Vision and Cognitive Systems

SCQ1097939 - LM CS,DS,CYB,PD

Convolutional Neural Networks - Part 2

Prof. Lamberto Ballan

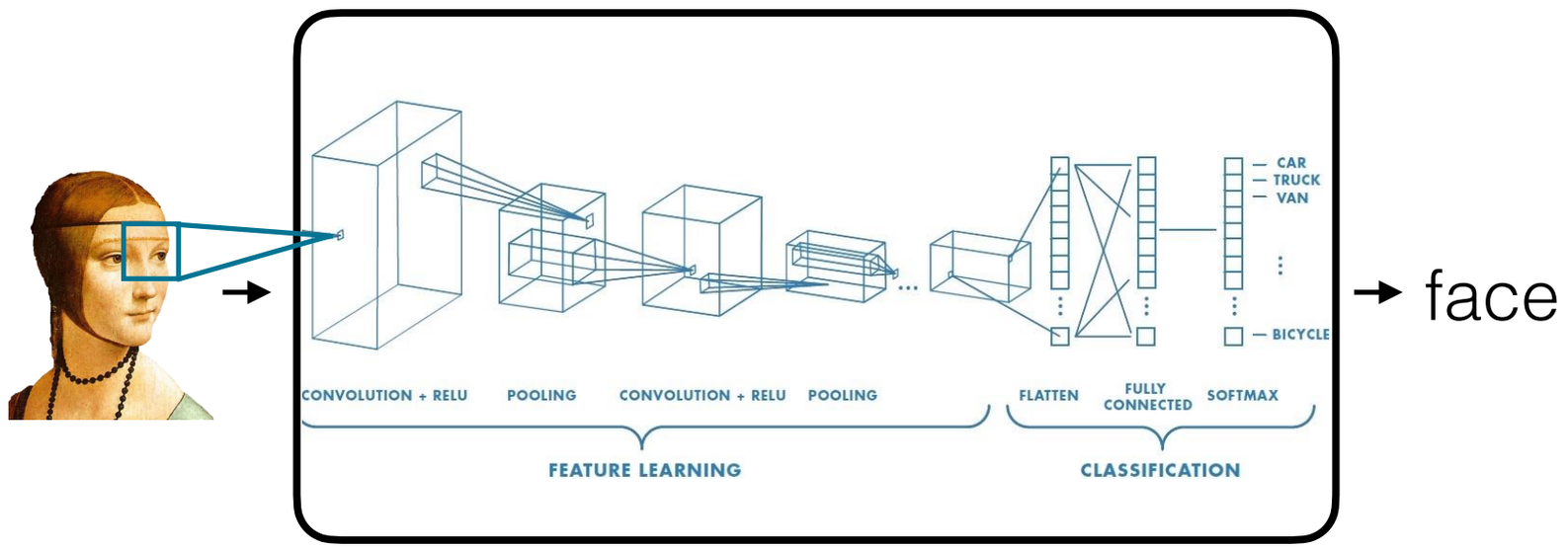
University of Padova, A.Y. 2023/24 December 4, 2023

DIPARTIMENTO MATEMATICA
Department of Mathematics “Tullio Levi-Civita”

What we learned until now

(a brief summary)

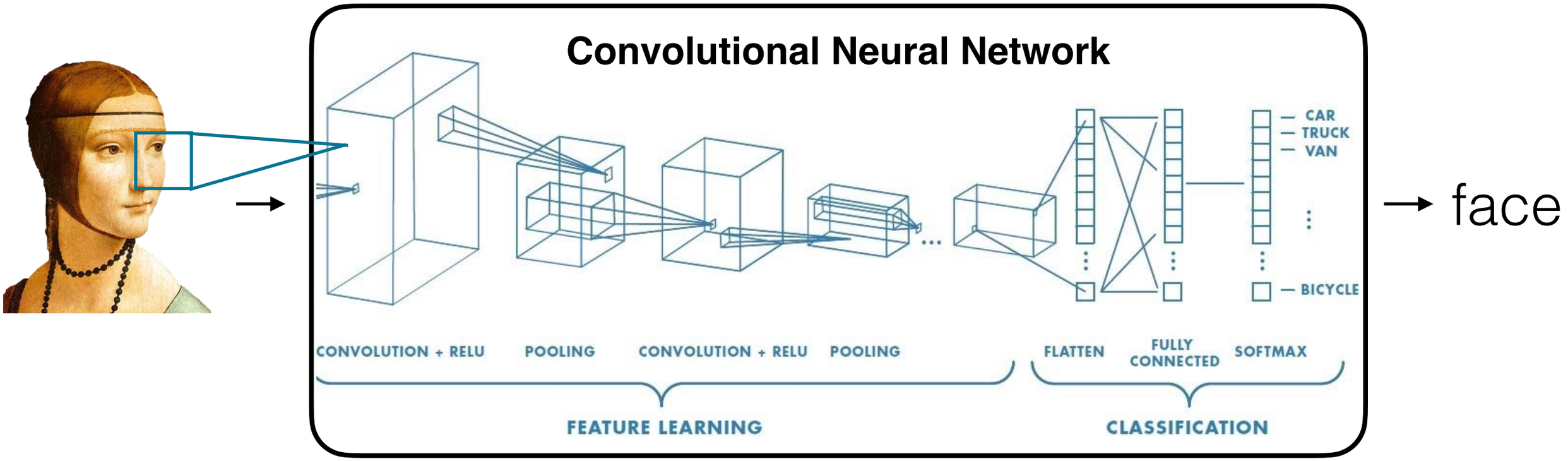
- From handcrafted features to representation learning
- Intro to Convolutional Neural Networks
- Conv, Pool and Fully Connected Layers
  - Conv layers, a closer look at spatial dimensions



Recap: representation learning

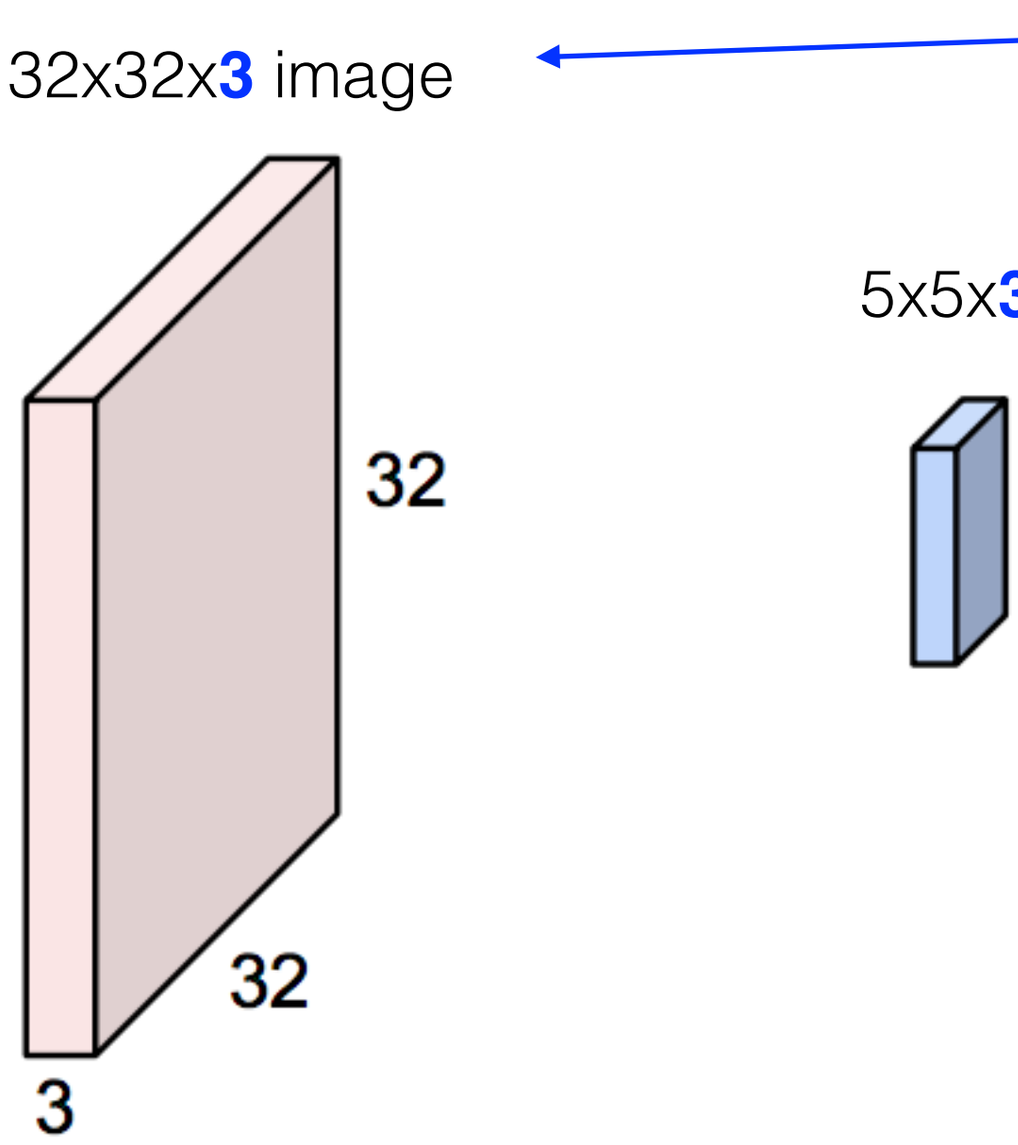
- Intuition: learn also the (image) representation directly from the data (*end-to-end learning)*

Deep Learning can be summarized as learning both the representation and the classifier out of data



Recap: convolutional layer

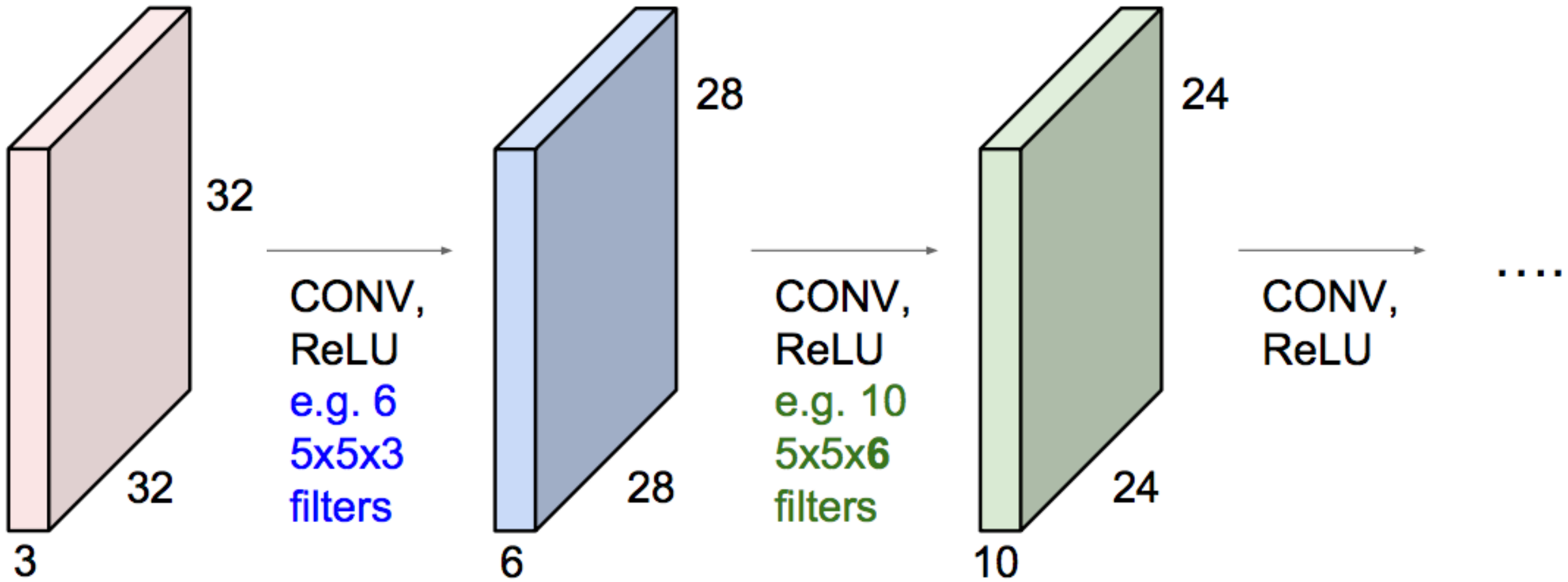
- input: 32x32x3 image -> *preserve spatial structure*



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

Recap: ConvNets (CNNs)

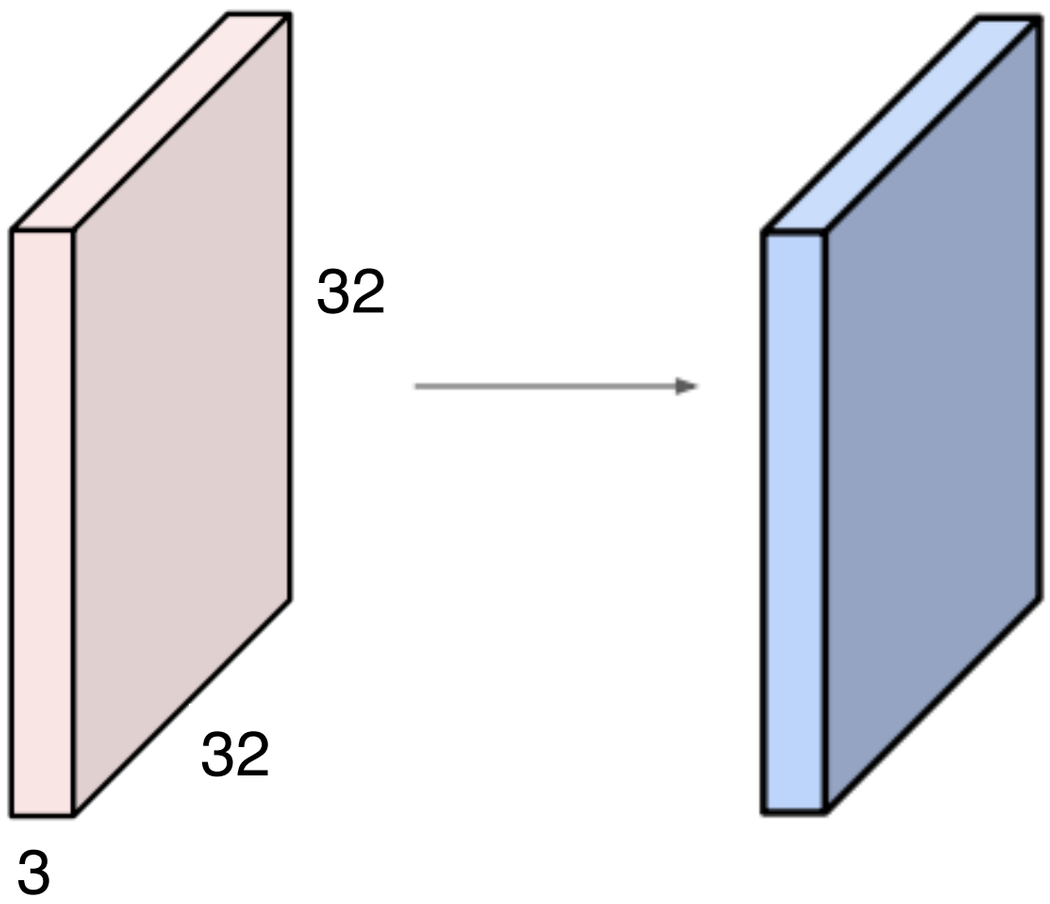
- Remember: 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially (i.e. 32, 28, 24, …)



Shrinking too fast is bad. It doesn't work well.

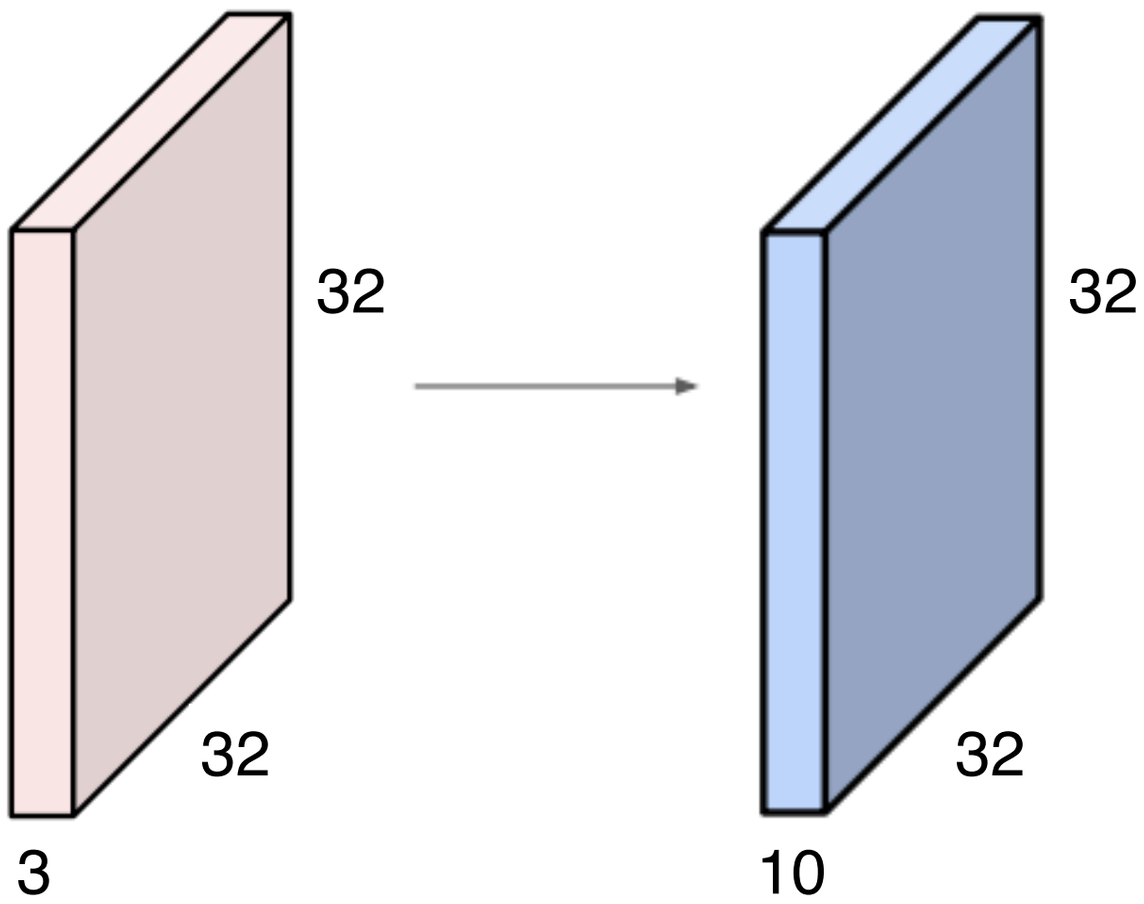
Examples

- Input volume: **32x32x3**; 10 5x5(x3) filters with stride 1, pad 2
- Output volume: **?**



Examples

- Input volume: **32x32x3**; 10 5x5(x3) filters with stride 1, pad 2
- Output volume: (32-5+2*2)/1+1 = 32 spatially => **32x32x10**



How many learnable parameters in this layer?

Each filter has: 5*5*3 + 1(bias) = 76 params
=> 76*10 = **760**

Implementations / Frameworks

- A conv layer in TensorFlow:

Convolutional Layer #1

In our first convolutional layer, we want to apply 32 5x5 filters to the input layer, with a ReLU activation function. We can use the `conv2d()` method in the `layers` module to create this layer as follows:

```
conv1 = tf.layers.conv2d(
    inputs=input_layer,
    filters=32,
    kernel_size=[5, 5],
    padding="same",
    activation=tf.nn.relu)
```

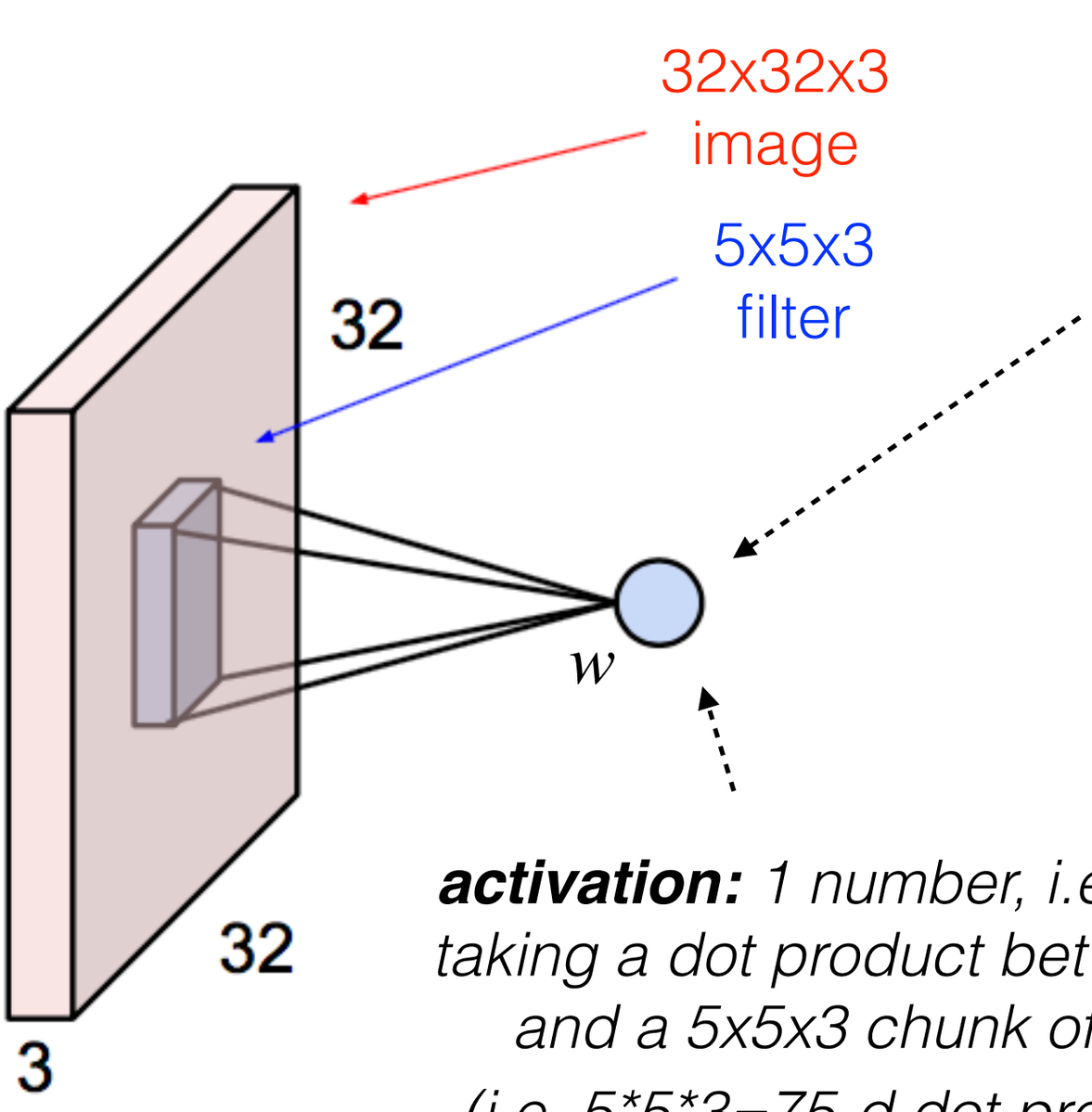
The `inputs` argument specifies our input tensor, which must have the shape `[batch_size, image_height, image_width, channels]`. Here, we're connecting our first convolutional layer to `input_layer`, which has the shape `[batch_size, 28, 28, 1]`.

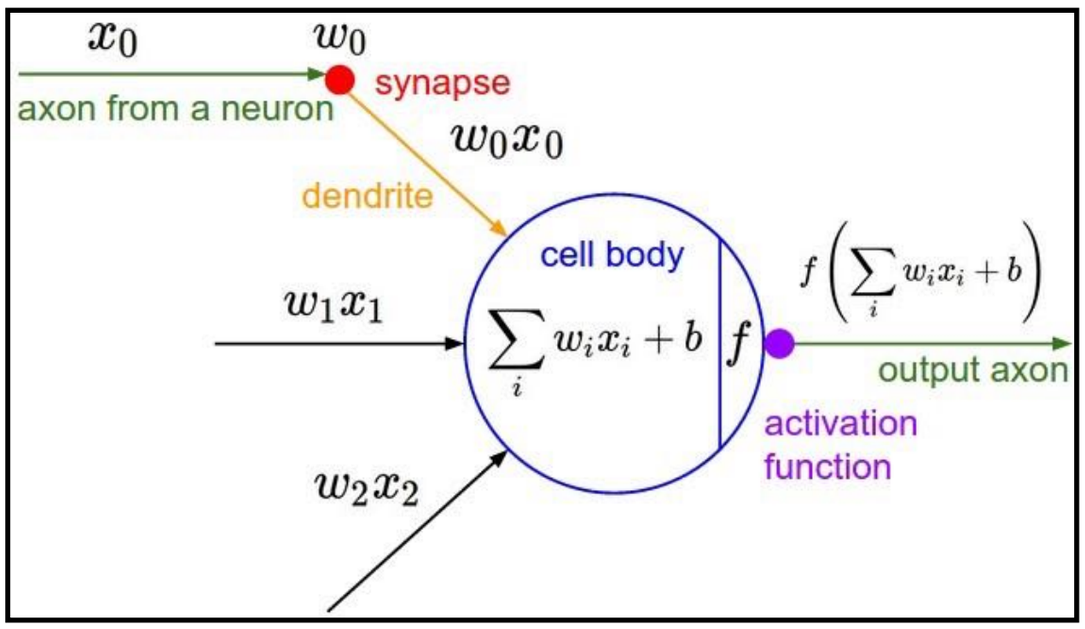
> **Note:** `conv2d()` will instead accept a shape of `[batch_size, channels, image_height, image_width]` when passed the argument `data_format=channels_first`.

