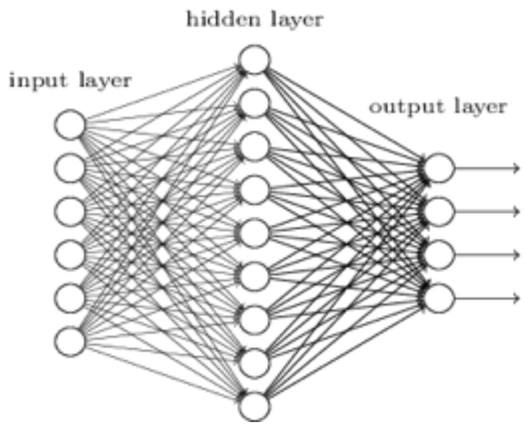
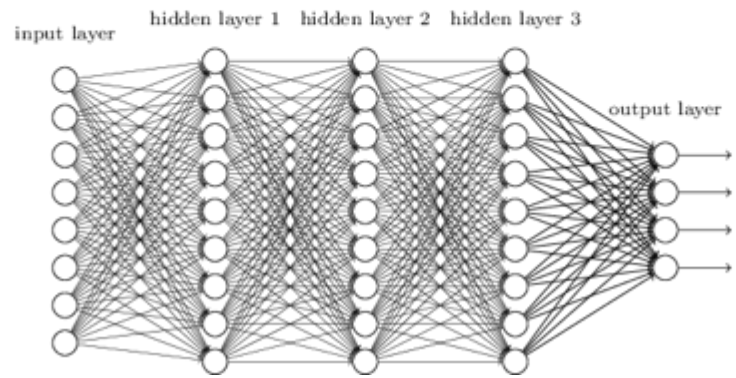
CMU 15-781

Lecture 18:
Deep learning and Vision: Convolutional neural networks

Teacher:
Gianni A. Di Caro

Deep, shallow, connected, sparse?



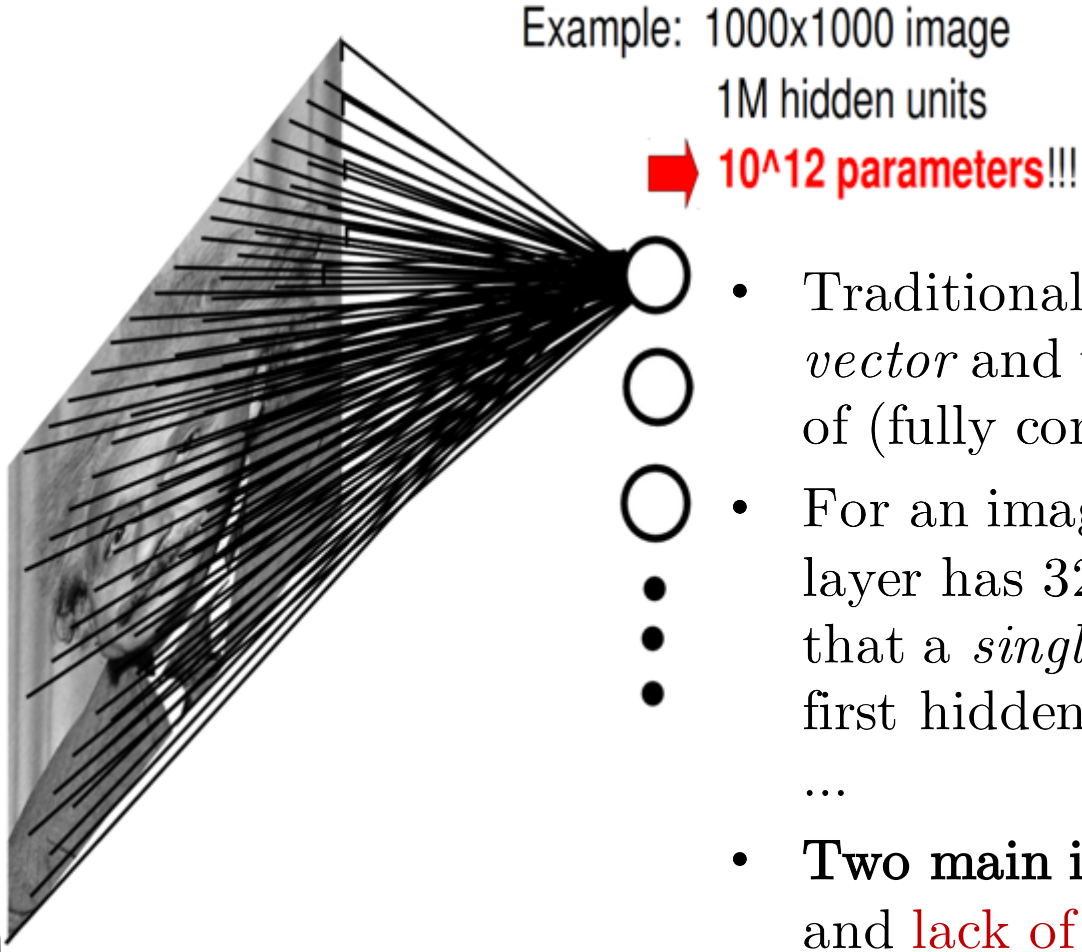


- Fully connected multi-layer feed-forward perceptrons:
  - More powerful than single layer networks:

$$F(\boldsymbol{x}) = f_1(f_2(f_3 \ldots f_n(\boldsymbol{x})\ldots)))$$

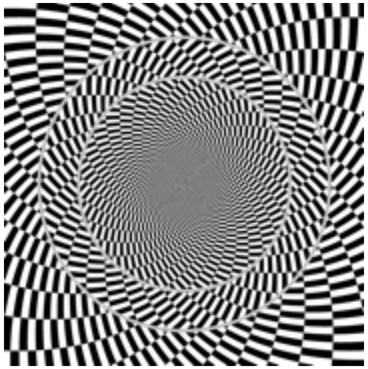
  - Can potentially learn hierarchical feature representations
  - A lot of parameters to learn! Need time, CPU, optimization algorithms, data, and has to avoid overfitting

Vision Example



- Traditional MLPs receive as input a *single vector* and transforms it through a series of (fully connected) hidden layers
- For an image (32w, 32h, 3c), the input layer has $32 \times 32 \times 3 = 3072$ neurons, such that a *single* fully-connected neuron in the first hidden layer would have 3072 weights …
- **Two main issues**: space-time complexity and lack of structure, locality of information

IMAGES ARE "MULTI-DIMENSIONAL"

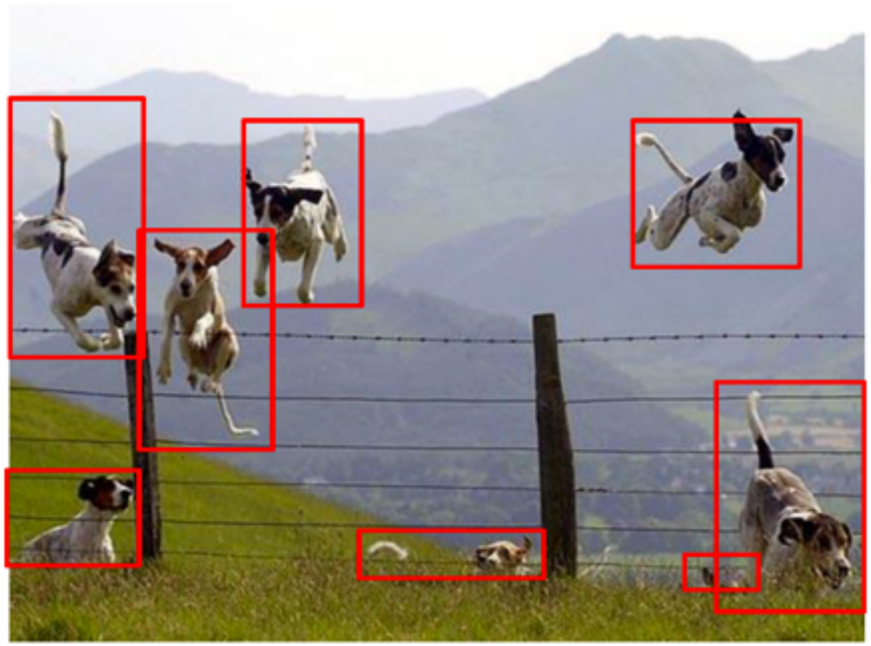


Have a local *structure* and *correlations*

Have distinctive *features* in space and in frequency domains

Same (useful) features can be rotated, translated, scaled ...

