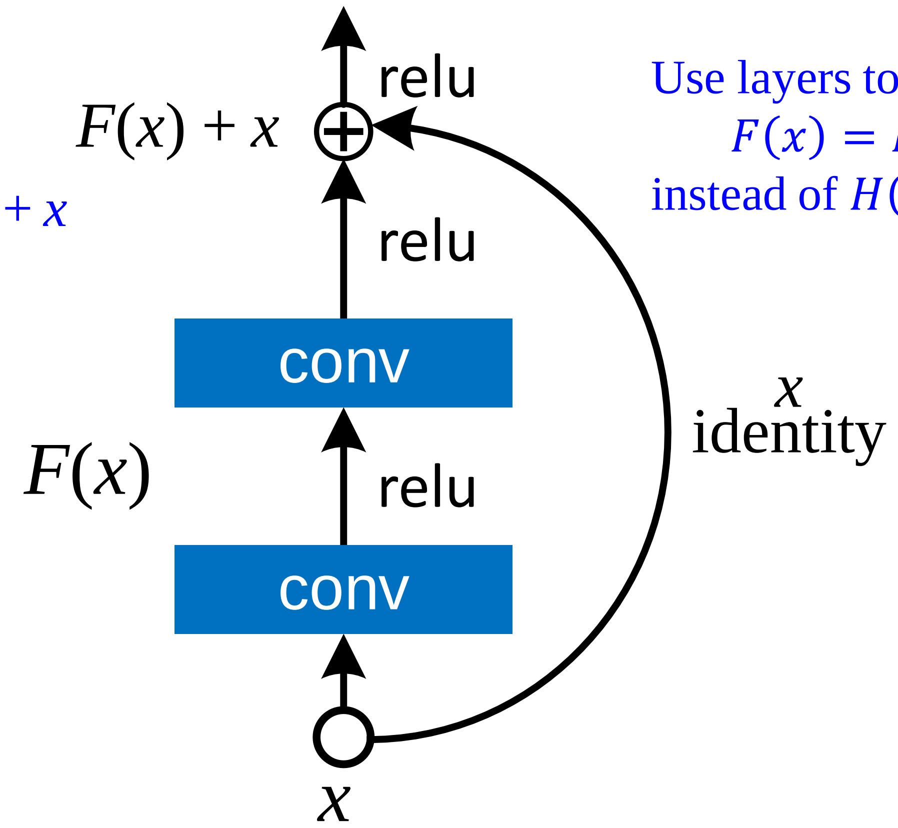
[He et al., 2015]

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



Plain block

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



Residual block

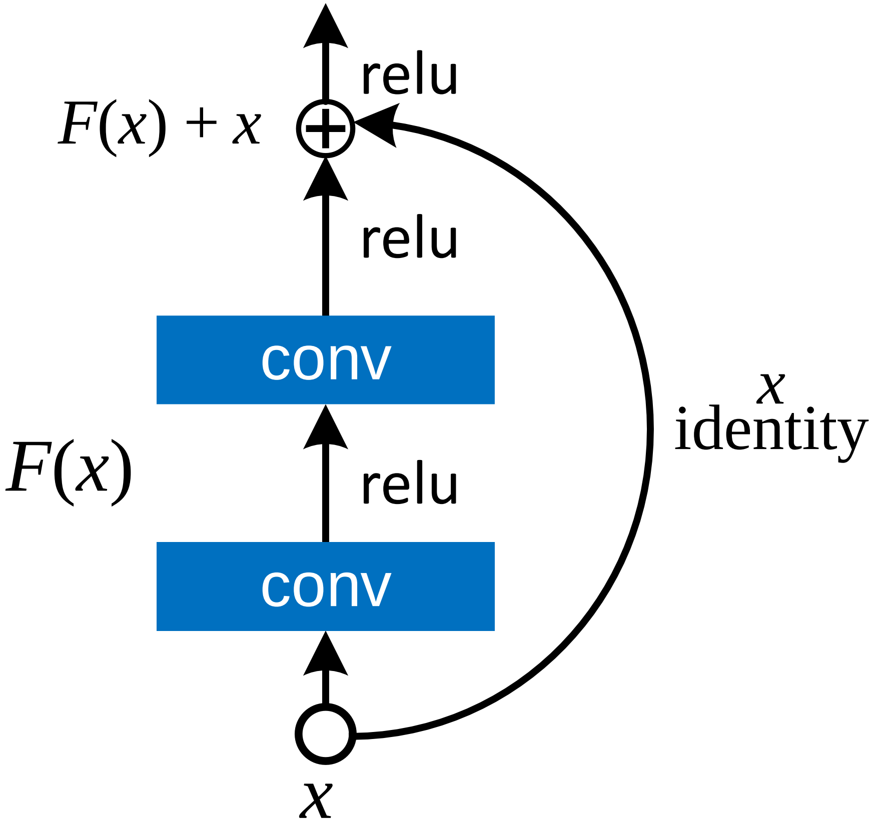
Use layers to fit residual $F(x) = H(x) - x$ instead of $H(x)$ directly