The `filters` argument specifies the number of filters to apply (here, 32), and `kernel_size` specifies the dimensions of the filters as `[height, width]` (here, `[5, 5]`).

More on convolutional layers

- The brain/neuron view of conv layers:





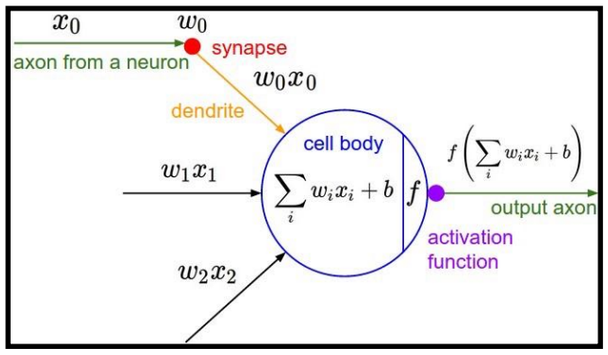
It's just a neuron with **local connectivity**!

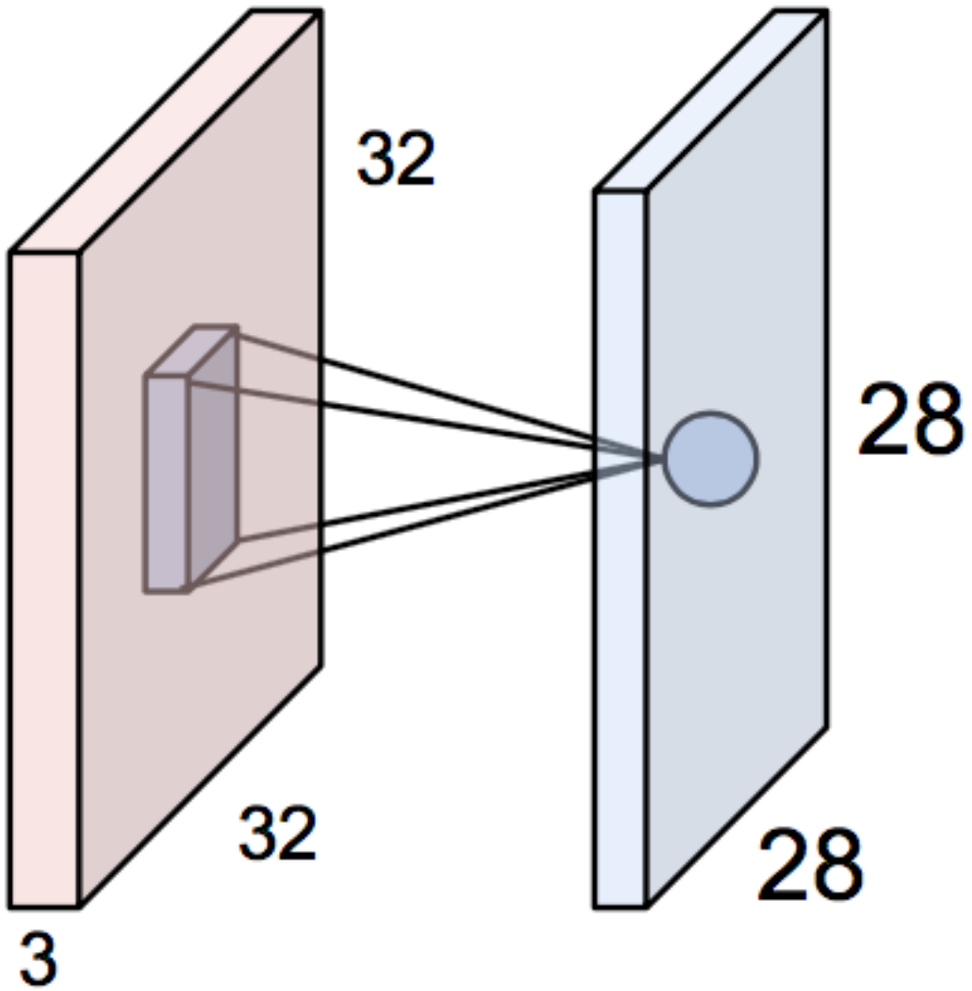
activation: *1 number, i.e. the result of taking a dot product between the filter and a 5x5x3 chunk of the image*

*(i.e. 5*5*3=75-d dot product + bias)*

More on convolutional layers

- The brain/neuron view of conv layers:





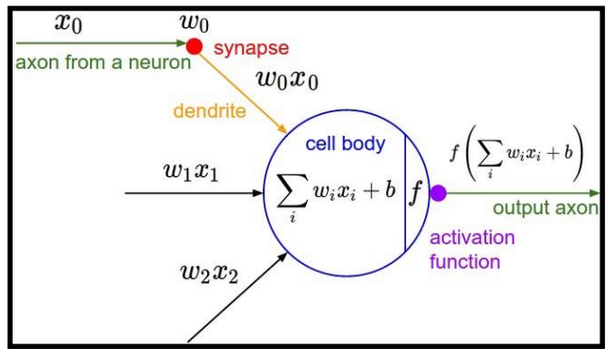
An activation map is a 28x28 *sheet* of neuron outputs:

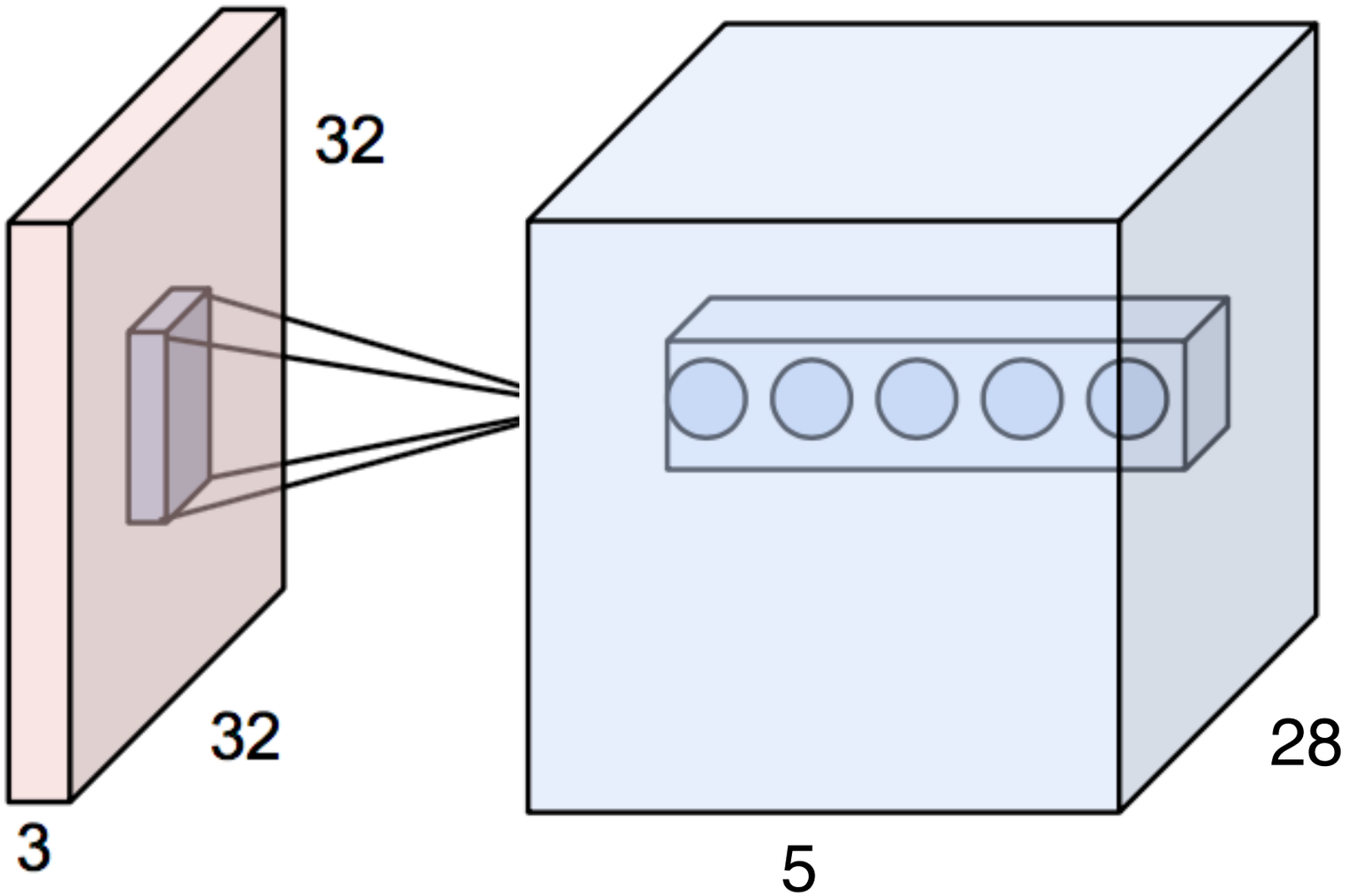
- Each neuron is connected to a small region in the input
- All of them share parameters

5x5 filter => 5x5 **receptive field**

More on convolutional layers

- The brain/neuron view of conv layers:



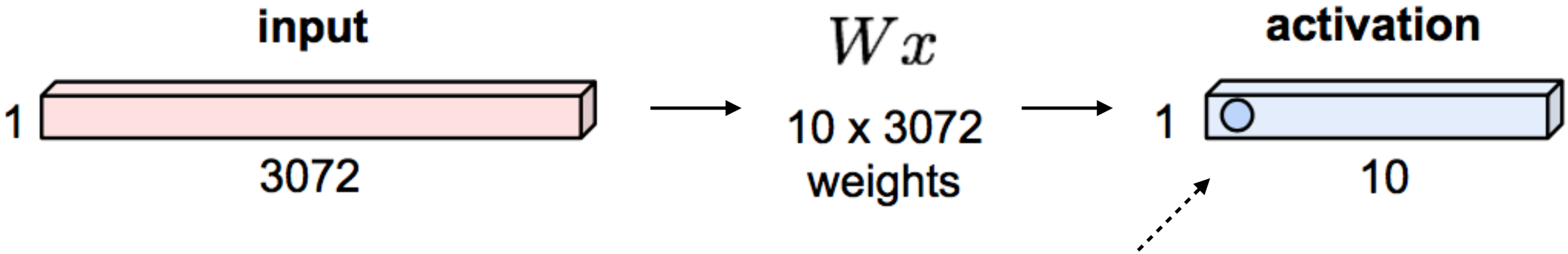


Output: neurons arranged in a 28x28x5 volume

There are 5 different neurons all looking at the **same region** in the input volume

More on convolutional layers

- Convolutional layer *vs* Fully Connected layer



activation: *1 number, i.e. the result of taking a dot product between a row of W and the input (a 3072-d dot product)*

In a Fully Connected layer each neuron looks at the **full input** volume

CNN's key property

- What's the key property (inductive bias) of CNN?

- From "Deep Learning", MIT Press, 2016 (I. Goodfellow, A. Courville, and Y. Bengio):
  - [...] the particular form of parameter sharing causes the layer to have a property called equivariance to translation
  - [...] pooling helps to make the representation become approximately invariant to small translations of the input

Convolutional Neural Networks

- Visualization of a simple ConvNet architecture:

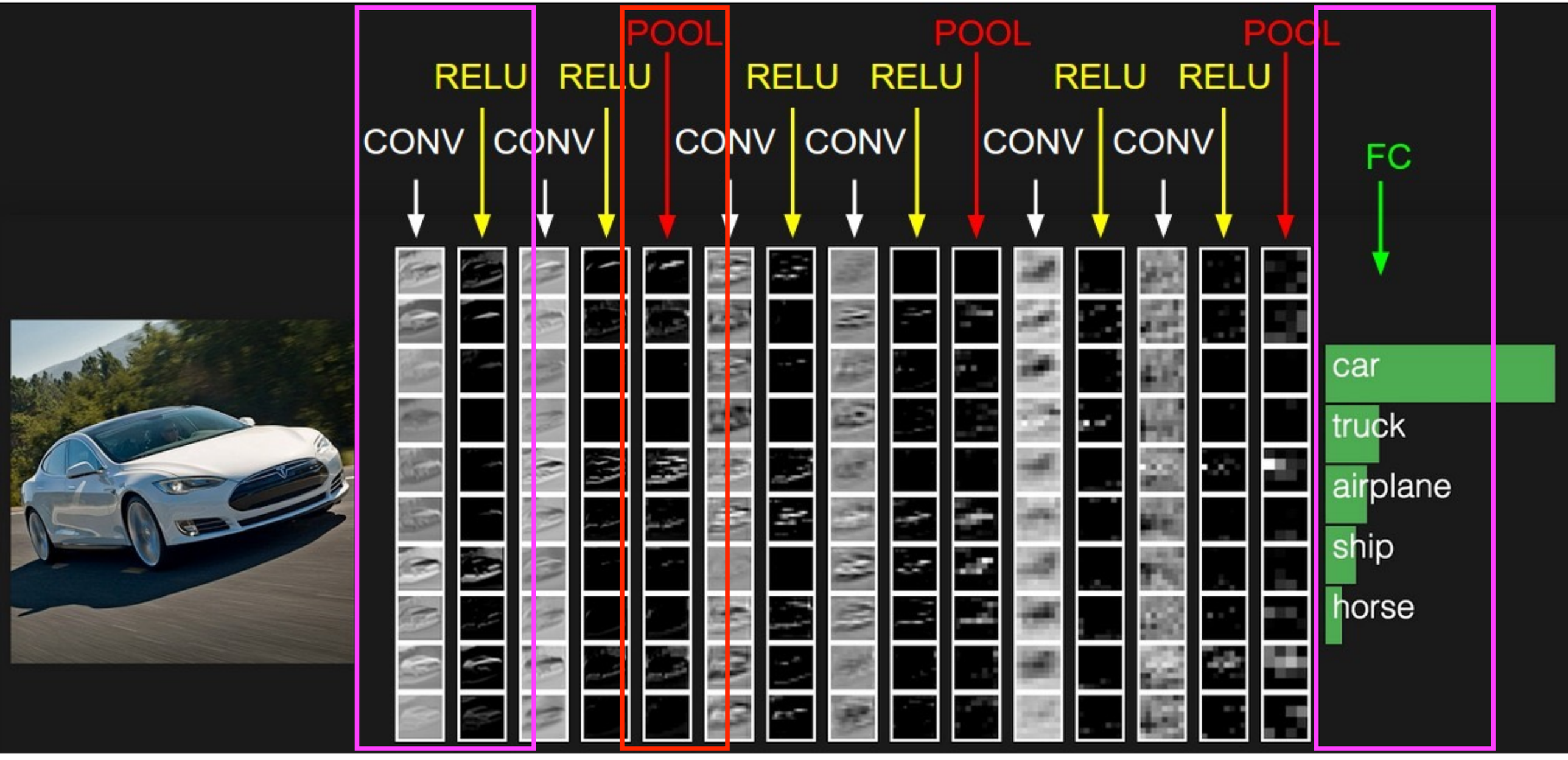
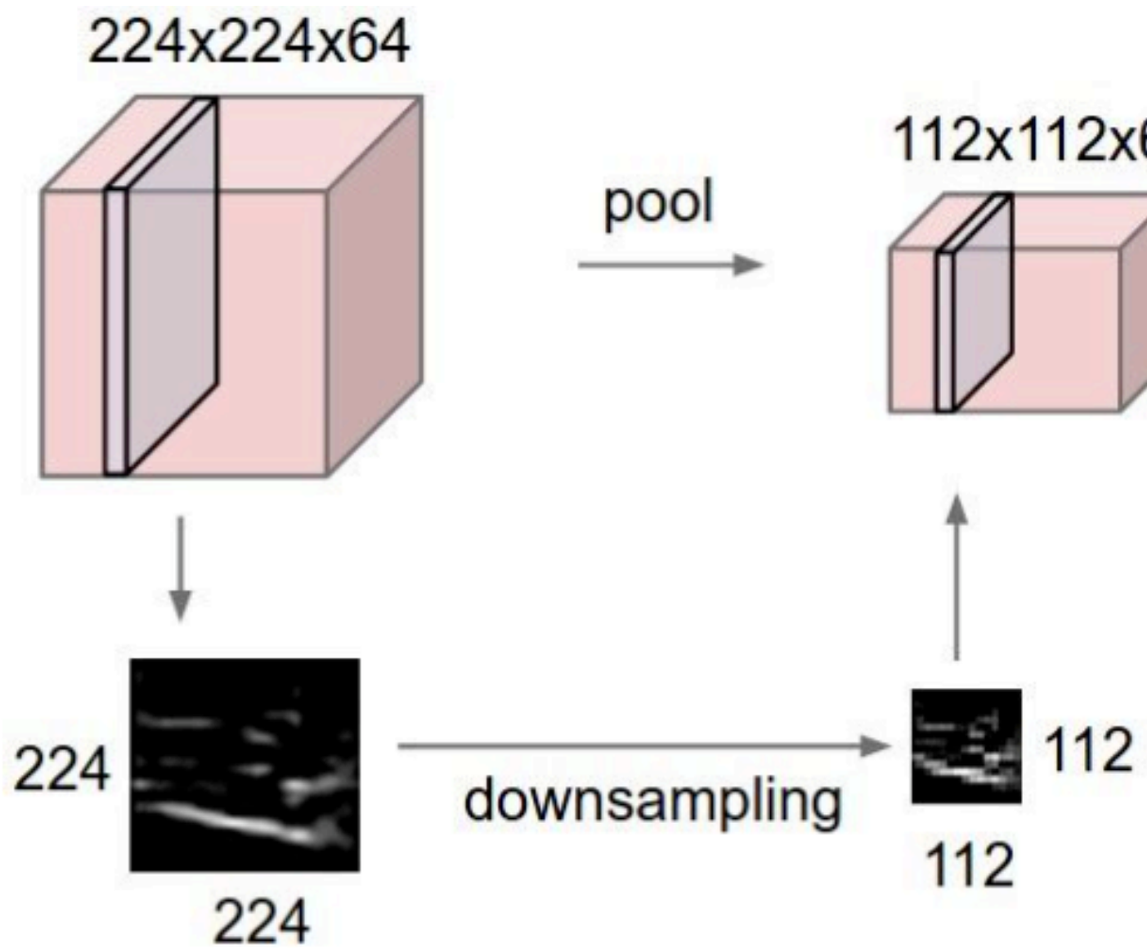


Image credit: Fei-Fei, Karpathy, Johnson

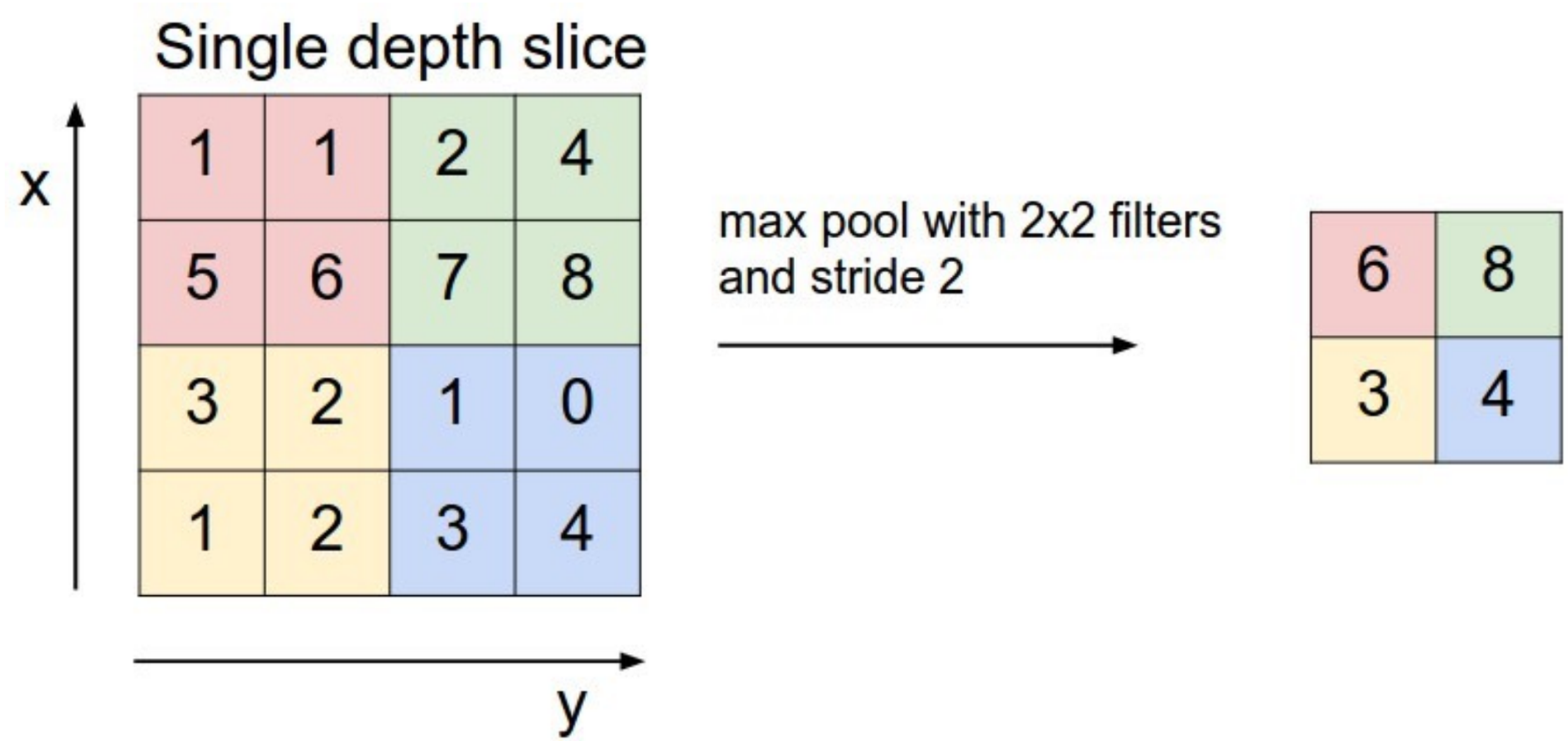
Pooling layer

- Reduces the spatial size of the representation to reduce the amount of parameters and computation
- Operates over each activation map independently



Pooling layer

- The most common strategy is **max** pooling (in general we can use other functions such as average)



The most common form is a pooling layer with filters of size F=2 applied with a stride S=2

Another option is "overlapping pooling": i.e. F=3, S=2

Summary

- From handcrafted features to representation learning
- Intro to Convolutional Neural Networks
- Conv, Pool and Fully Connected layers
  - Conv layers and a closer look at spatial dimensions

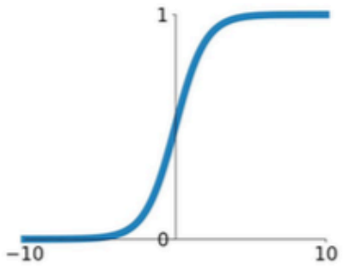
Neural Networks recap: activation functions

- **Feed-forward**: each neuron performs a dot product with the input and its weights, adds the bias and applies the activation function (or *non-linearity*)

- There are several activation functions you may encounter:

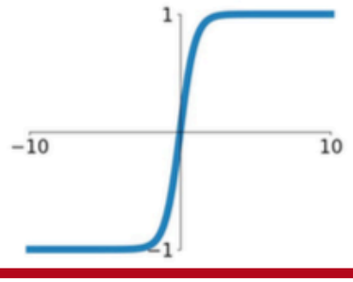
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



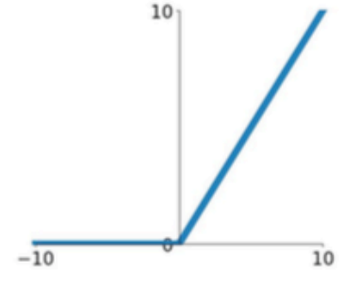
tanh

$$\tanh(x)$$



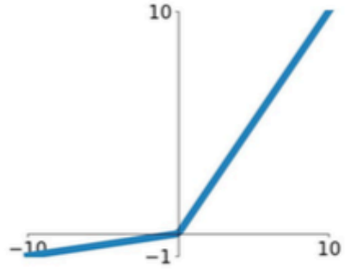
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

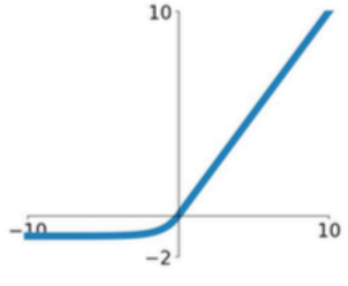


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

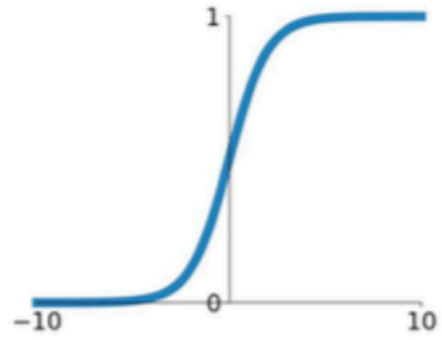


Activation functions: sigmoid

- Sigmoid (as well as tanh) is a “traditional” choice

Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



  - Squashes numbers to range [0,1]
  - Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

