Cognitive Services

LM Informatica & Data Science - 2nd semester - 6 CFU

Intro to Convolutional Neural Networks- Part 2

Lamberto Ballan

University of Padova, A.Y. 2018/19 May 14, 2019

Administrative

- Our next lecture (Monday 20) will be in lab
- In that lecture we will give more info/suggestions about the course projects
- Today we will spend ~15 minutes collecting your feedbacks about the course with questionnaires

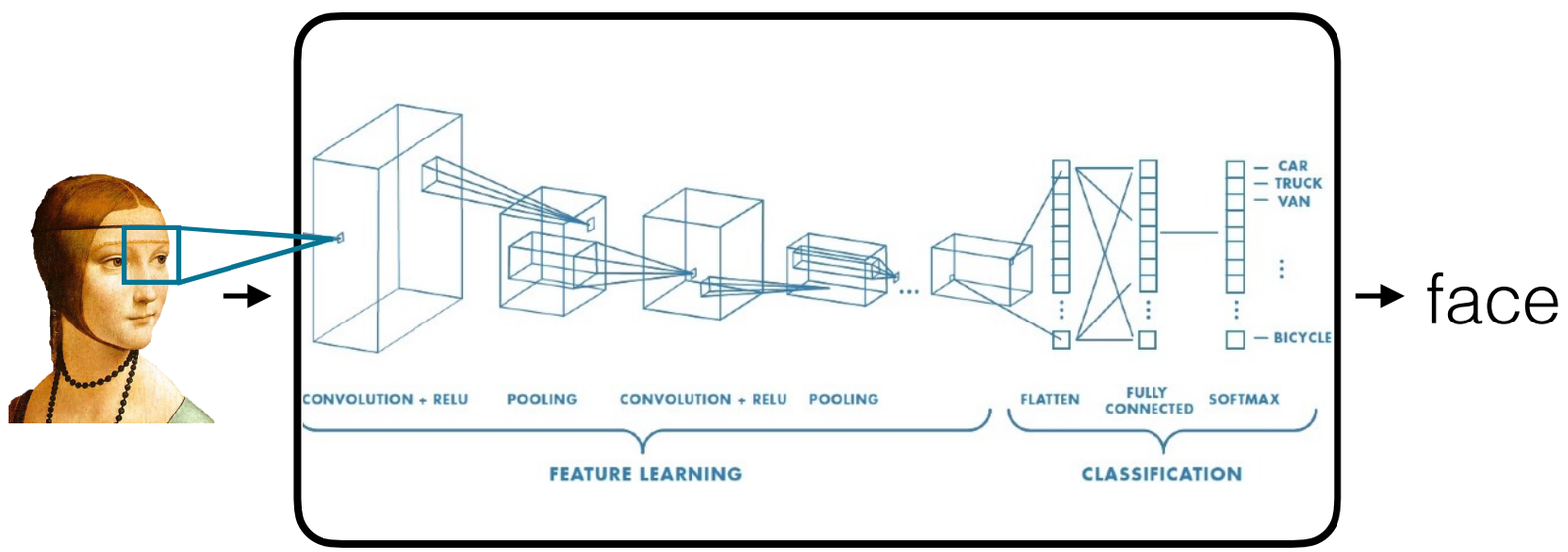
Questionnaires

- We kindly ask you to provide your evaluation of your personal experience with this course
- There are three open questions: for each of them please answer with respect to the two parts (P1 & P2)
- Questionnaires will remain anonymous
- Please write on top of your questionnaire if you are:
  - A student enrolled in the program “LM Informatica”
  - “LM Data Science”
  - “LT Informatica”
  - “Others”

What we learned in the last class

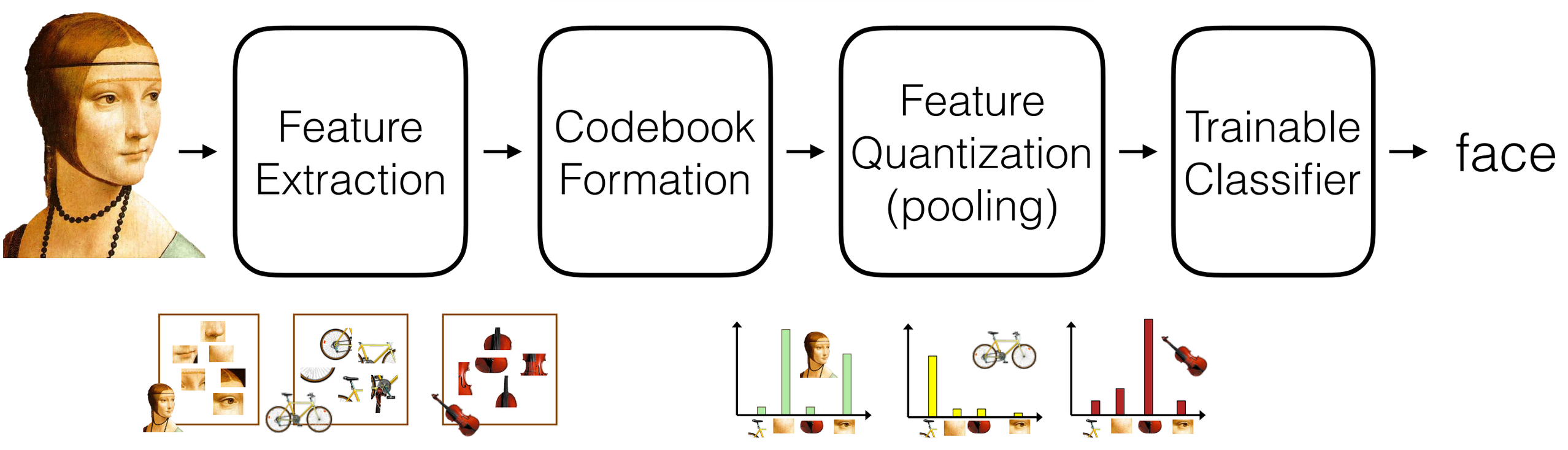
(a brief summary)

- 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



From hand-crafted representations to…

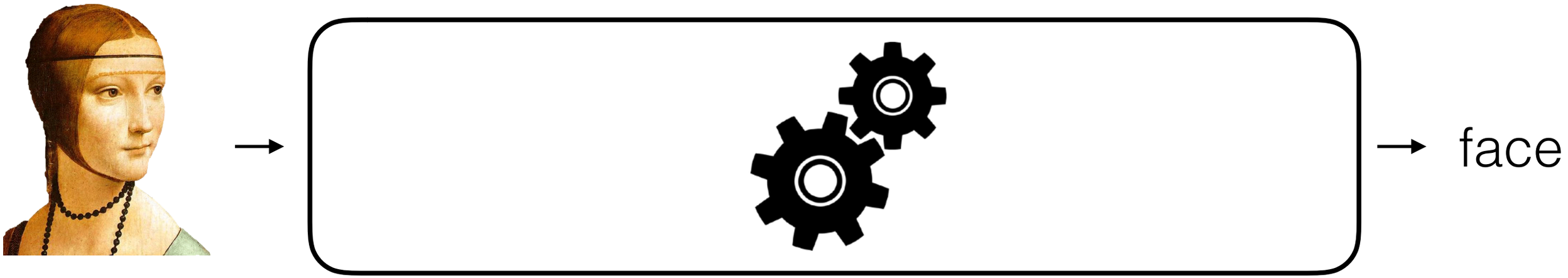
- Since now we have seen hand-crafted feature representations
- Let's take a look (again) at our bow pipeline:



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



What we will learn today?

- More on convolutional layers
- CNN architectures, examples and case studies
- Training CNNs, applications
- Takeaways

Several slides adapted from Stanford CS231n by L. Fei-Fei, A. Karpathy, J. Johnson, S. Yeung

Resources and background material:

- **http://cs231n.stanford.edu/** (slides, videos)
- **http://cs231n.github.io/** (notes and tutorials)

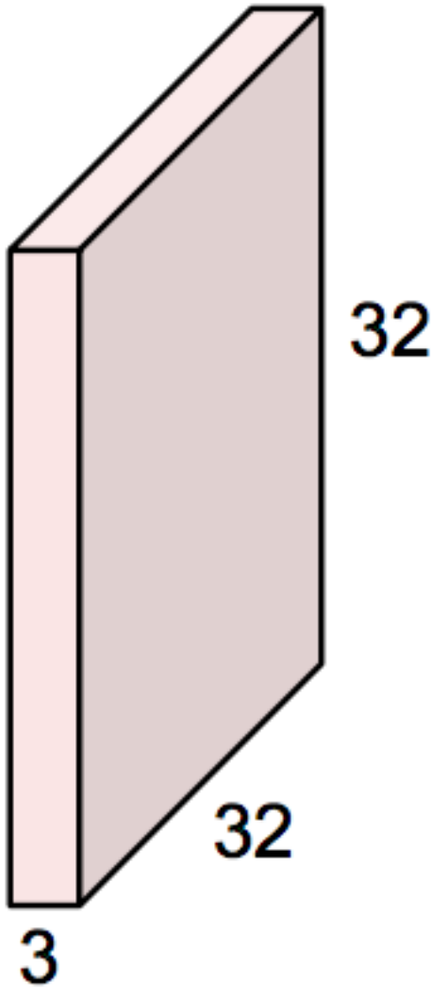
Convolutional layer

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

32x32x**3** image

Note: filters always extend the full depth of the input volume

5x5x**3** filter

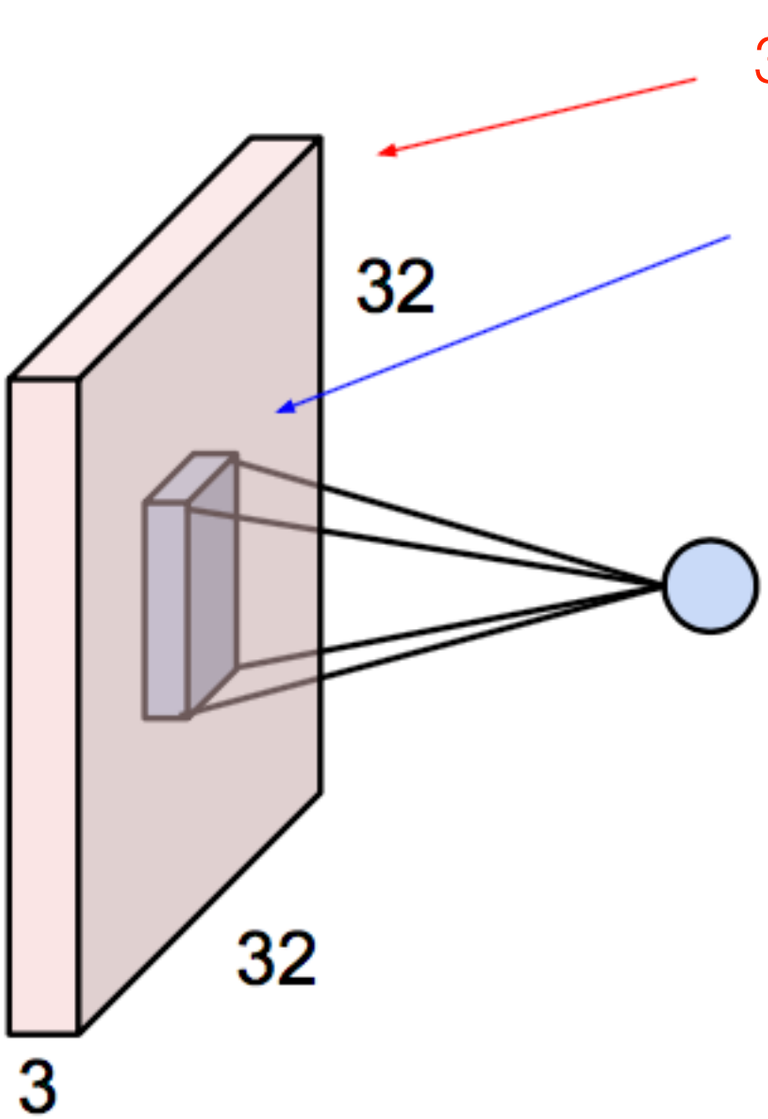


Convolve the filter with the image, i.e. “slide over the image spatially, computing dot products”

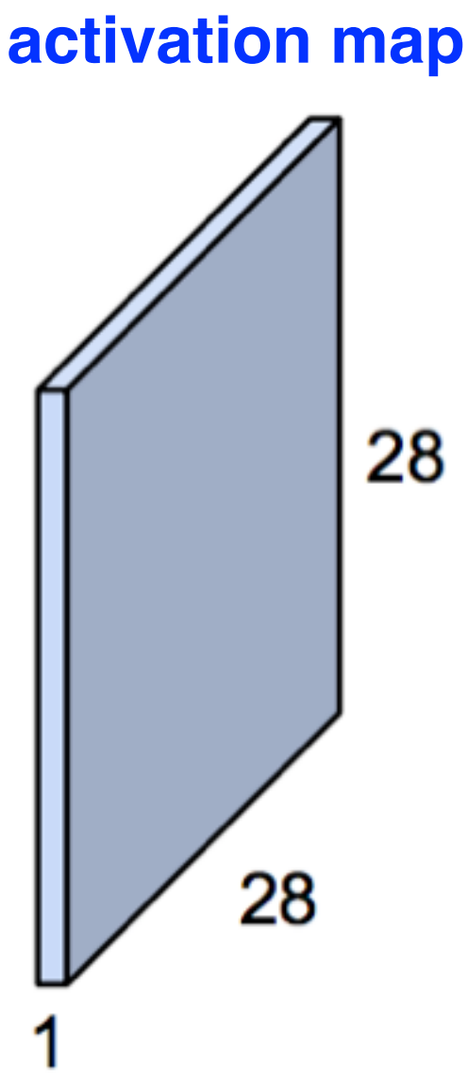
Slide credit: Fei-Fei, Karpathy, Johnson

Convolutional layer

- A closer look at spatial dimensions:

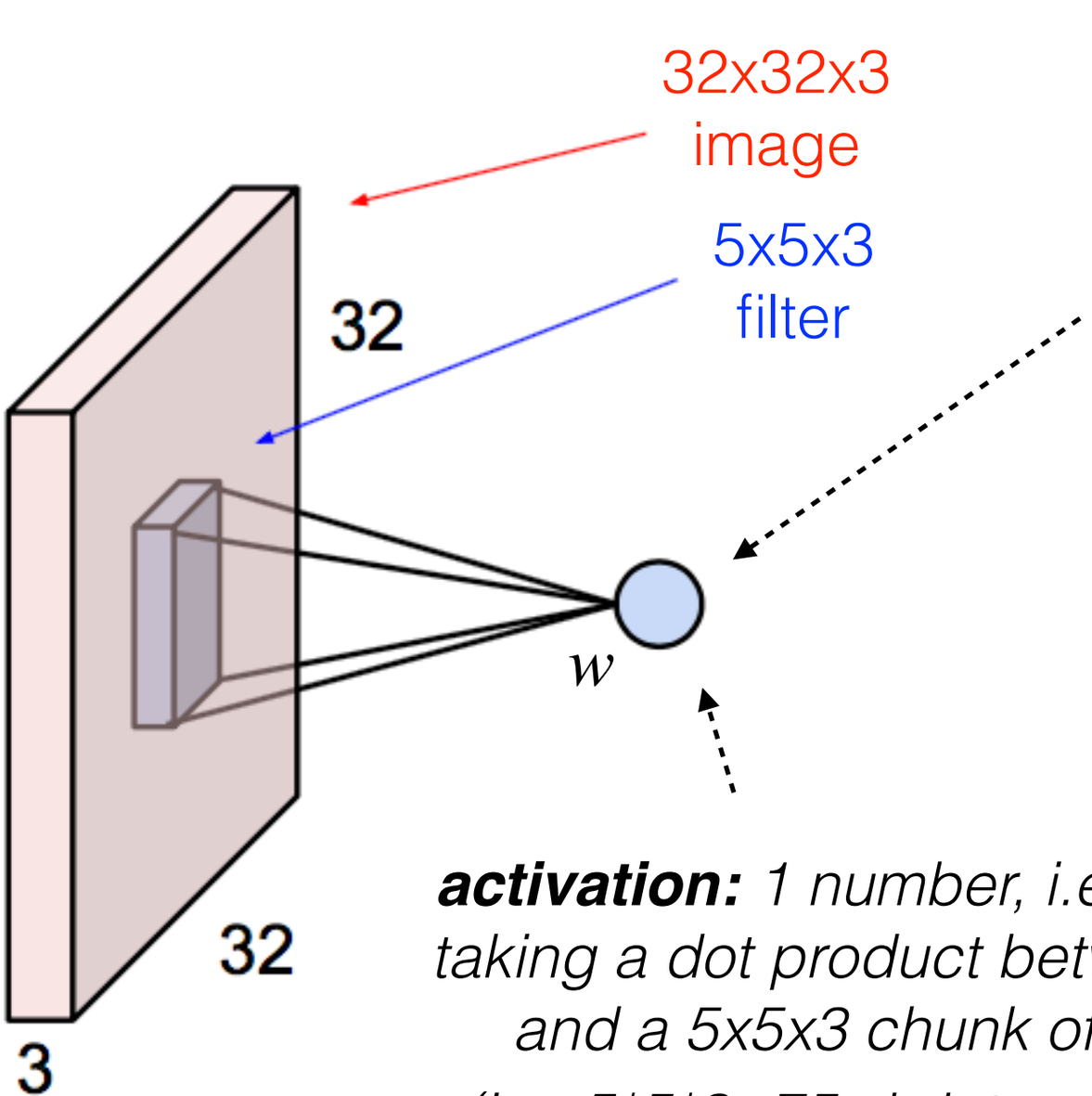


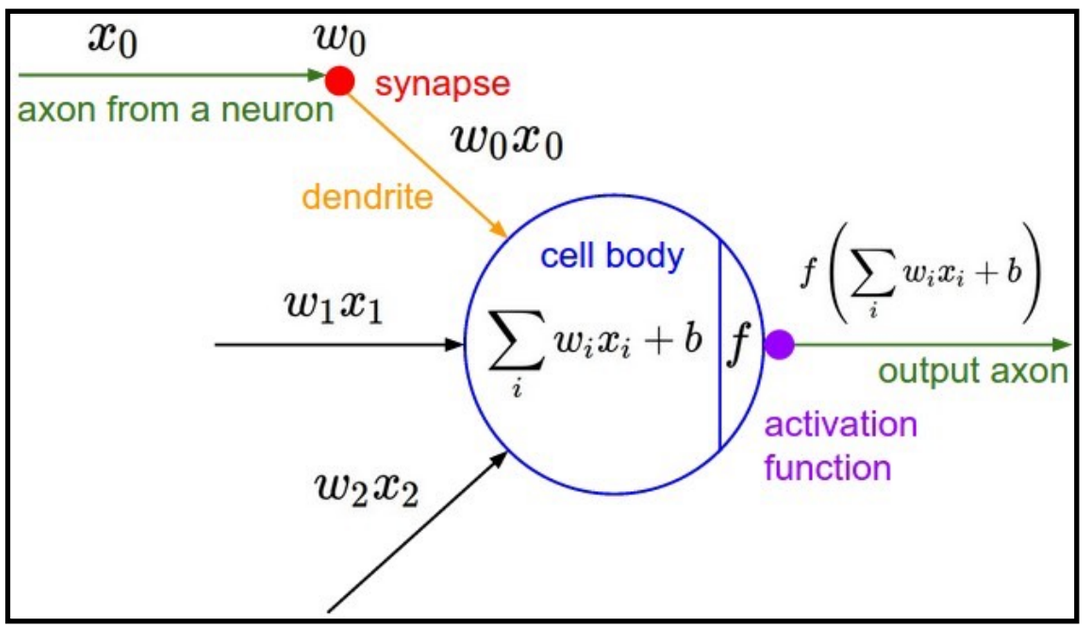
Convolve (slide) over all spatial locations