Case Study: ResNet

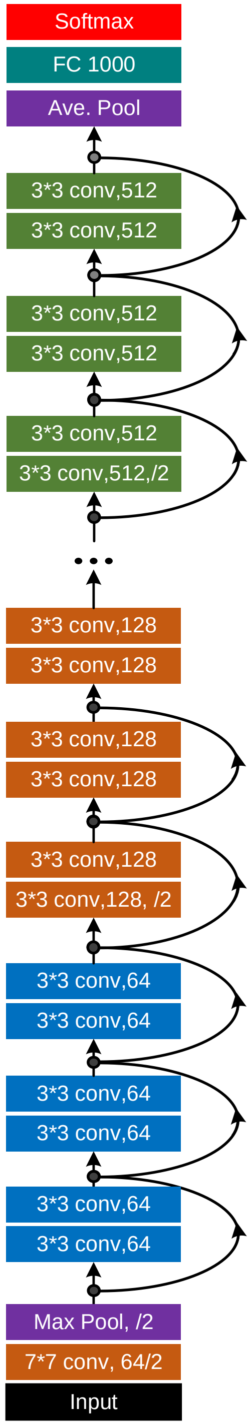
[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double the number of filters and subsample spatially using stride 2
- Additional conv layer at the beginning
- No FC layers at the end (only FC 1000 to output classes)



Residual block



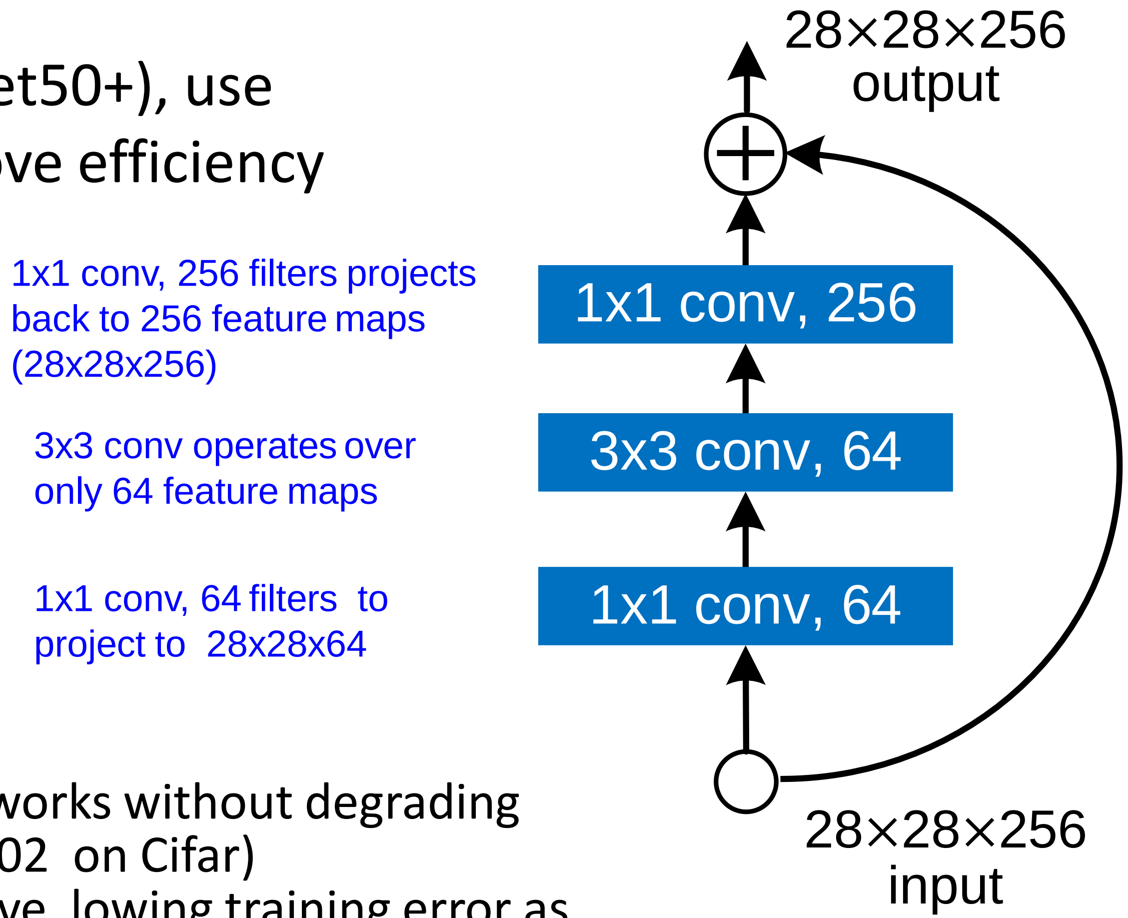
Total depths of 34, 50, 101, or 152 layers for ImageNet