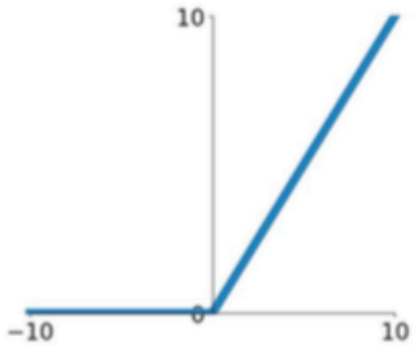
- Main problems:
  - Sigmoid saturate and kill the gradients
  - `exp()` is a bit expensive to compute
  - Sigmoid outputs are not zero-centred (fixed by tanh)

Activation functions: ReLu

- ReLU: Rectified Linear Unit

ReLU

$$\max(0, x)$$

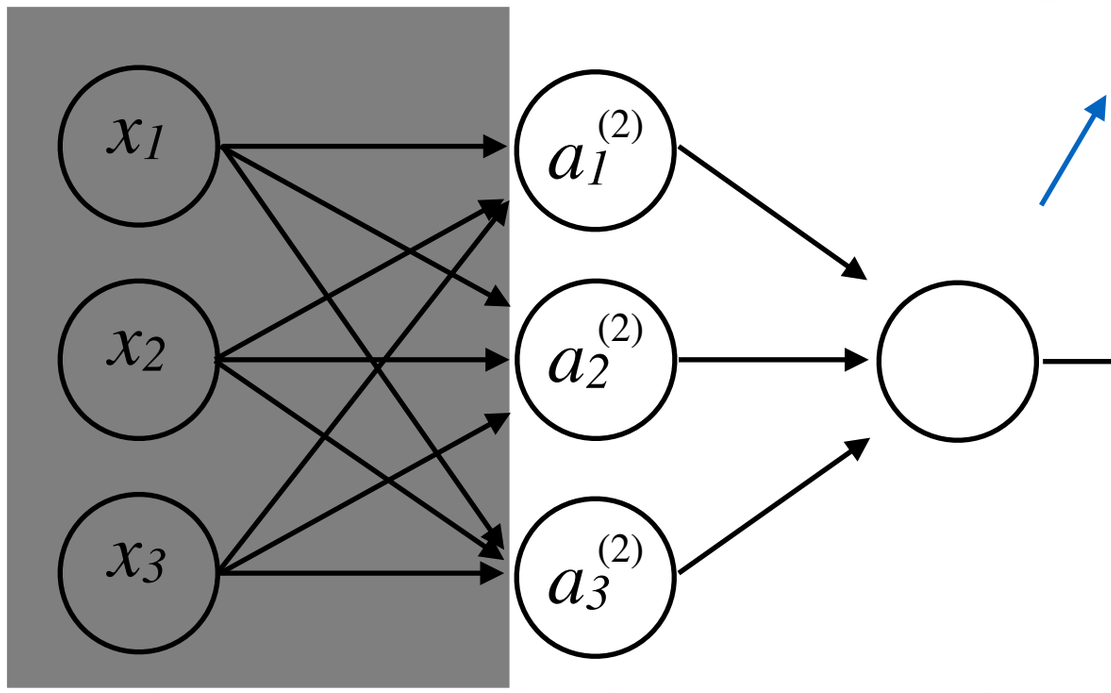


  - Does not saturate (in +region)
  - Very computationally efficient
  - Converges much faster than sigmoid/tanh (e.g. 6x)
  - Actually more biologically plausible than sigmoid

- Main problems:
  - ReLU outputs are not zero-centred (kill the gradient in -region)

Neural Networks recap: feed-forward

- This forward propagation view also help us to understand what neural networks might be doing



This is basically logistic regression…

$$f(z) = \frac{1}{1 + e^{-z}}$$

$$h_\theta(\mathbf{x}) = f(\Theta^{(2)} \boldsymbol{a}^{(2)})$$

… but now features ($\boldsymbol{a}^{(2)}$) are learned by the network

Adapted from slide by A. Ng

Neural Networks recap: feed-forward

- Gradient computation:

"forward propagation"

$$\mathbf{z}^{(2)} = \Theta^{(1)} \boldsymbol{a}^{(1)}$$

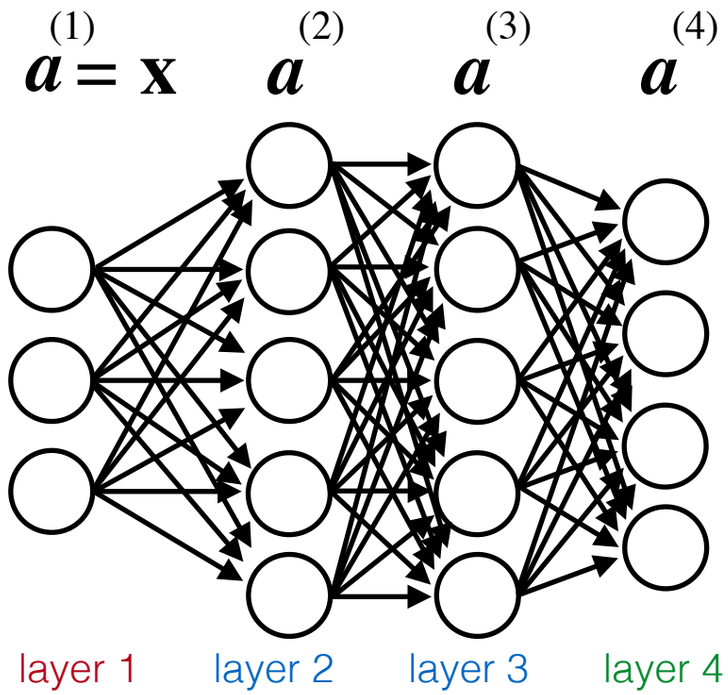
$$\boldsymbol{a}^{(2)} = f(\mathbf{z}^{(2)}) \qquad \text{(add bias: } a_0^{(2)} = 1\text{)}$$

$$\mathbf{z}^{(3)} = \Theta^{(2)} \boldsymbol{a}^{(2)}$$

$$\boldsymbol{a}^{(3)} = f(\mathbf{z}^{(3)}) \qquad \text{(add bias: } a_0^{(3)} = 1\text{)}$$

$$\mathbf{z}^{(4)} = \Theta^{(3)} \boldsymbol{a}^{(3)}$$

$$\boldsymbol{a}^{(4)} = f(\mathbf{z}^{(4)}) = h_\theta(\mathbf{x})$$



Next, in order to compute the partial derivatives, we are going to use an algorithm called "backpropagation"

Neural Networks recap: backpropagation

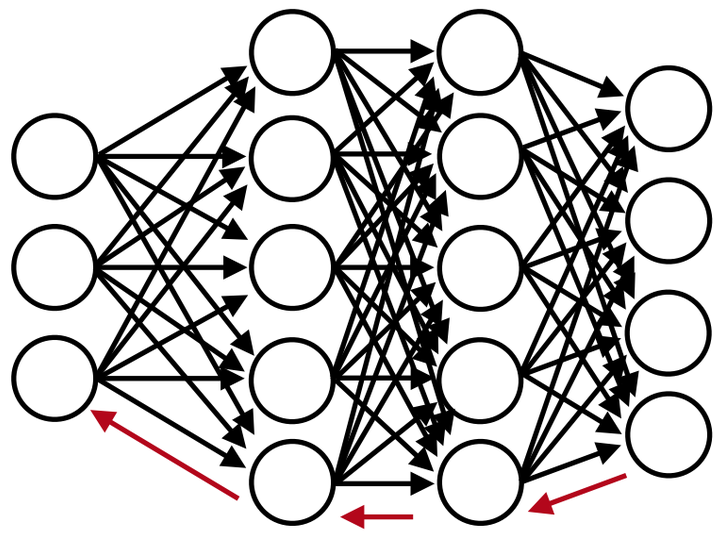
- Gradient computation:

"backpropagation"

Intuition: we shall compute $\delta_j^{(l)}$

("error" of unit j in layer l)

$a_j^{(l)}$: activation of unit j in layer l



$\delta^{(2)} \leftarrow \delta^{(3)} \leftarrow \delta^{(4)}$

Therefore, for each output unit:

(e.g. l=4 in this example)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

Then we compute the error terms for the previous layers:

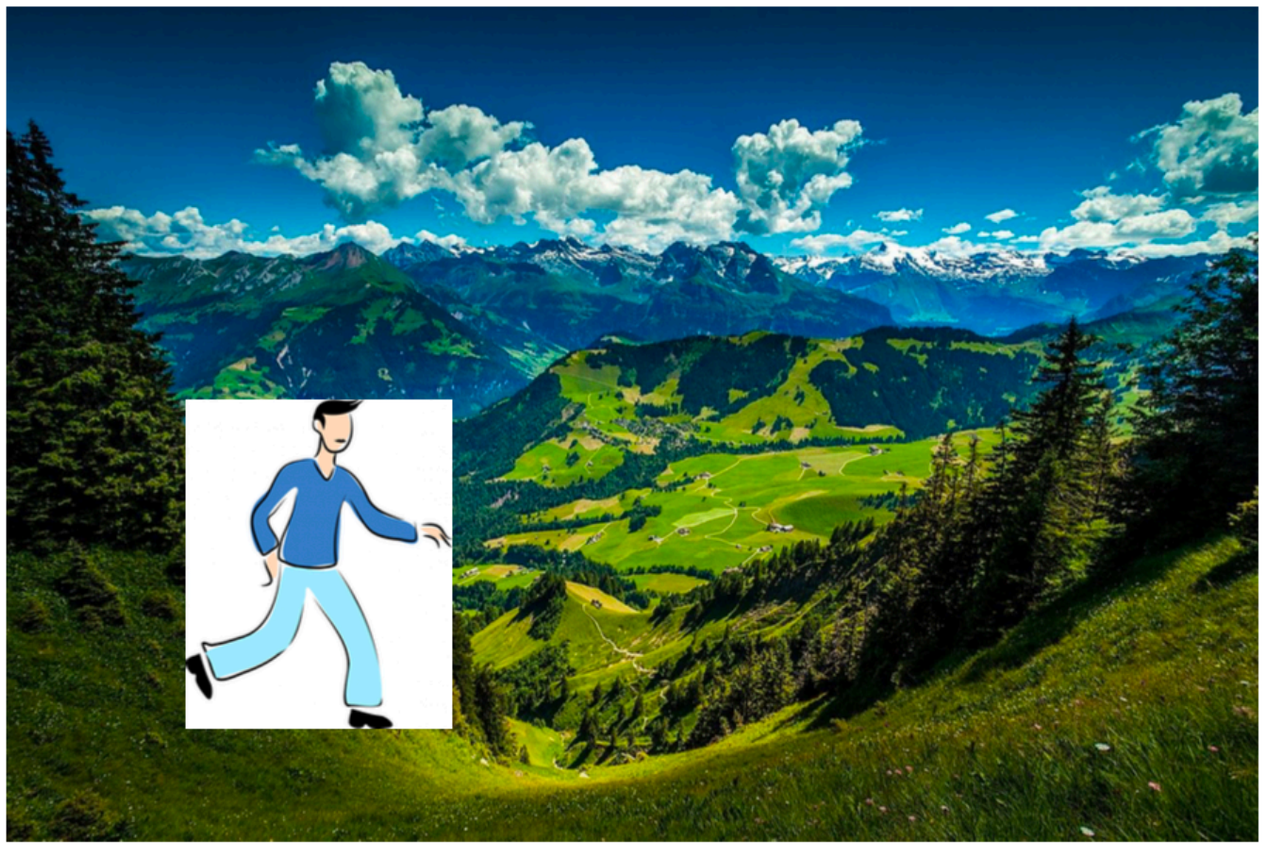
$$\delta^{(3)} = (\theta^{(3)})^T \delta^{(4)} \; .* \; f'(z^{(3)}) \longrightarrow a^{(3)} \; .* \; (1 - a^{(3)})$$

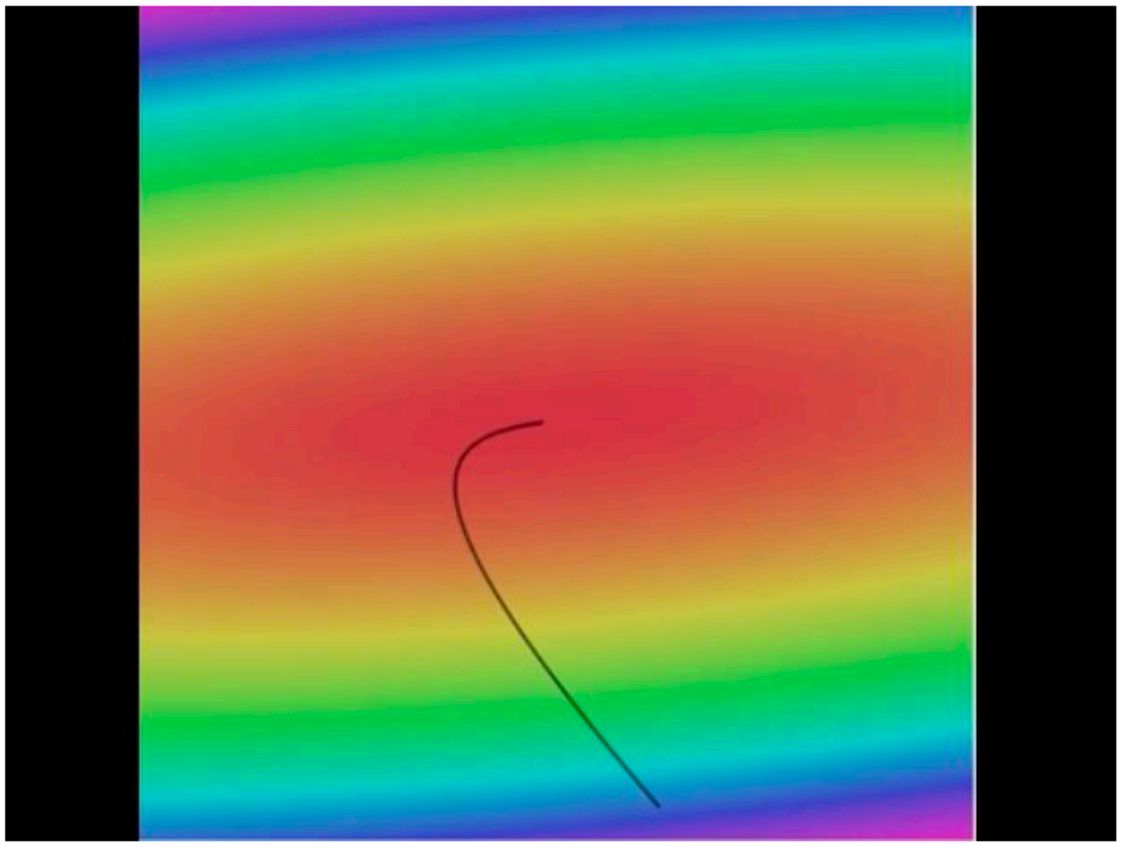
$$\delta^{(2)} = (\theta^{(2)})^T \delta^{(3)} \; .* \; f'(z^{(2)}) \quad \text{(no } \delta^{(1)}\text{)}$$