More on convolutional layers

- The brain/neuron view of conv layers:





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

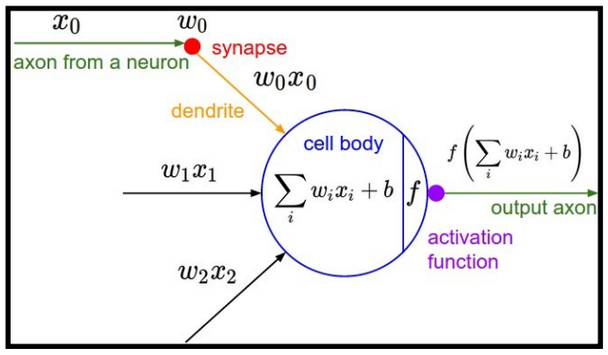
activation: *1 number, i.e. the result of taking a dot product between the filter and a 5x5x3 chunk of the image*

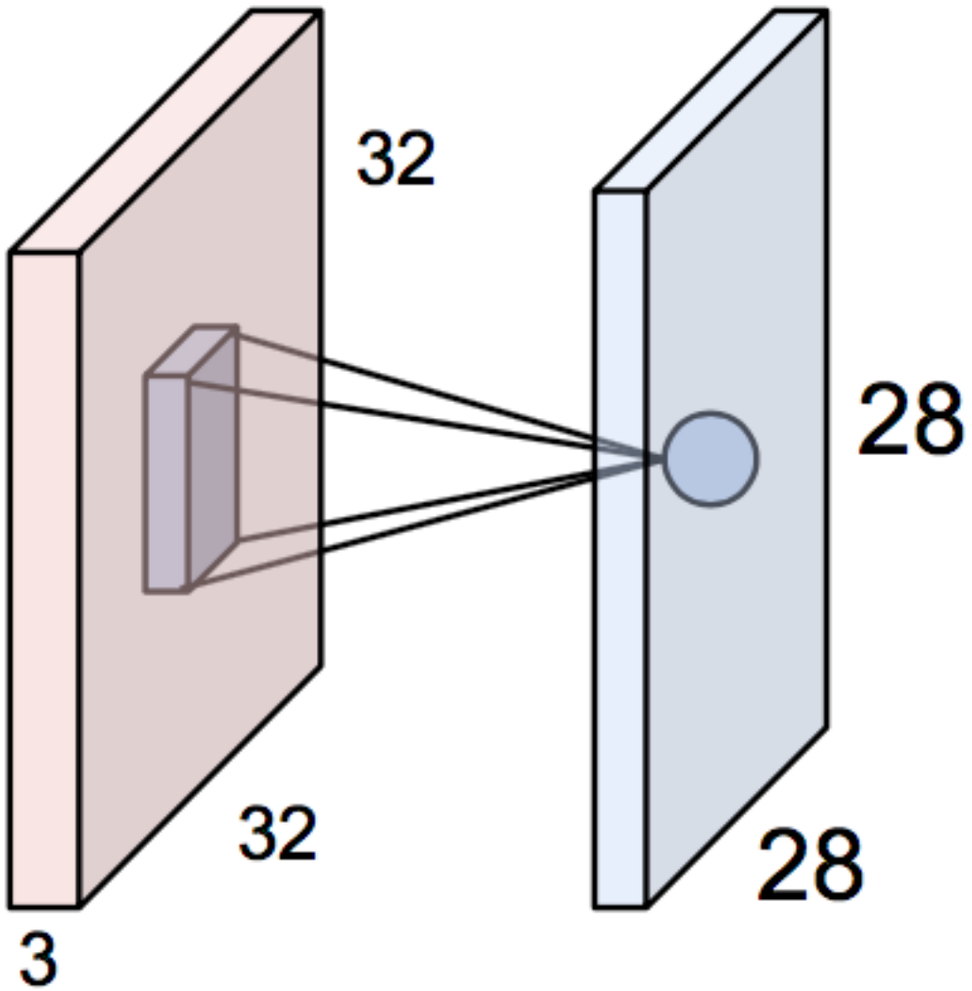
*(i.e. 5*5*3=75-d dot product + bias)*

Slide credit: Fei-Fei, Karpathy, Johnson

More on convolutional layers

- The brain/neuron view of conv layers:





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

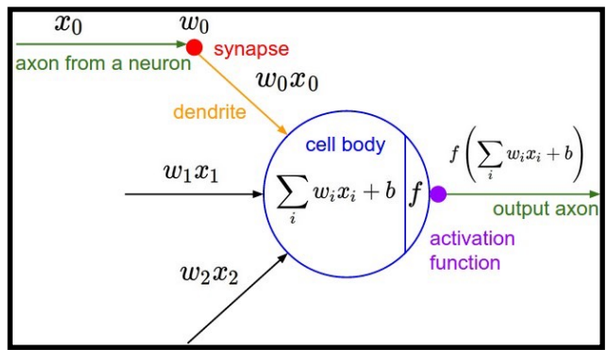
- Each neuron is connected to a small region in the input
- All of them share parameters

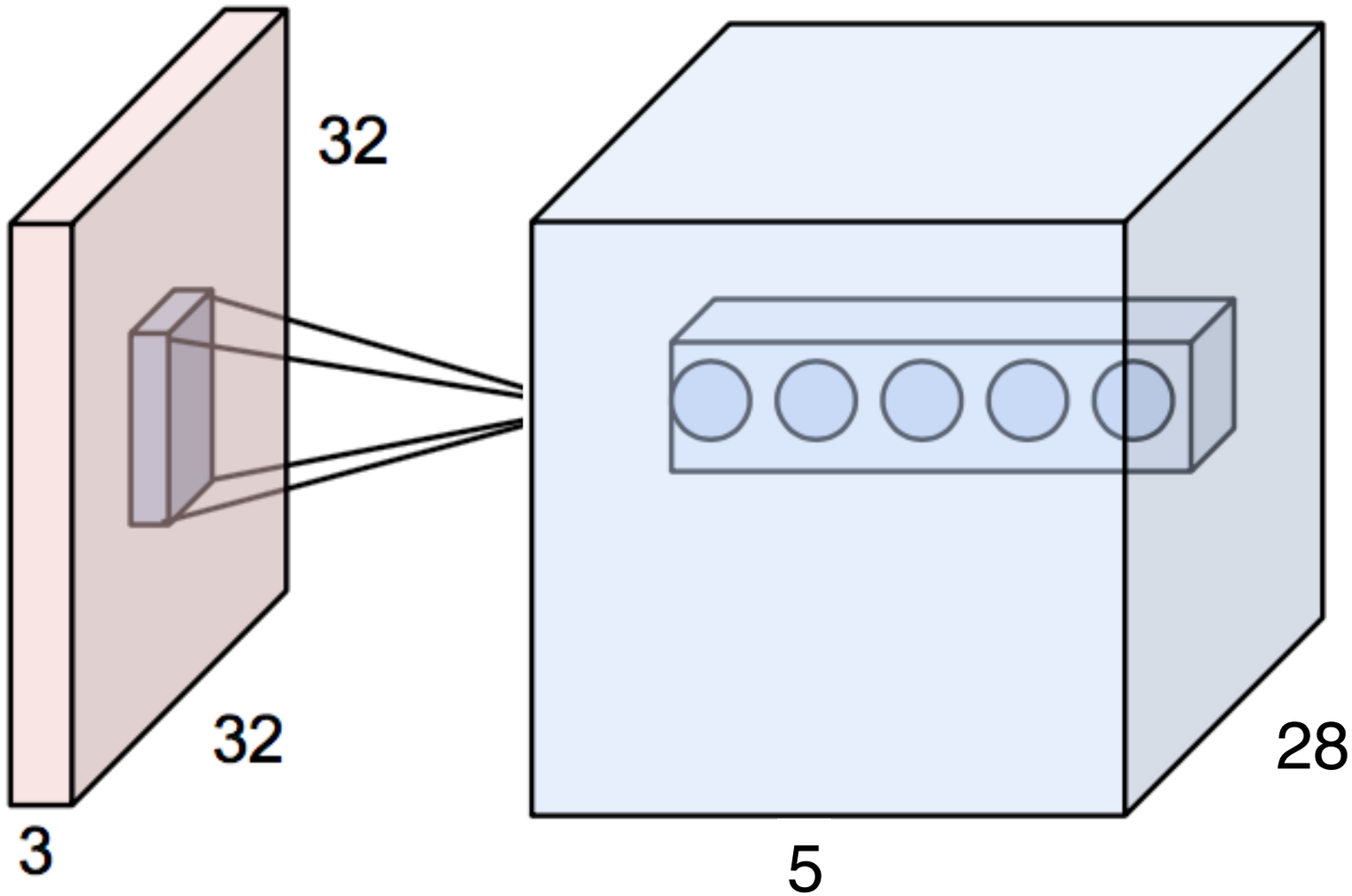
5x5 filter => 5x5 **receptive field**

Slide credit: Fei-Fei, Karpathy, Johnson

More on convolutional layers

- The brain/neuron view of conv layers:





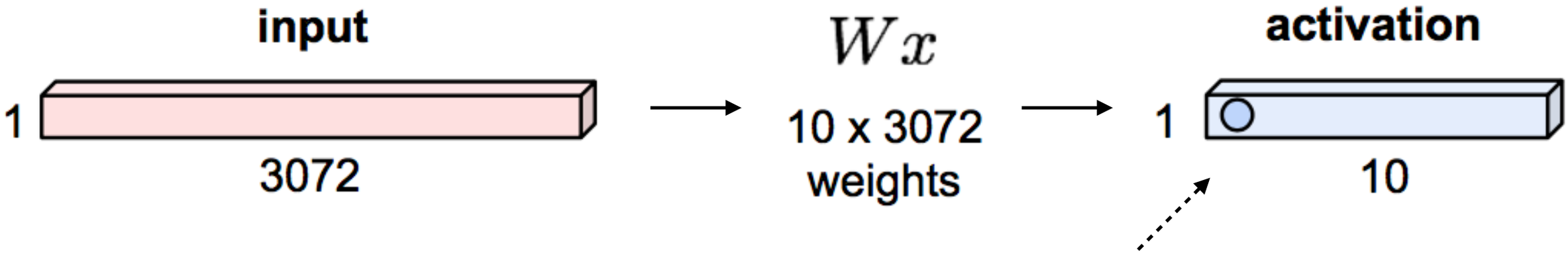
Output: neurons arranged in a 28x28x5 volume

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

Slide credit: Fei-Fei, Karpathy, Johnson

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

Slide credit: Fei-Fei, Karpathy, Johnson

Convolutional Neural Networks

- Visualization of a simple ConvNet architecture:

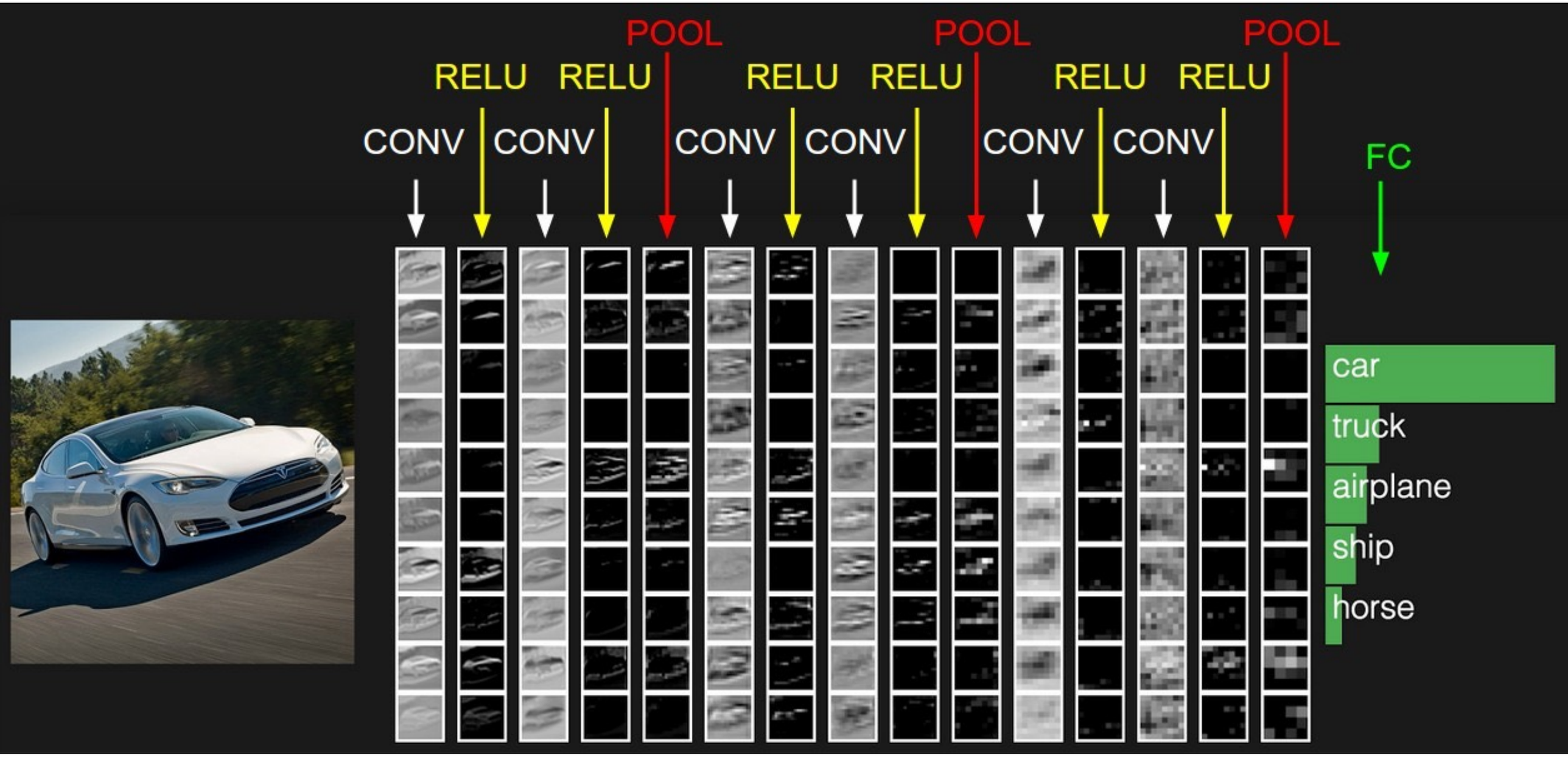
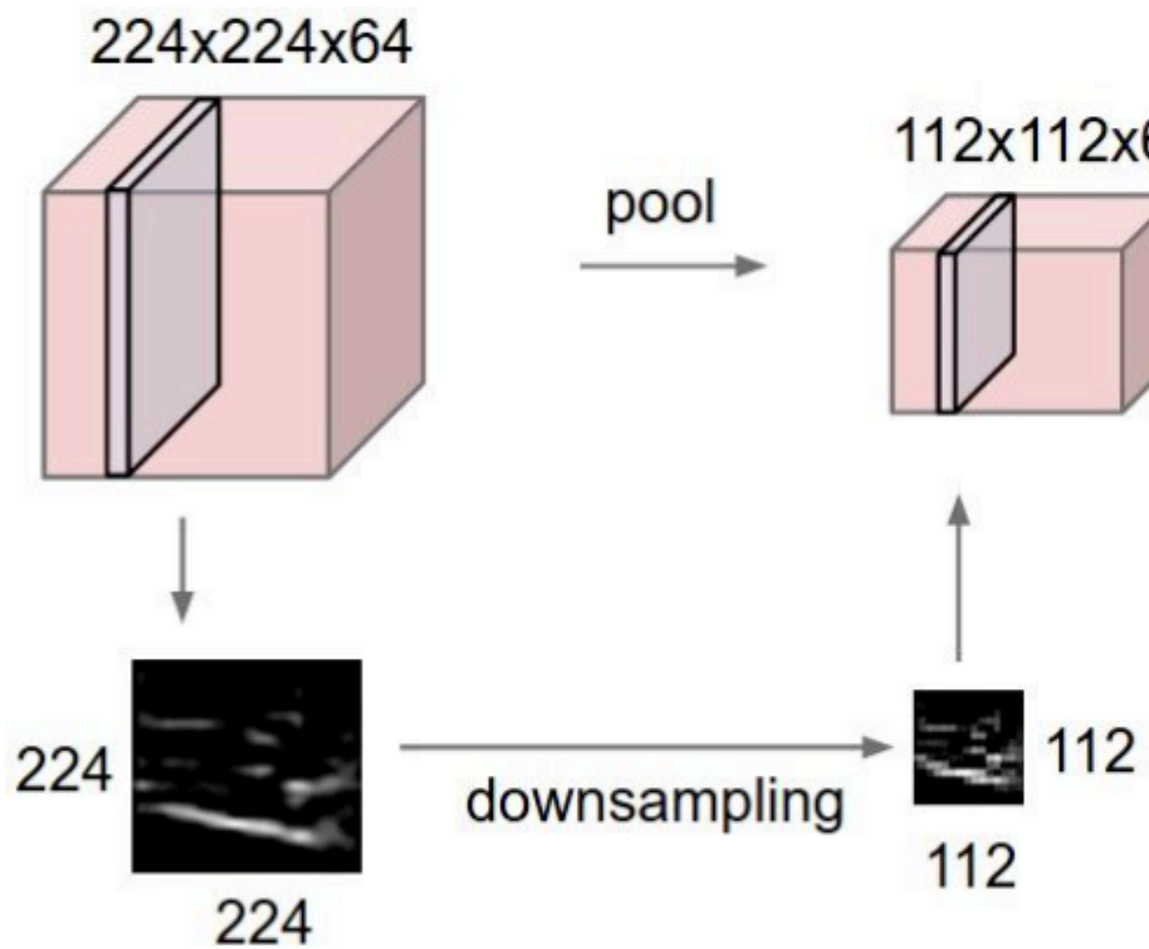