Case Study: ResNet

[He et al., 2015]

For deeper networks (ResNet50+), use "bottleneck" layer to improve efficiency (similar to GoogLeNet)

ILSVRC 2015 classification winner (3.6% top 5 error)→better than "human performance"!



Experimental Results

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

Typical CNN Architectures

Case Studies

- AlexNet
- VGG
- GoogLeNet
- ResNet

about other architecture...

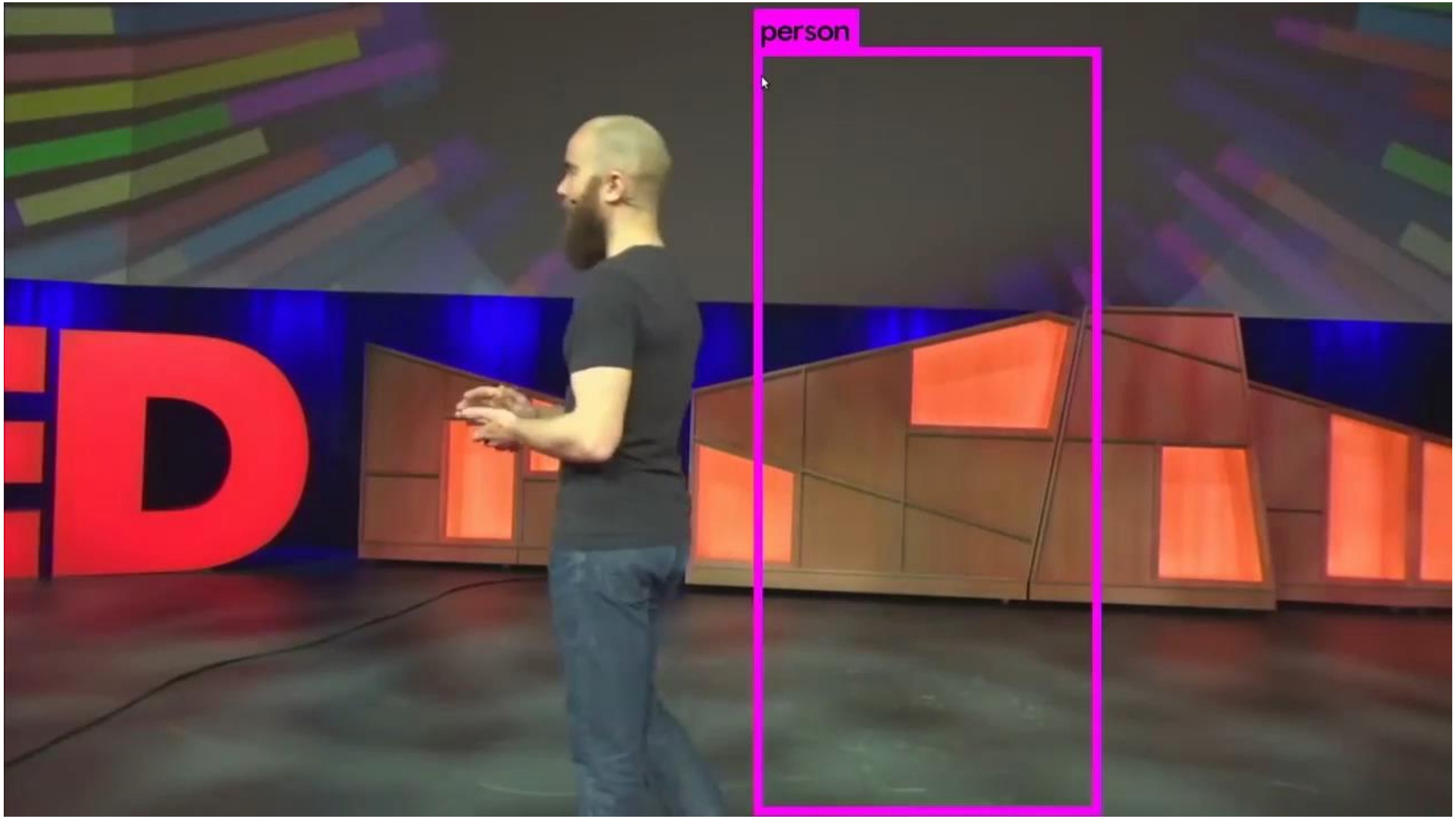
- NIN (Network in Network)
- Wide ResNet
- ResNeXT
- Stochastic Depth