. is the element-wise multiplication*

Neural Networks recap: learning parameters

- Learning through optimization: (Stochastic) Gradient Descent





```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Training a (Conv) Neural Network

- Common strategy: mini-batch SGD
- Loop:
  - **Sample** a batch of data
  - **Forward** prop it through the graph (network), get loss
  - **Backprop** to calculate the gradients
  - **Update** the parameters using the gradient

The rise of deep learning

- ImageNet and ILSVRC results

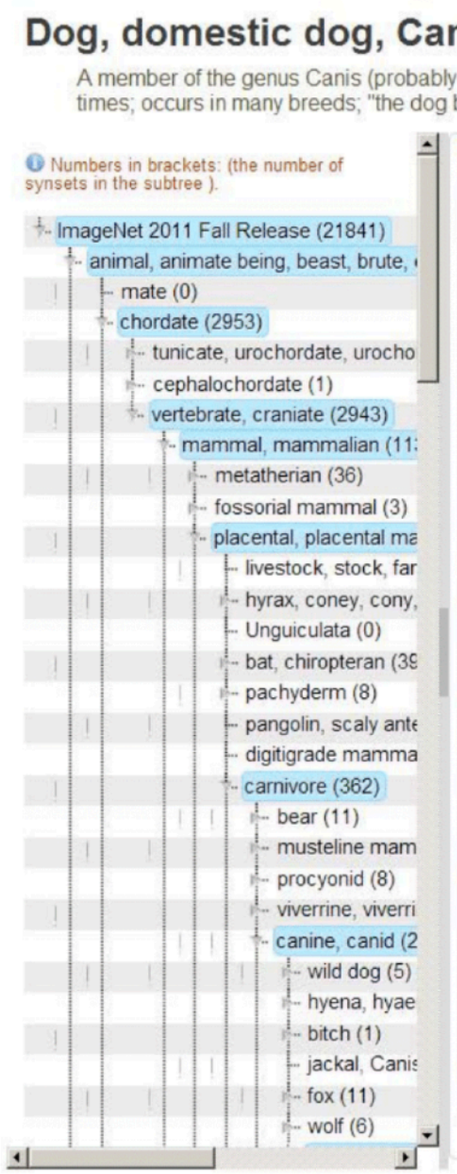
15M images, 22K classes

PRINCETON UNIVERSITY

WordNet

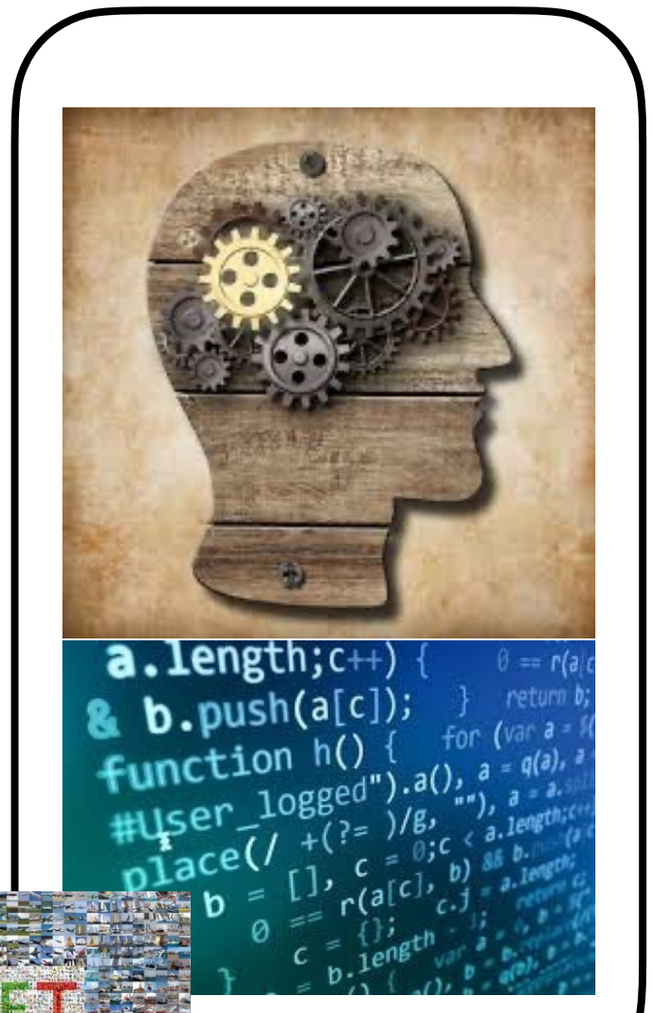
A Lexical Database for English

WordNet is a lexical database of English; nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept.

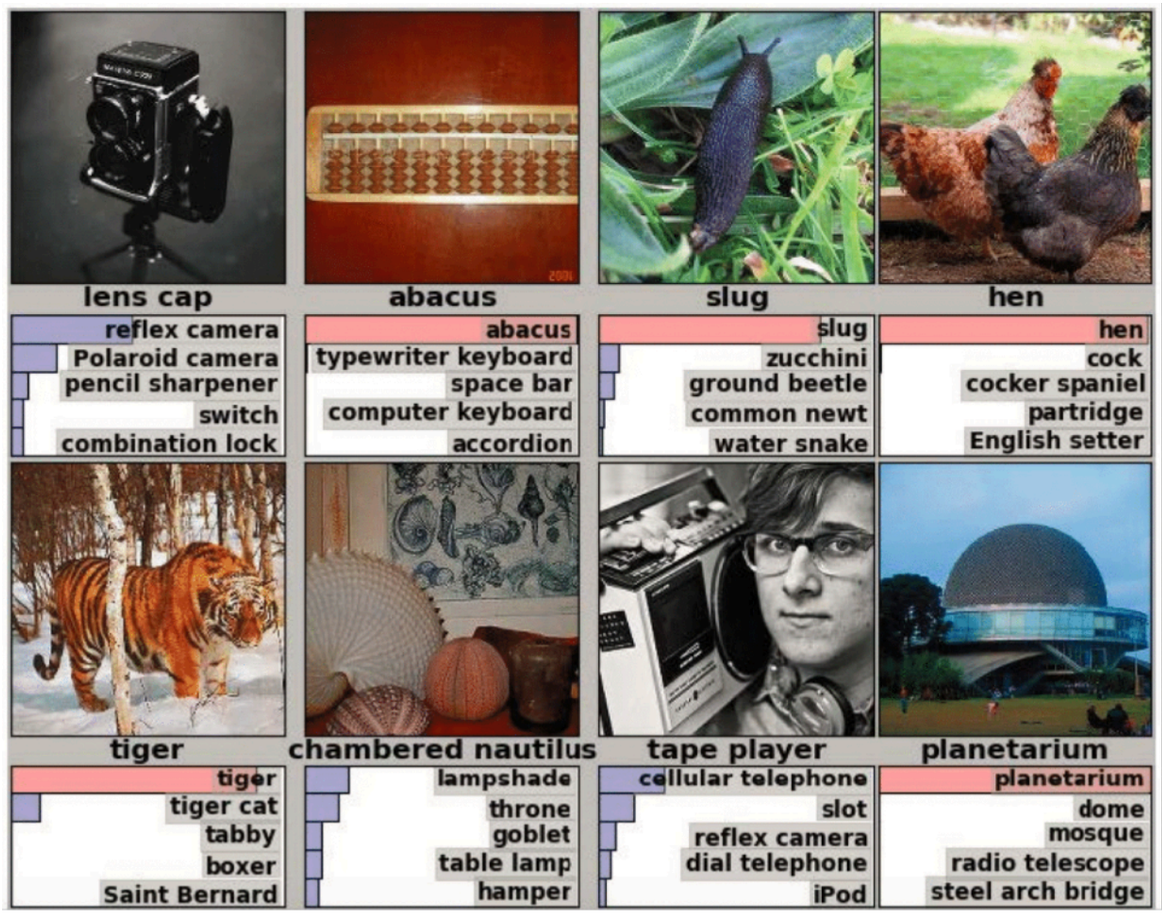


The rise of deep learning

- ImageNet and ILSVRC (classification) results



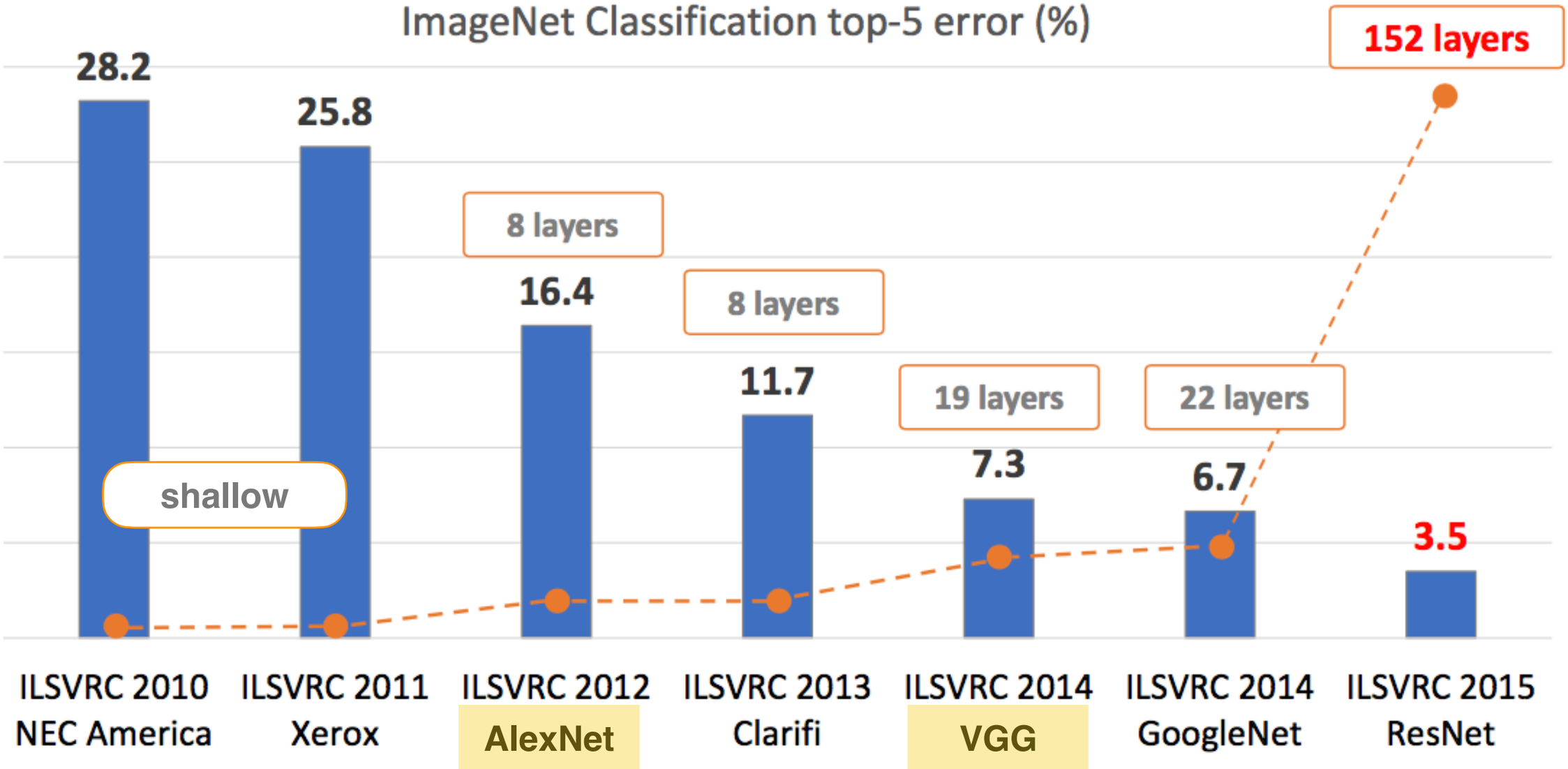
1.2M images, 1K classes



Evaluation metric: top-5 error (%)

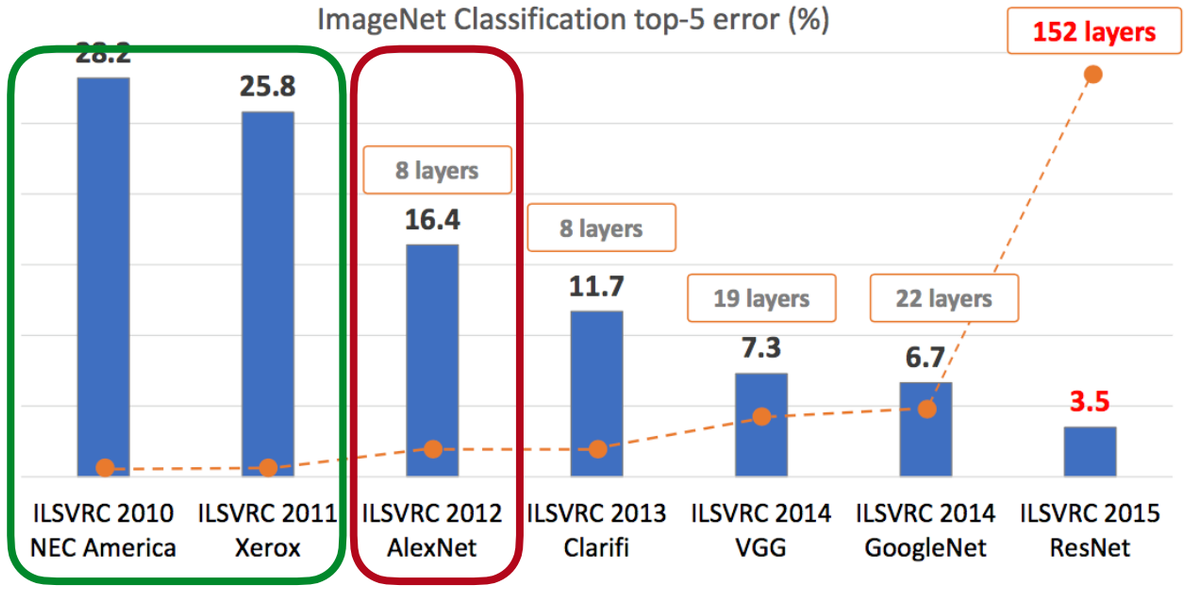
The rise of deep learning

- Revolution of depth: how deep is deep?



The “CNN story”: ILSVRC results

- ILSVRC 2012 Results:

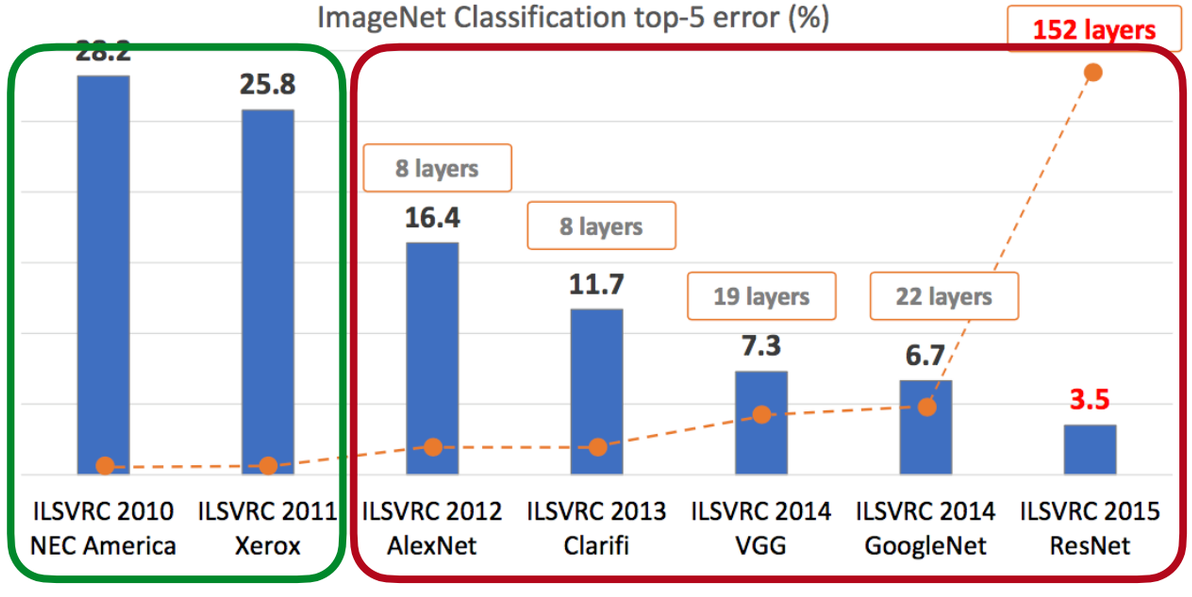


N	Error-5	Algorithm	Team	Authors
1	**0.164**	Deep Conv. Neural Network	Univ. of Toronto	Krizhevsky et al
2	0.262	Features + Fisher Vectors + Linear classifier	ISI	Gunji et al
3	0.270	Features + FV + SVM	OXFORD_VGG	Simonyan et al
4	0.271	SIFT + FV + PQ + SVM	XRCE/INRIA	Perronin et al
5	0.300	Color desc. + SVM	Univ. of Amsterdam	van de Sande et al

image-net.org/challenges/LSVRC/2012/results.html

The “CNN story”: ILSVRC results

- ILSVRC 2013 Results:

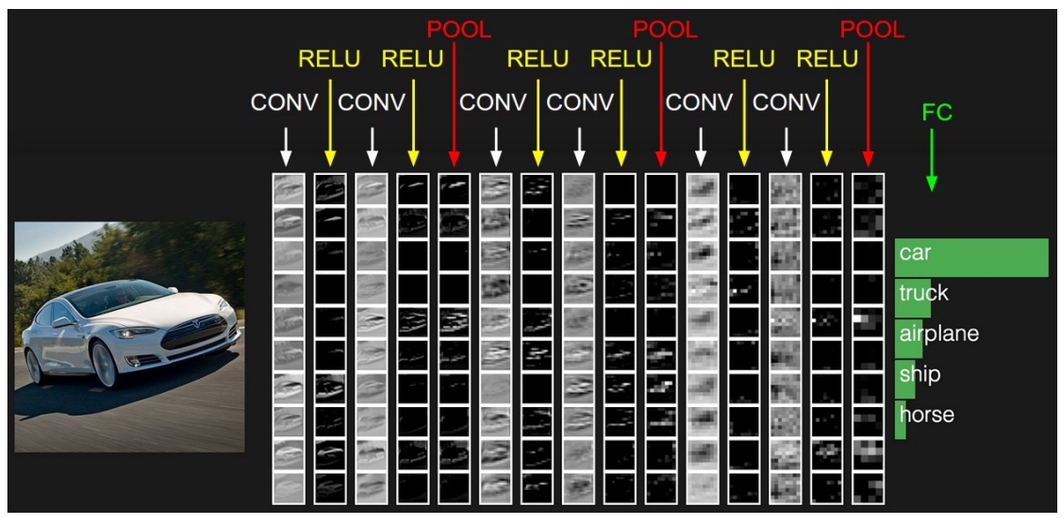


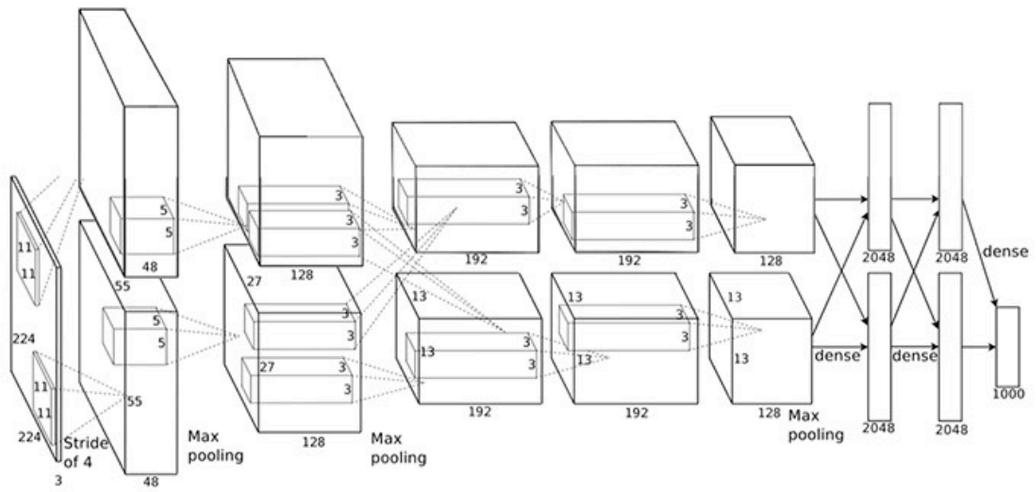
N	Error-5	Algorithm	Team	Authors
1	0.117	Deep Convolutional Neural Network	Clarifi	Zeiler
2	0.129	Deep Convolutional Neural Networks	Nat.Univ Singapore	Min LIN
3	0.135	Deep Convolutional Neural Networks	NYU	Zeiler Fergus
4	0.135	Deep Convolutional Neural Networks		Andrew Howard
5	0.137	Deep Convolutional Neural Networks	Overfeat NYU	Pierre Sermanet et al

image-net.org/challenges/LSVRC/2013/results.php

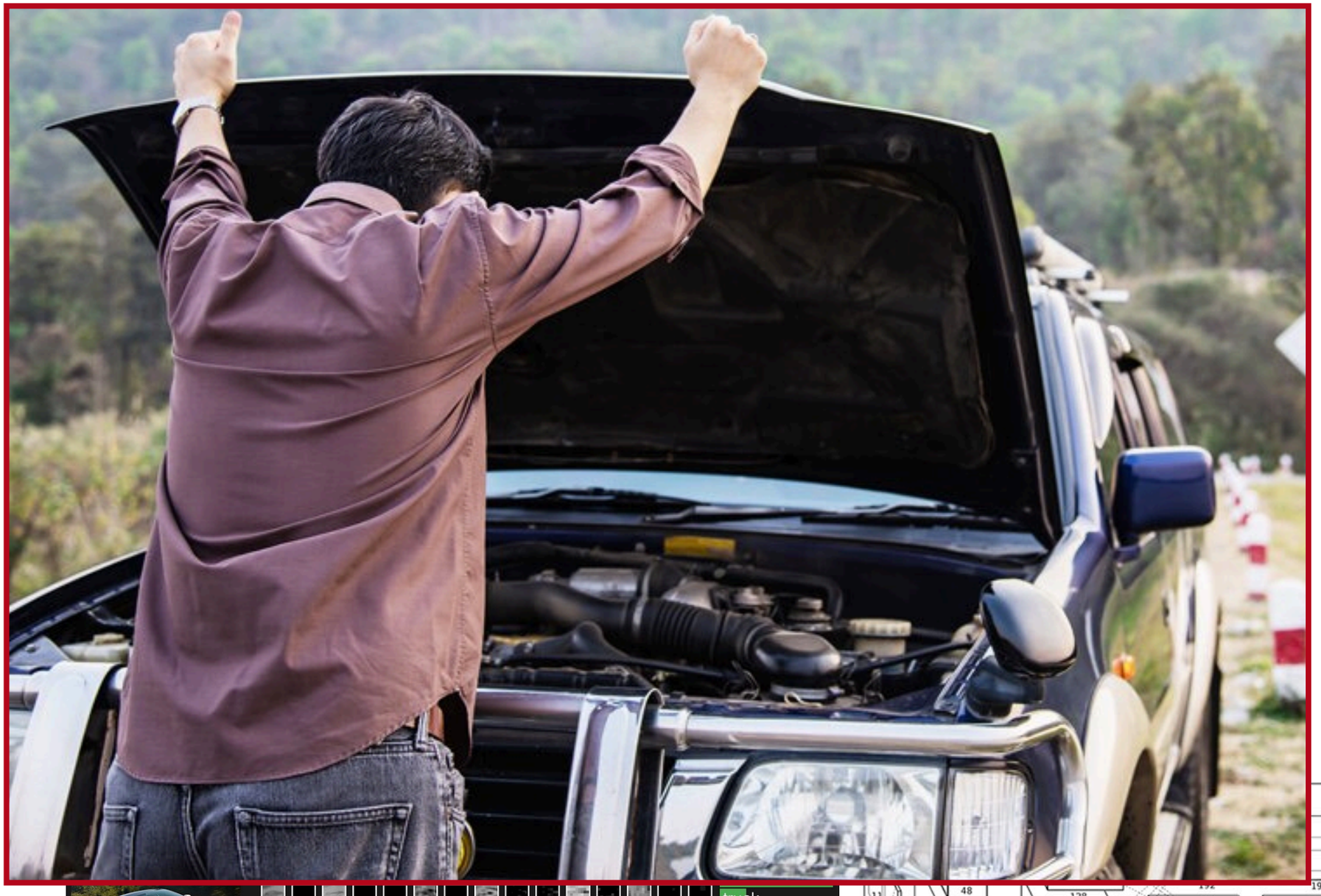
CNN Architectures: summary

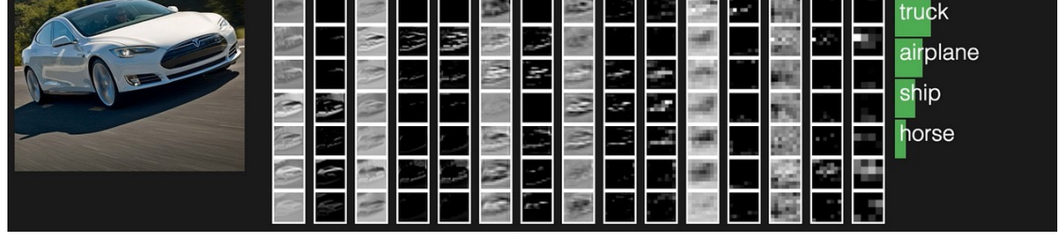
- ConvNets stack CONV, POOL, FC layers
- Typical architectures look like:
  - [(CONV,RELU)*N,POOL]*M,(FC,RELU)*K,SOFTMAX where N is usually up to ~5, M is large, 0≤K≤2
  - but recent architectures such as ResNet and GoogleNet (Inception) challenge this paradigm

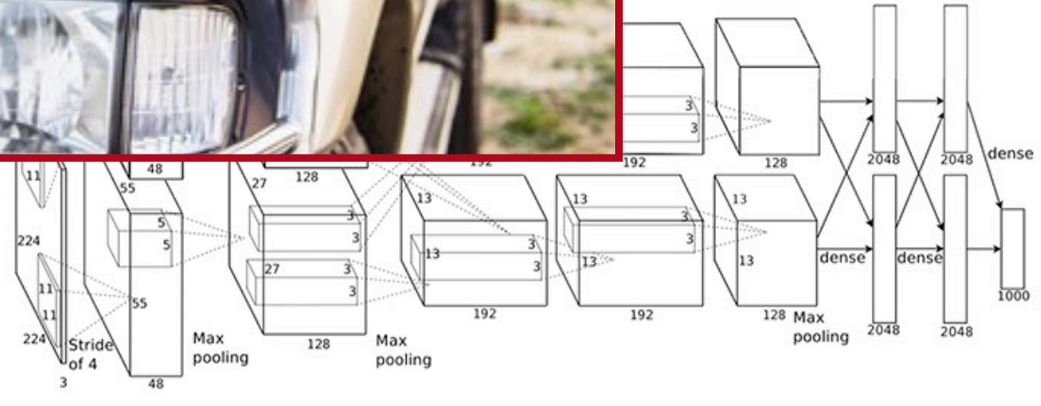




CNN Architectures: under the hood





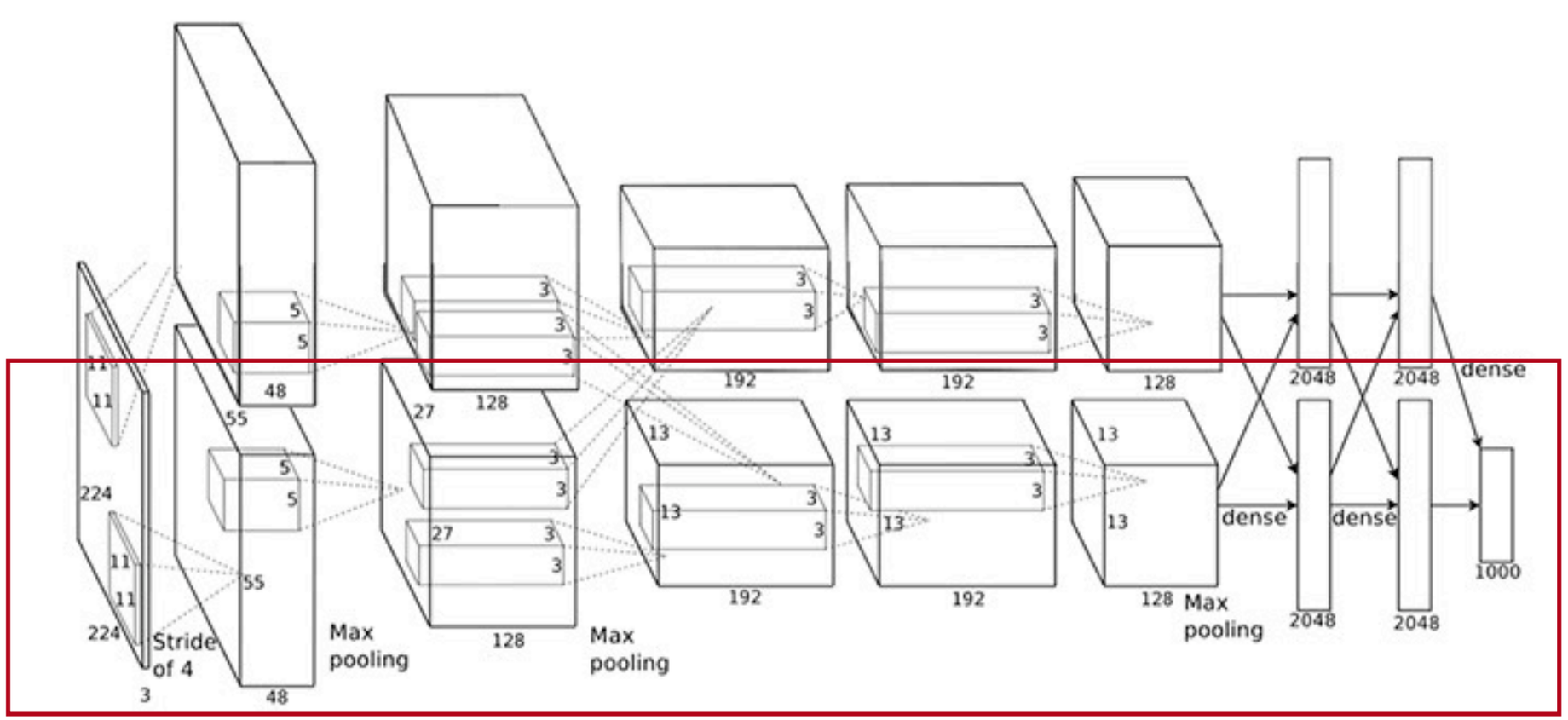


Let's dive into a few (real) case studies…

Case study: AlexNet

- Return of the CNN: first strong “modern” results

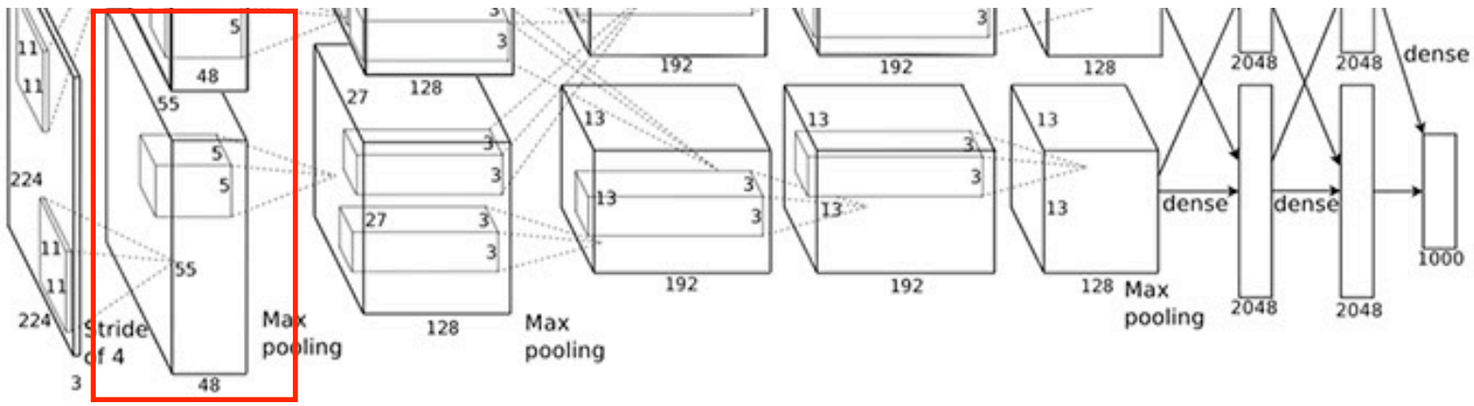
Architecture:
CONV1
MAX POOL1
NORM1
CONV2
MAX POOL2
NORM2
CONV3
CONV4
CONV5
Max POOL3
FC6
FC7
FC8



A.Krizhevsky, I.Sutskever, G.Hinton, “ImageNet Classification with Deep Convolutional Neural Networks”, NIPS 2012