Image credit: Fei-Fei, Karpathy, Johnson

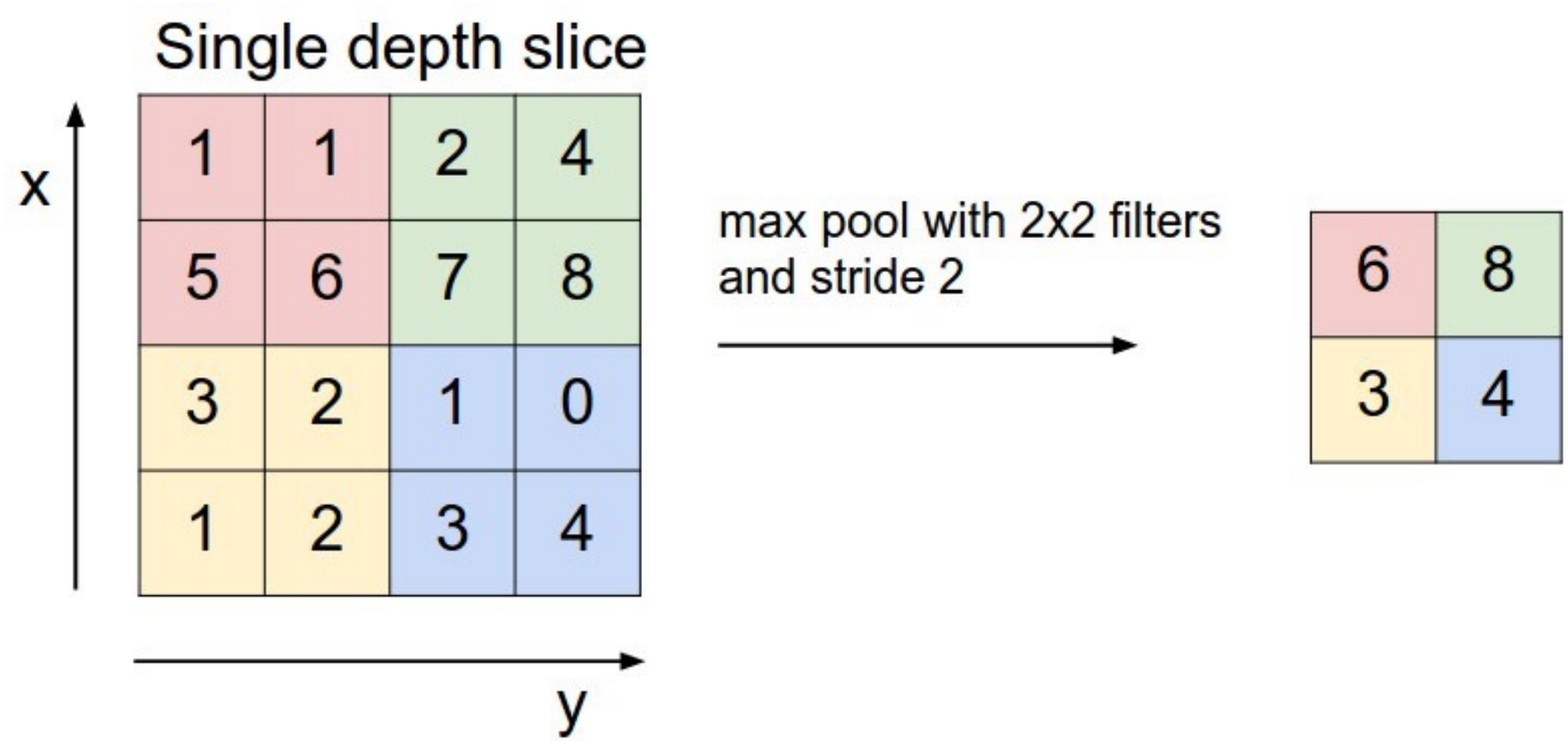
Pooling layer

- 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

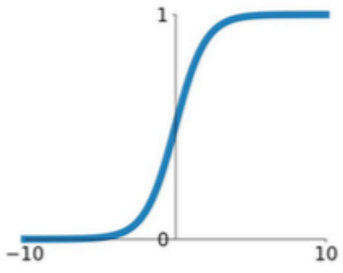
Neural Networks recap: activation functions

- **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:

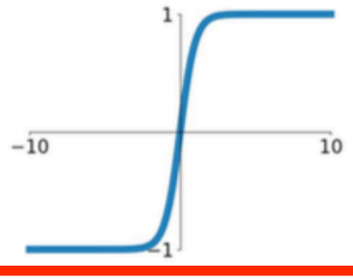
Sigmoid

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



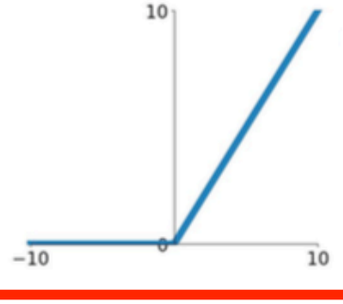
tanh

$$\tanh(x)$$



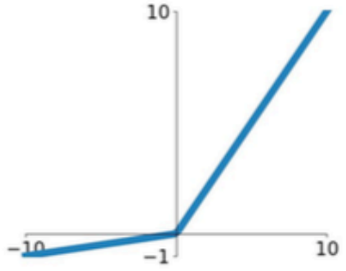
ReLU

$$\max(0, x)$$



Leaky ReLU

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

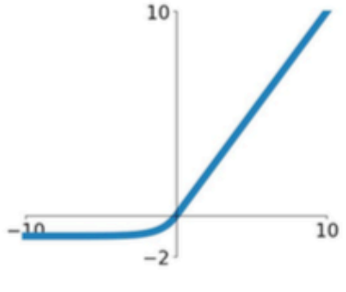


Maxout

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

ELU

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

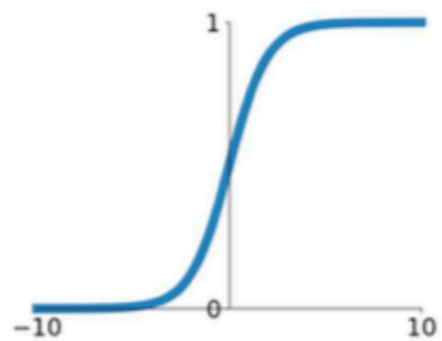


Activation functions: sigmoid

- 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

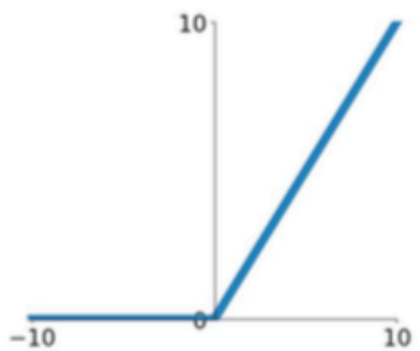
- Main problems:
  - Sigmoid saturate and kill the gradients
  - `exp()` is a bit expensive to compute
  - Sigmoid outputs are not zero-centered (fixed by tanh)

Activation functions: ReLu

- ReLU: Rectified Linear Unit

ReLU

$$\max(0, x)$$

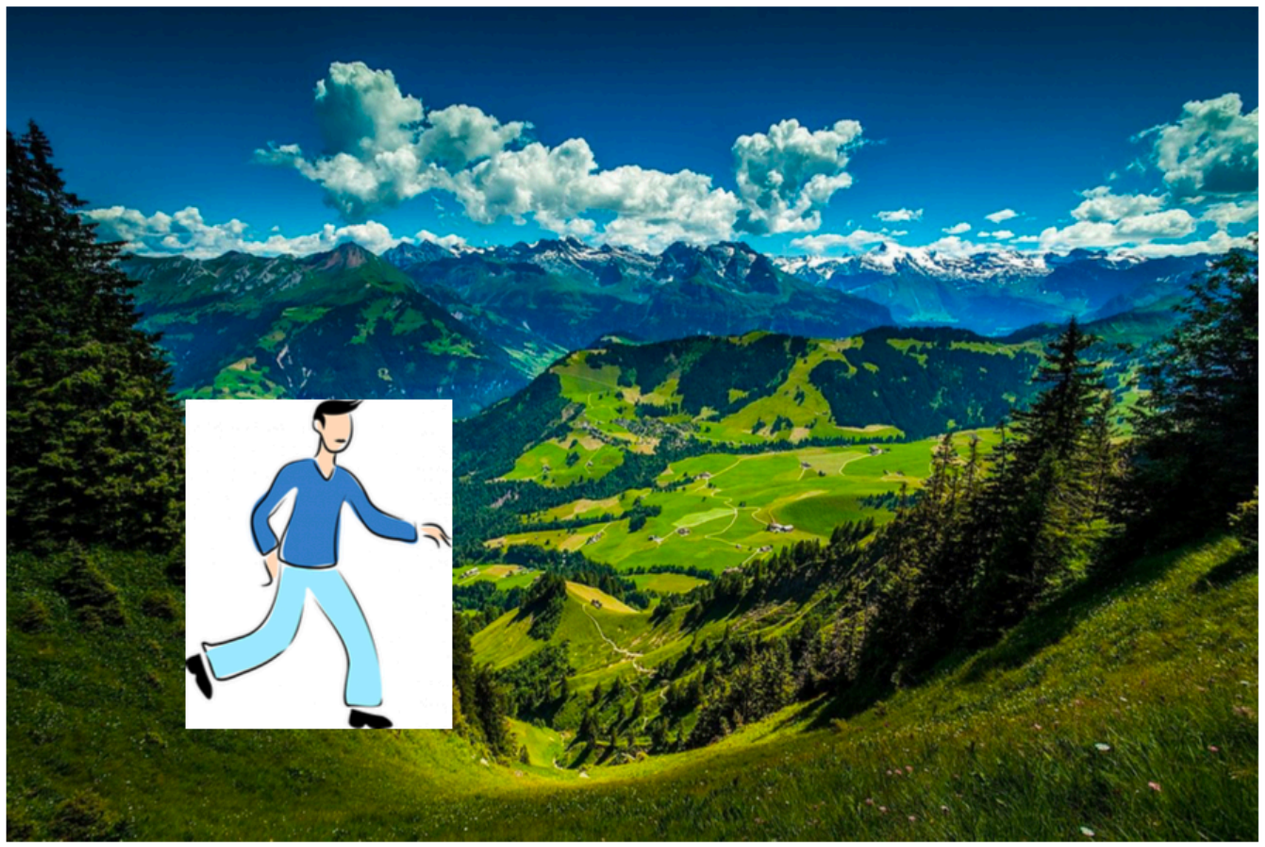


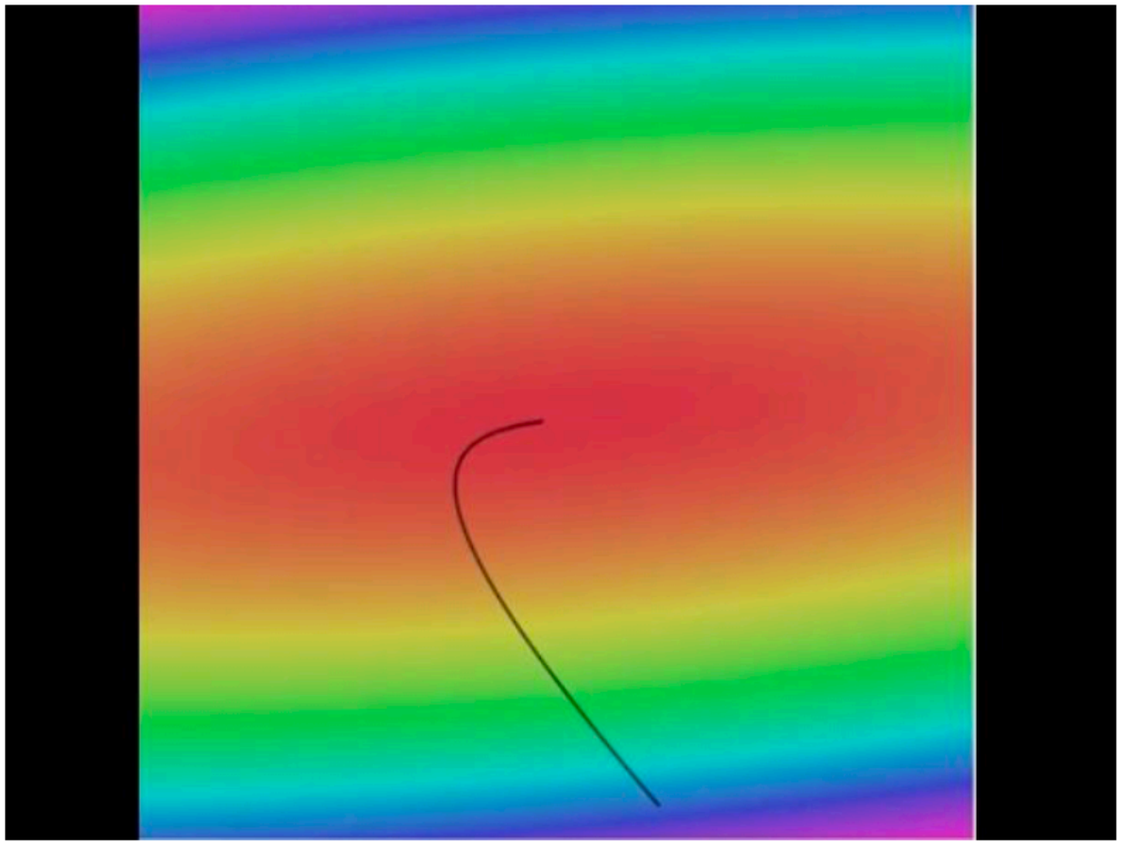
  - Does not saturate (in +region)
  - Very computationally efficient
  - Converges much faster than sigmoid/tanh (e.g. 6x)
  - Actually more biologically plausible than sigmoid