Object detection



Where are the objects of interest?

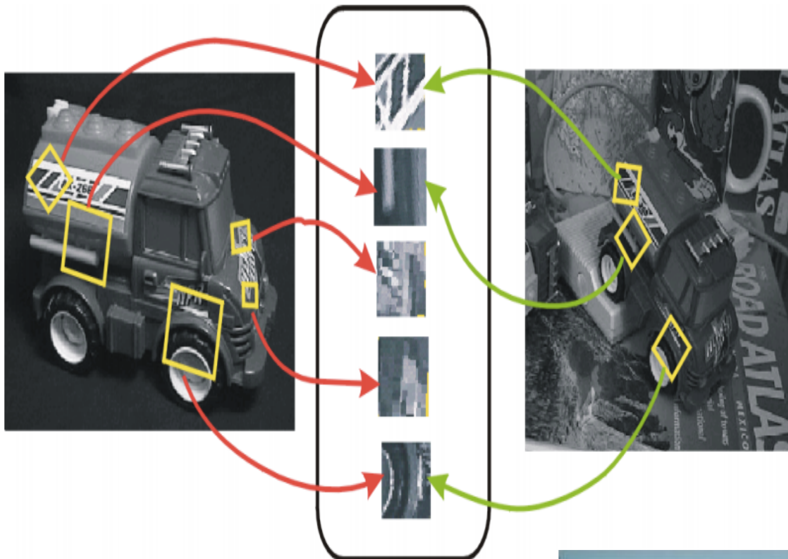
Finding / Extracting *good* features is fundamental in vision processing tasks

But it's not an easy task!

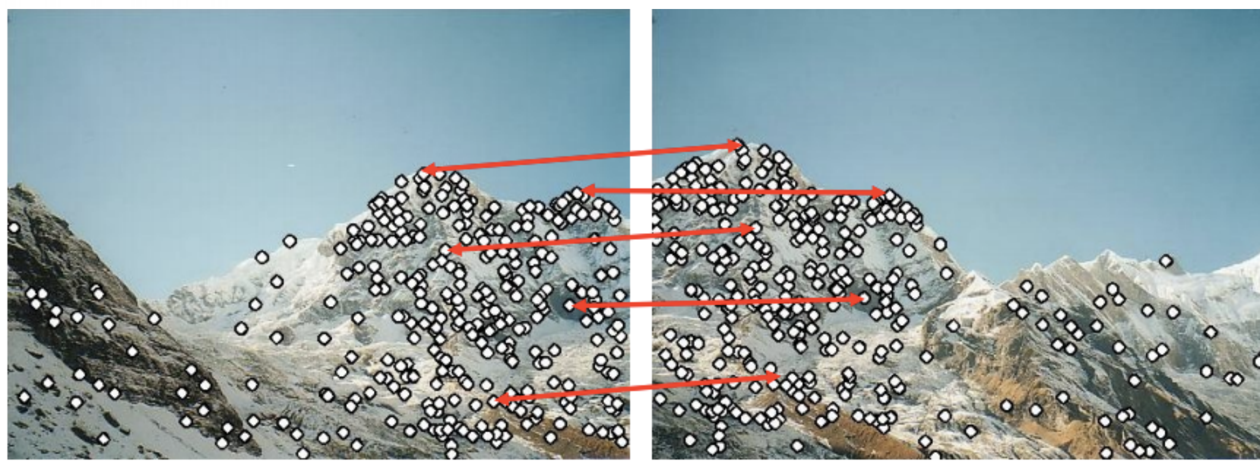
BTW ... Features?

- Want *uniqueness*
- Want *invariance*
  - *Geometric invariance:* translation, rotation, scale
  - *Photometric invariance*: brightness, exposure, ...
- Leads to unambiguous matches in other images or wrt to know entities of interest
- Look for "interest points": image regions that are *unusual*
- Typically non-linear, "complex"
- How to define "unusual"?

BTW … Features?



Feature Descriptors



SIFT

Basic idea:

- Take 16x16 square window around detected interest point (8x8 shown below)
- Compute edge orientation (angle of the gradient minus 90°) for each pixel
- Throw out weak edges (threshold gradient magnitude)
- Create histogram of surviving edge orientations (8 bins)

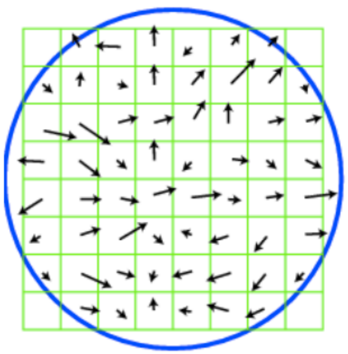
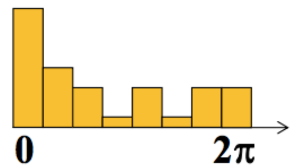
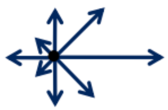


Image gradients



angle histogram



SIFT
Scale Invariant Feature Transform

Full version

- Divide the 16x16 window into a 4x4 grid of cells
  (8x8 window and 2x2 grid shown below for simplicity)
- Compute an orientation histogram for each cell
- 16 cells * 8 orientations = 128 dimensional descriptor

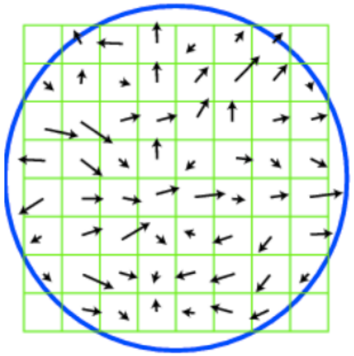
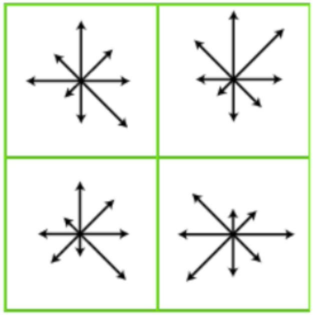


Image gradients



Keypoint descriptor

SIFT

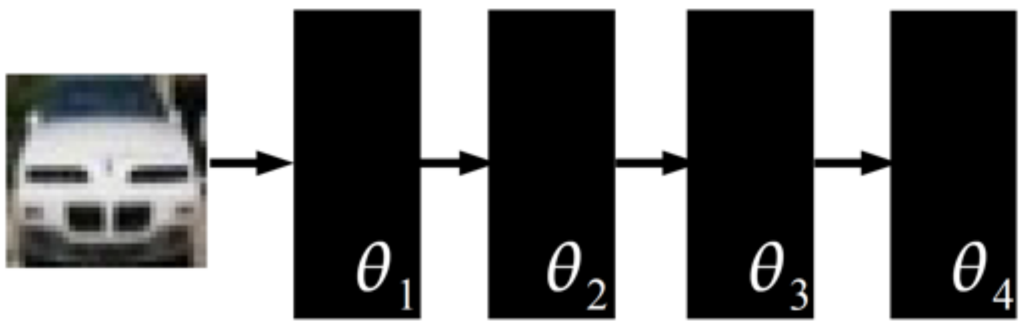
Different recipes

"Classical"
Pattern recognition

Solution #1: freeze first N-1 layer (engineer the features)
It makes it shallow!