Case study: AlexNet

Full (simplified) architecture:



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

[55x55x48] x 2

Historical note:

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

Slide credit: Fei-Fei, Karpathy, Johnson

Recall: conv spatial arrangement

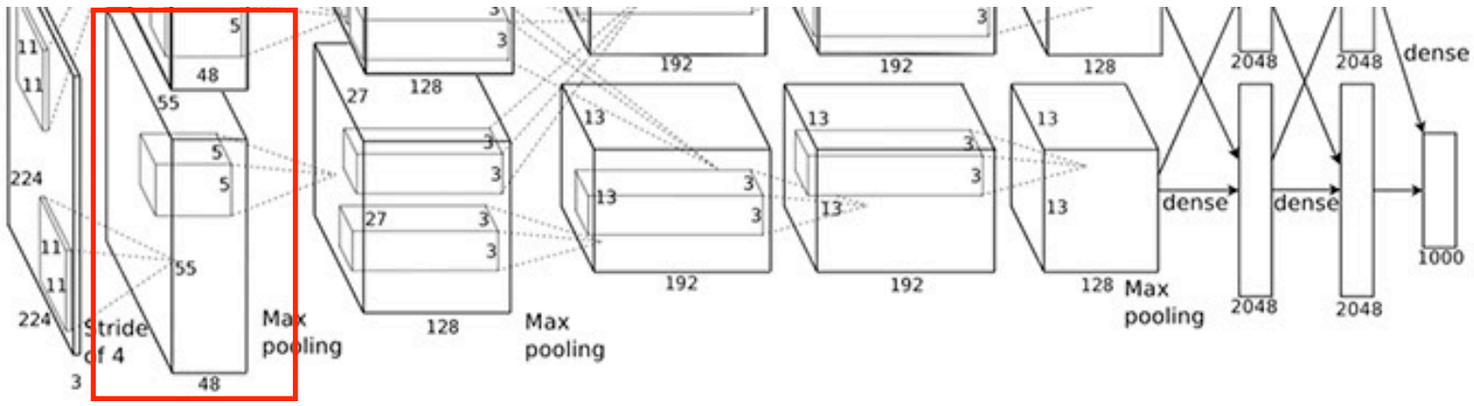
- Three hyper-parameters control the size of the output volume: *depth*, *stride* and *zero-padding*
- Output volume size: *(W-F+2*P)/S+1*
  - *W*: input volume size
  - *F*: filter size (commonly referred as to *receptive field*)
  - *S*: the stride with which filter(s) are applied
  - *P*: the amount of zero padding used on the border

A simple example (from previous class):

*7x7 input, 3x3 filter, stride 1 and pad 0 => 5x5 output, i.e. (7-3+2*0)/1+1=5*

Case study: AlexNet

Full (simplified) 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 layer: what is the output volume size?

A: (227-11)/4+1=55
i.e. [55x55x96]

What is the number of parameters at this layer?

*A: (11*11*3)*96=35K*

Case study: VGGNet

- Its main contribution was in showing that the depth of the network is a critical component
- A downside is that it is expensive to evaluate and uses a lot more memory and parameters (140M)
- Lets break down the VGGNet in detail:
  - It is composed of CONV layers that perform 3x3 convolutions with stride 1 and pad 1
  - POOL layers that perform 2x2 max pooling with stride 2 (no padding)

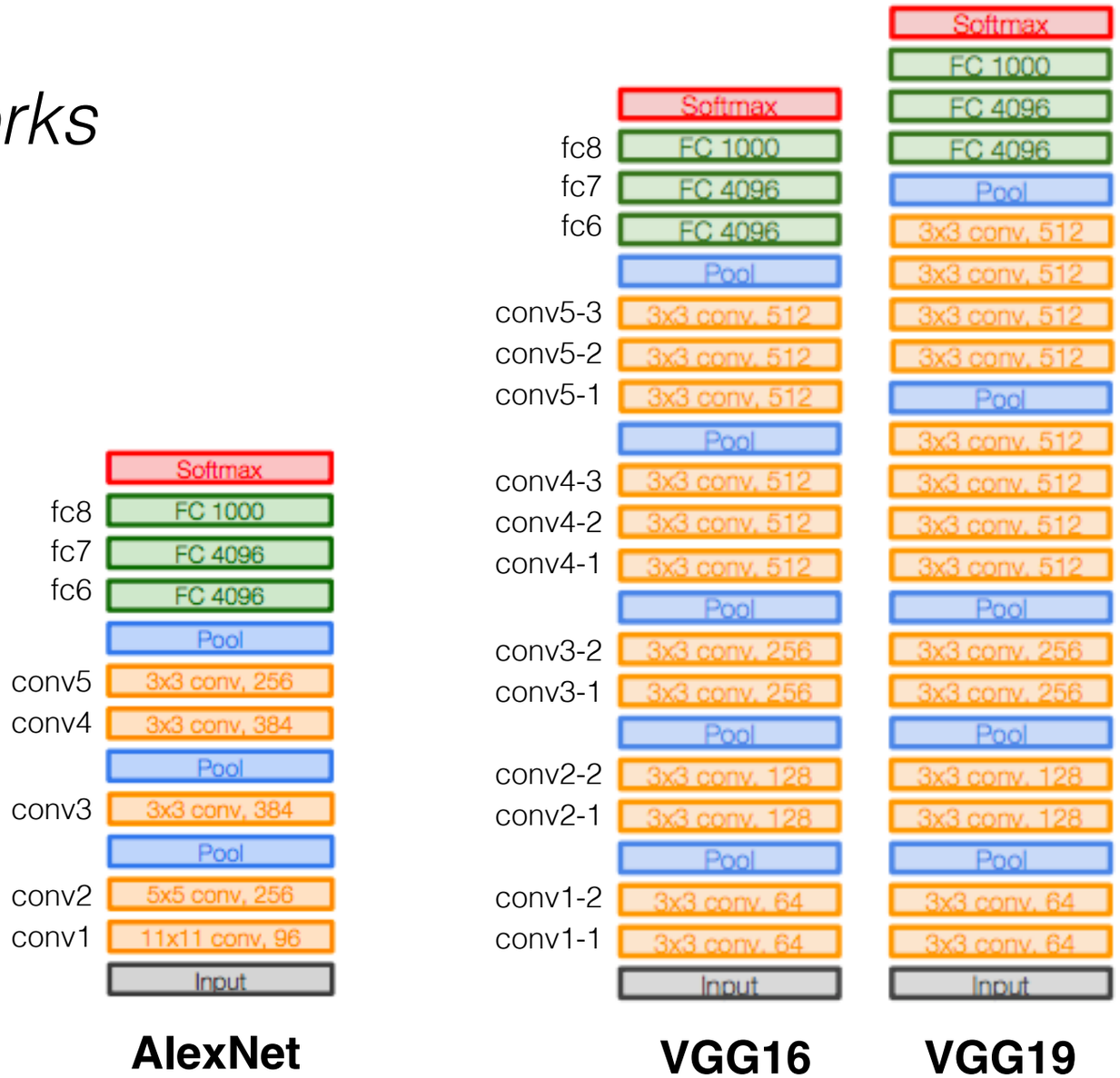
K.Simonyan, A.Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition”, ICLR 2015

URL: http://www.robots.ox.ac.uk/~vgg/research/very_deep/

Case study: VGGNet

VGGNet *vs* AlexNet:

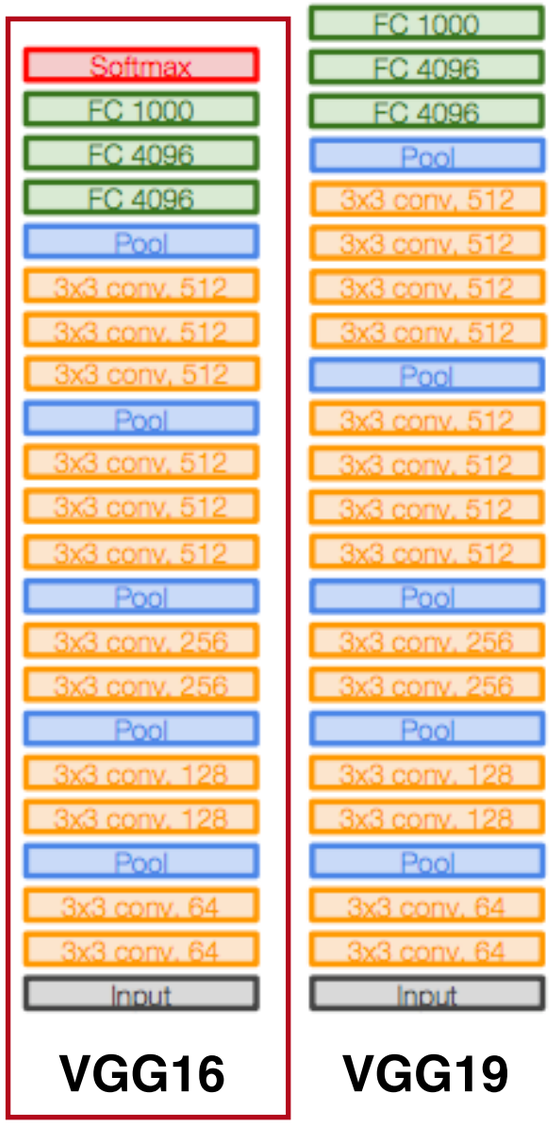
Small filters, Deeper networks



Case study: VGGNet

(not counting biases)

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



Case study: VGGNet

(not counting biases)

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 parameters are in late FC

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

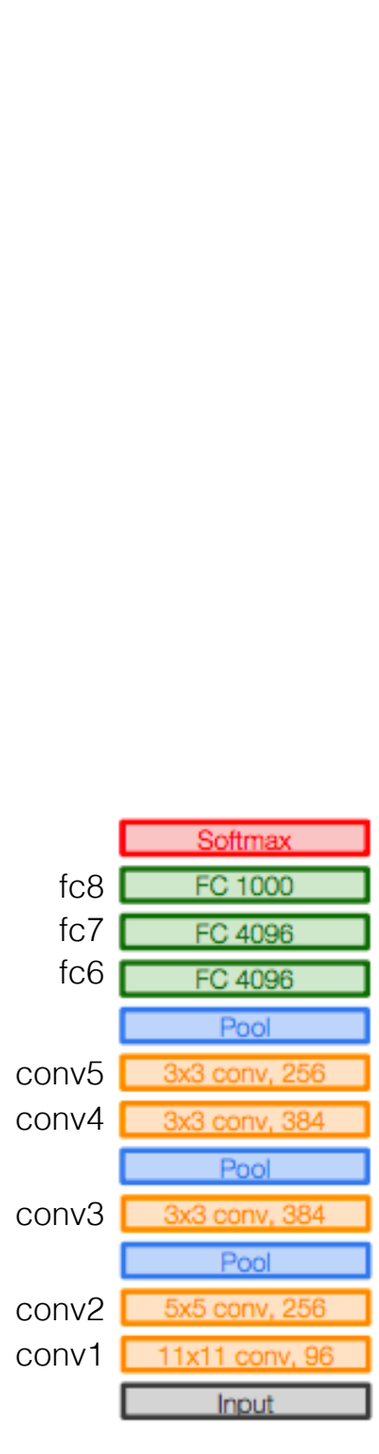
Case study: GoogLeNet (Inception)

Deeper networks, with <u>computational efficiency</u>

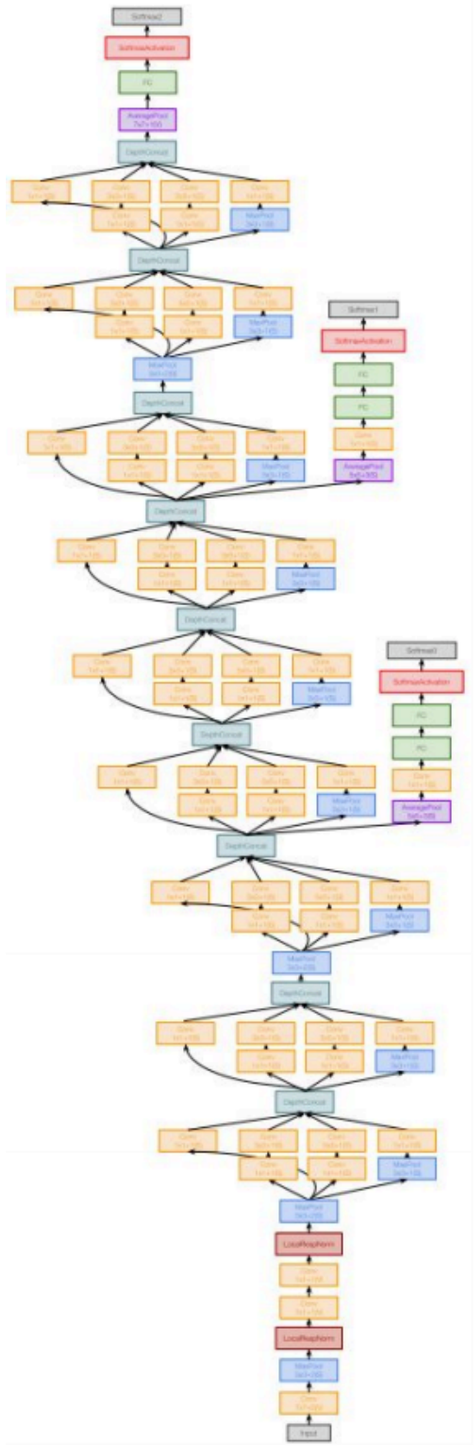
22 layers with no FC layers

Efficient "Inception" module

5M params (12x less AlexNet)



AlexNet



GoogLeNet

Case study: GoogLeNet (Inception)

Stem network

At the start aggressively downsamples input
(recall: in VGG most of the compute was at the start)

Layer	Input size C	H / W	Layer filters	kernel	stride	pad	Output size C	H/W	memory (KB)	params (K)	flop (M)
conv	3	224	64	7	2	3	64	112	3136	9	118
max-pool	64	112		3	2	1	64	56	784	0	2
conv	64	56	64	1	1	0	64	56	784	4	13
conv	64	56	192	3	1	1	192	56	2352	111	347
max-pool	192	56		3	2	1	192	28	588	0	1

Total from 224 to 28 spatial resolution

Memory: 7.5 MB

Parameters: 124K

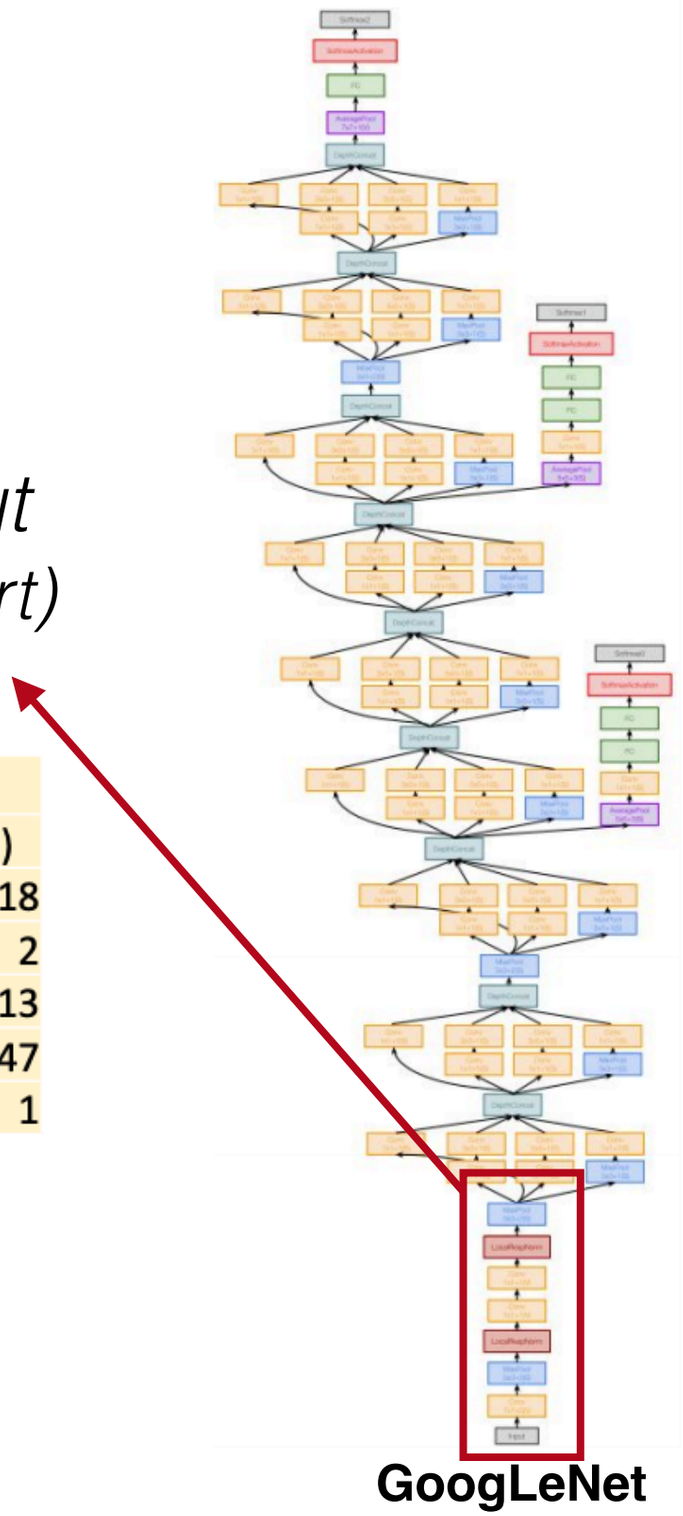
MFLOP: 418

Compare VGG-16:

Memory: 42.9 MB (5.7x)

Parameters: 1.1M (8.9x)

MFLOP: 7485 (17.8x)

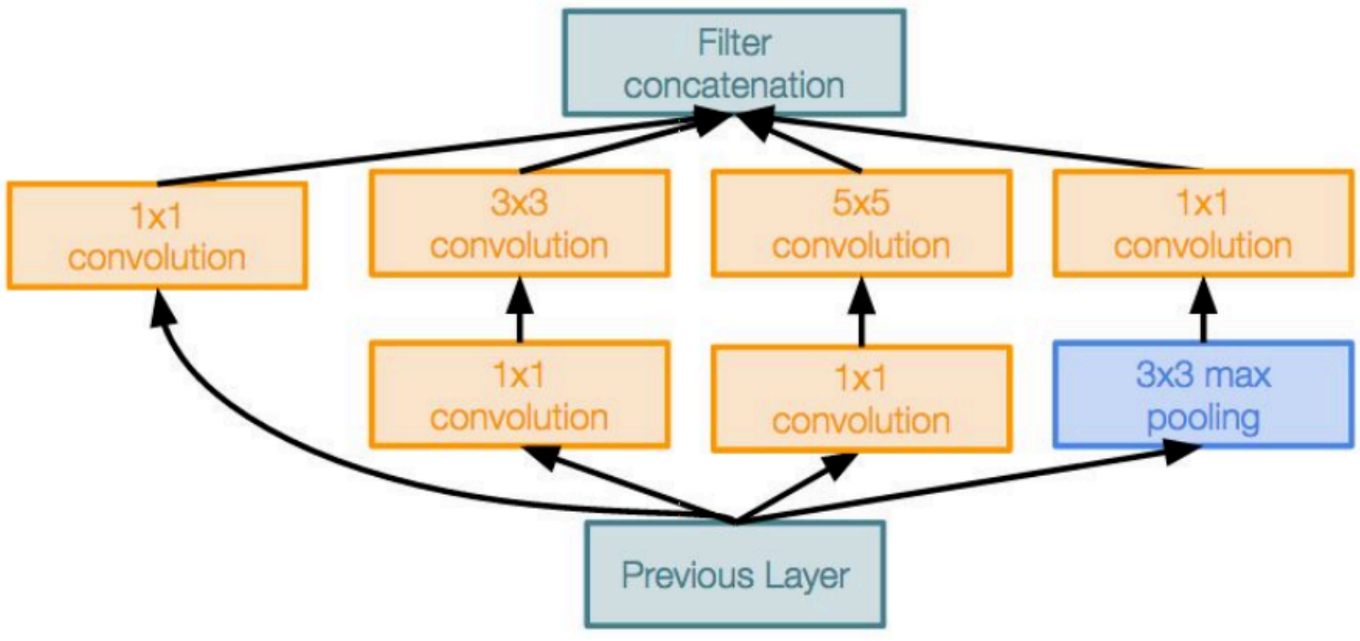


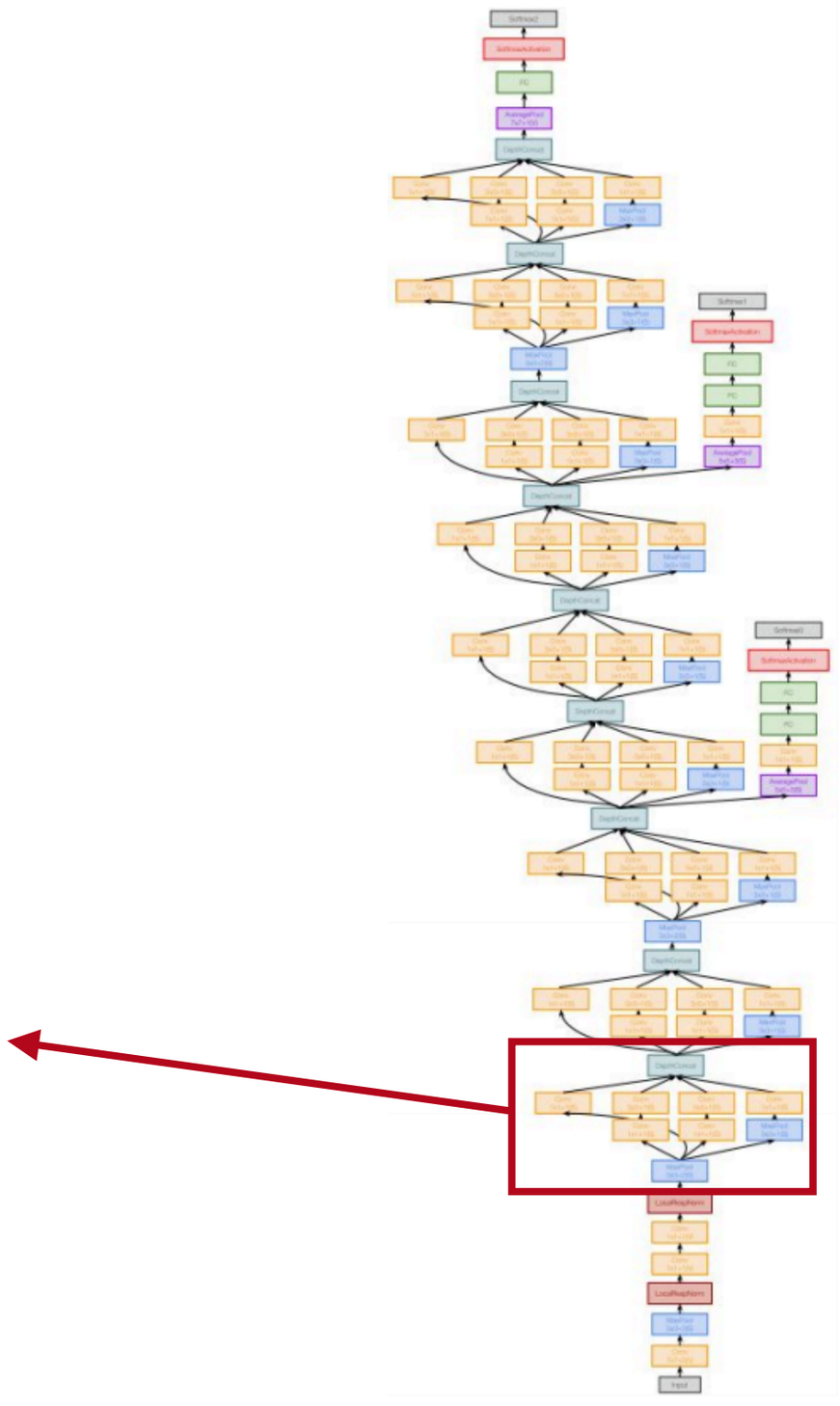
GoogLeNet

Case study: GoogLeNet (Inception)