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

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
```

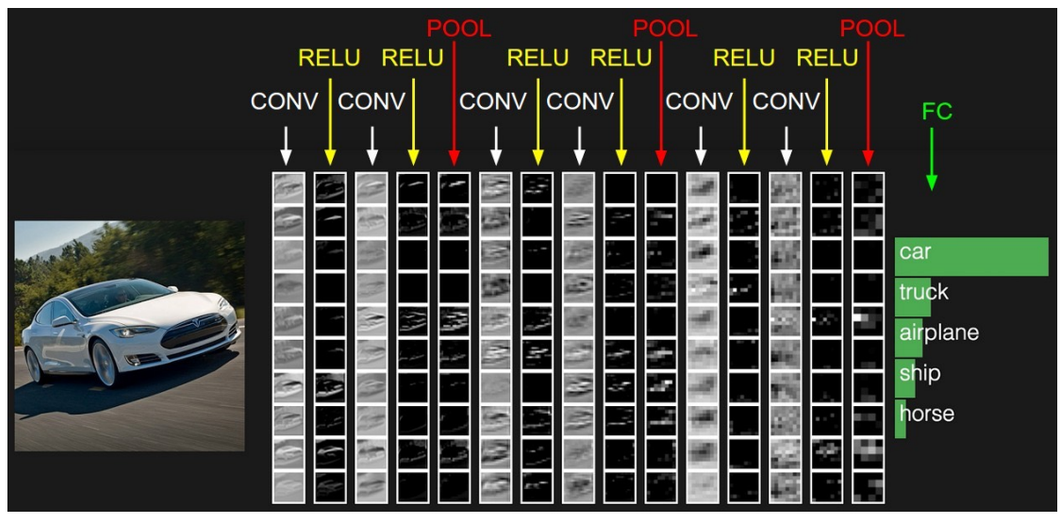

Slide credit: Fei-Fei, Karpathy, Johnson

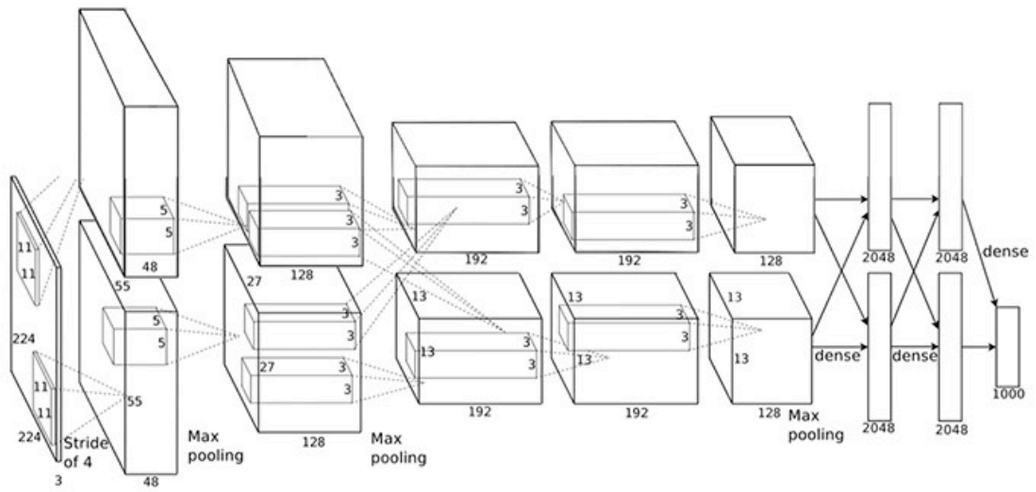
Training a (Conv) Neural Network

- 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

CNN Architectures: summary

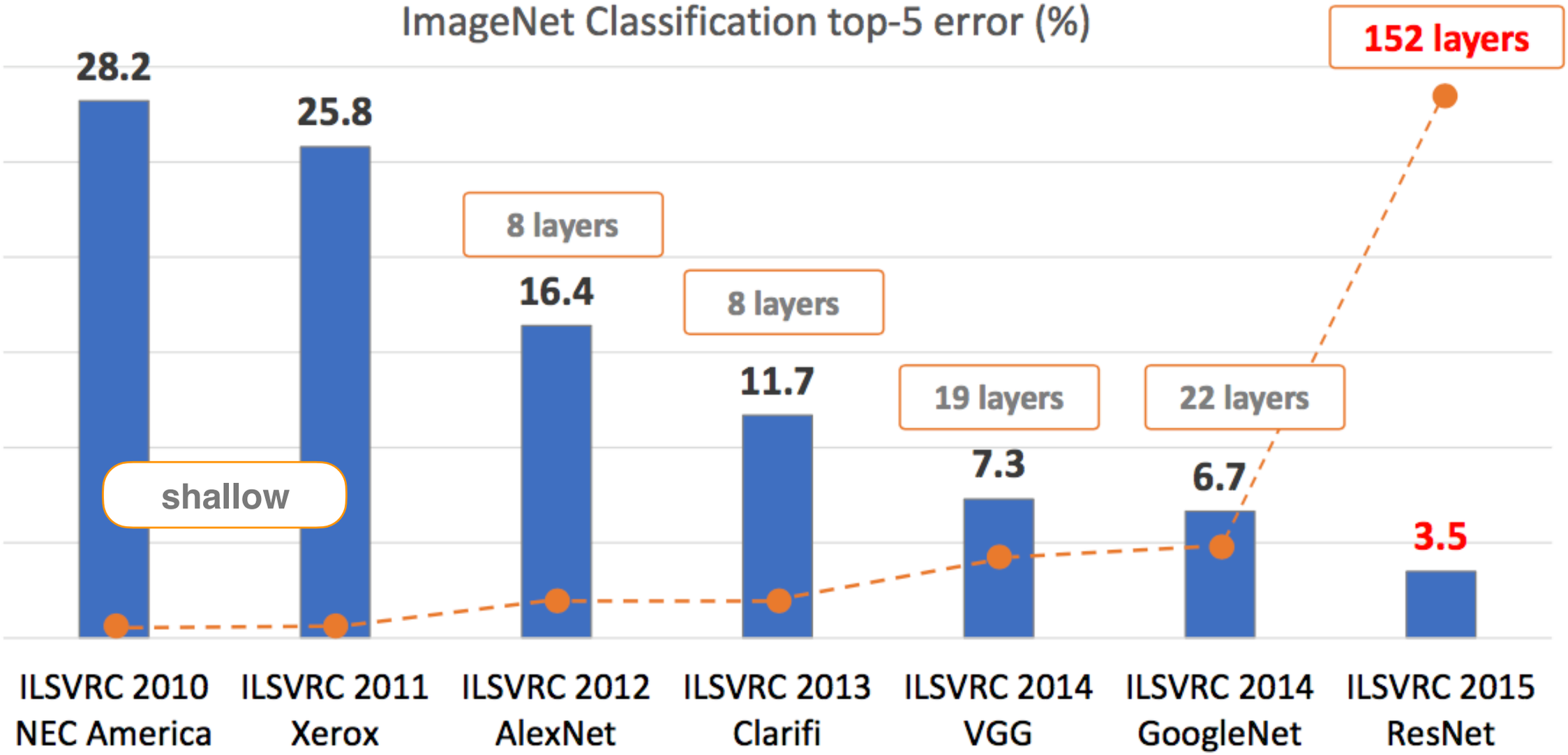
- ConvNets stack CONV,POOL,FC layers
- Typical architectures look like:
  - [(CONV,RELU)*N,POOL]*M,(FC,RELU)*K,SOFTMAX where N is usually up to ~5, M is large, 0≤K≤2
  - but recent architectures such as ResNet and GoogleNet (Inception) challenge this paradigm





The rise of deep learning

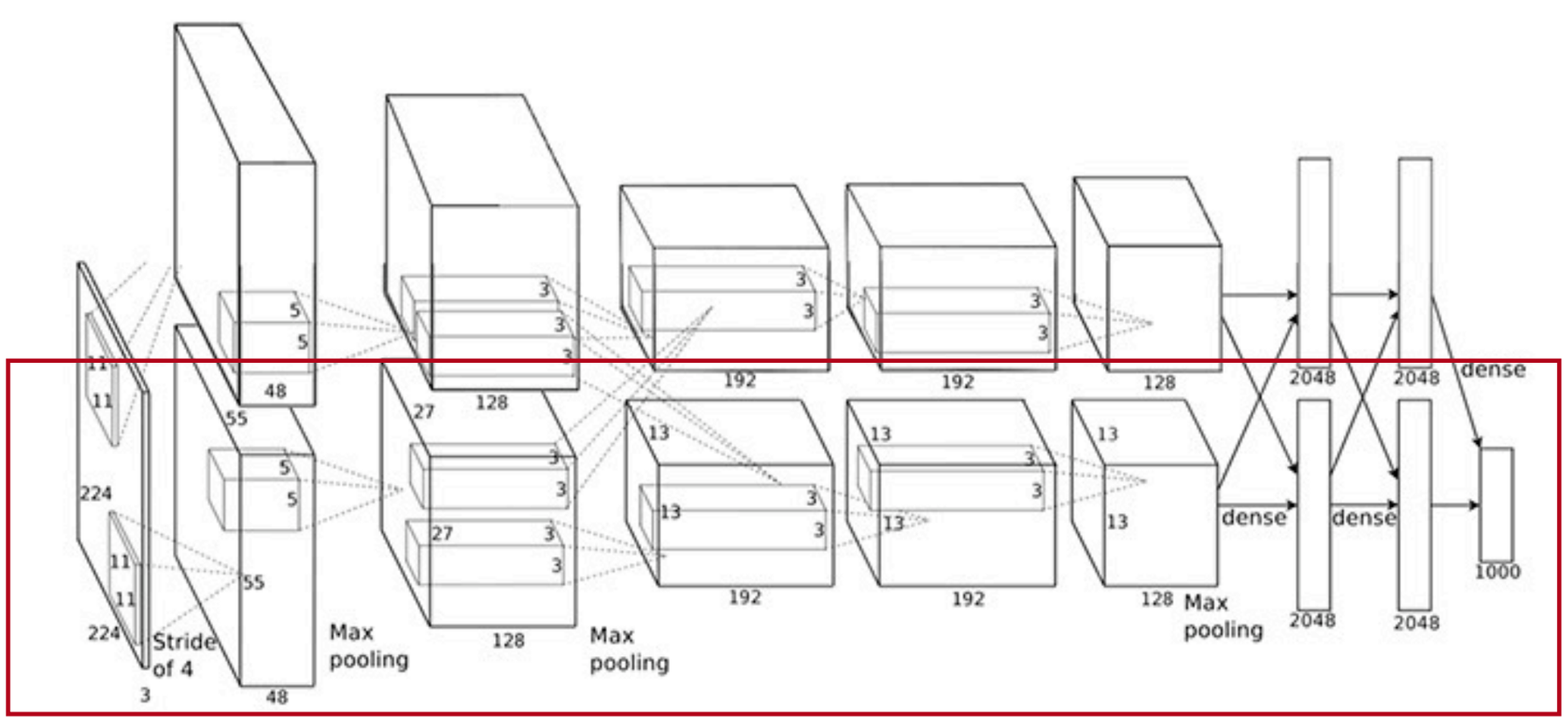
- Revolution of depth: how deep is deep?



Case study: AlexNet

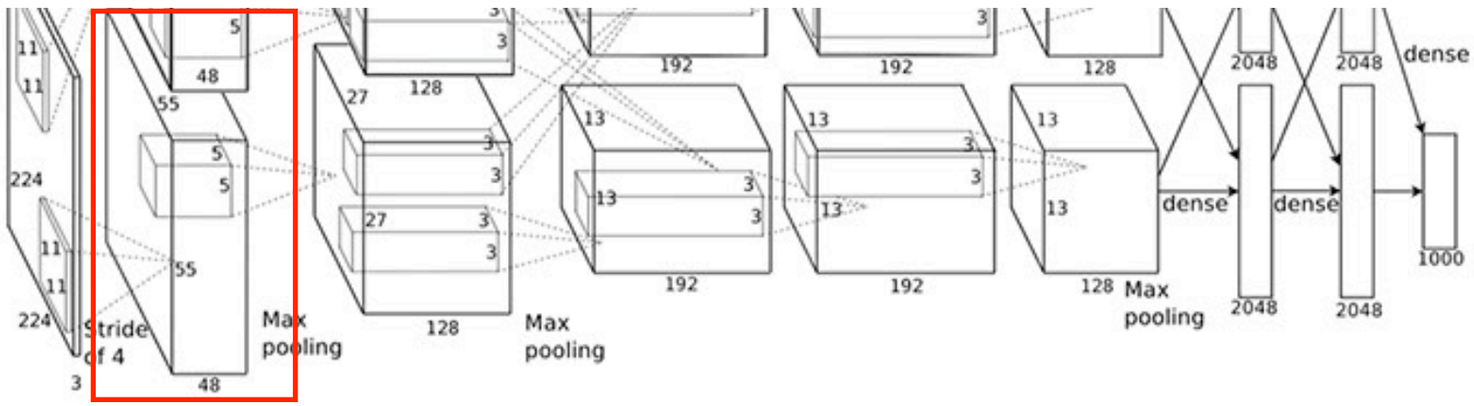
- Return of the CNN: first strong “modern” results

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

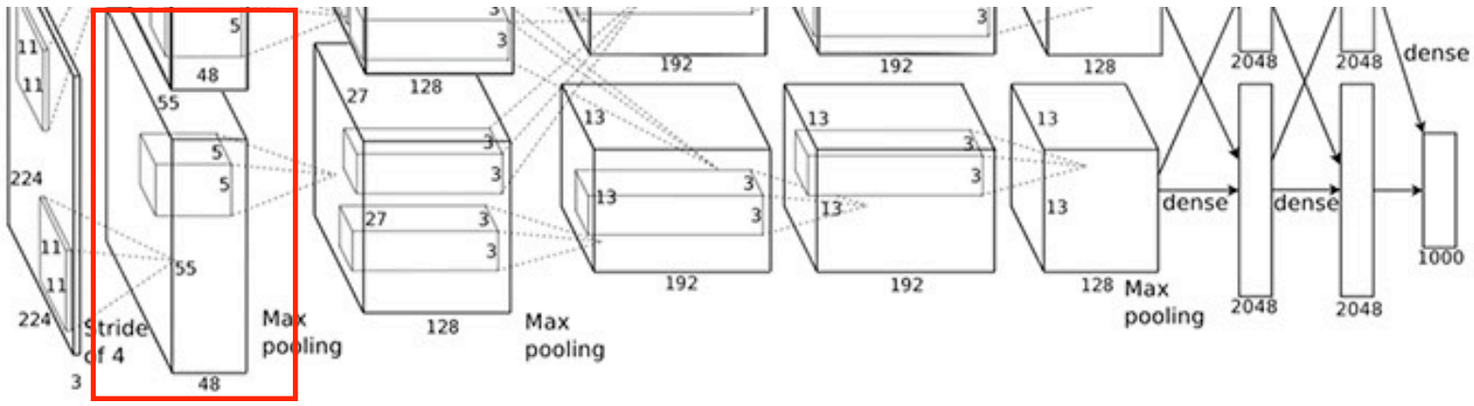
Recall: conv spatial arrangement

- Three hyper-parameters control the size of the output volume: the *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

Previous example:

*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)