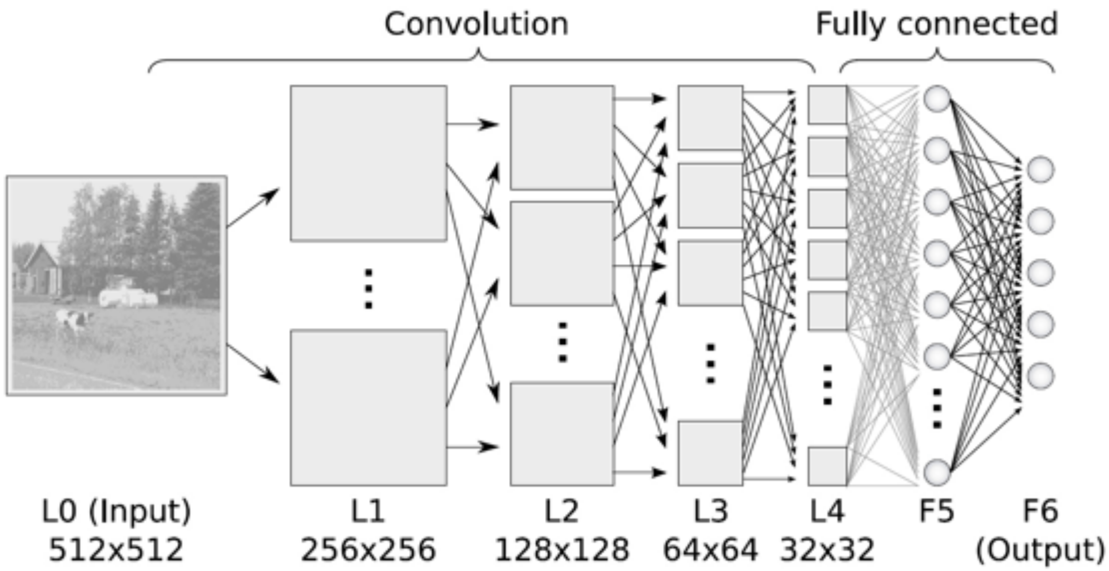
Optimization is difficult: non-convex, non-linear system

"Hot"
Convolutional
Neural Networks



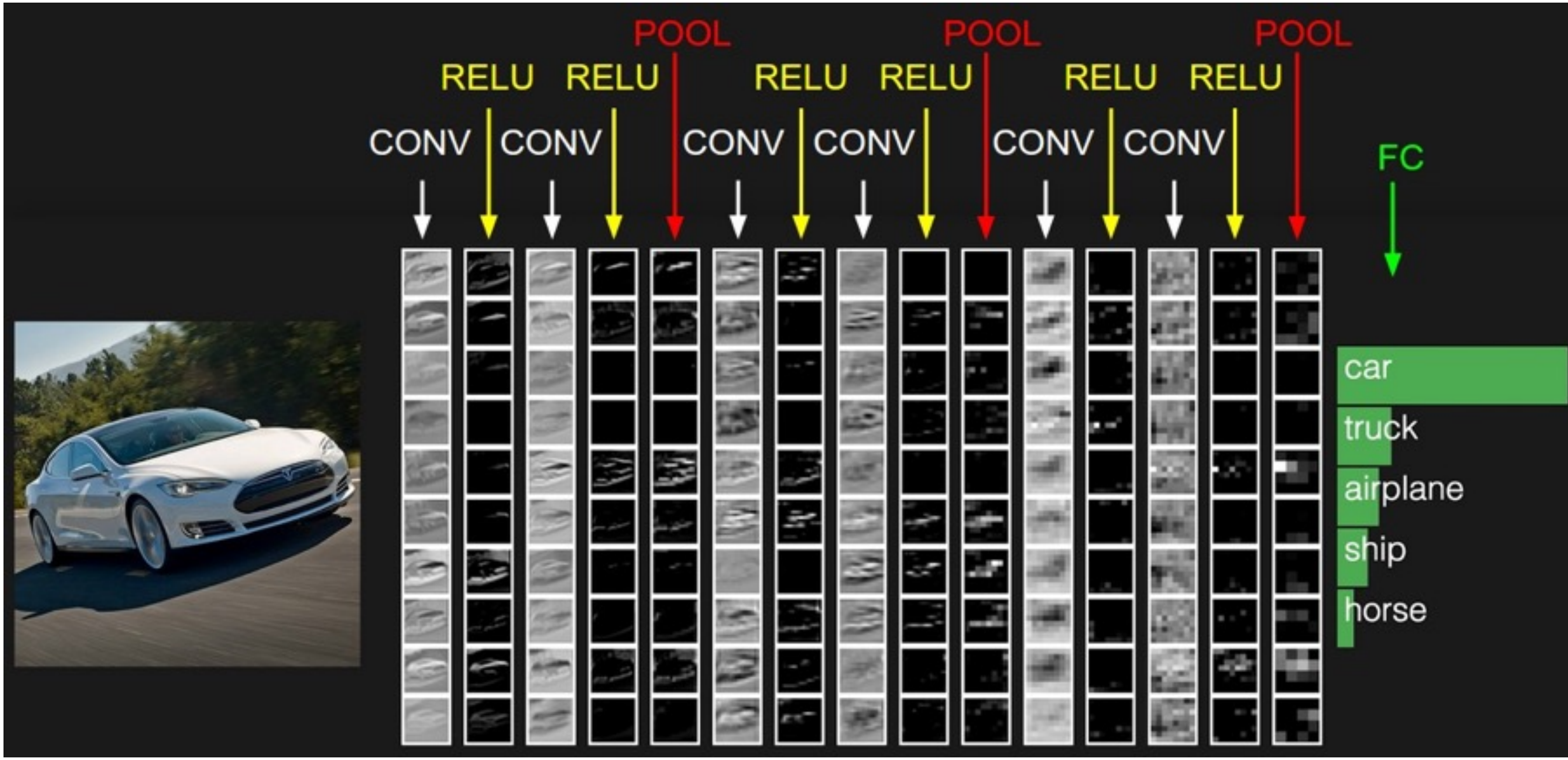
Solution #2: live with it!
It will converge to a local minimum.

Convolutional NNs



- Not anymore fully connected
- Locality of processing
- Weight sharing for parameter reduction
- Learn the parameters of multiple convolutional filter banks
- Compress to extract salient features and favor generalization