Inception module

The key idea is to design a good local topology (network within a network) and then stack these modules on top of each other



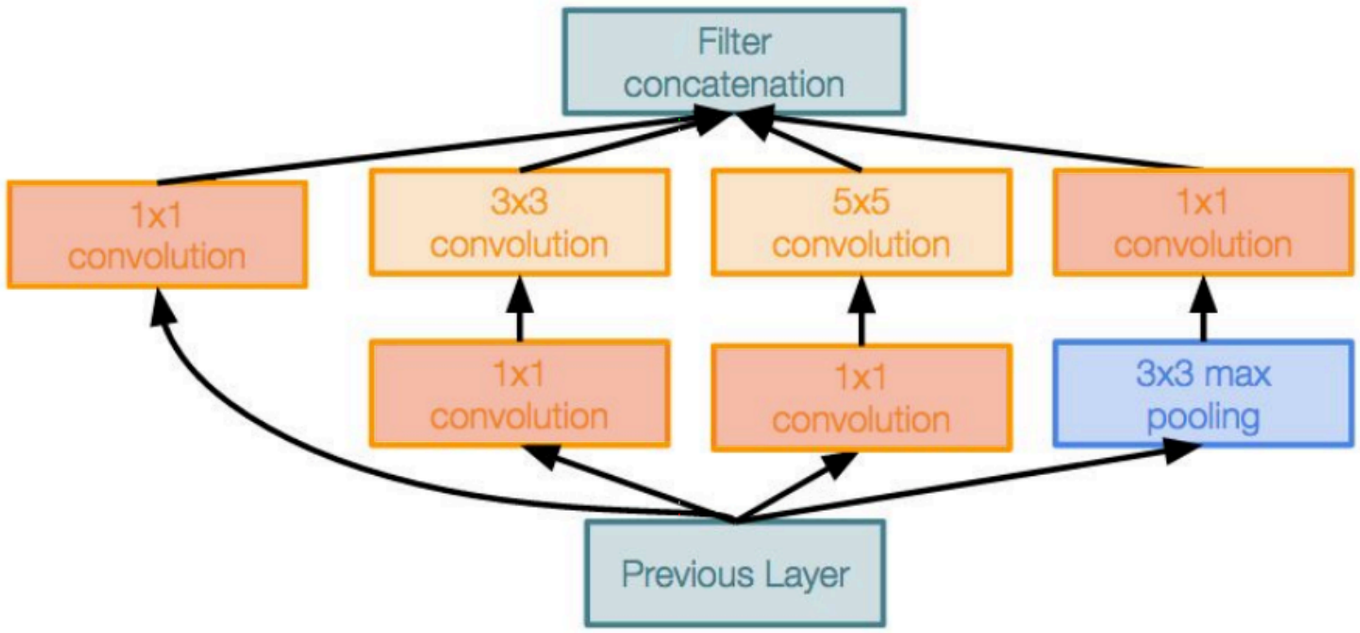


GoogLeNet

Case study: GoogLeNet (Inception)

Inception module

The key idea is to design a good local topology (network within a network) and then stack these modules on top of each other



Apply parallel filter operations on the input layer:

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

Concatenate all filter outputs together depth-wise

Use 1x1 “bottleneck” layers to reduce channel dimension before expensive convolution *(we’ll see a similar thing in ResNet)*

Case study: GoogLeNet (Inception)

Global average pooling

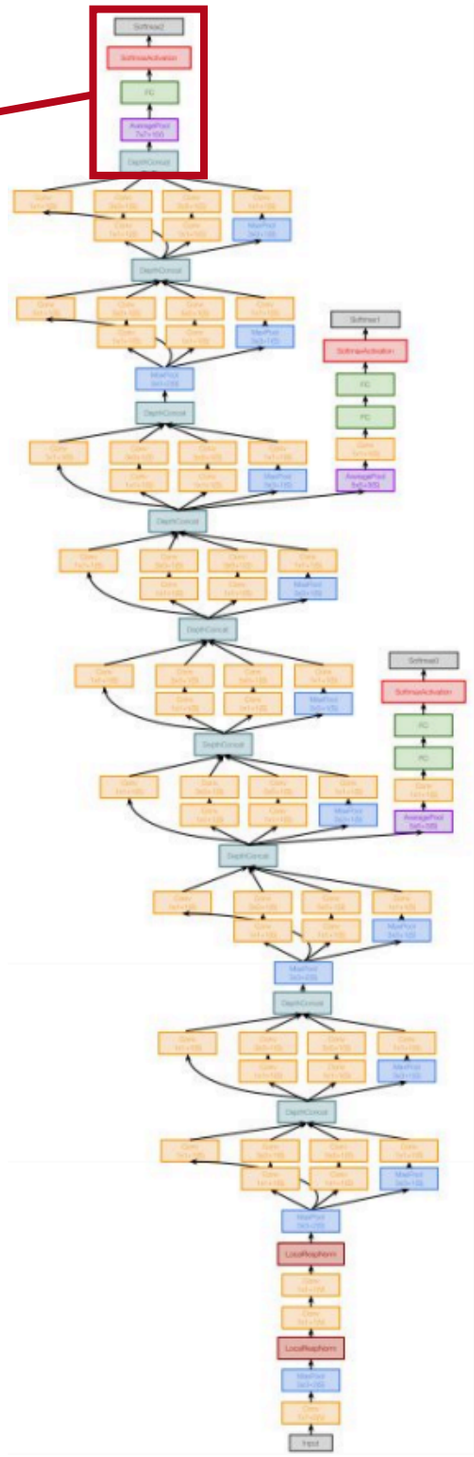
No large FC layers at the end! Instead it uses global average pooling to collapse spatial dimensions, and one linear layer to produce class scores

	Input size		Layer				Output size				
Layer	C	H/W	filters	kernel	stride	pad	C	H/W	memory (KB)	params (k)	flop (M)
avg-pool	1024	7		7	1	0	1024	1	4	0	0
fc	1024		1000				1000		0	1025	1

(recall: in VGG most of the parameters were in the FC layers)

Compare VGG-16:

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

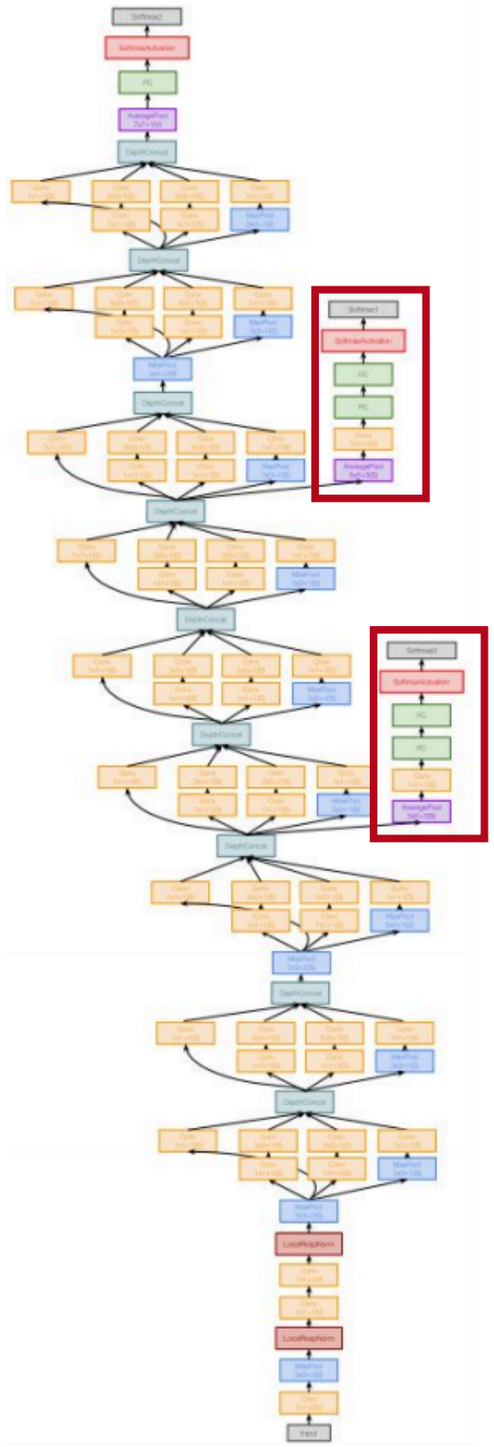


GoogLeNet

Case study: GoogLeNet (Inception)

Auxiliary classifiers

Training using loss at the end of the network didn't work well: Network is too deep, gradients don't propagate cleanly



GoogLeNet

To mitigate this problem, GoogLeNet introduces a hack:

- Attach "auxiliary classifiers" at several intermediate points in the network that also try to classify the image and receive loss
- This was before the introduction of batch normalization; with BatchNorm no longer need to use this trick!

Case study: ResNet

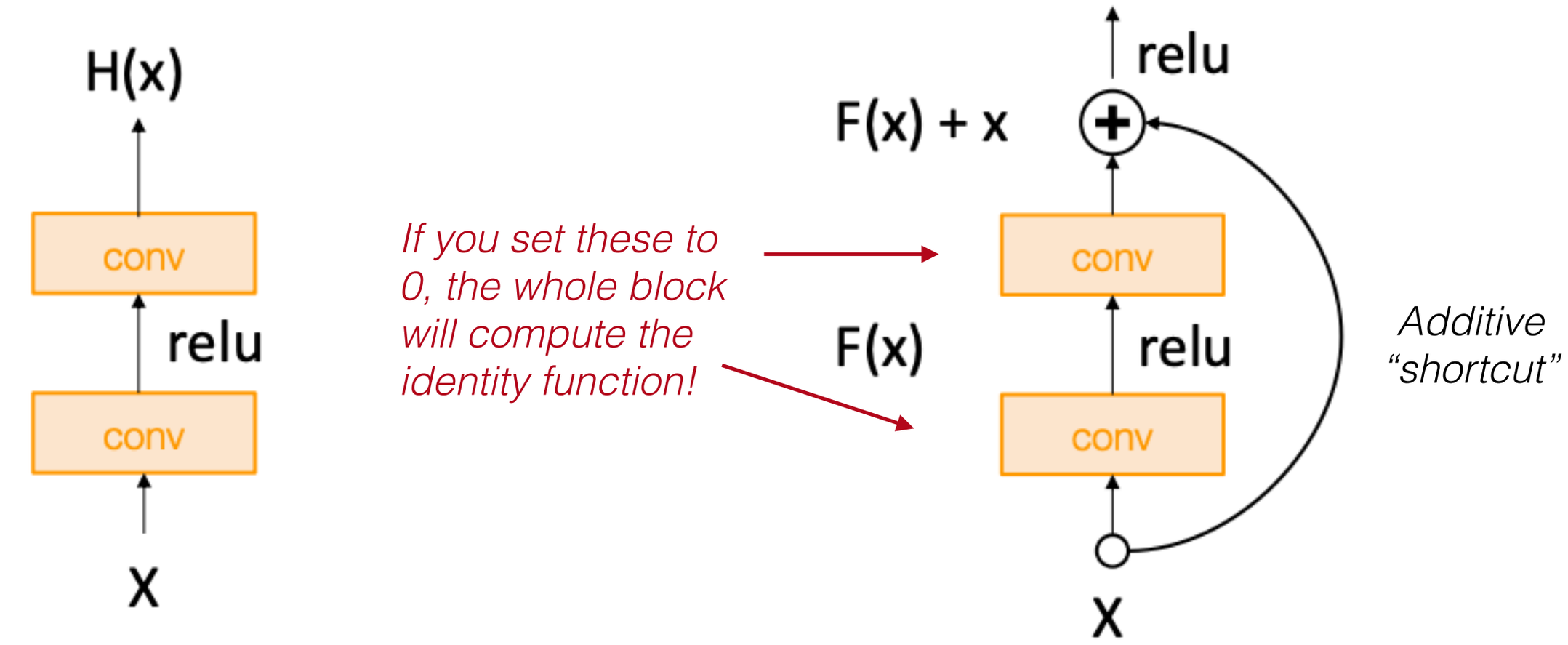
- Going deeper is challenging…
- Residual Networks (ResNet)
  - A deeper model can emulate a shallower model: copy layers from shallower model, set extra layers to identity
    - *Thus deeper models should do at least as good as shallow models!*
  - **Hypothesis:** this is an optimization problem. Deeper models are harder to optimize, and in particular don't learn identity functions to emulate shallow models…
  - **Solution:** change the network so learning identity functions with extra layers is easy!

Adapted from slide by J. Johnson

Case study: ResNet

- Key idea: residual blocks

(Solution: change the network so learning identity functions with extra layers is easy!)



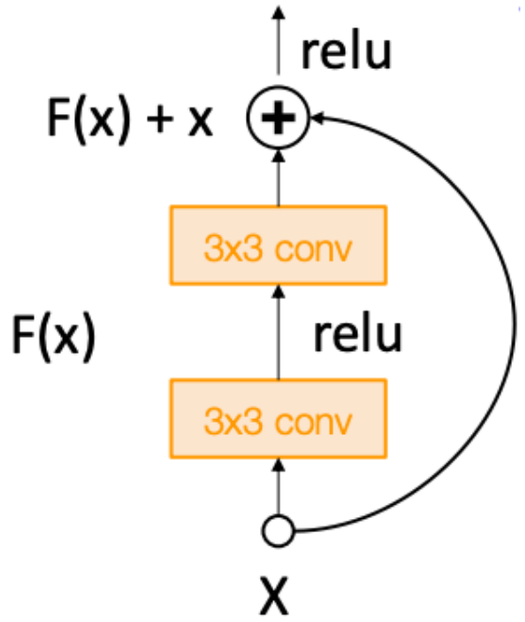
"Plain" block

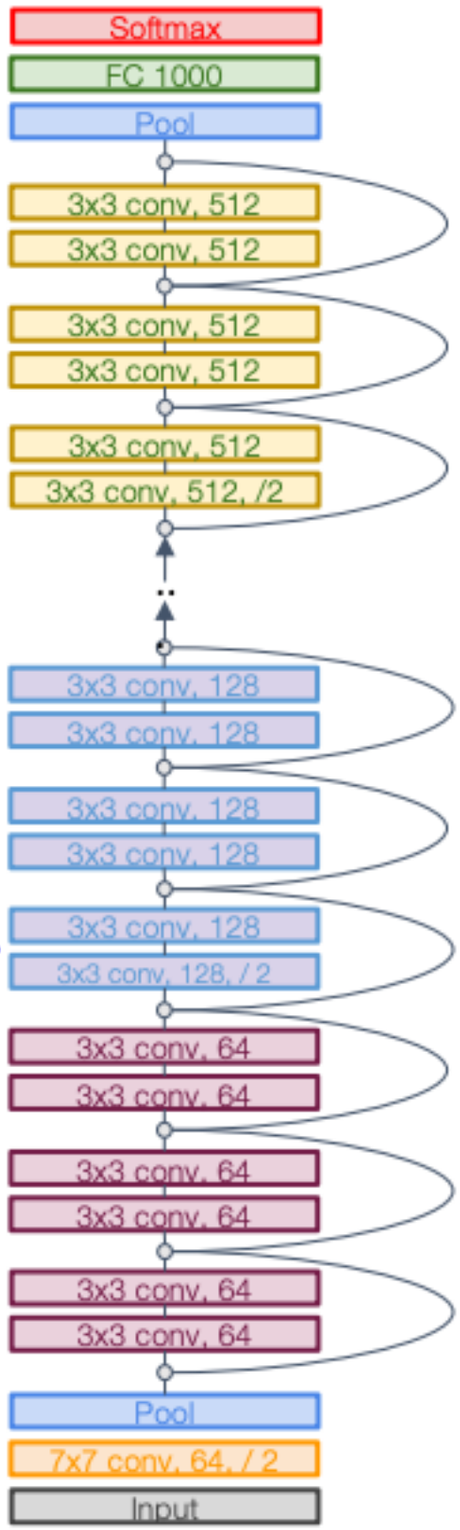
Residual block

Case study: ResNet

- A residual network is a stack of many residual blocks…

Regular design (like VGG): each residual block has two 3x3 conv filters



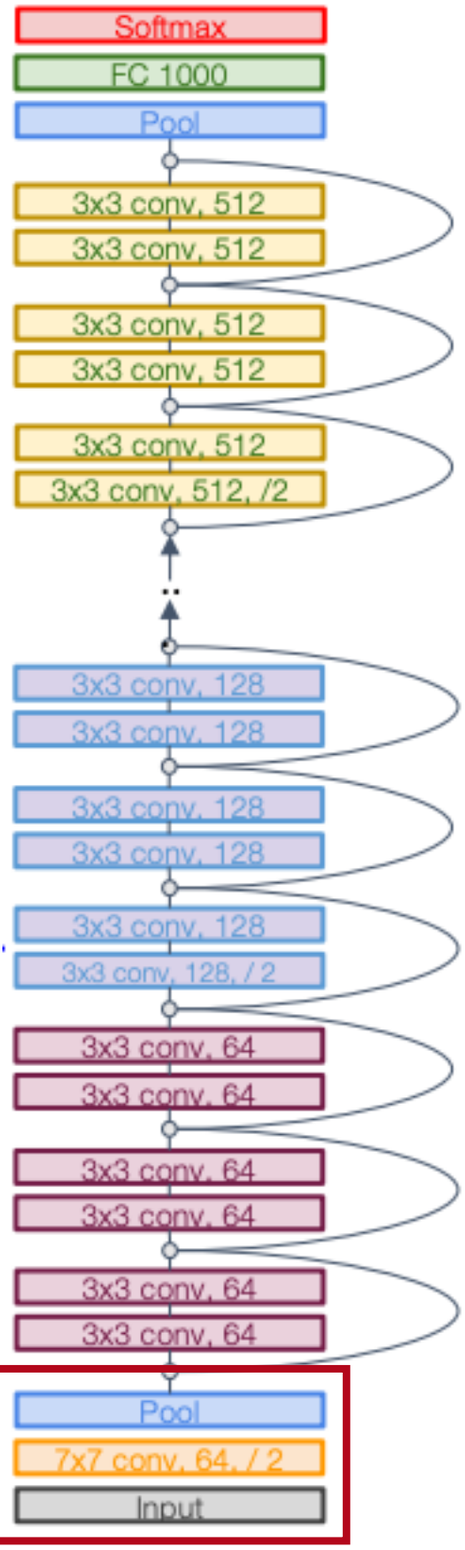


Network is divided into **stages**: the first block of each stage halves the resolution (with stride-2 conv) and doubles the number of channels

Adapted from slide by J. Johnson

Case study: ResNet