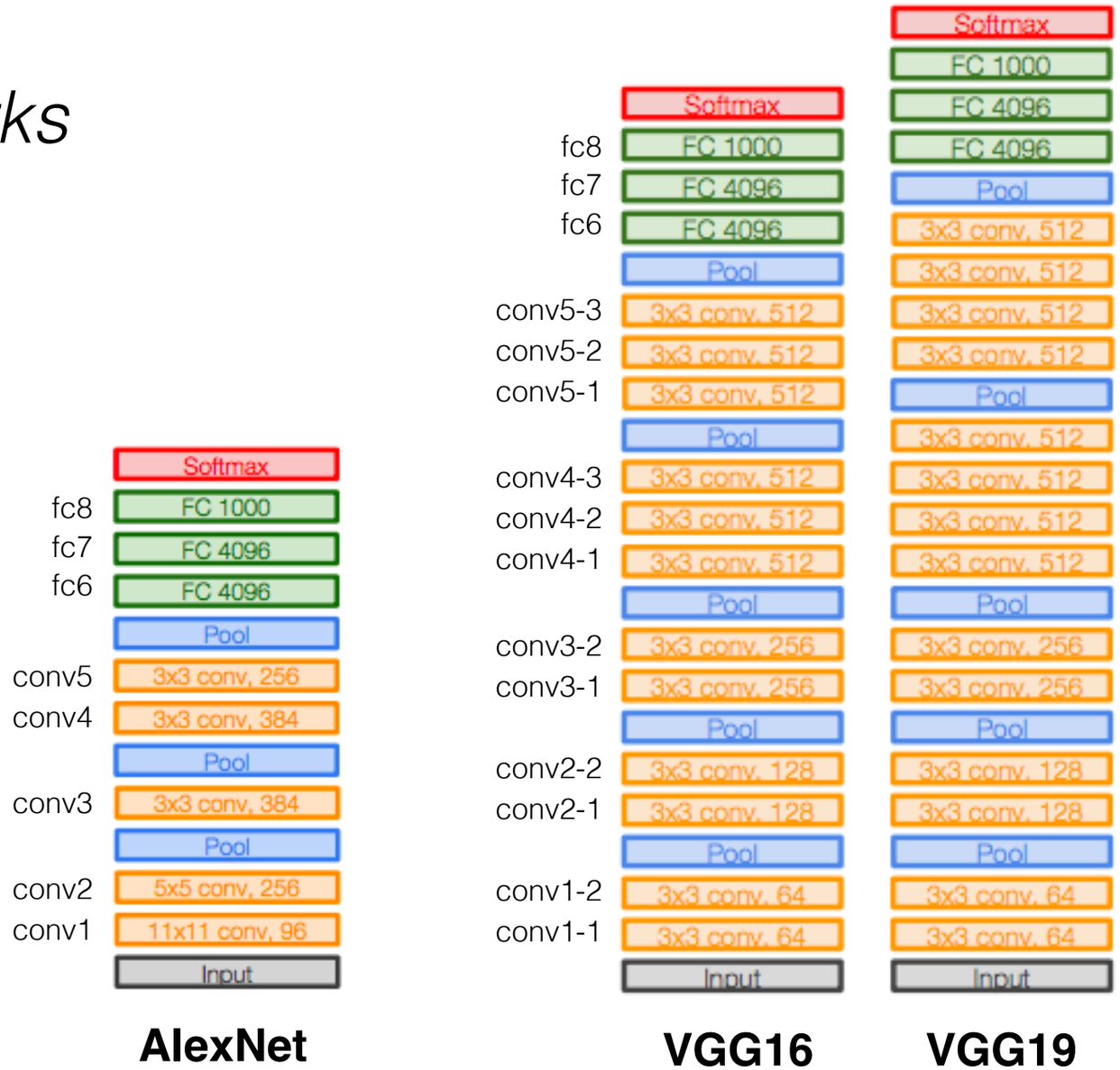
K.Simonyan, A.Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition”, ICLR 2015

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

Case study: VGGNet

VGGNet *vs* AlexNet:

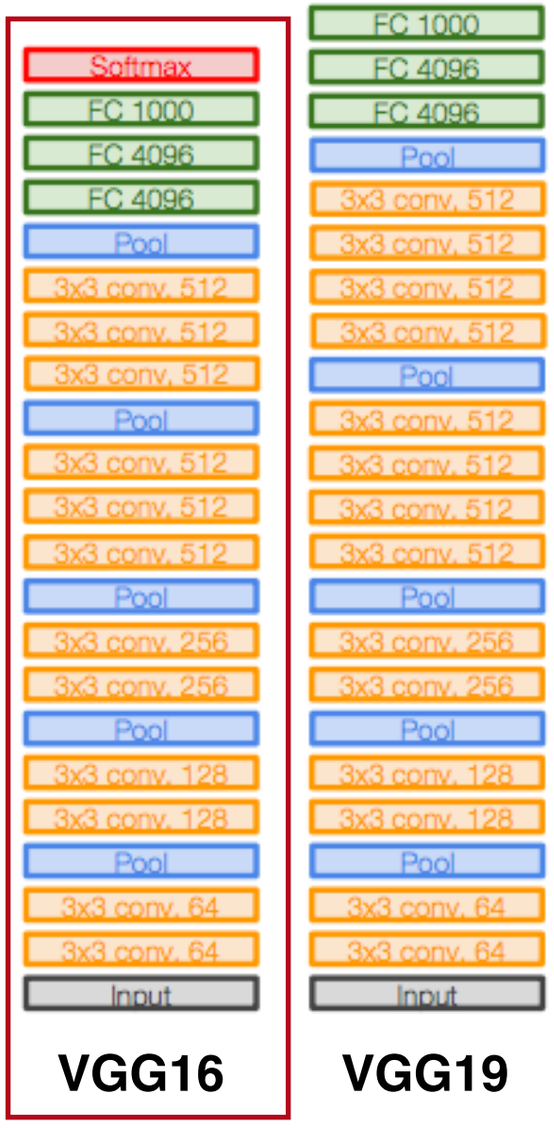
Small filters, Deeper networks



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

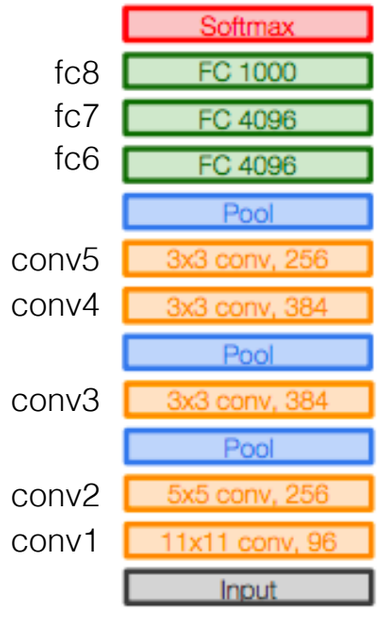
Case study: GoogLeNet (Inception)

Deeper networks, with computational efficiency

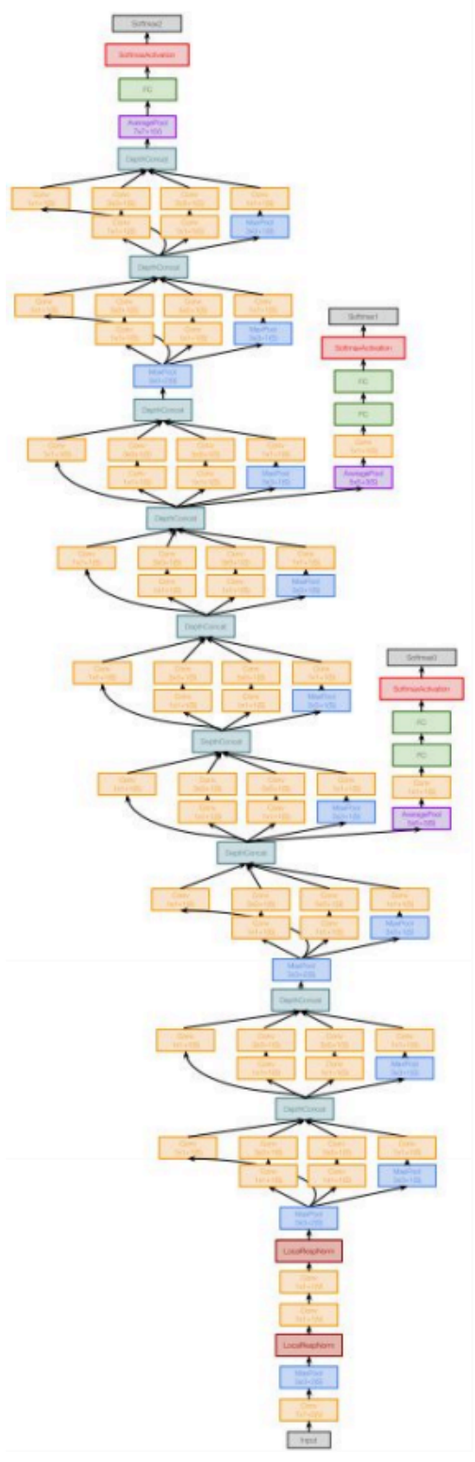
22 layers with no FC layers

Efficient "Inception" module

5M params (12x less AlexNet)



AlexNet

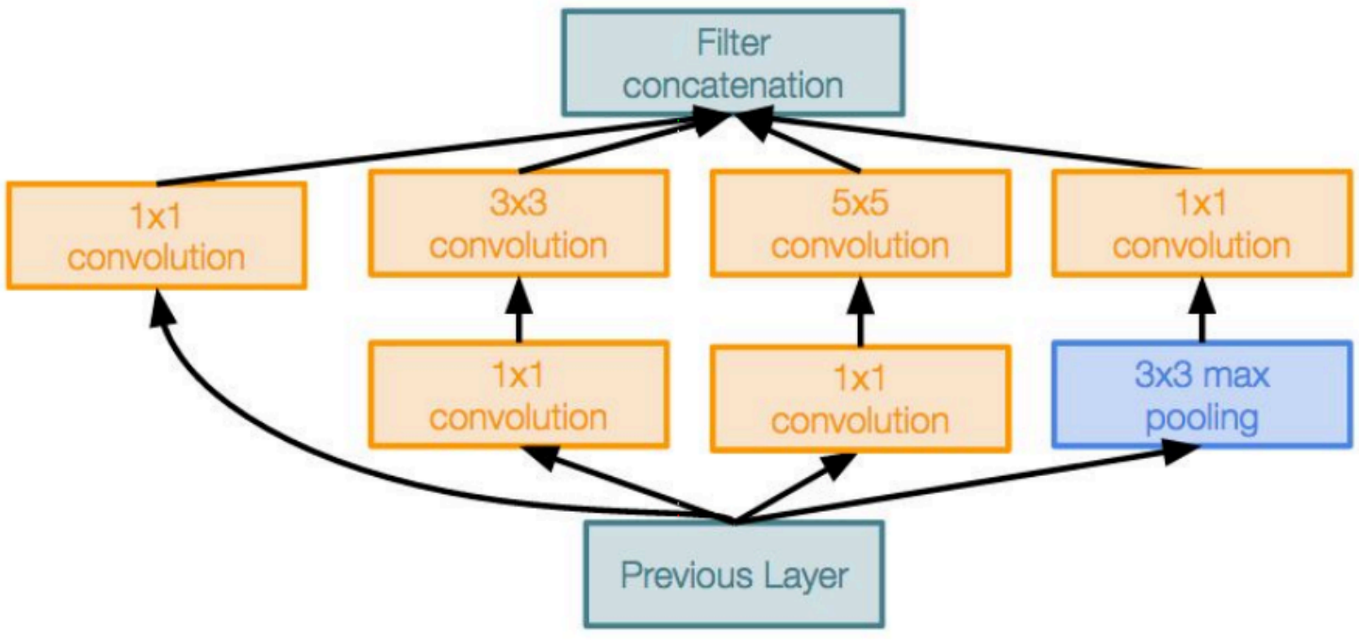


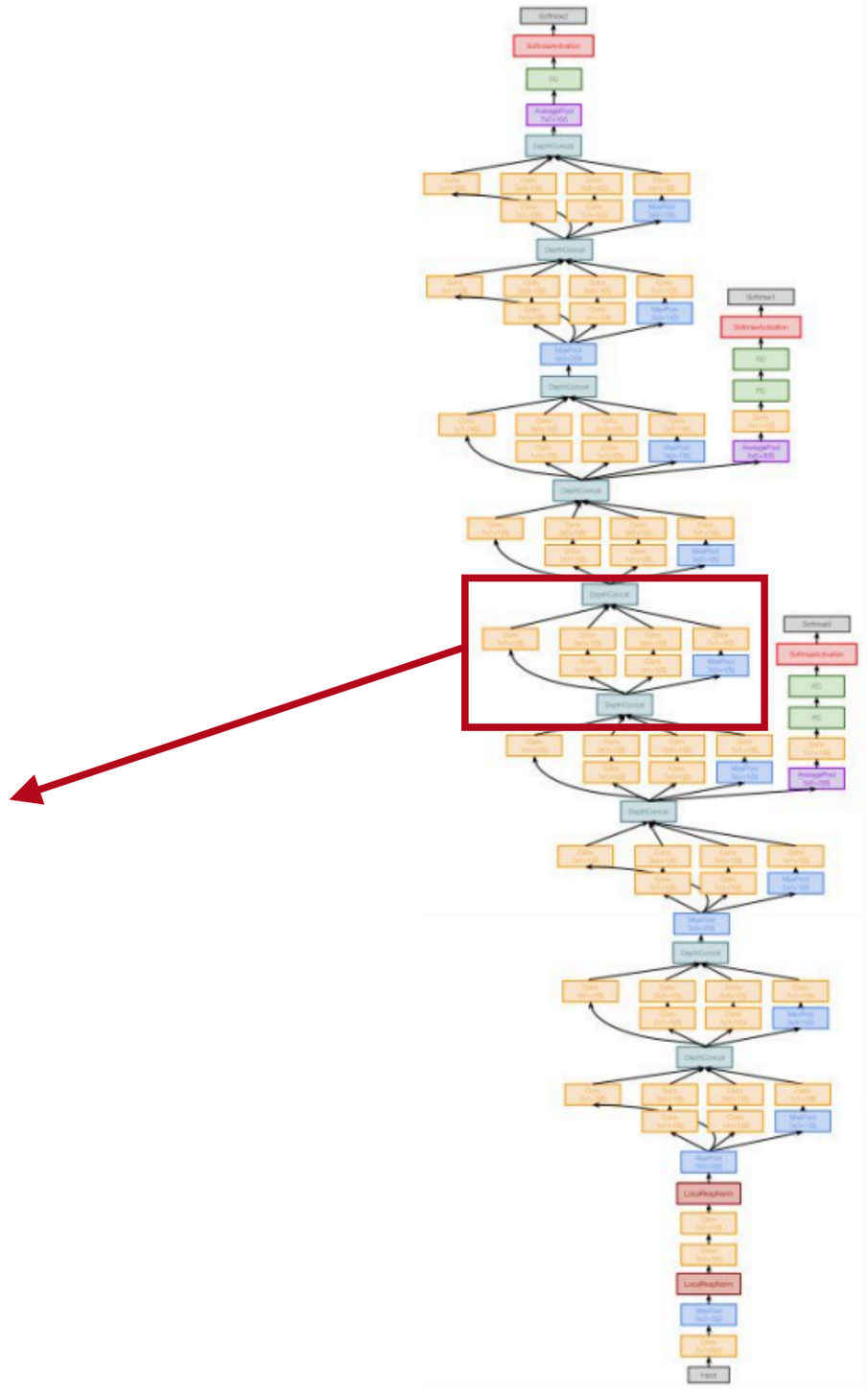
GoogLeNet

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



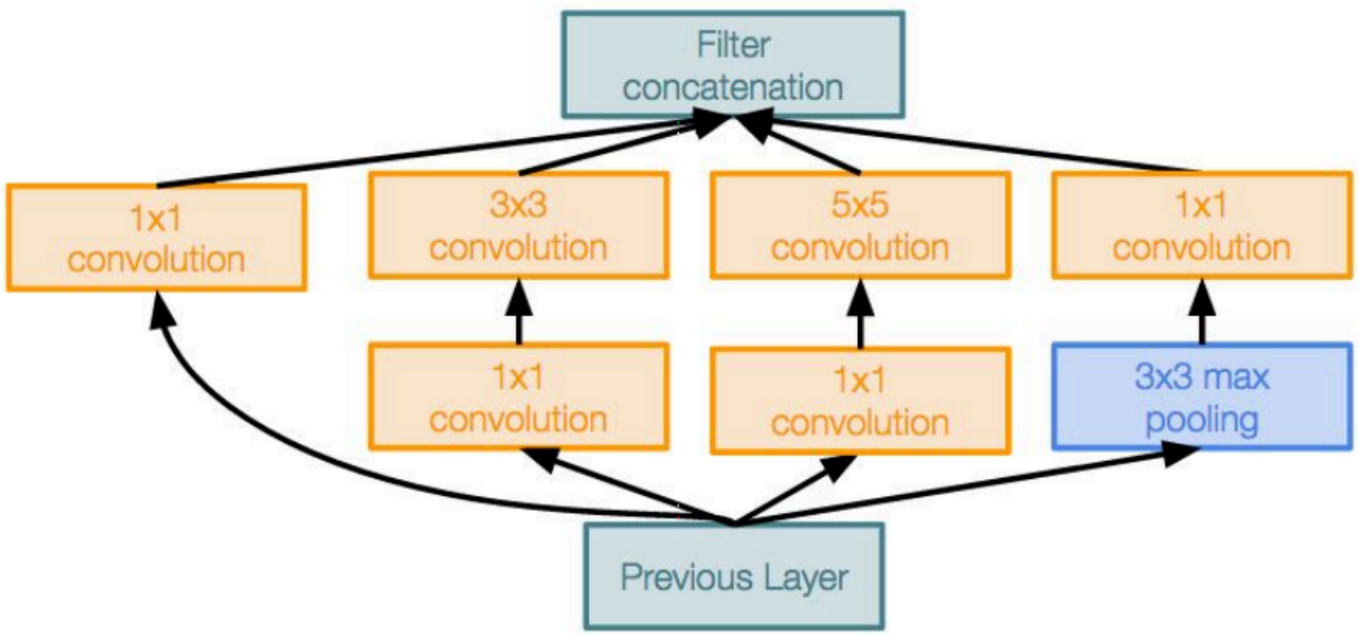


GoogLeNet

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



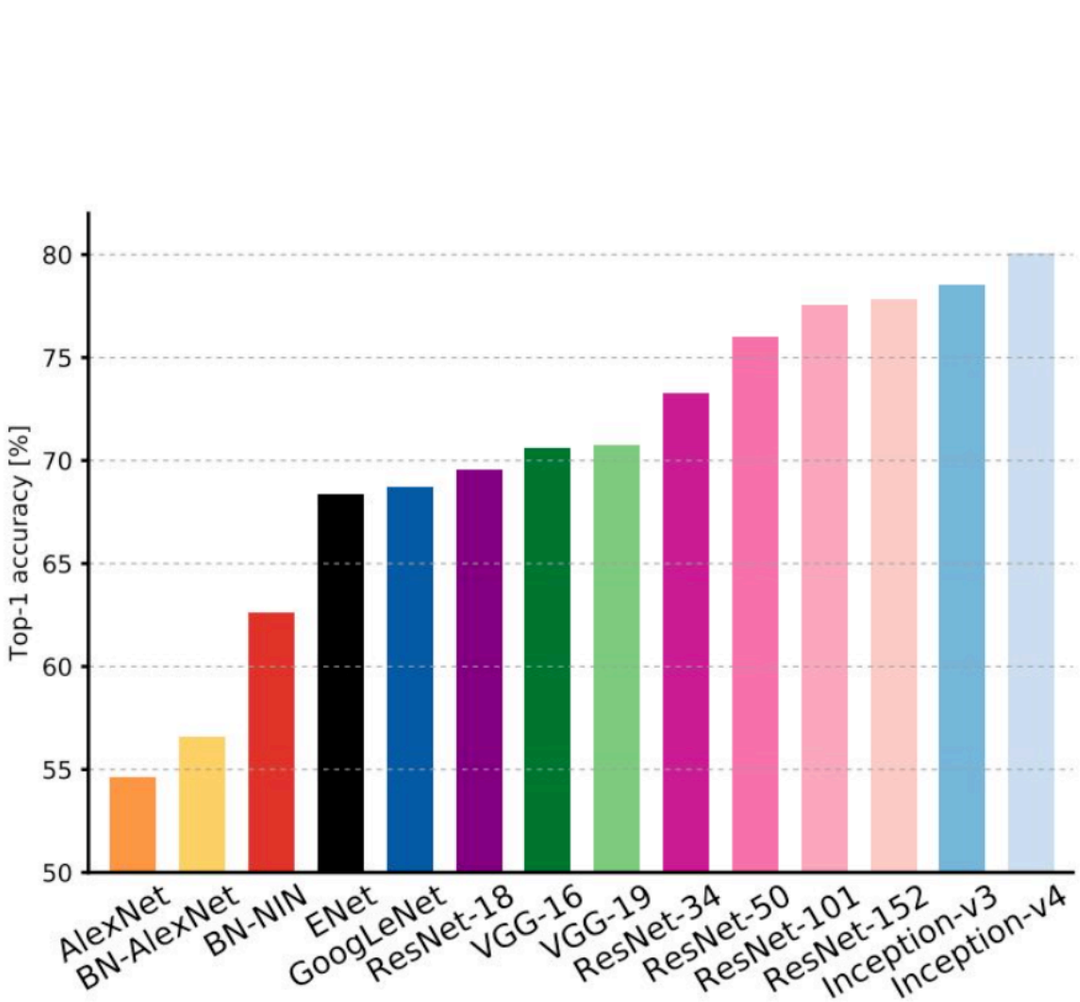
Apply parallel filter operations on the input layer:

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

Concatenate all filter outputs together depth-wise

CNN Architectures

- Comparing complexity:



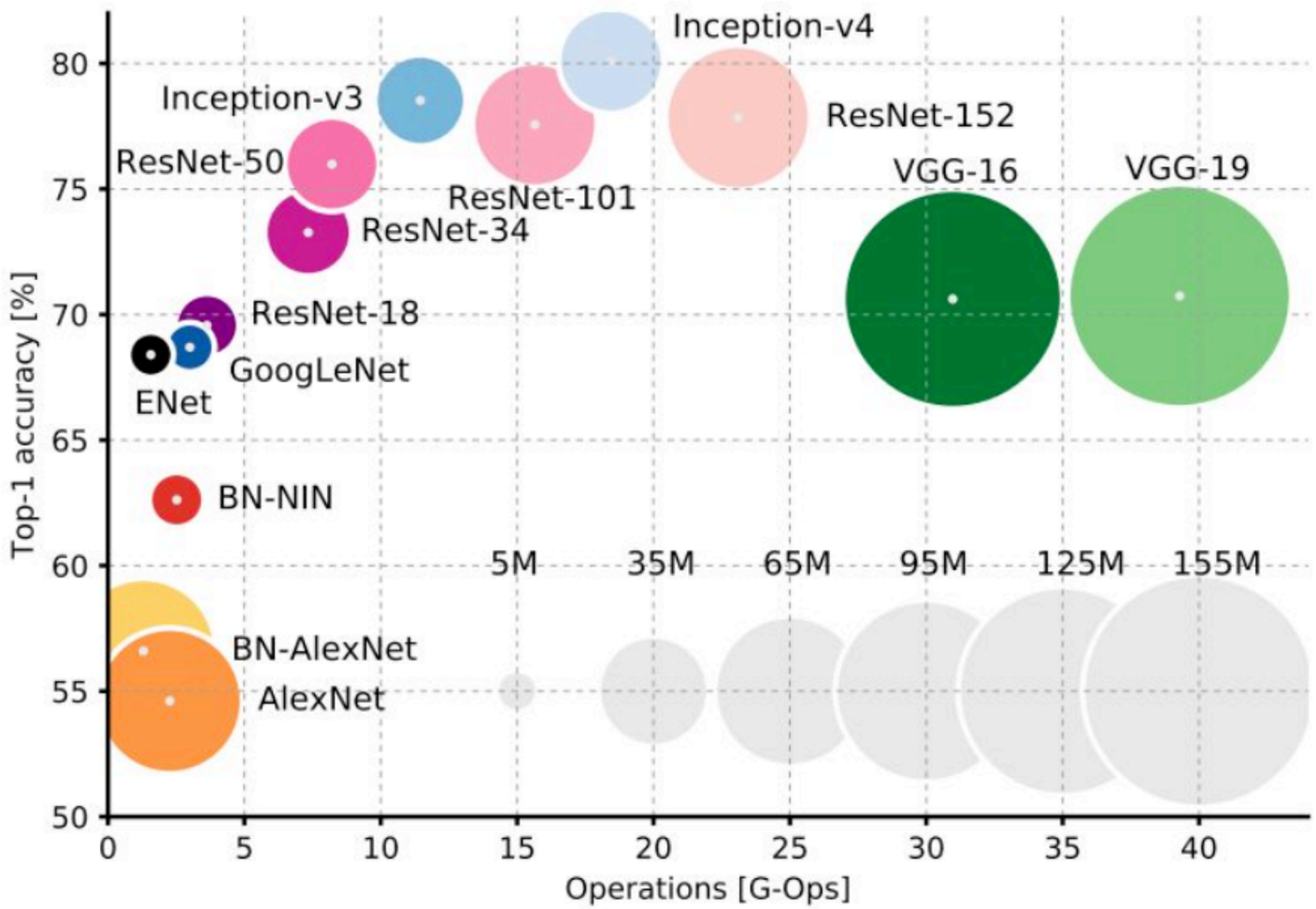