- DenseNet
- SENet
- FractalNet
- SqueezeNet

No time to talk about, leave for yourself

Summary: CNN Architectures

- VGG, GoogLeNet, ResNet all in wide use, available in model zoos
- ResNet current best default
- Trend towards extremely deep networks
- Significant research centers around design of layer / skip connections and improving gradient flow
- Even more recent trend towards examining necessity of depth vs. width and residual connections
- Trends in network compression and acceleration
- Trends in network architecture search


person

International Journal of Computer Vision
https://doi.org/10.1007/s11263-019-01247-4

Check for updates

Deep Learning for Generic Object Detection: A Survey

Li Liu[1,2] · Wanli Ouyang[3] · Xiaogang Wang[4] · Paul Fieguth[5] · Jie Chen[2] · Xinwang Liu[1] · Matti Pietikäinen[2]

Received: 6 September 2018 / Accepted: 26 September 2019

Abstract
Object detection, one of the most fundamental and challenging problems in computer vision, seeks to locate object instances from a large number of predefined categories in natural images. Deep learning techniques have emerged as a powerful strategy for learning feature representations directly from data and have led to remarkable breakthroughs in the field of generic object detection. Given this period of rapid evolution, the goal of this paper is to provide a comprehensive survey of the recent achievements in this field brought about by deep learning techniques. More than 300 research contributions are included in this survey, covering many aspects of generic object detection: detection frameworks, object feature representation, object proposal generation, context modeling, training strategies, and evaluation metrics. We finish the survey by identifying promising directions for future research.

Keywords Object detection · Deep learning · Convolutional neural networks · Object recognition

https://link.springer.com/content/pdf/10.1007%2Fs11263-019-01247-4.pdf

International Journal of Computer Vision (2019) 127:74–109
https://doi.org/10.1007/s11263-018-1125-z

CrossMark

From BoW to CNN: Two Decades of Texture Representation for Texture Classification

Li Liu[1,2] · Jie Chen[2] · Paul Fieguth[3] · Guoying Zhao[2] · Rama Chellappa[4] · Matti Pietikäinen[2]

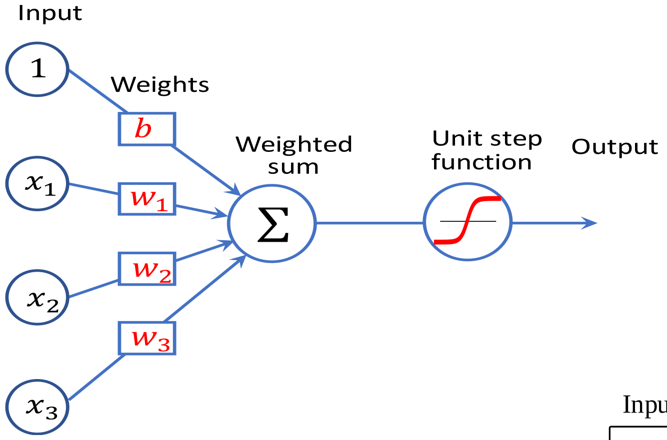
Received: 6 January 2018 / Accepted: 6 October 2018 / Published online: 8 November 2018