- A residual network is a stack of many residual blocks…



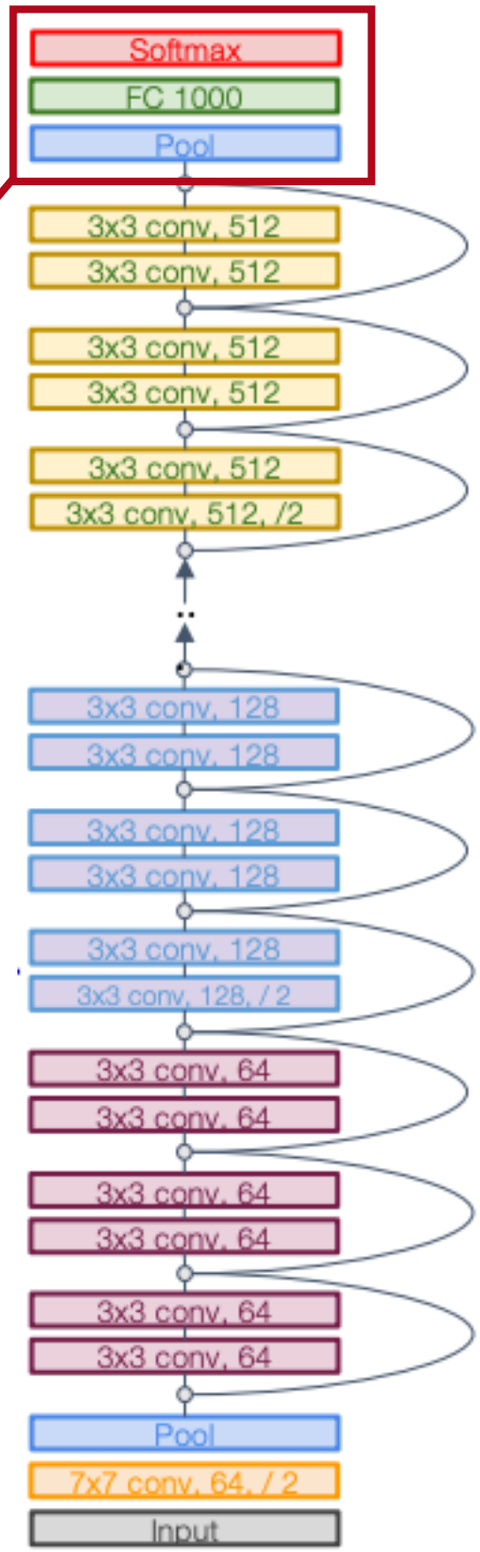
Stem network: it uses the same aggressive stem as GoogleNet to downsample the input 4x before applying residual blocks

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

Adapted from slide by J. Johnson

Case study: ResNet

- A residual network is a stack of many residual blocks…

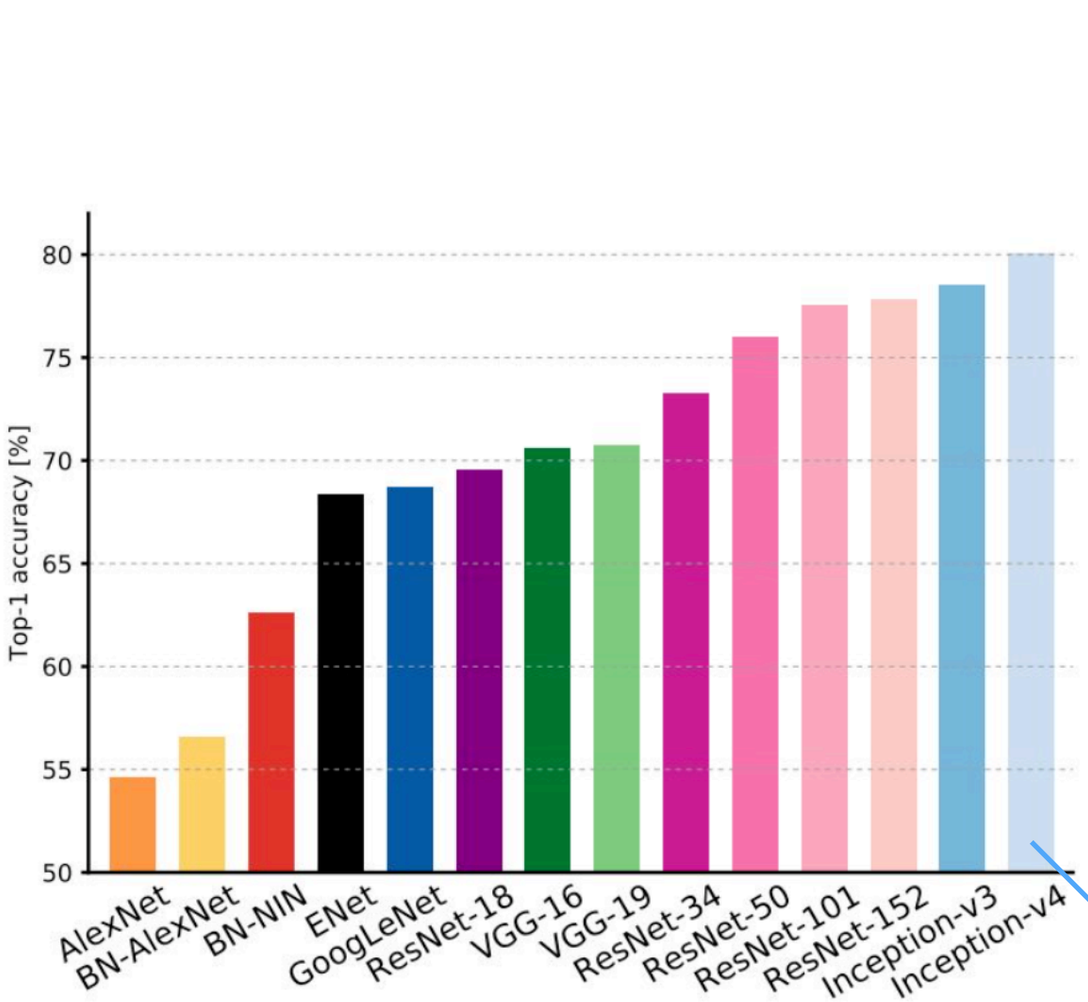


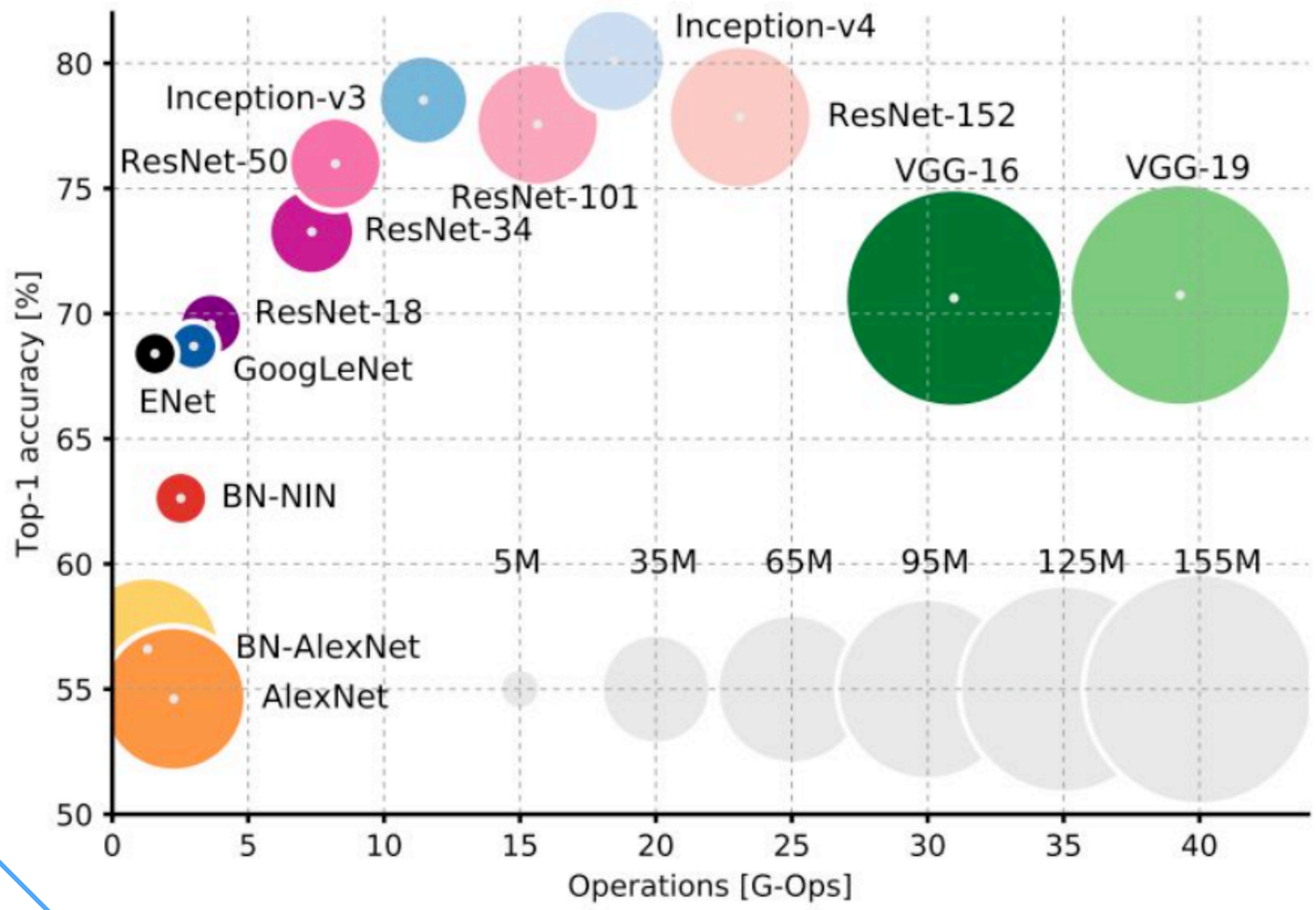
Global average pooling: like GoogLeNet, no big fully-connected-layers: instead use global average pooling and a single linear layer at the end

Adapted from slide by J. Johnson

CNN Architectures

- Comparing complexity:





Note: Inception-v4 = ResNet + Inception!

Image credit: Canziani, Culurciello, Paszke

What about modern GPUs?

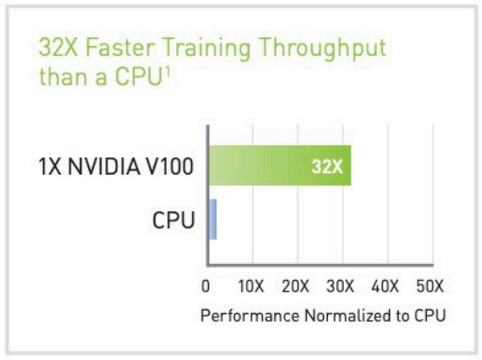
not anymore →

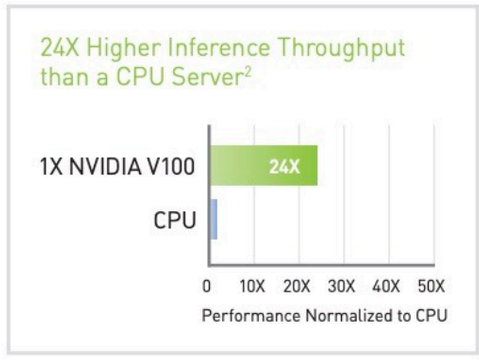
~~The World's Most Powerful GPU~~

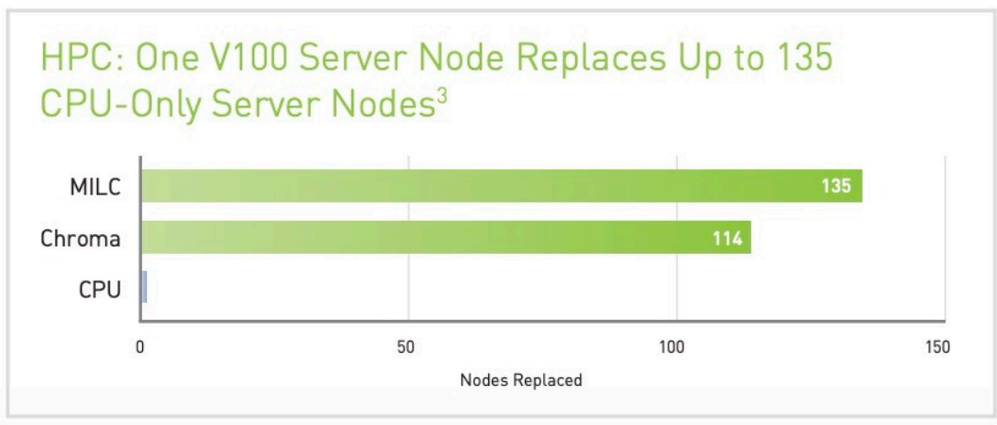
The NVIDIA® V100 Tensor Core GPU is the world's most powerful accelerator for deep learning, machine learning, high-performance computing (HPC), and graphics. Powered by NVIDIA Volta™, a single V100 Tensor Core GPU offers the performance of nearly 32 CPUs—enabling researchers to tackle challenges that were once unsolvable. The V100 won MLPerf, the first industry-wide AI benchmark, validating itself as the world's most powerful, scalable, and versatile computing platform.

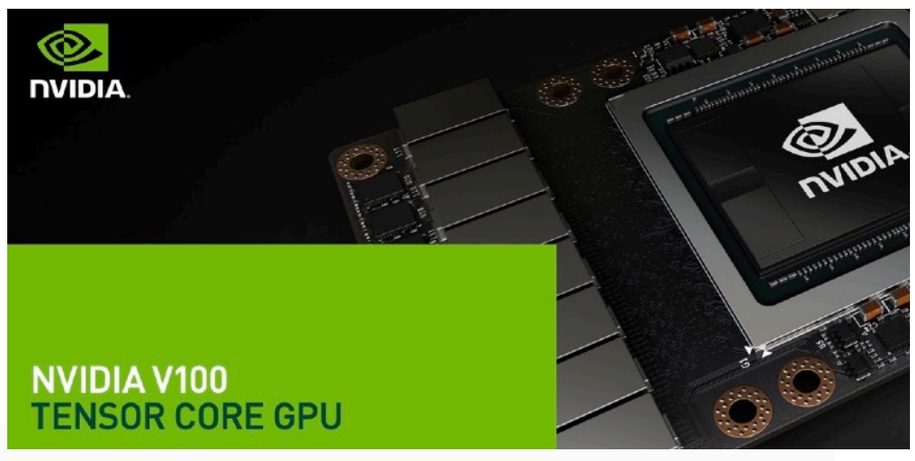
SPECIFICATIONS

	V100 PCIe	V100 SXM2	V100S PCIe
GPU Architecture	NVIDIA Volta		
NVIDIA Tensor Cores	640		
NVIDIA CUDA® Cores	5,120		
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS	8.2 TFLOPS
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS	16.4 TFLOPS
Tensor Performance	112 TFLOPS	125 TFLOPS	130 TFLOPS
GPU Memory	32 GB /16 GB HBM2		32 GB HBM2
Memory Bandwidth	900 GB/sec		1134 GB/sec
ECC	Yes		
Interconnect Bandwidth	32 GB/sec	300 GB/sec	32 GB/sec
System Interface	PCIe Gen3	NVIDIA NVLink™	PCIe Gen3
Form Factor	PCIe Full Height/Length	SXM2	PCIe Full Height/Length
Max Power Comsumption	250 W	300 W	250 W
Thermal Solution	Passive		
Compute APIs	CUDA, DirectCompute, OpenCL™, OpenACC®		





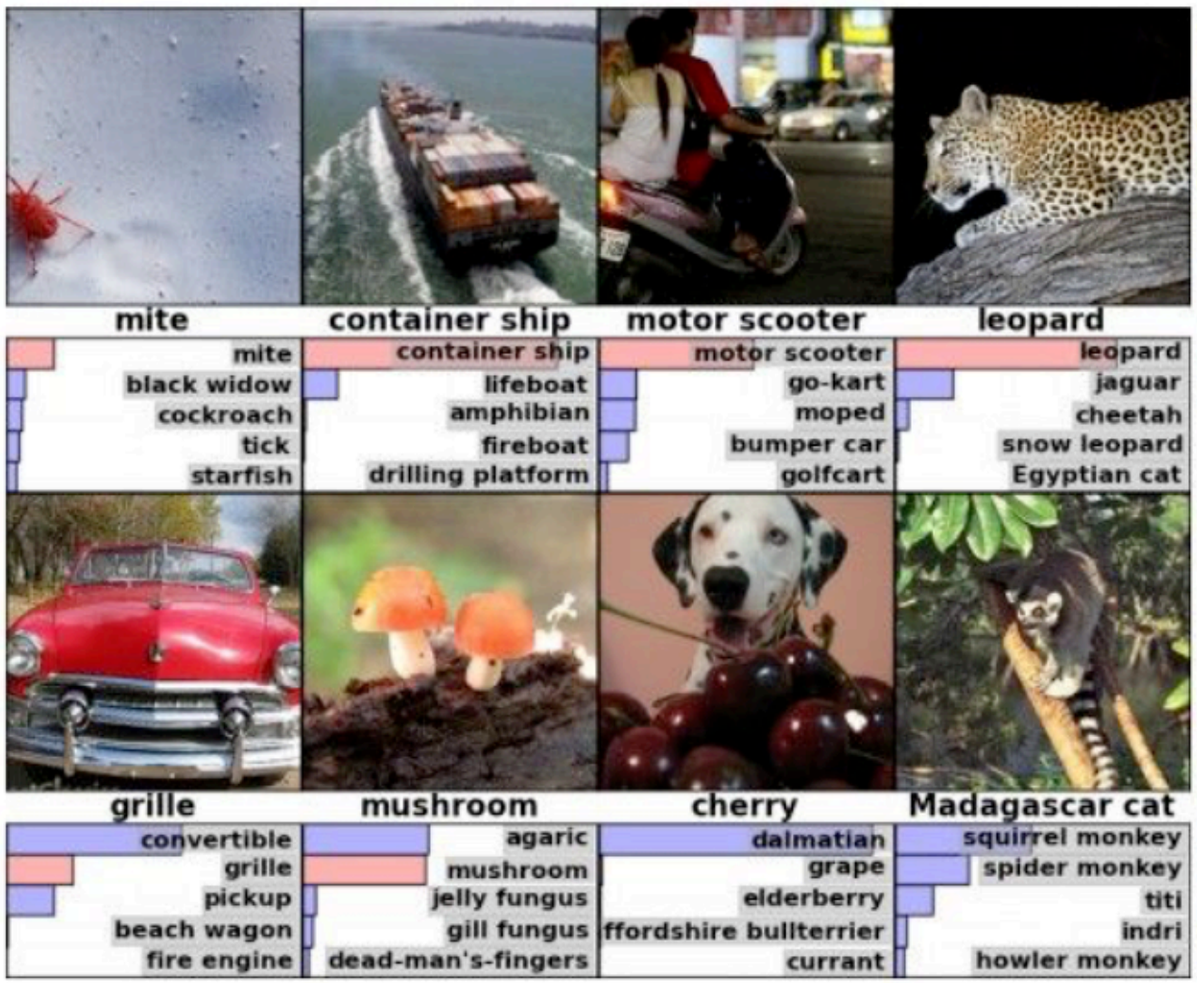




CNNs are everywhere…

Classification

Retrieval



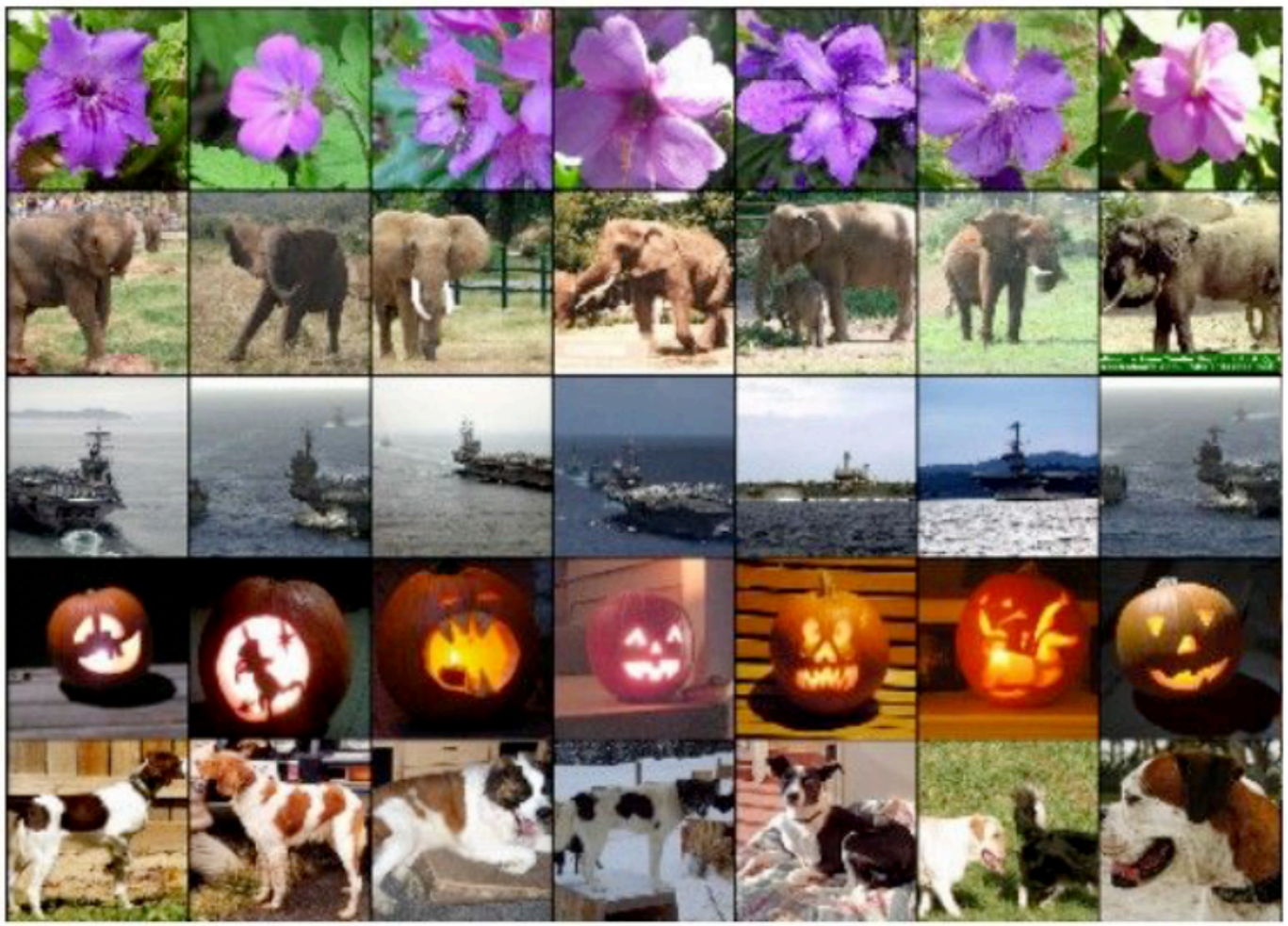
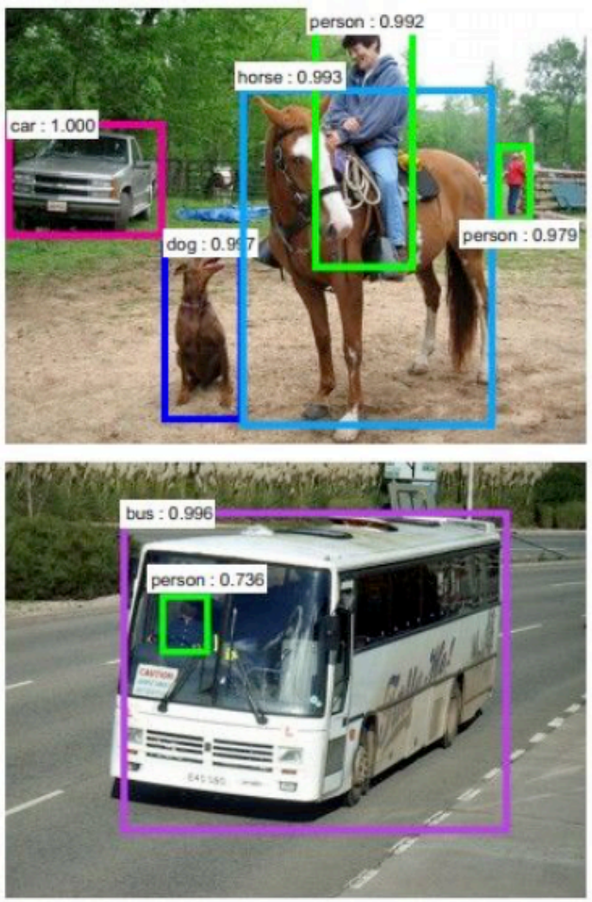


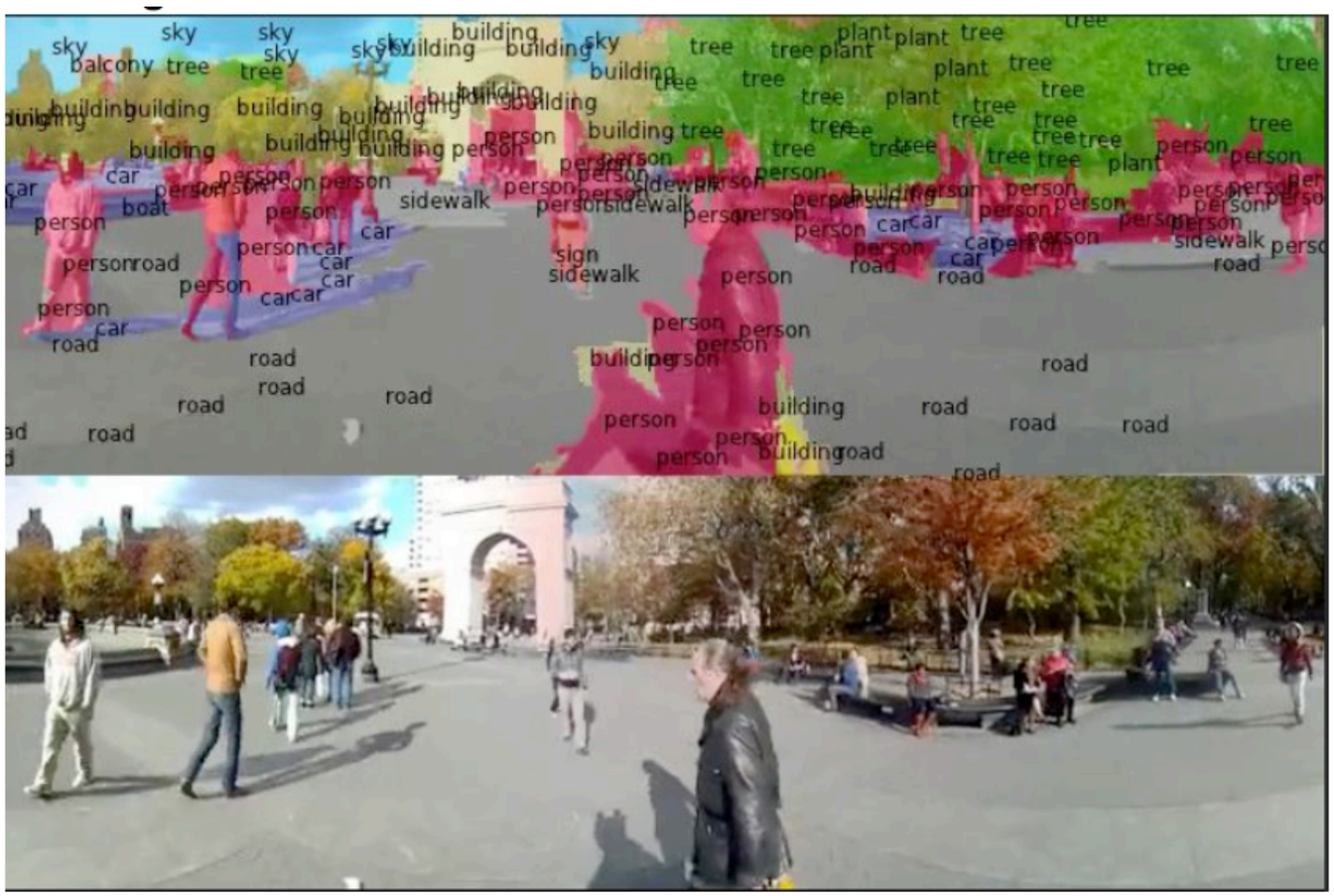
Image credit: Fei-Fei, Karpathy, Johnson

CNNs are everywhere…

Detection

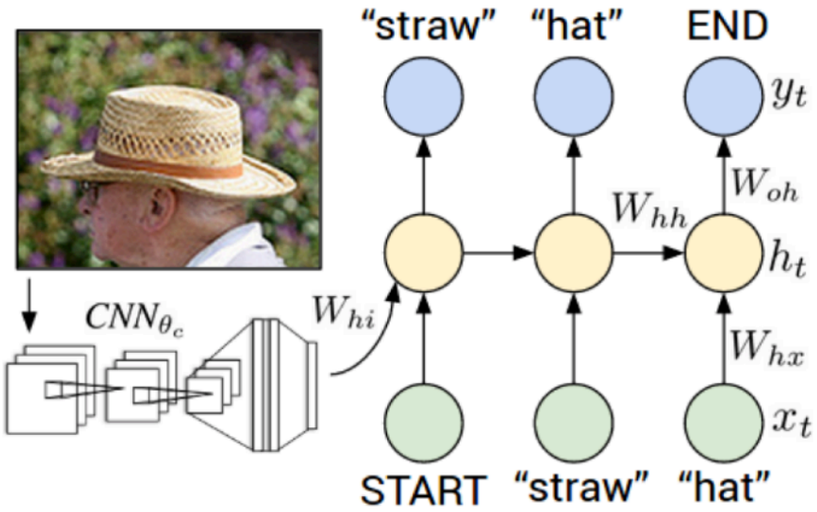
Segmentation





CNNs are everywhere…

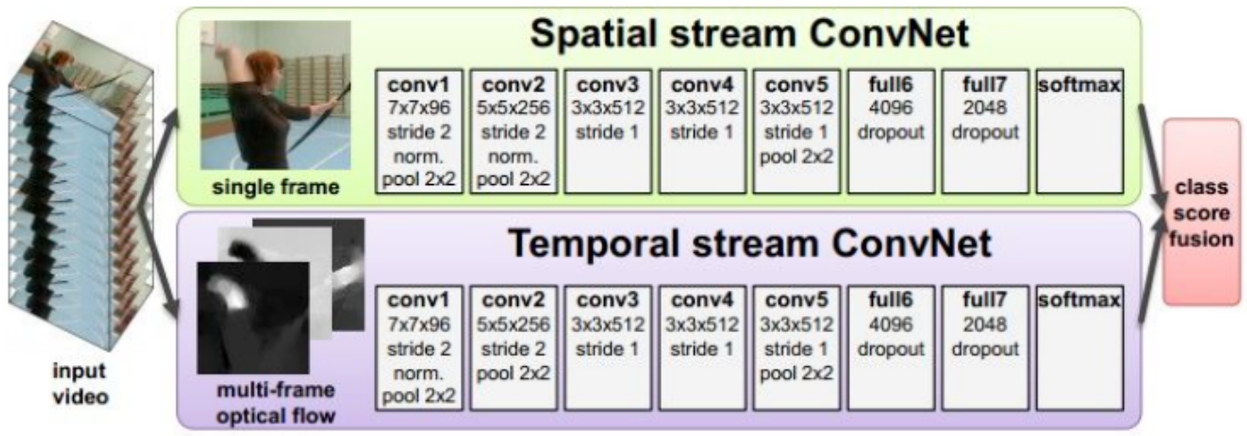
Captioning



"black and white dog jumps over bar."

"two young girls are playing with lego toy."

Video understanding



ACTIVITYNET

CNNs are everywhere: applications

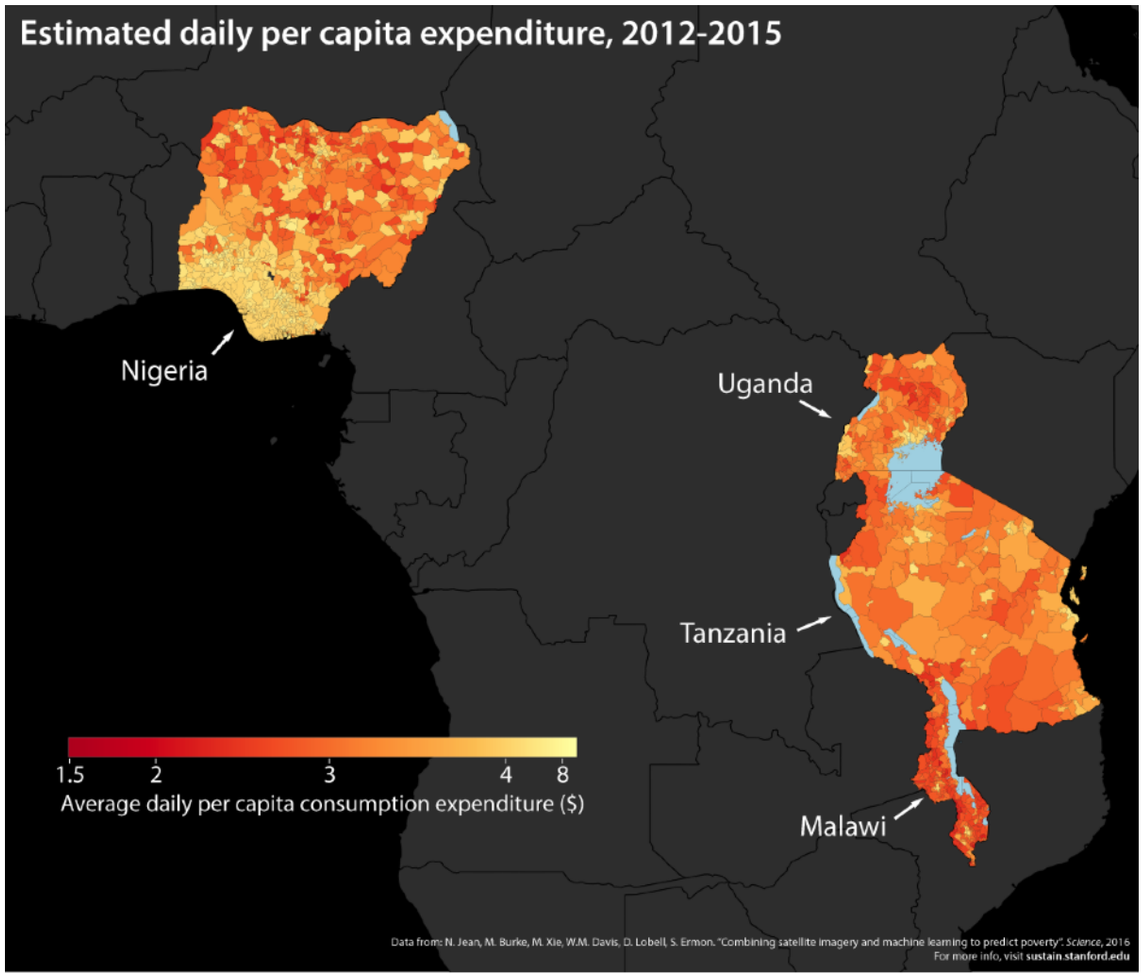
Intelligent transportation

Style transfer

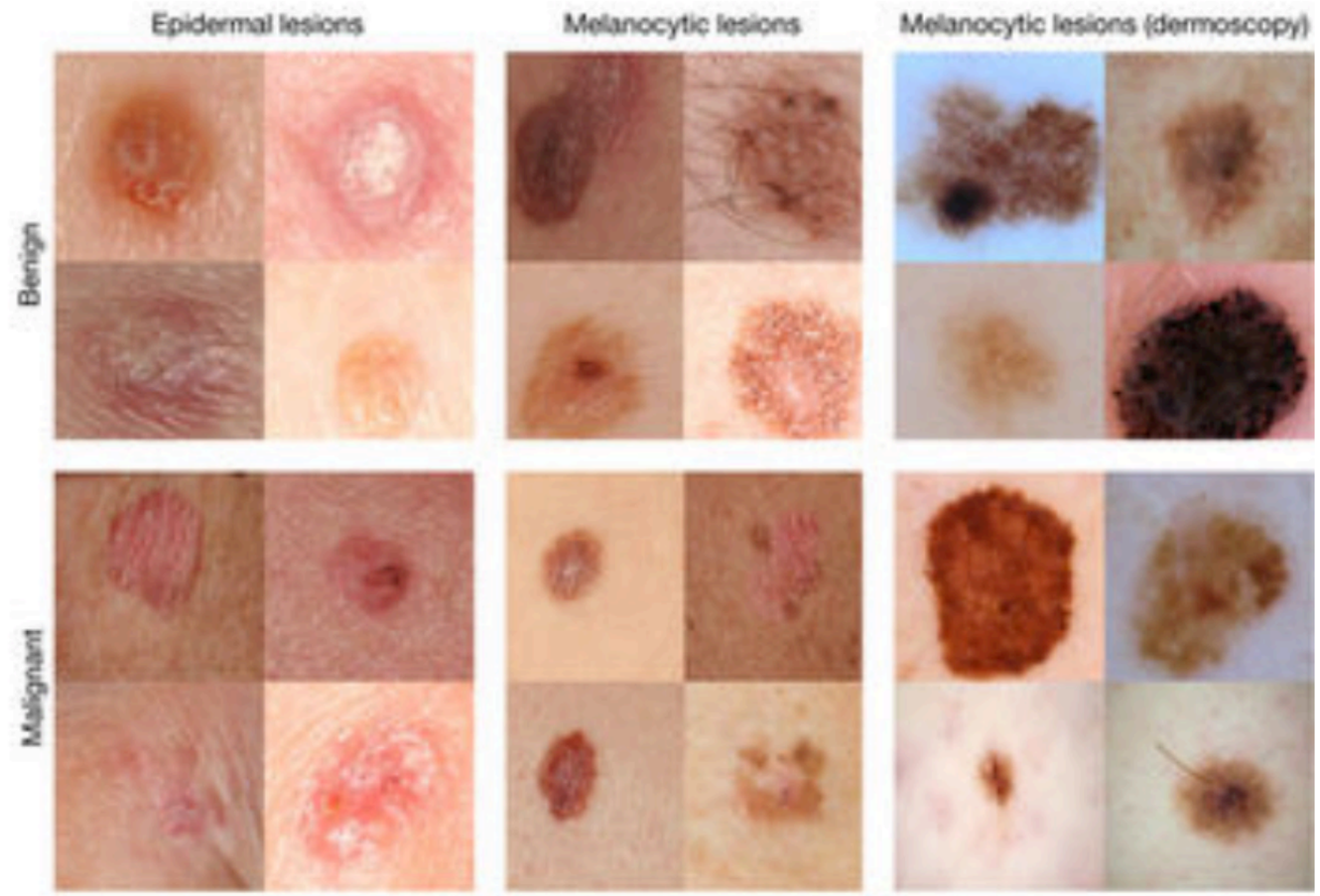
Image credit: Fei-Fei, Karpathy, Johnson

CNNs are everywhere: applications

Monitoring environmental changes

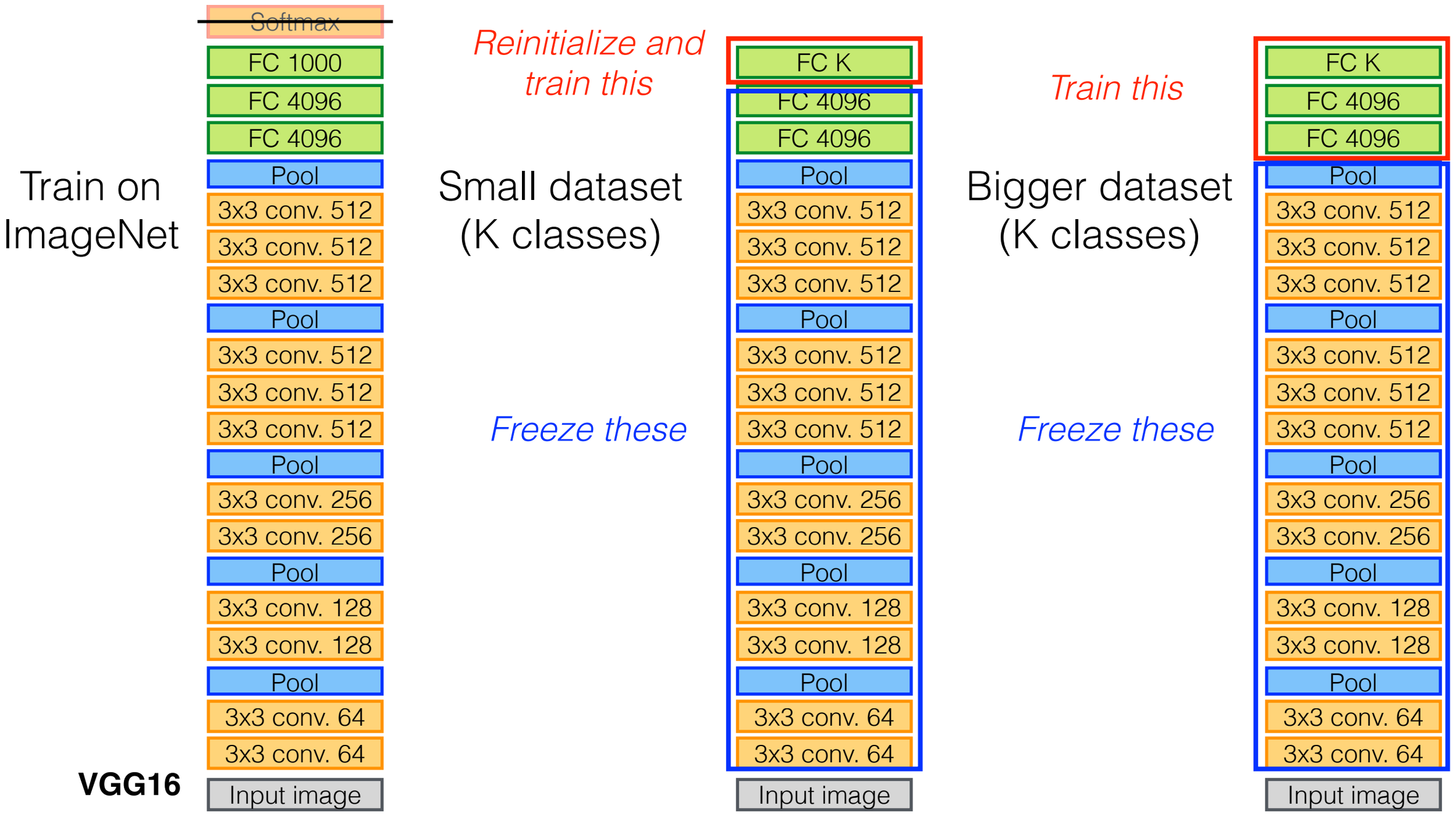


Medical imaging / Healthcare