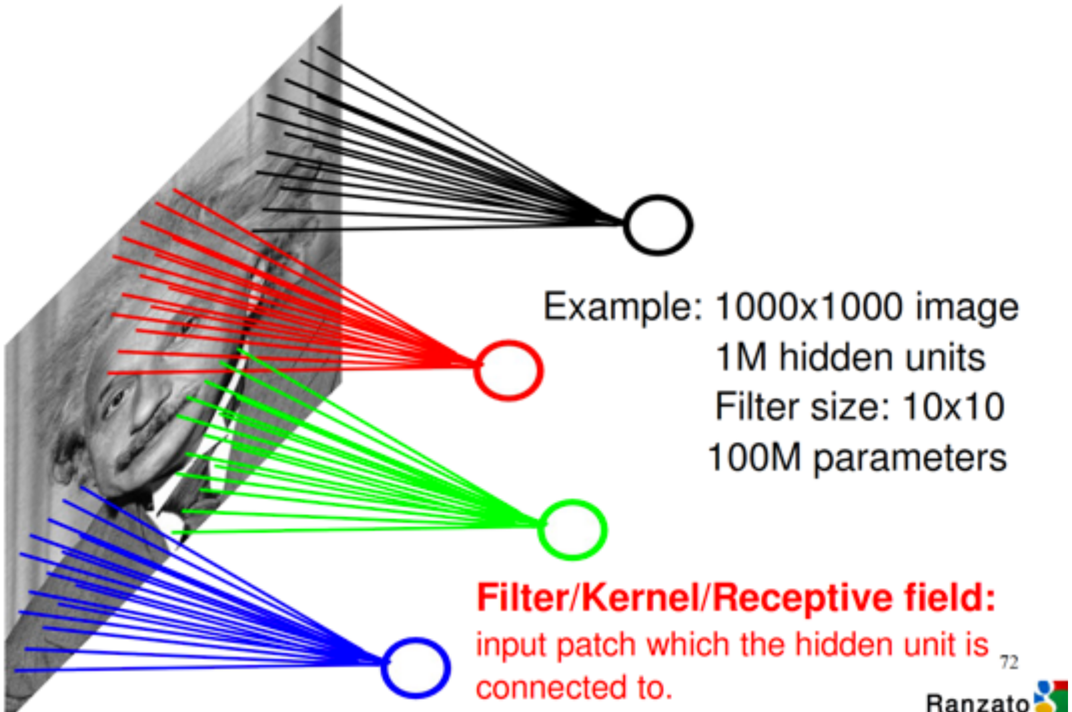
CONVOLUTIONAL NN

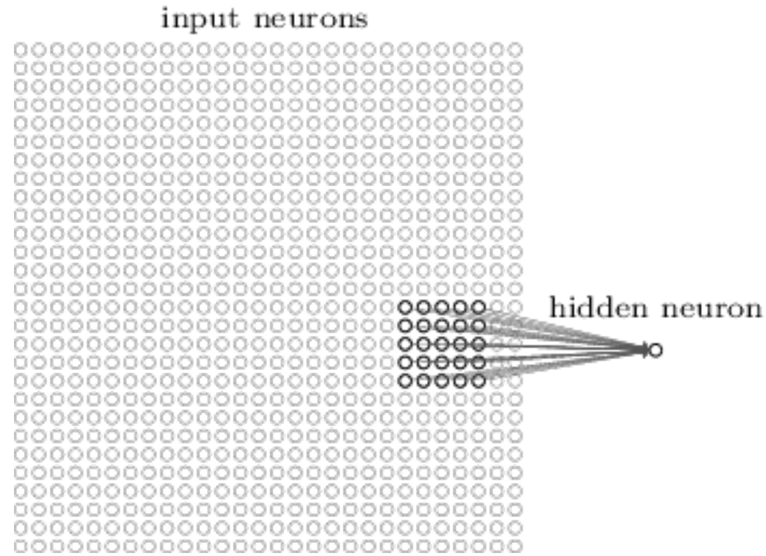


http://cs231n.github.io/

LOCALITY OF INFORMATION: RECEPTIVE FIELDS

$28 \times 28 = 784$
input image



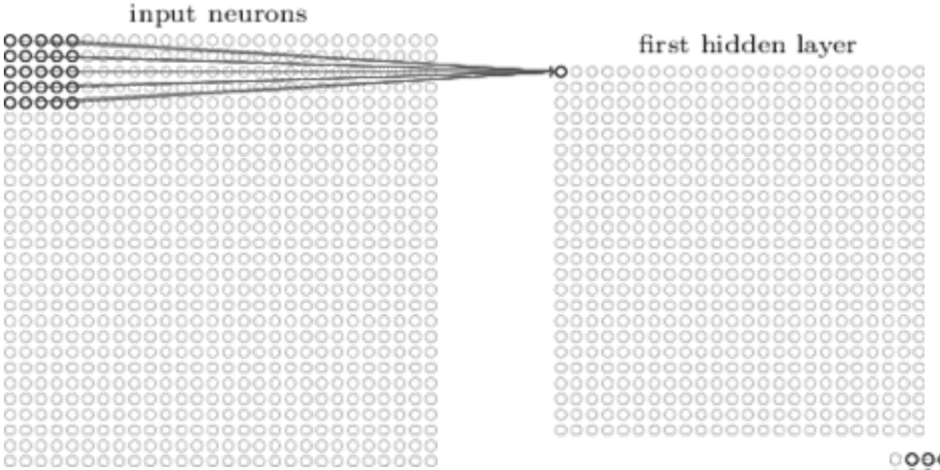


$5 \times 5 = 25$
input pixels

How many neurons in the 1^{st} hidden layer?

(Filter) Stride

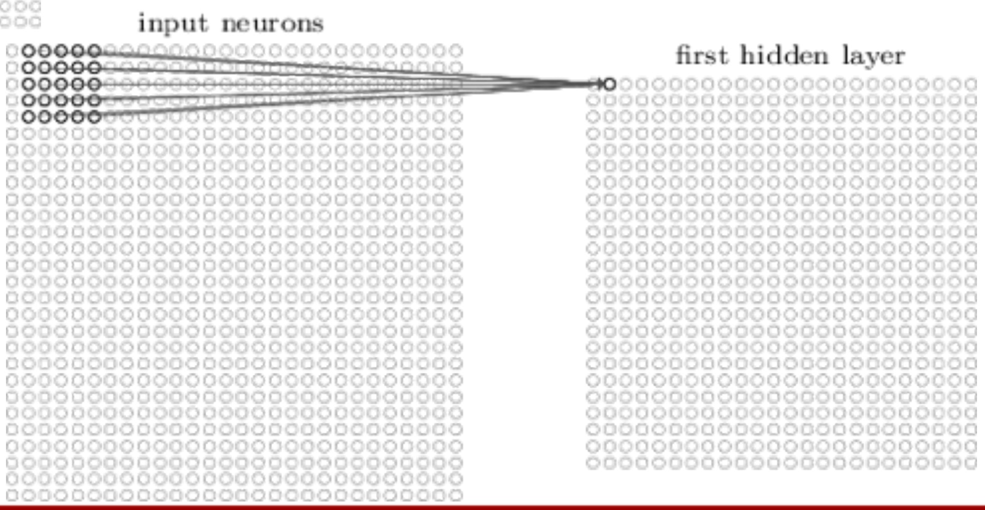
Let's *slide* the 5×5 mask over all input pixels



Stride length = 1

Any stride can be used ...