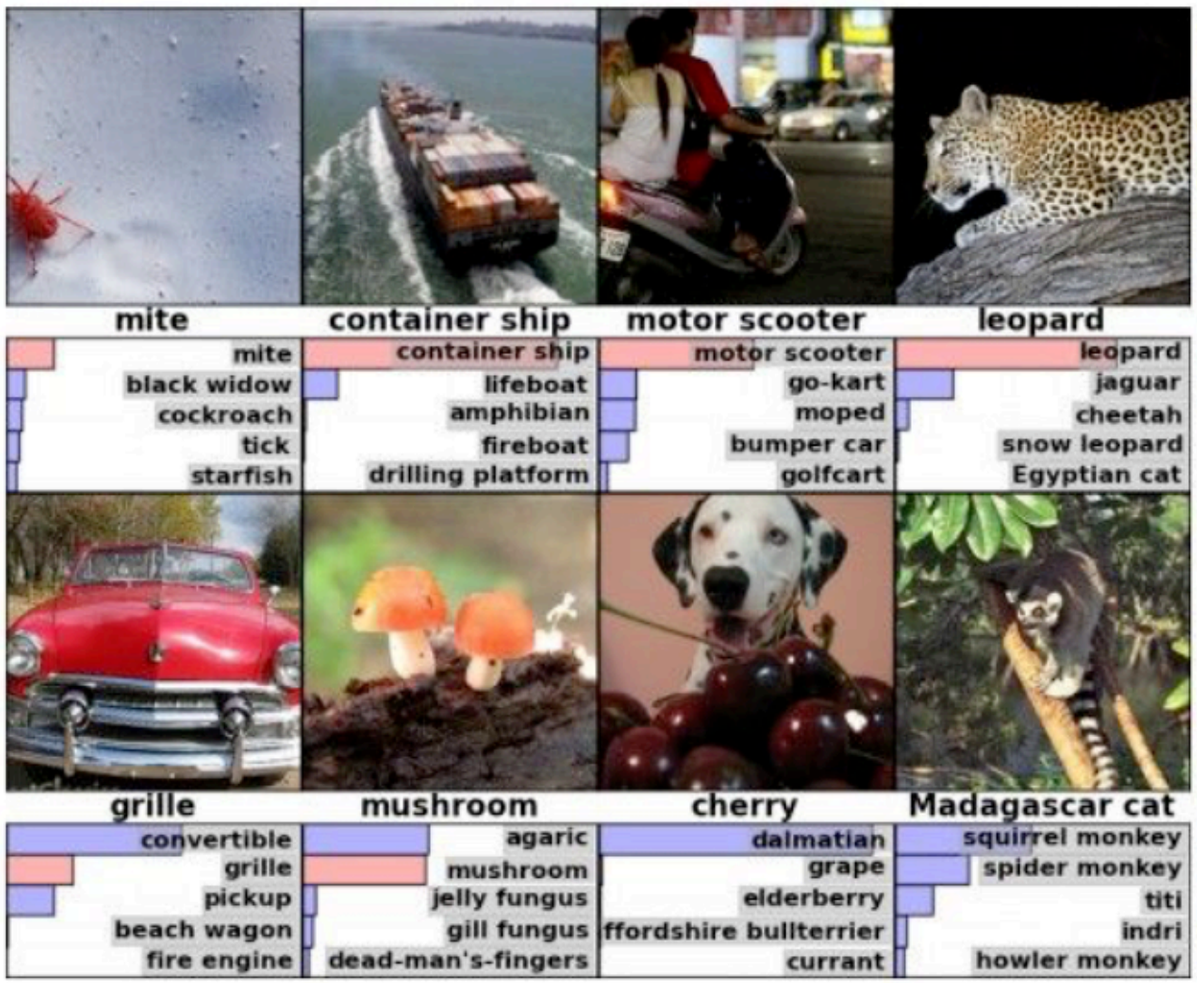


Image credit: Canziani, Culurciello, Paszke

CNNs are everywhere…

Classification



Retrieval

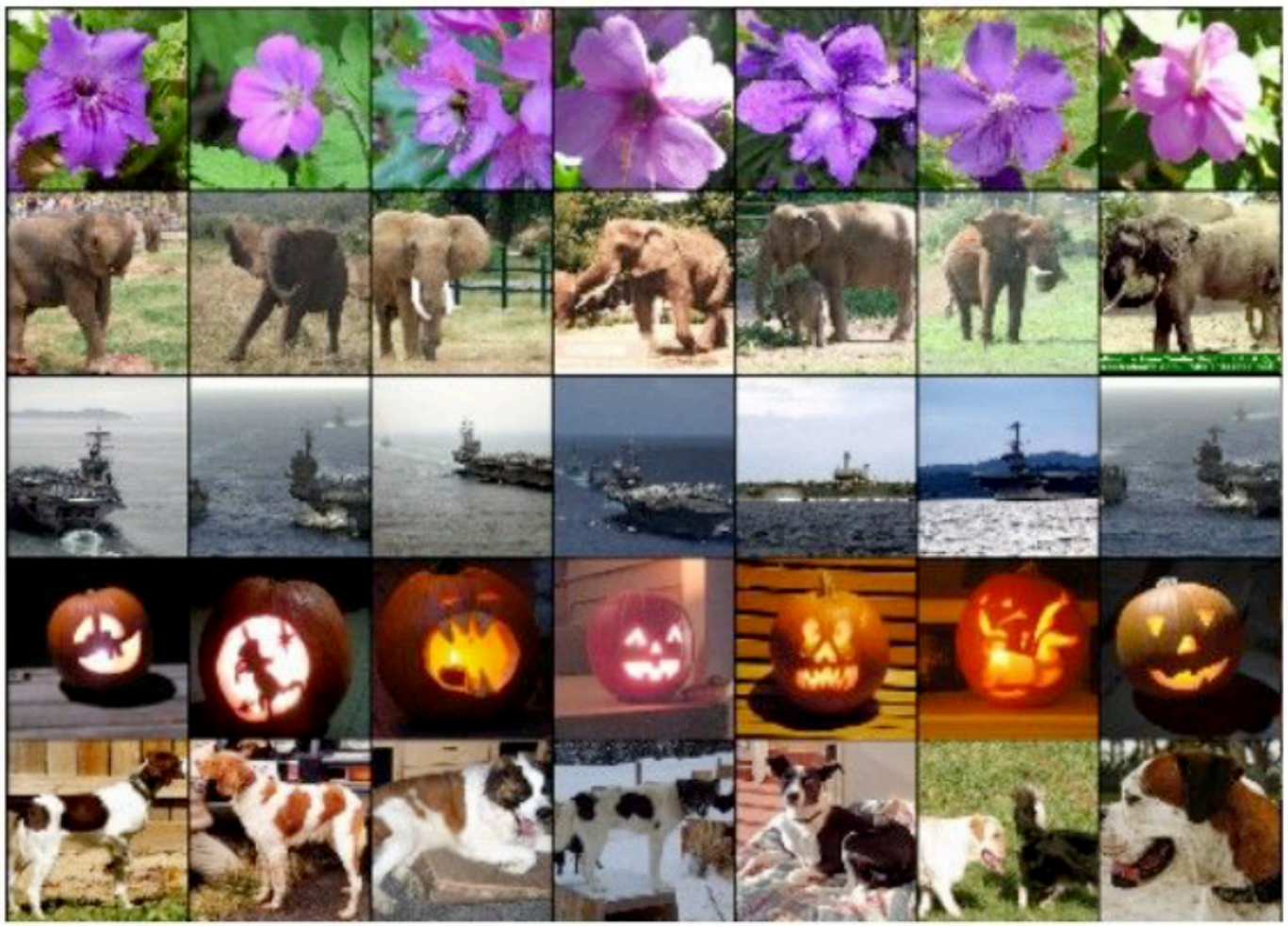
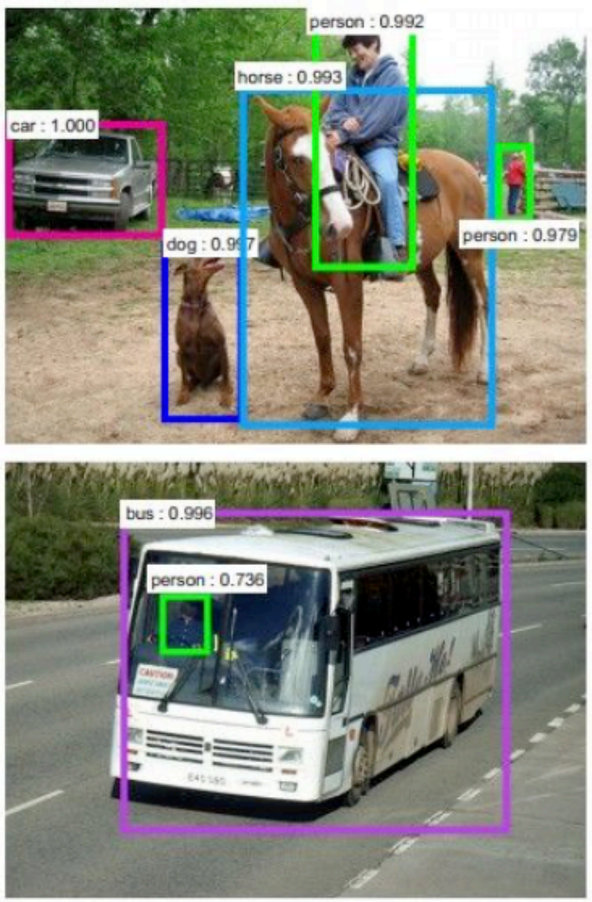


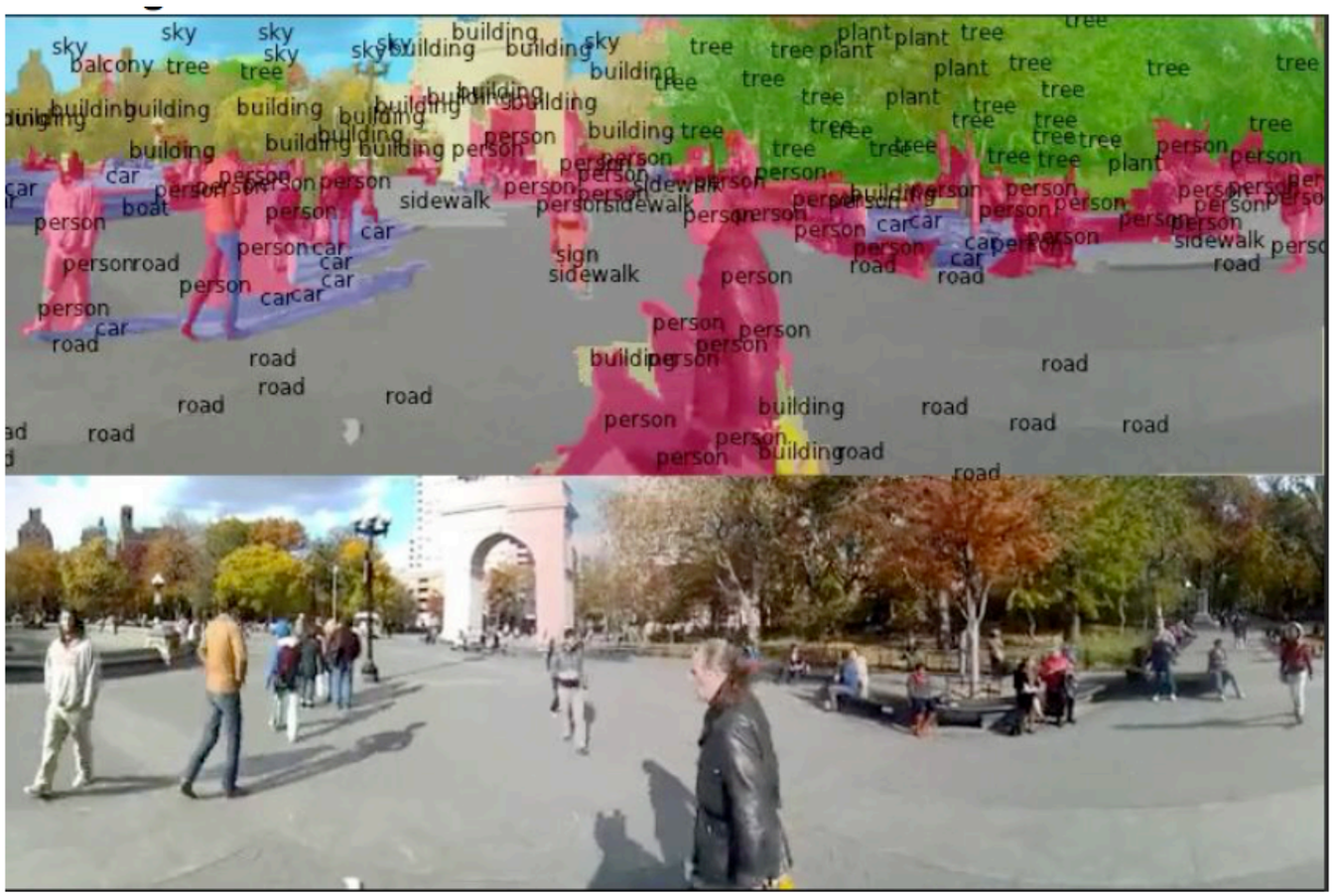
Image credit: Fei-Fei, Karpathy, Johnson

CNNs are everywhere…

Detection

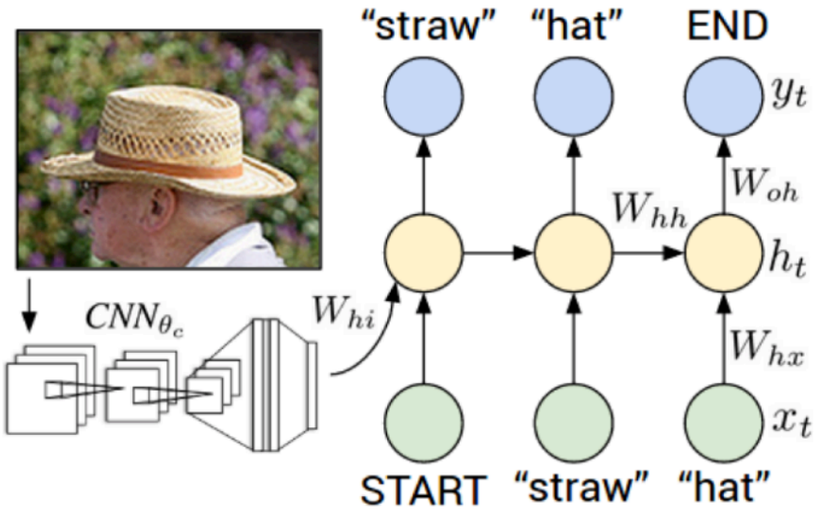
Segmentation





CNNs are everywhere…

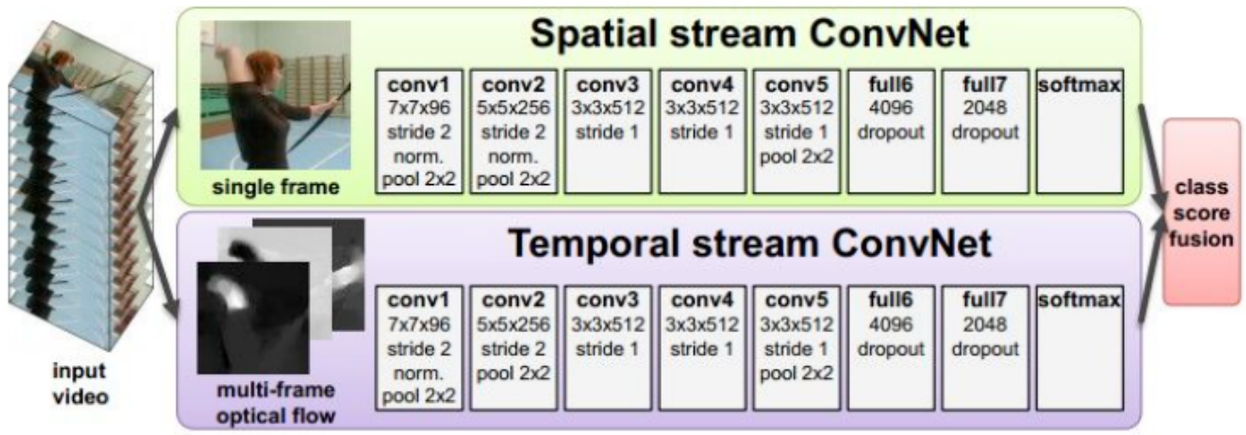
Captioning



"black and white dog jumps over bar."

"two young girls are playing with lego toy."

Video understanding



ACTIVITYNET

CNNs are everywhere: applications

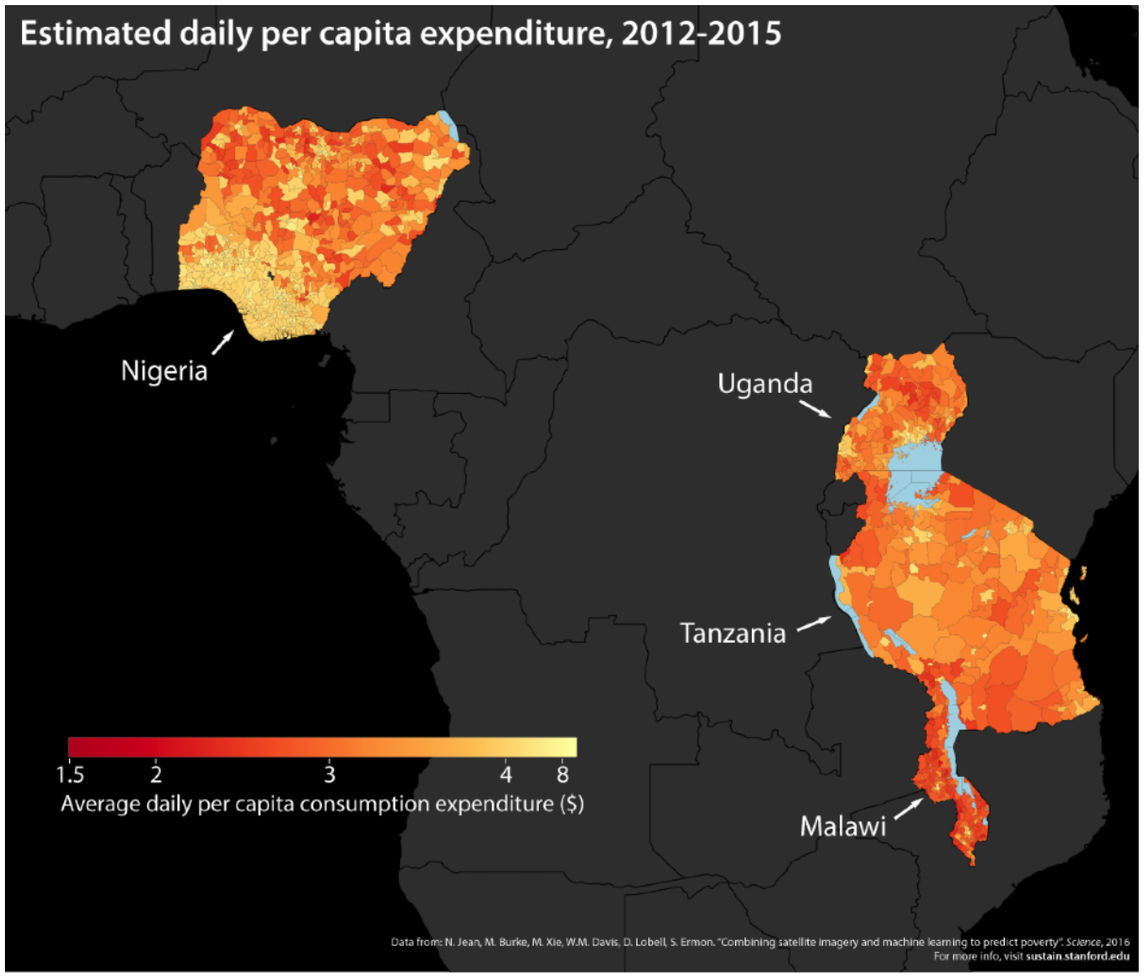
Intelligent transportation

Style transfer

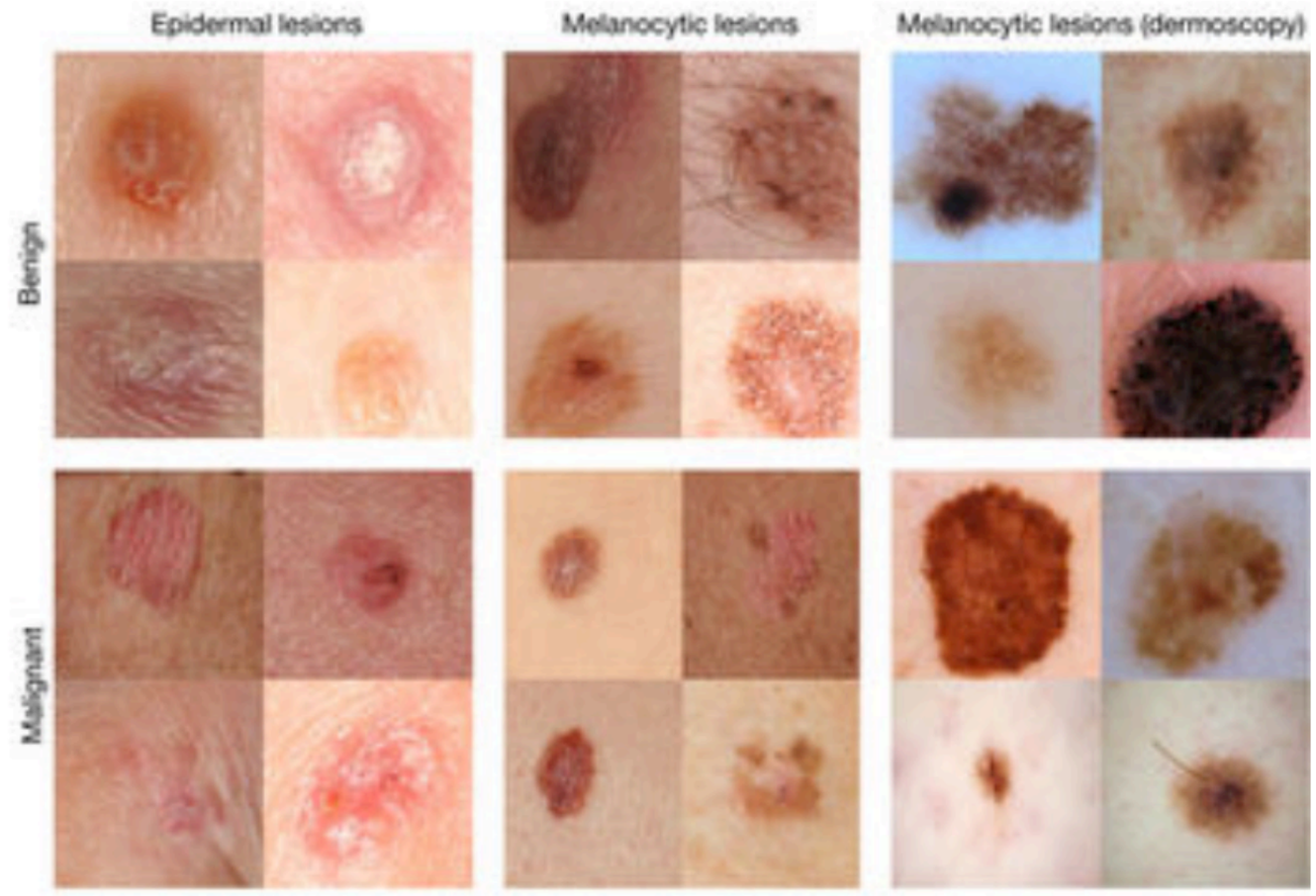
Image credit: Fei-Fei, Karpathy, Johnson

CNNs are everywhere: applications

Monitoring environmental changes

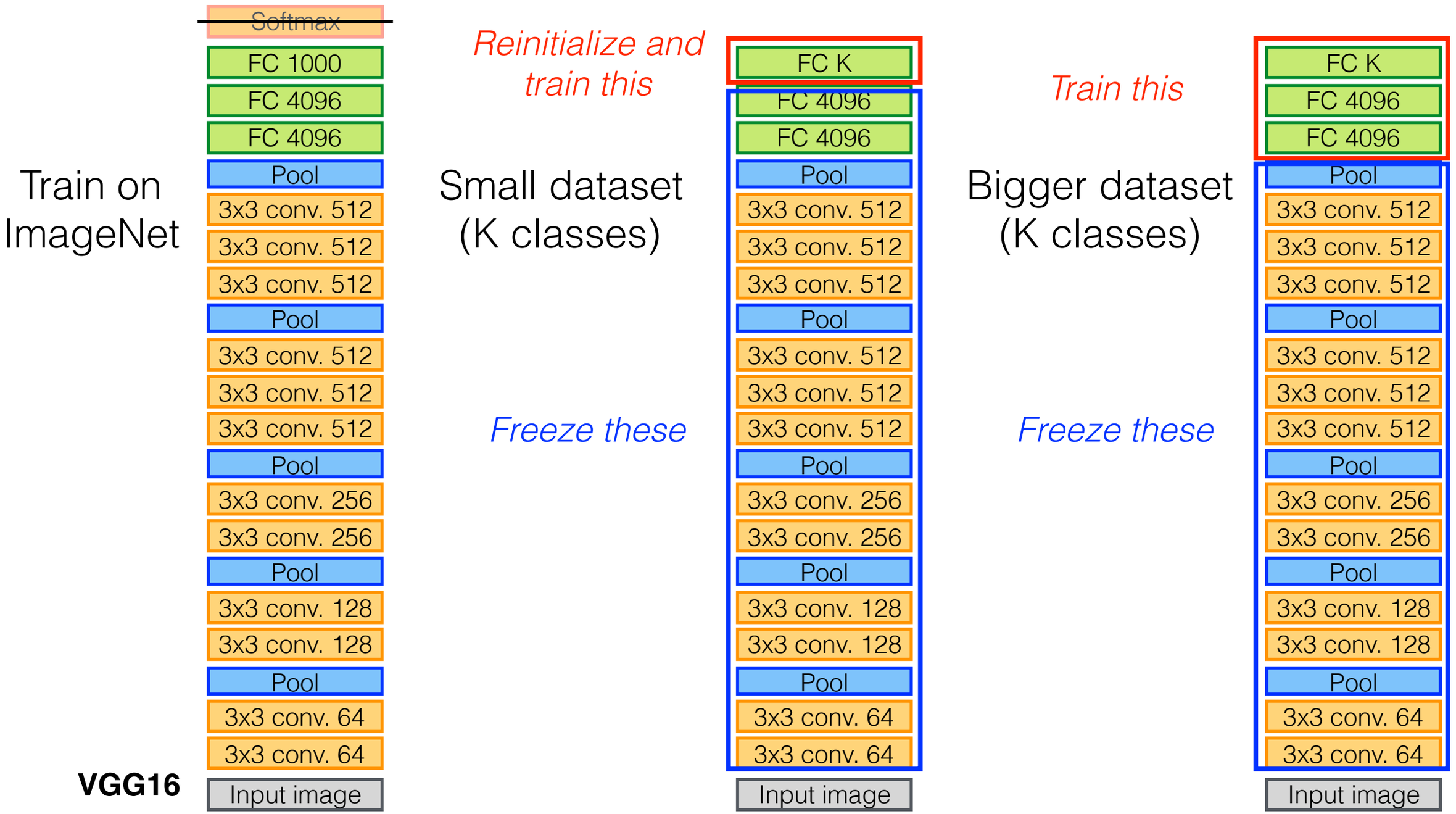


Medical imaging / Healthcare



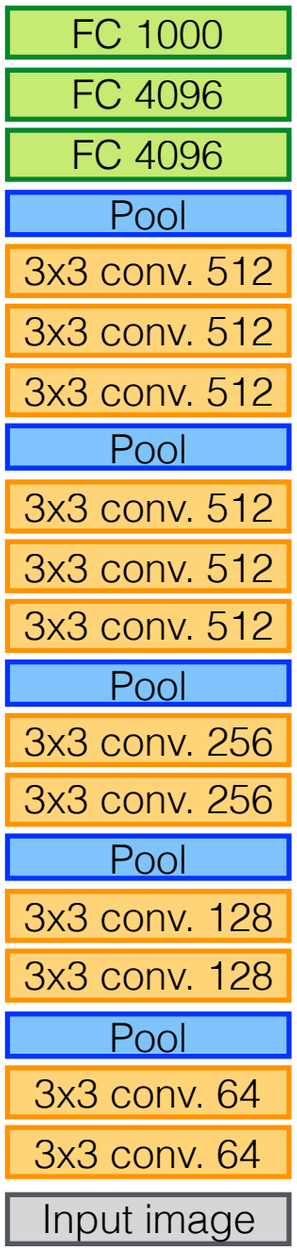
Transfer Learning / Fine Tuning

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



Transfer Learning / Fine Tuning

- Fine tuning: a few “guidelines”



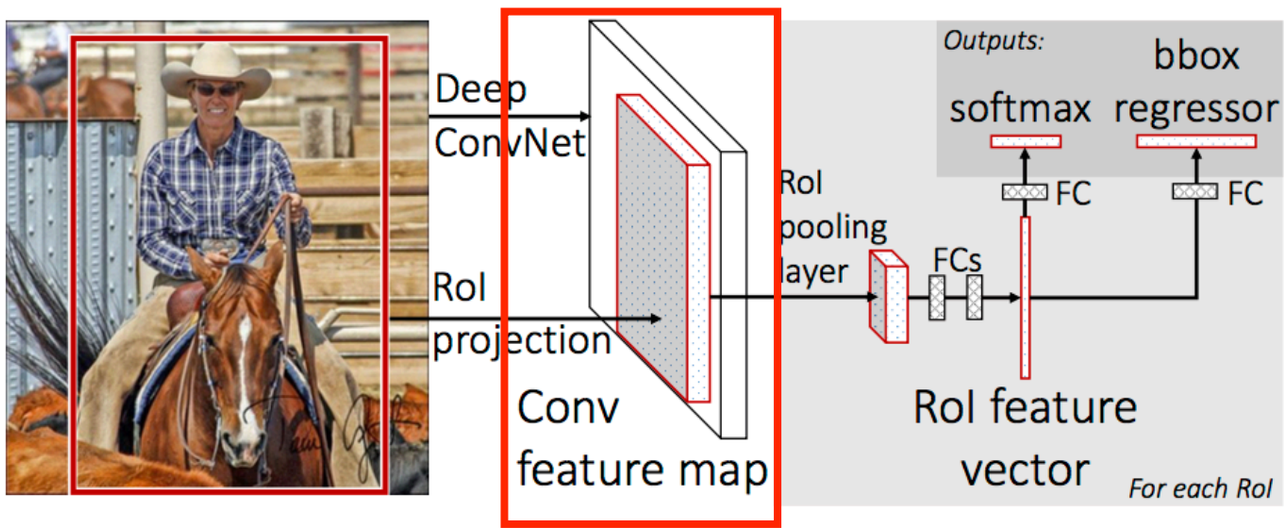