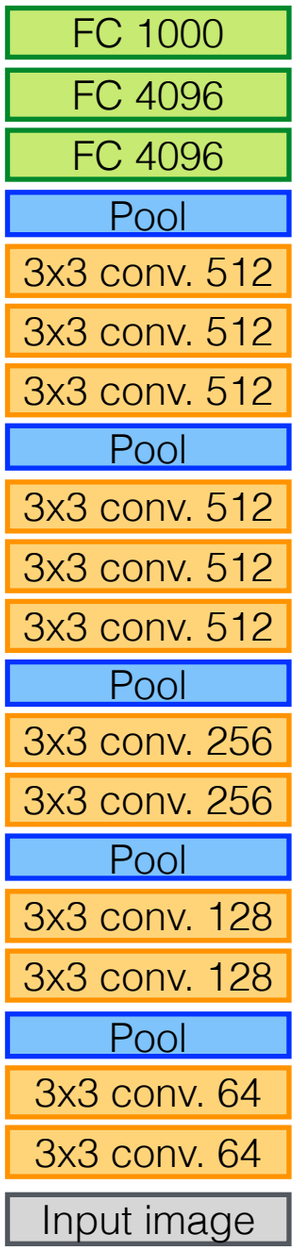
Transfer Learning / Fine Tuning

- You need a lot of a data if you want to train CNNs…



Transfer Learning / Fine Tuning

- Fine tuning: a few “guidelines”



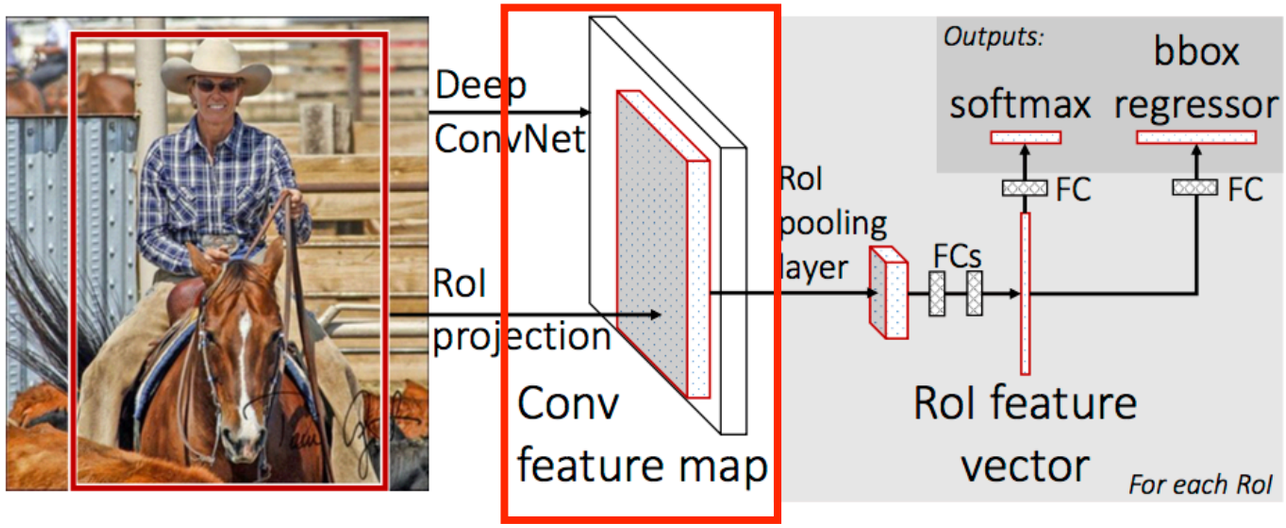
	Very similar dataset	**Very different dataset**
Very little data	Use linear classifier on top layer	This is bad… try linear classifier from different stages
Quite a lot of data	Finetune a few layers	Finetune a large number of layers

Slide credit: Fei-Fei, Karpathy, Johnson

Transfer Learning / Fine Tuning

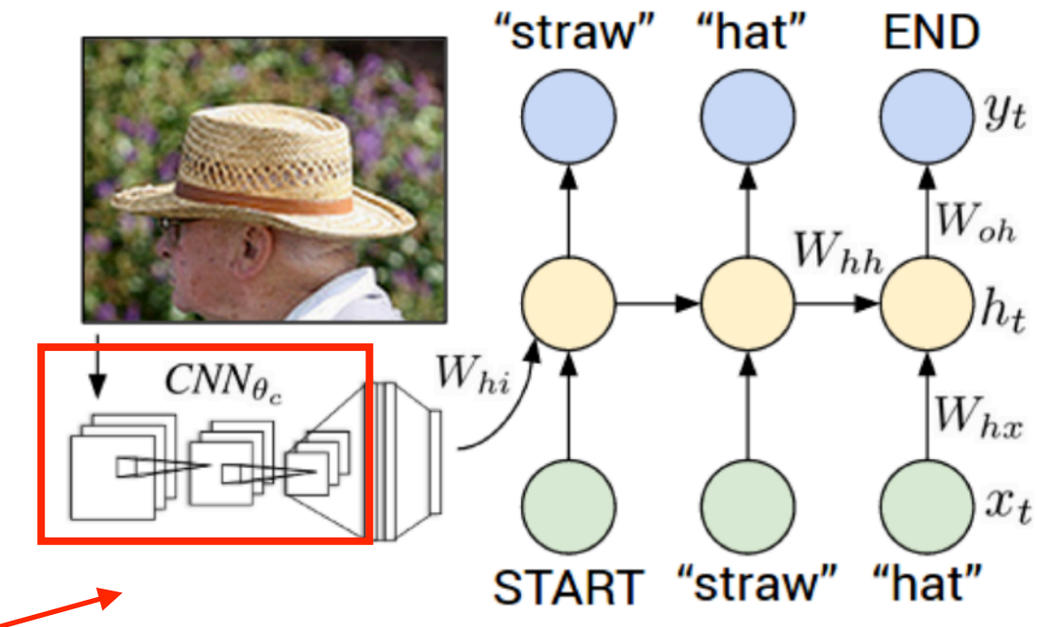
- Transfer learning with CNNs is pervasive, it's the norm, not an exception

Object Detection: Fast R-CNN



R.Girshick, "Fast R-CNN", ICCV 2015

Image Captioning: CNN+RNN



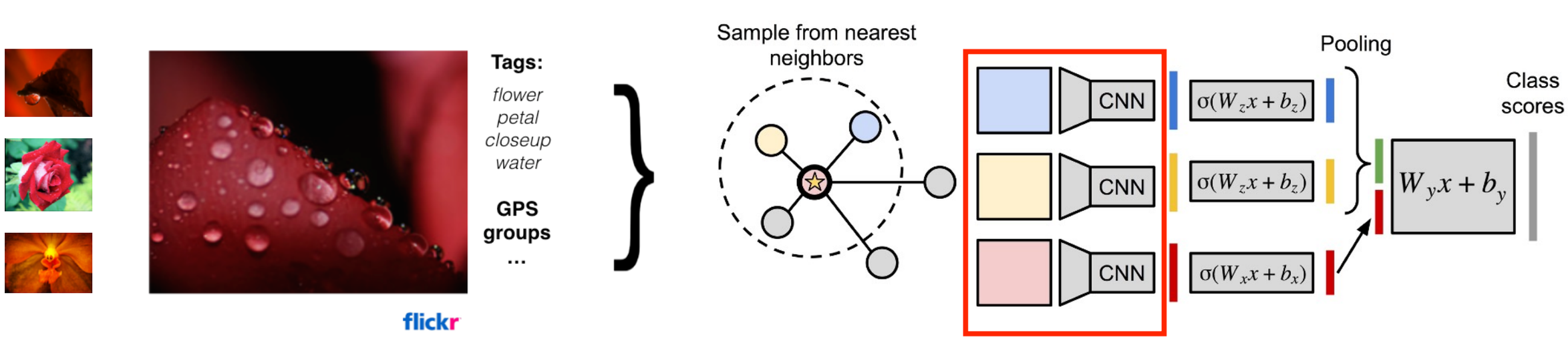
A.Karpathy, L.Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

CNN pretrained on ImageNet

Transfer Learning / Fine Tuning

- Transfer learning with CNNs is pervasive, it's the norm, not an exception

Exploiting weak labels: image tagging in social media



CNN pretrained on ImageNet

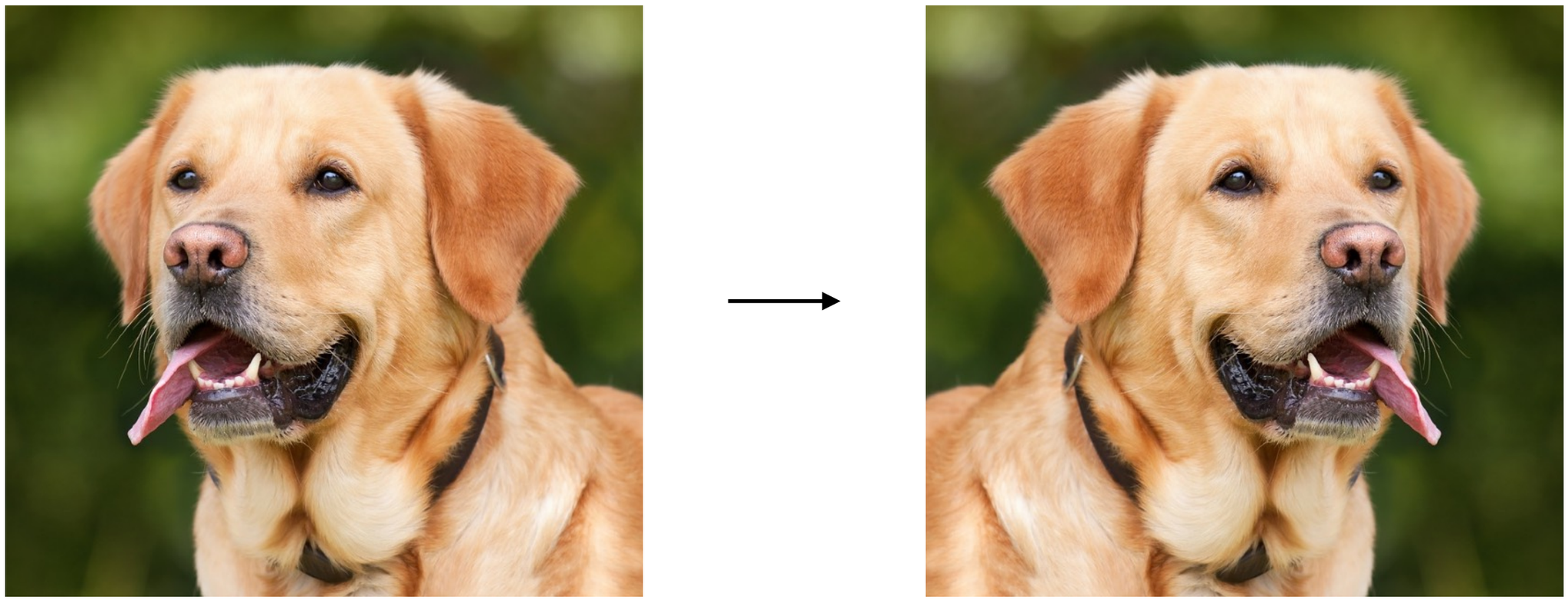
J.Johnson, L.Ballan, L.Fei-Fei, "Love Thy Neighbors: Image Annotation by Exploiting Image Metadata", ICCV 2015

Data augmentation

- You need a lot of training data: get creative in order to augment your original training samples
- Random mix/combinations of:
  - Translation
  - Rotation
  - Scaling and stretching
  - Color jittering
  - Go crazy… lens distortions, etc.

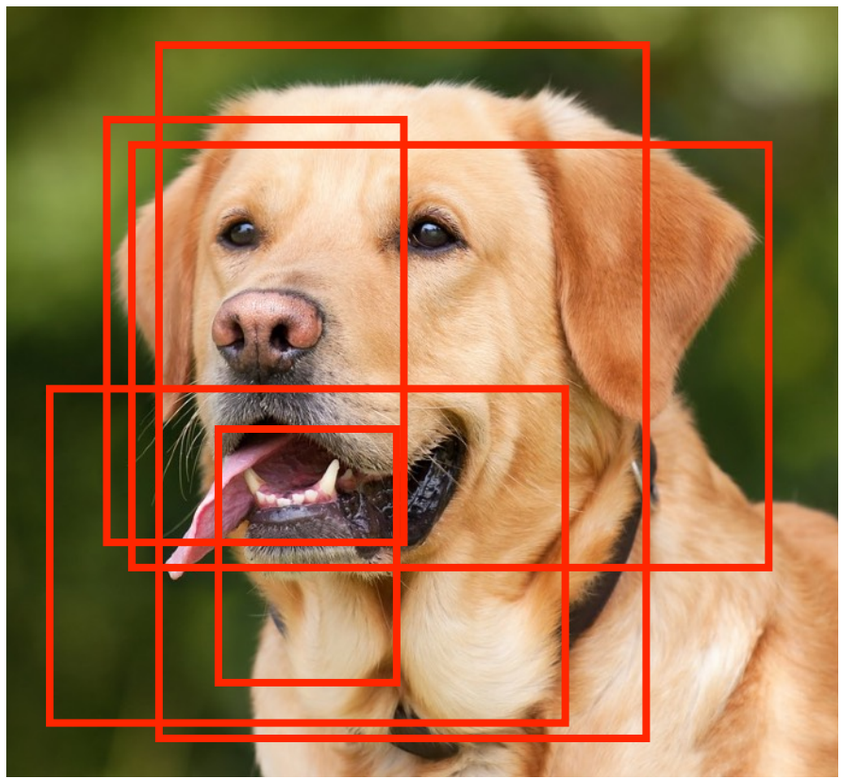
Data augmentation

- Examples: horizontal flips



Data augmentation

- Examples: random crops and scales



Training (e.g. ResNet):

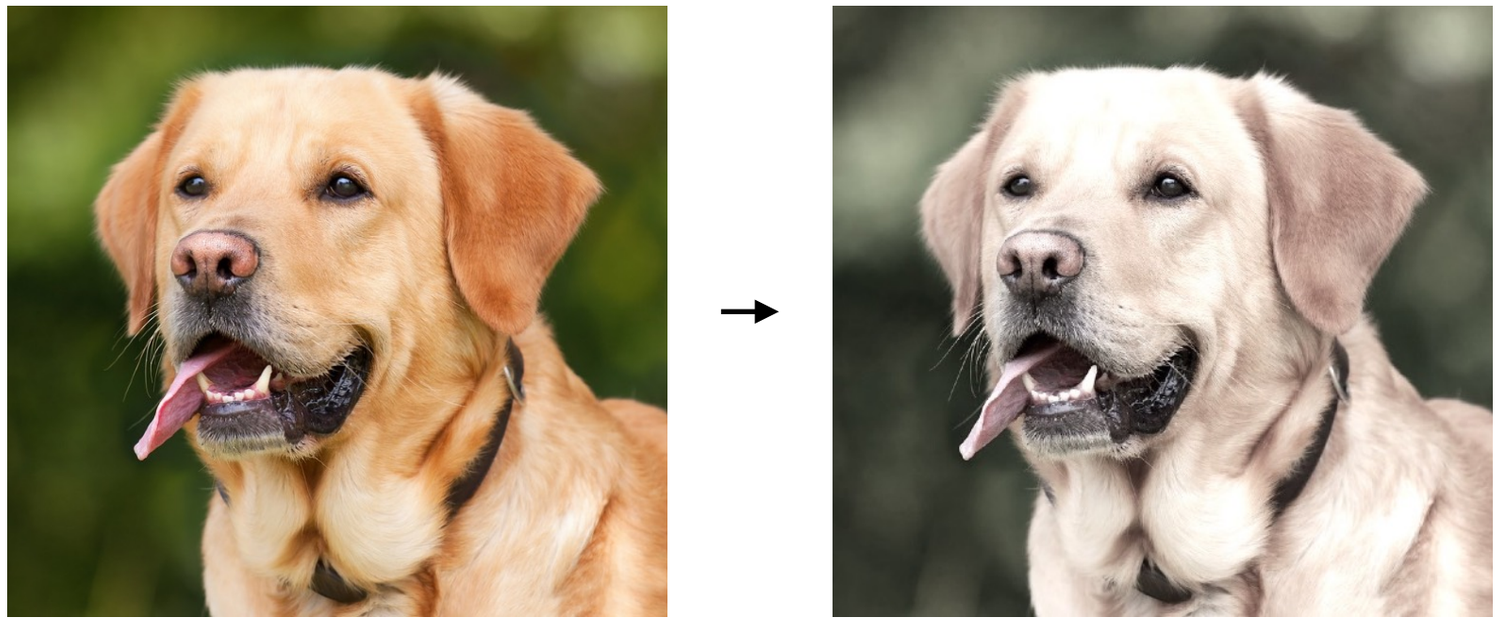
- Pick random L in range [256, 480]
- Resize training image: short-side=L
- Sample random 224 x 224 patch

Testing (e.g. ResNet):

- Resize image at 5 scales: {224, 256, 384, 480, 640}
- For each size, use ten 224x224 crops: 4 corners + center, + flips

Data augmentation

- Examples: color jitter
  - Simple: just randomize contrast and brightness



  - Advanced: apply PCA to all RGB pixels in training set
  - Then sample a color offset along principal component directions and add offset to all pixels of a training image

Takeaways for your projects

- Do you have some dataset of interest but it has less than 1 Million images?
  - Find a large dataset that has similar data (e.g. ImageNet) and train a CNN there
  - Transfer learn / finetune to your “target” dataset

- Deep learning frameworks provide a “Model zoo” of pretrained models so you don’t need to train them
  - PyTorch: https://github.com/pytorch/vision
  - TensorFlow: https://github.com/tensorflow/models
  - Caffe: https://github.com/BVLC/caffe/wiki/Model-Zoo
  - Keras: https://github.com/albertomontesg/keras-model-zoo

Contact

- **Office:** Torre Archimede, room 6CD3
- **Office hours** (ricevimento)**:** Friday 9:00-11:00

✉ lamberto.ballan@unipd.it

🏠 http://www.lambertoballan.net

🏠 http://vimp.math.unipd.it

@ twitter.com/lambertoballan