Abstract
Texture is a fundamental characteristic of many types of images, and texture representation is one of the essential and challenging problems in computer vision and pattern recognition which has attracted extensive research attention over several decades. Since 2000, texture representations based on Bag of Words and on Convolutional Neural Networks have been extensively studied with impressive performance. Given this period of remarkable evolution, this paper aims to present a comprehensive survey of advances in texture representation over the last two decades. More than 250 major publications are cited in this survey covering different aspects of the research, including benchmark datasets and state of the art results. In retrospect of what has been achieved so far, the survey discusses open challenges and directions for future research.

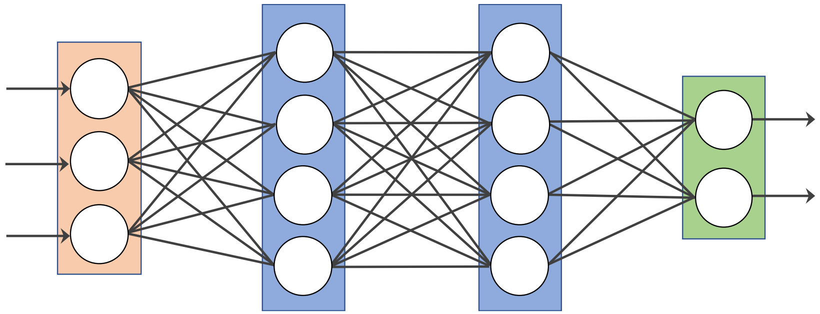
https://link.springer.com/content/pdf/10.1007%2Fs11263-018-1125-z.pdf

In this Course

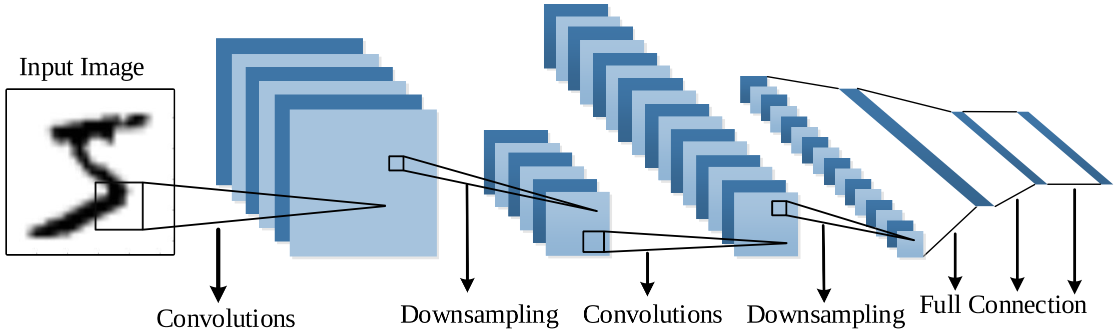
1. DL basics, linear regression, logistic regression etc.



2. Multilayer neural networks, backpropagation

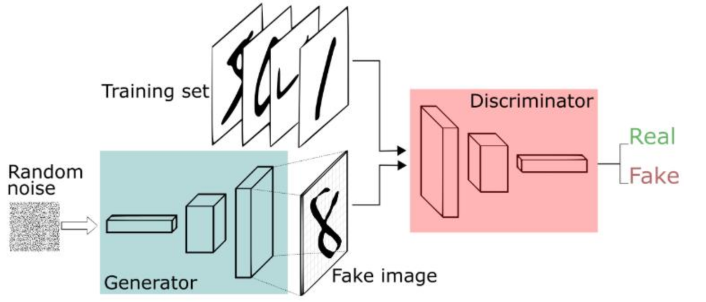


3. Convolutional Neural Networks and Applications



Next Lecture by Lam

4. Generative Adversarial Networks



5. Recurrent networks and applications