	Very similar dataset	**Very different dataset**
Very little data	Use linear classifier on top layer	This is bad… try linear classifier from different stages
Quite a lot of data	Finetune a few layers	Finetune a large number of layers

Slide credit: Fei-Fei, Karpathy, Johnson

Transfer Learning / Fine Tuning

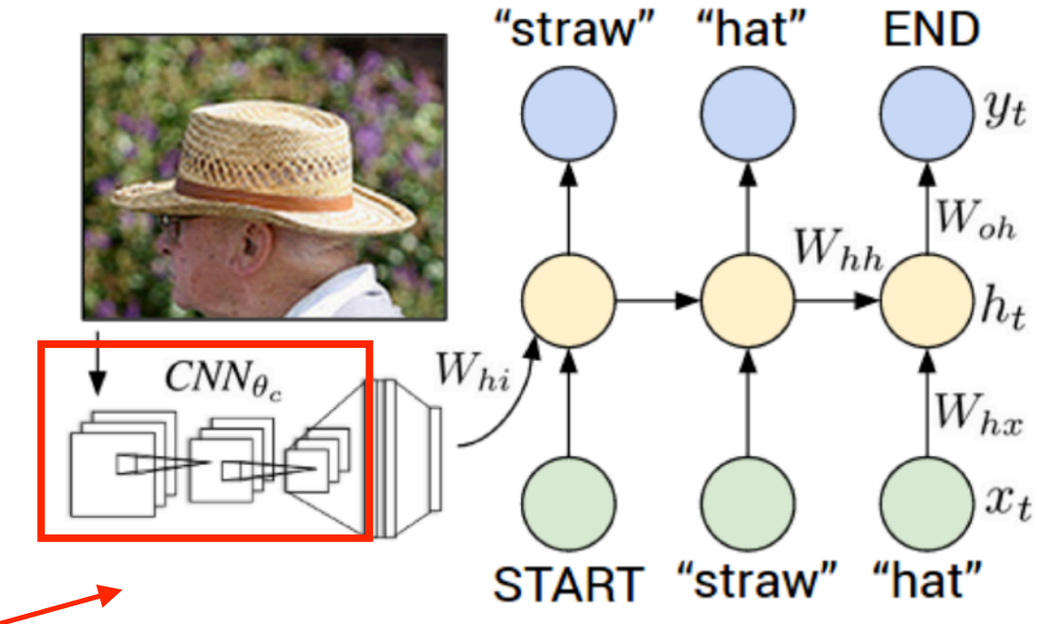
- Transfer learning with CNNs is pervasive, it's the norm, not an exception

Object Detection: Fast R-CNN



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

Image Captioning: CNN+RNN



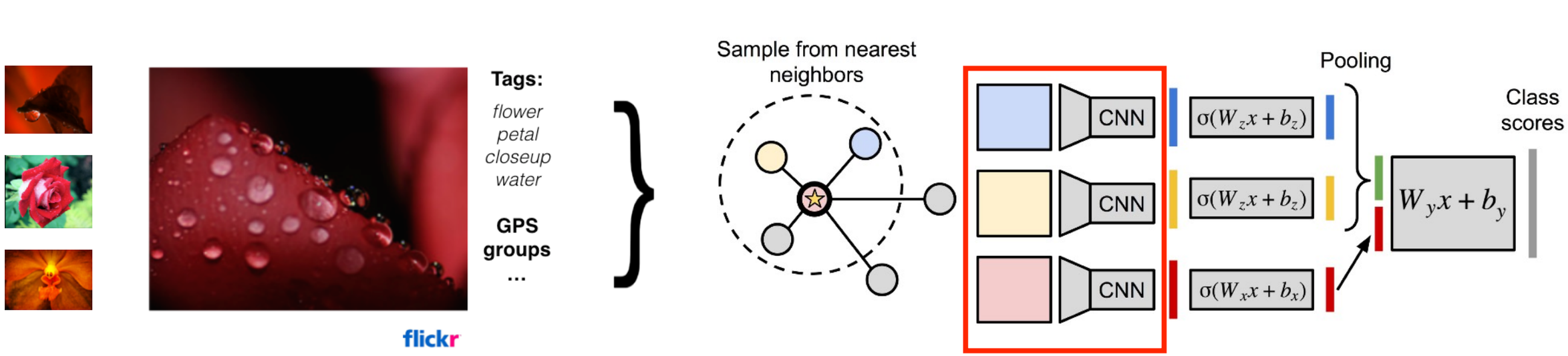
A.Karpathy, L.Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

CNN pretrained on ImageNet

Transfer Learning / Fine Tuning

- Additional note: 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

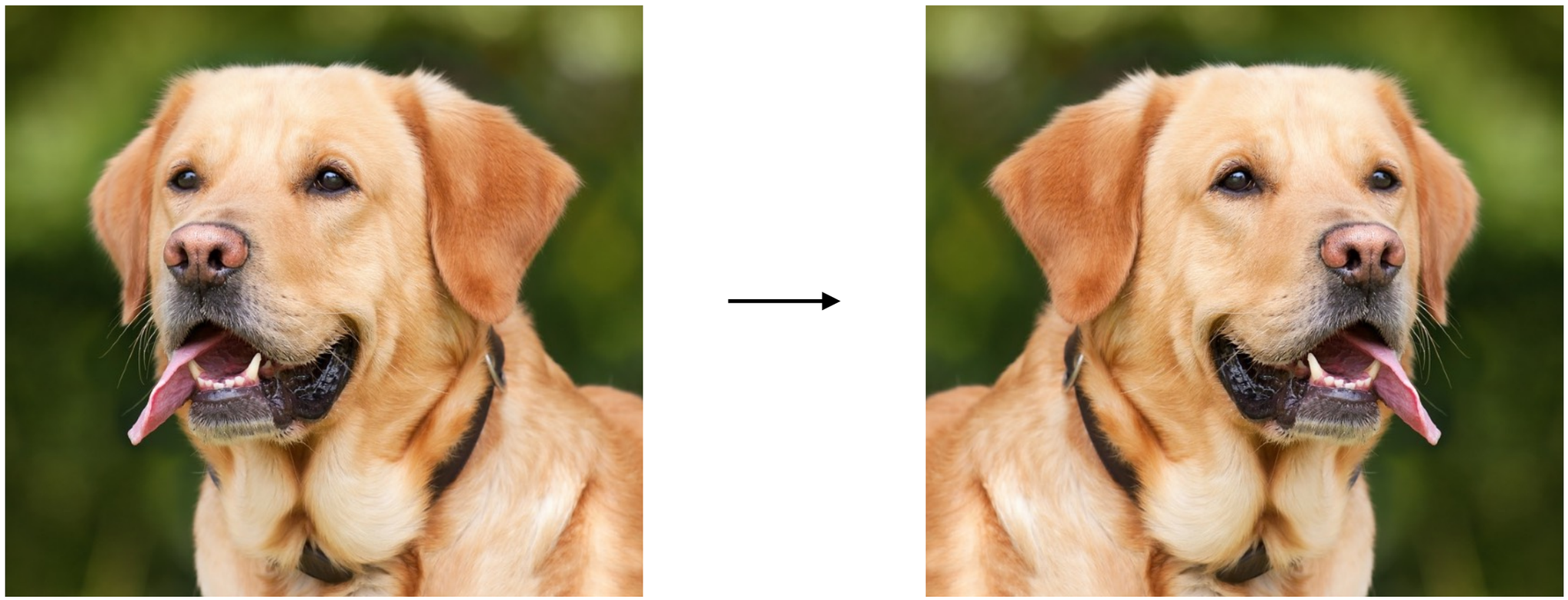
J.Johnson, L.Ballan, L.Fei-Fei, "Love Thy Neighbors: Image Annotation by Exploiting Image Metadata", ICCV 2015

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.

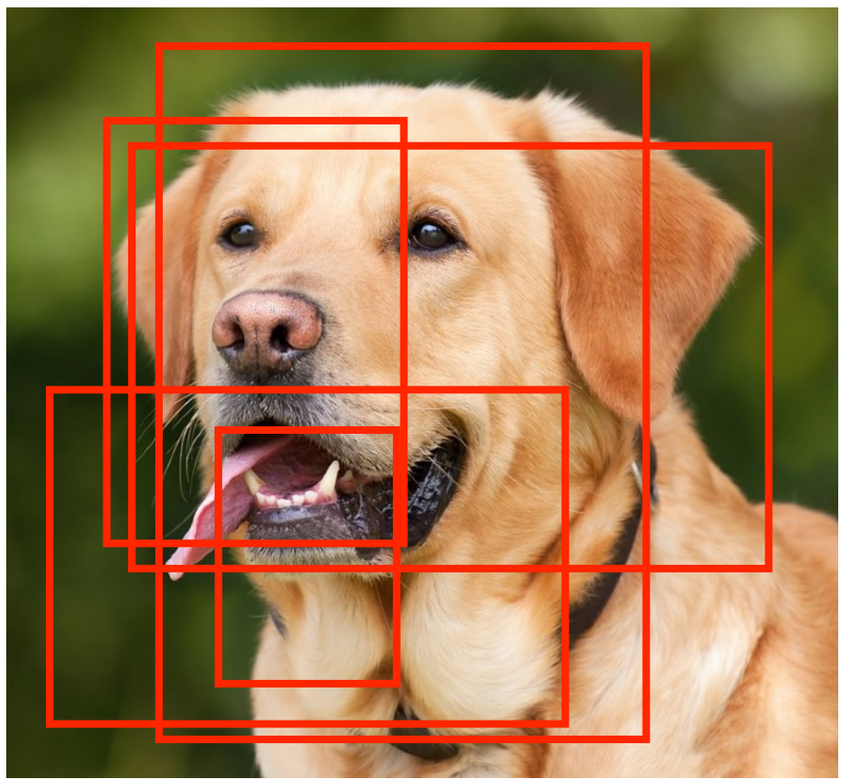
Data augmentation

- Examples: horizontal flips



Data augmentation

- Examples: random crops and scales



Training (e.g. ResNet):