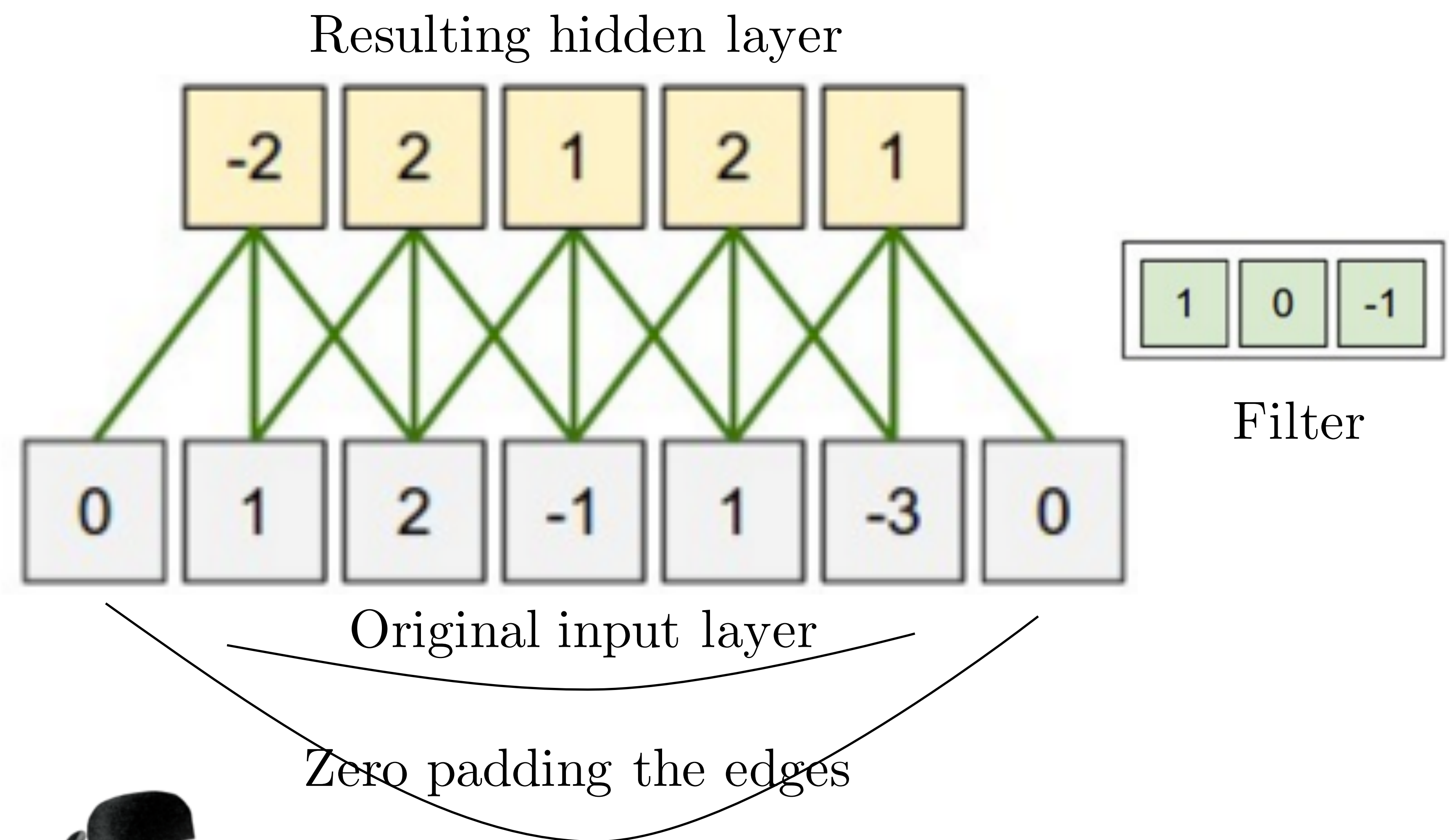
with some precautions



How many neurons in the 1^{st} hidden layer?

24×24

(Filter) Padding



Shared weights

- What is the precise relationship between the neurons in the receptive field and that in the hidden layer?
- What is the *activation value* of the hidden layer neuron?

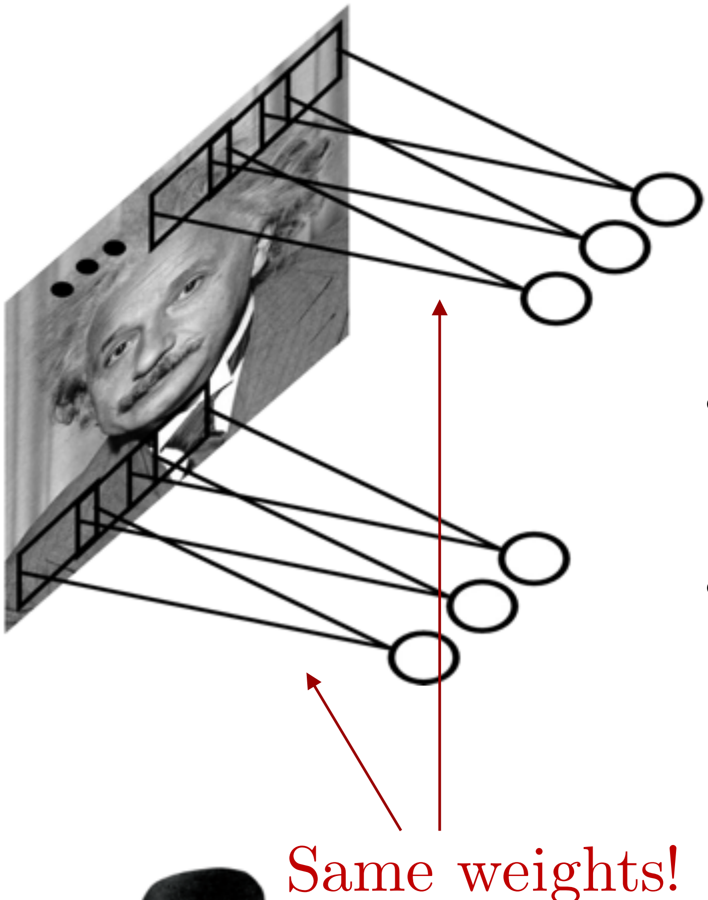
$$\sigma\left(b + \sum_{l=0}^{4} \sum_{m=0}^{4} w_{l,m} a_{j+l,k+m}\right)$$

- σ is the selected activation function, a are the activation values of the neurons in the receptive field
- The *same* weights w and bias b are used for each of the 24×24 hidden neurons

Feature map

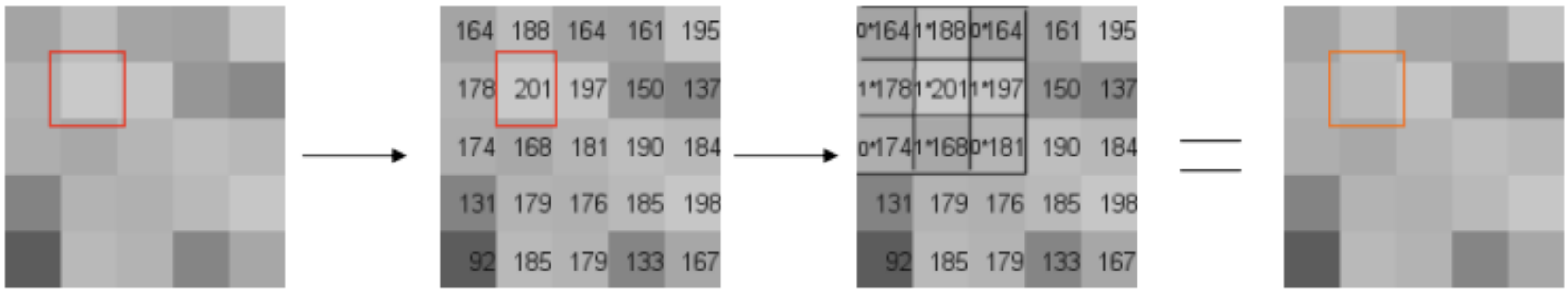
- All the neurons in the first hidden layer detect exactly the same **feature**, just at different locations in the input image.
- **"Feature":** the kind of input pattern (e.g., a local edge) that determine the neuron to "fire" or, more in general, produce a certain response level
- Why this makes sense? Suppose the weights and bias are (learned) such that the hidden neuron can pick out, a vertical edge in a particular local receptive field. That ability is also likely to be useful at other places in the image. And so it is useful to apply the same feature detector everywhere in the image.

FEATURE MAP



- The map from the input layer to the hidden layer is therefore a **feature map:** all nodes detect the same feature in different parts of the image
- The map is defined by the shared weights and bias
- The shared map is the result of the application of **convolutional filter** (defined by weights and bias)

Convolution image filter



Original image

Image with color values placed over it

Image with 3x3 kernel placed over it

Output image

164	188	164
178	201	197
174	168	181

Color values

×

0	1	0
1	1	1
0	1	0

Kernel

Divided by the sum of the kernel

932\5 = new pixel color

Convolution image filter

1	1	1
1	1	1
1	1	1

Unweighted 3x3 smoothing kernel

0	1	0
1	4	1
0	1	0

Weighted 3x3 smoothing kernel with Gaussian blur

0	-1	0
-1	5	-1
0	-1	0

Kernel to make image sharper

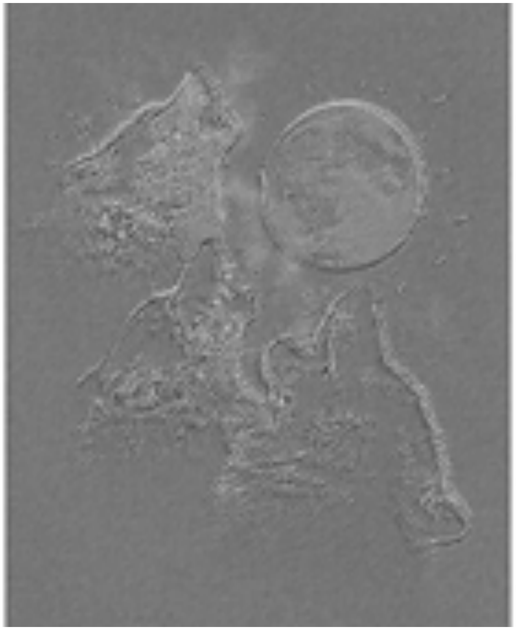
-1	-1	-1
-1	9	-1
-1	-1	-1

Intensified sharper image

Gaussian Blur

Sharpened image

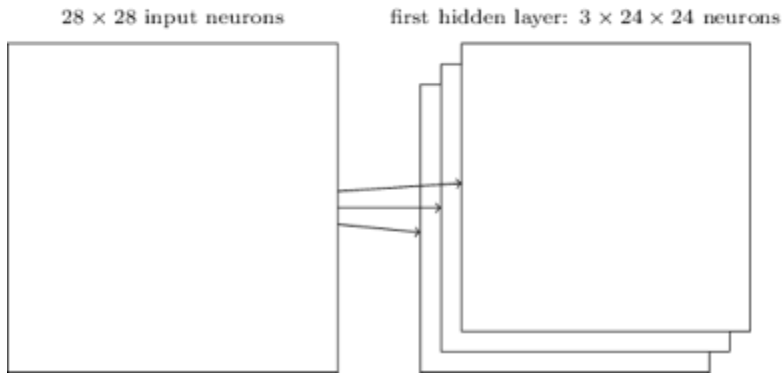
Convolution image filter



Filter banks

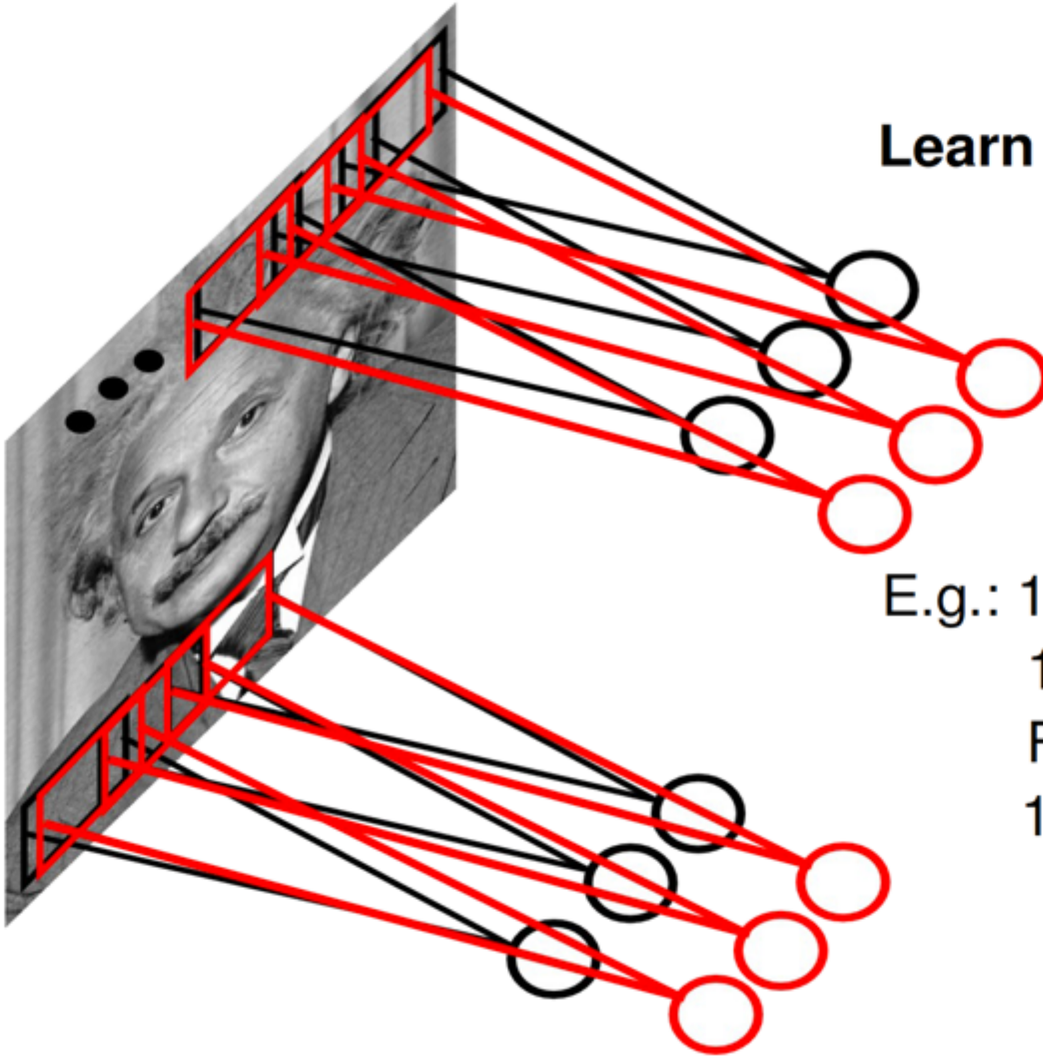
Why only one filter? (one feature map)

- At the i-th hidden layer n filters can be active in parallel
- A **bank of convolutional filters**, each learning a *different* feature (different weights and bias)



- 3 feature maps, each defined by a set of 5×5 shared weights and one bias
- The result is that the network can detect 3 different kinds of features, with each feature being detectable across the entire image.

Filter banks



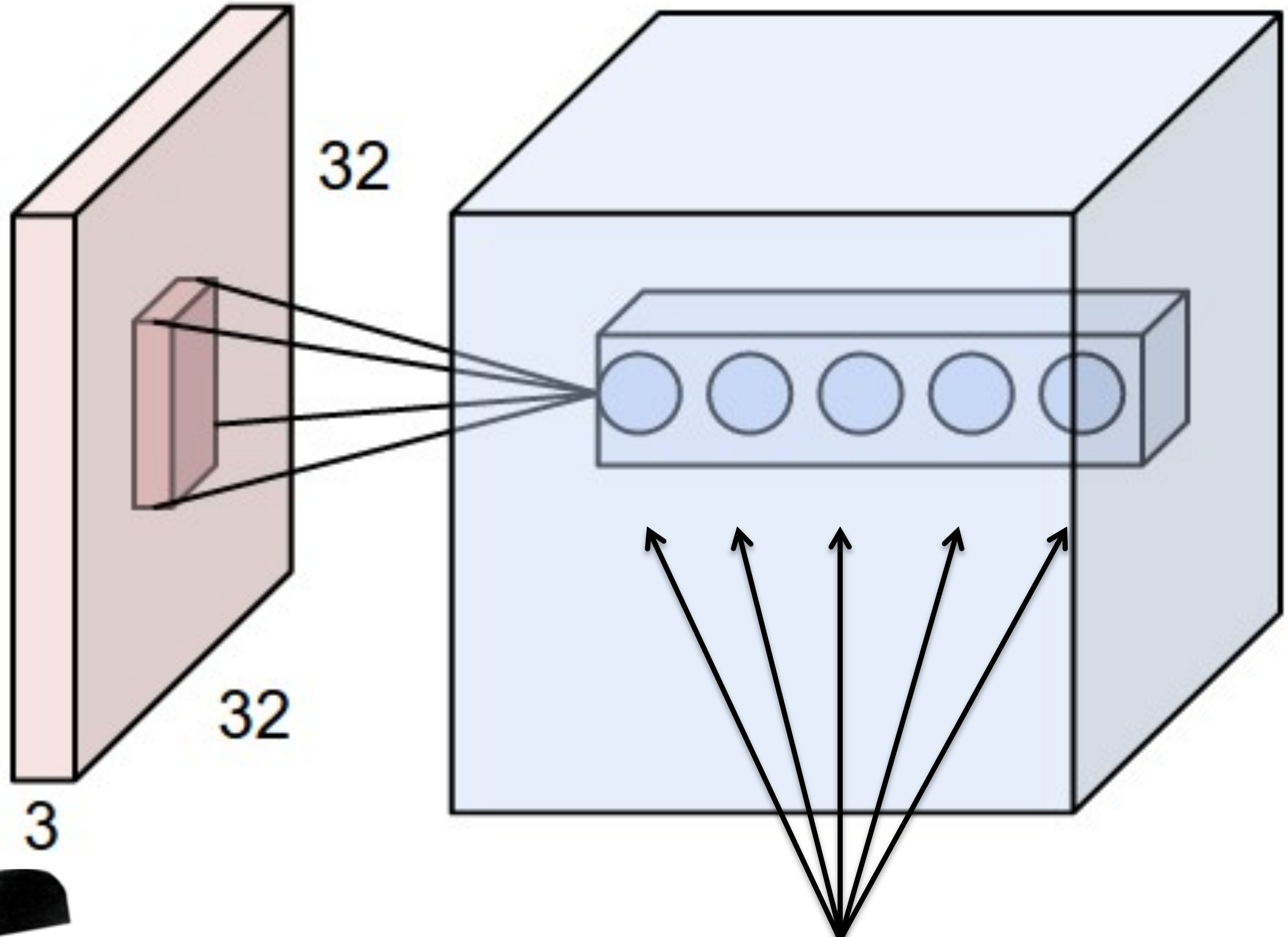
Learn multiple filters.