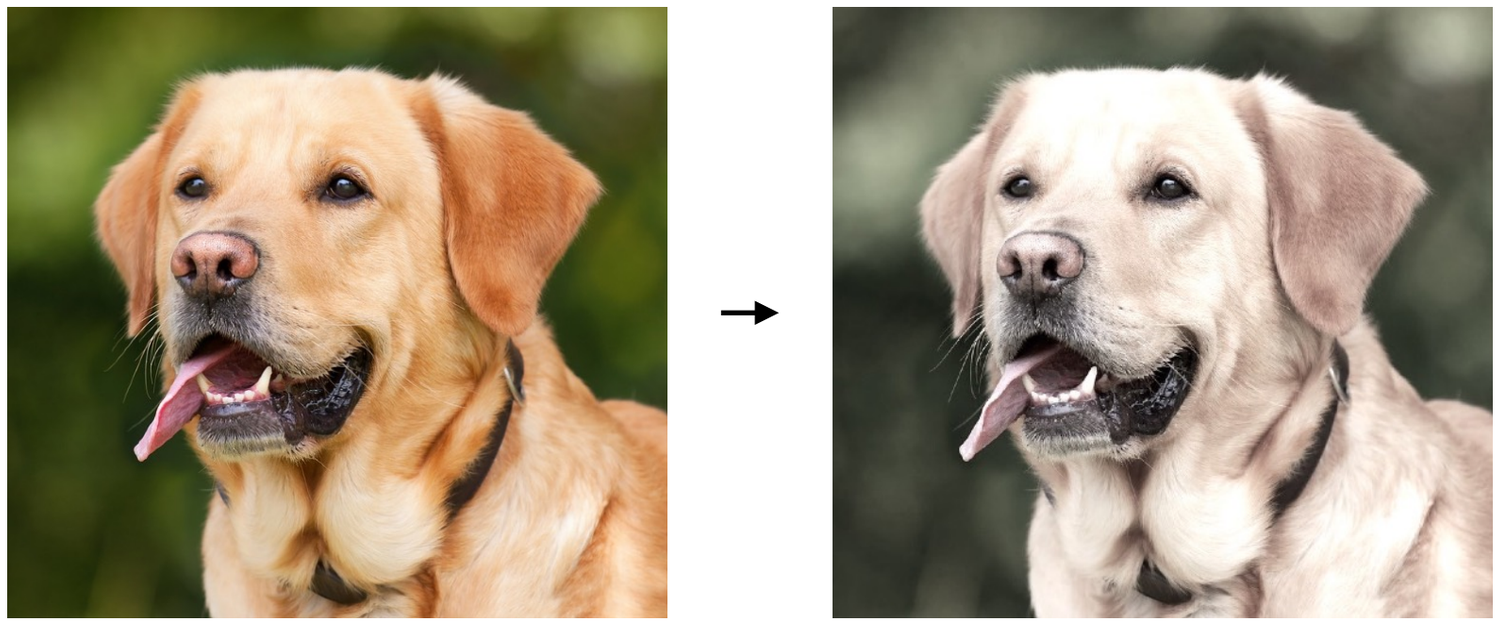
- Pick random L in range [256, 480]
- Resize training image: short-side=L
- Sample random 224 x 224 patch

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

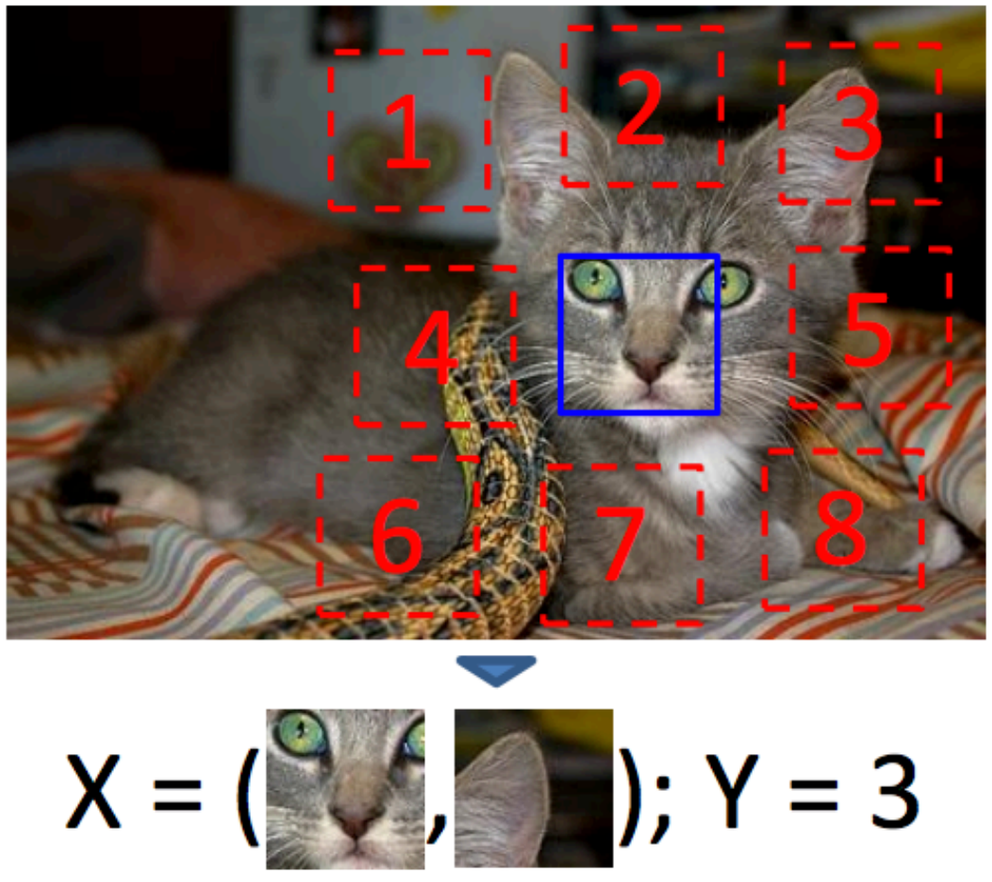
- Examples: color jitter
  - Simple: just randomize contrast and brightness

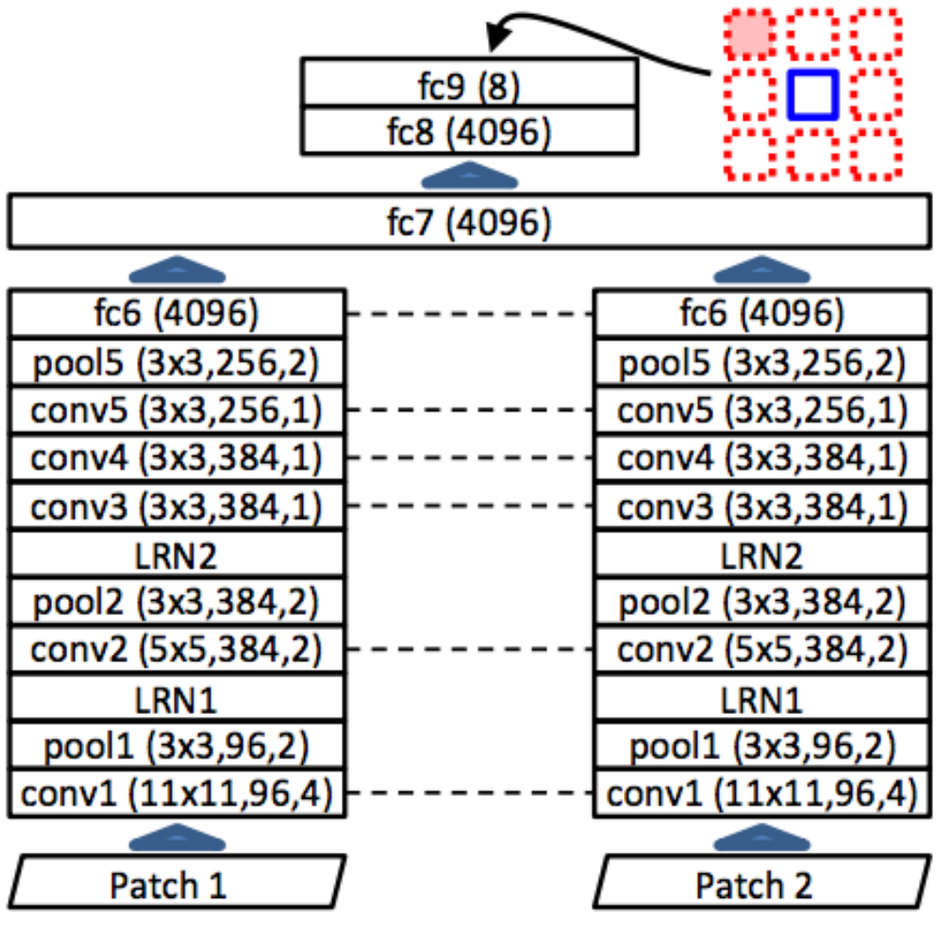


  - Advanced: apply PCA to all RGB pixels in training set
  - Then sample a color offset along principal component directions and add offset to all pixels of a training image

Self-supervised learning

- Use of (spatial) context as a source of free supervisory signal for representation learning





C.Doersch, A.Gupta, A.Efros, "Unsupervised Visual Representation Learning by Context Prediction", ICCV 2015

Image credit: Doersch, Gupta, Efros

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
  - TensorFlow: https://github.com/tensorflow/models
  - Caffe: https://github.com/BVLC/caffe/wiki/Model-Zoo
  - PyTorch: https://github.com/pytorch/vision
  - Keras: https://github.com/albertomontesg/keras-model-zoo

Summary

- More on convolutional layers
- CNN architectures, examples and case studies
- Training CNNs, applications
- Takeaways