E.g.: 1000x1000 image
100 Filters
Filter size: 10x10
10K parameters

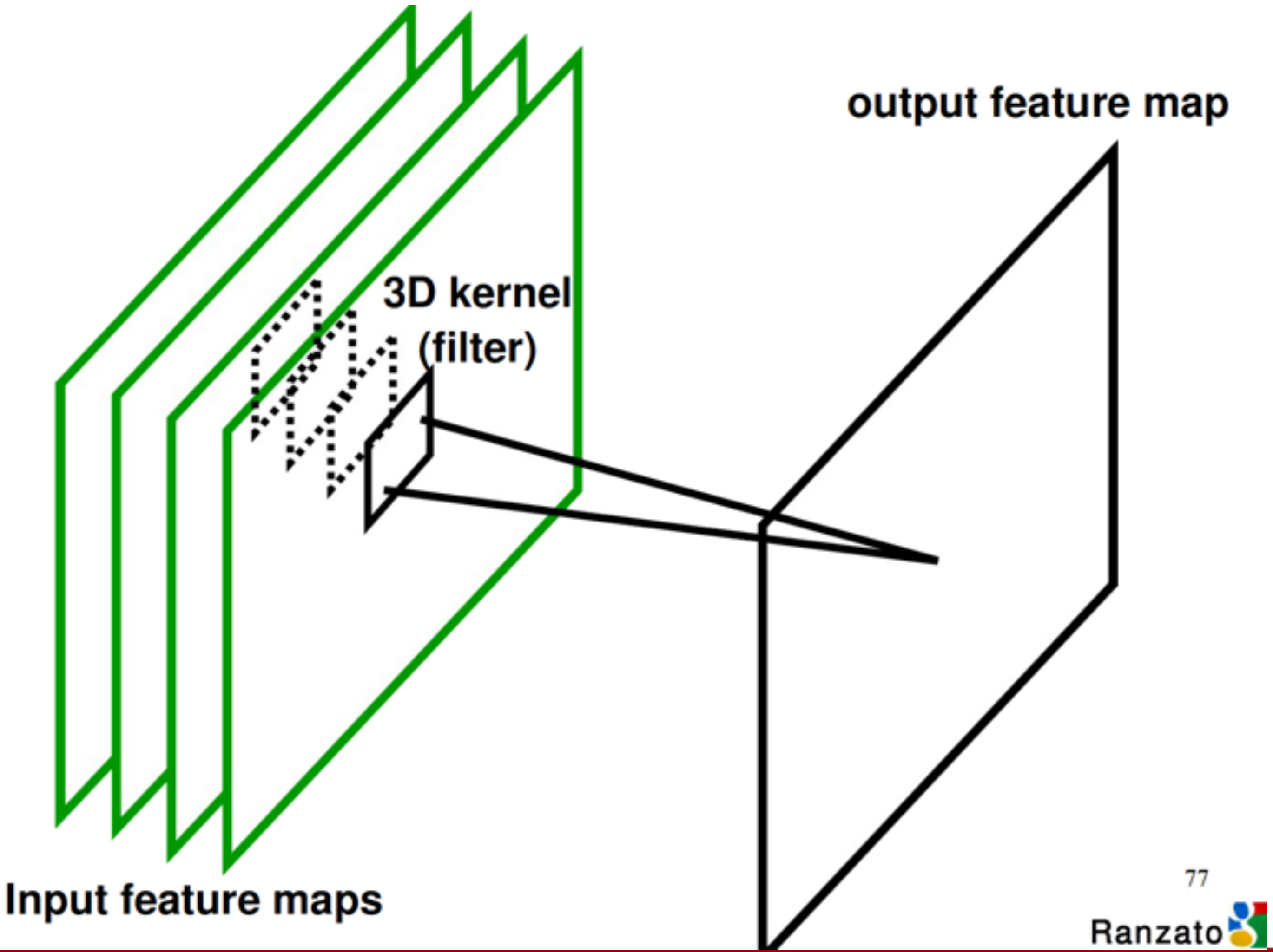
75

Ranzato

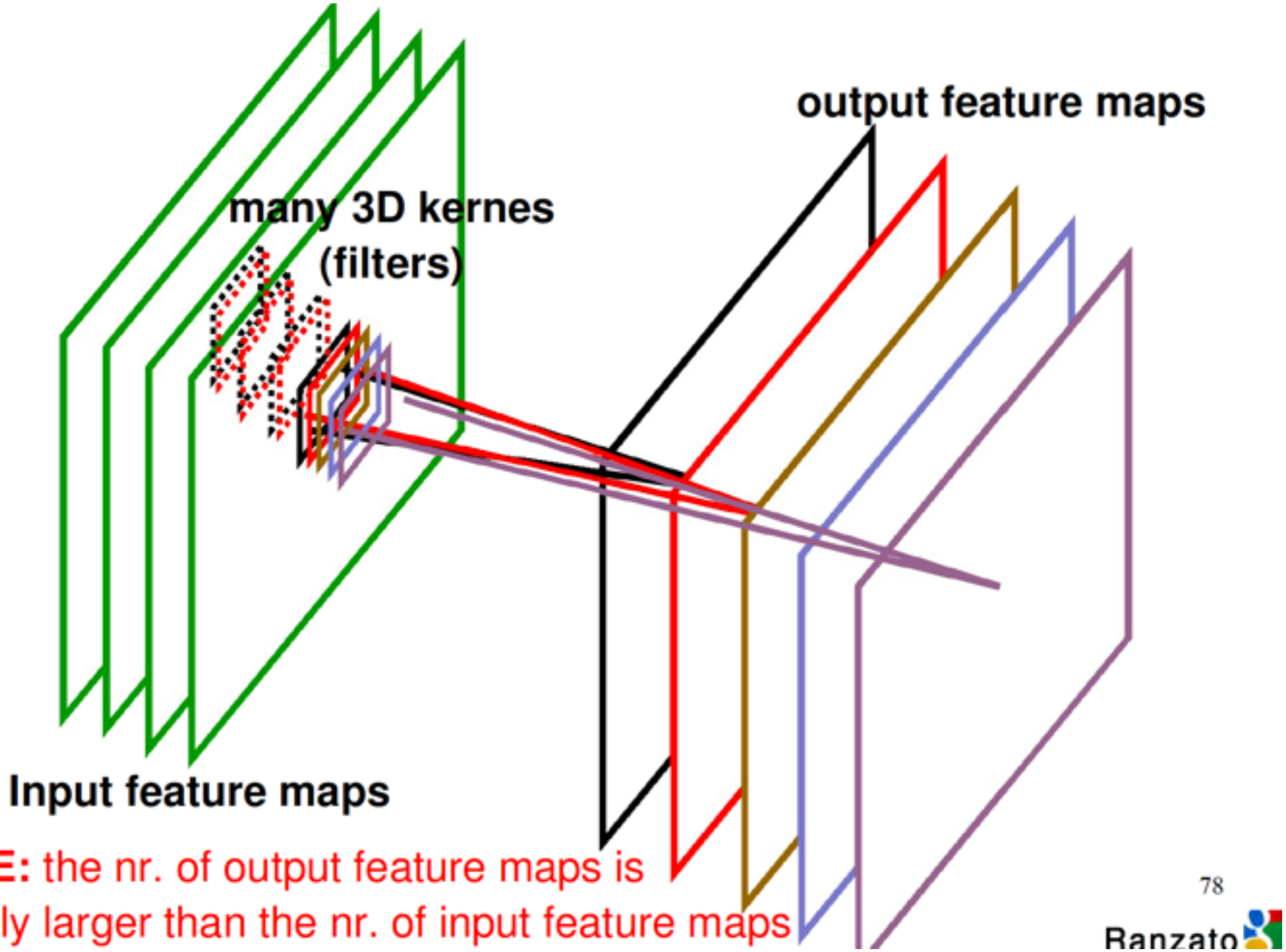
Volumes and Depths



Multiple feature maps

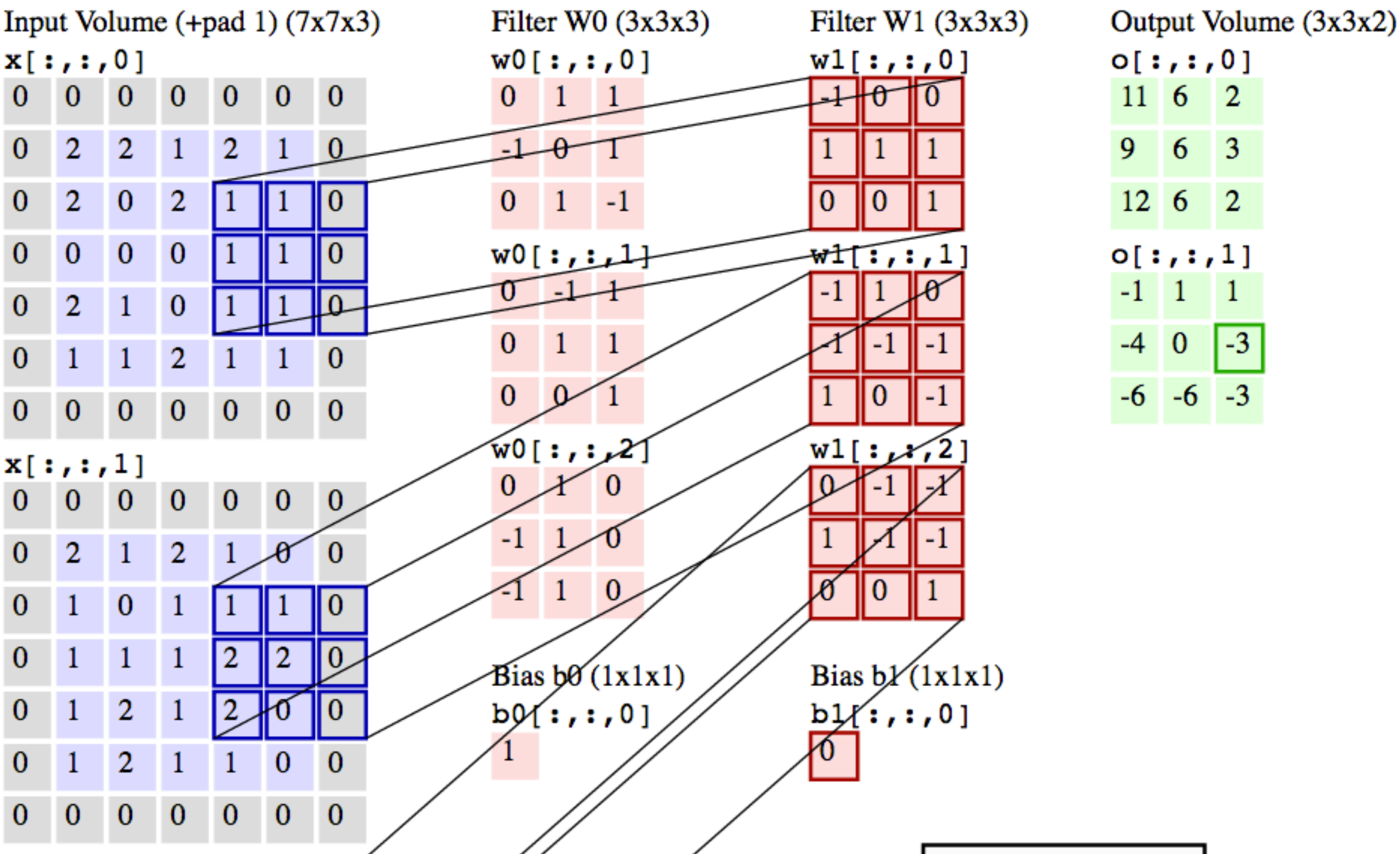


MULTIPLE FEATURE MAPS

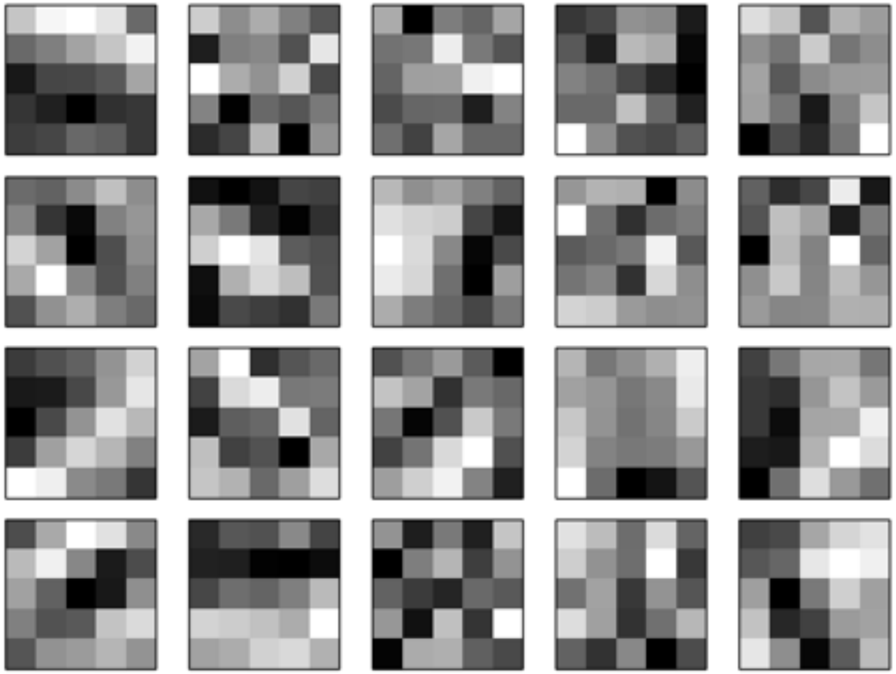


NOTE: the nr. of output feature maps is usually larger than the nr. of input feature maps

Numeric example

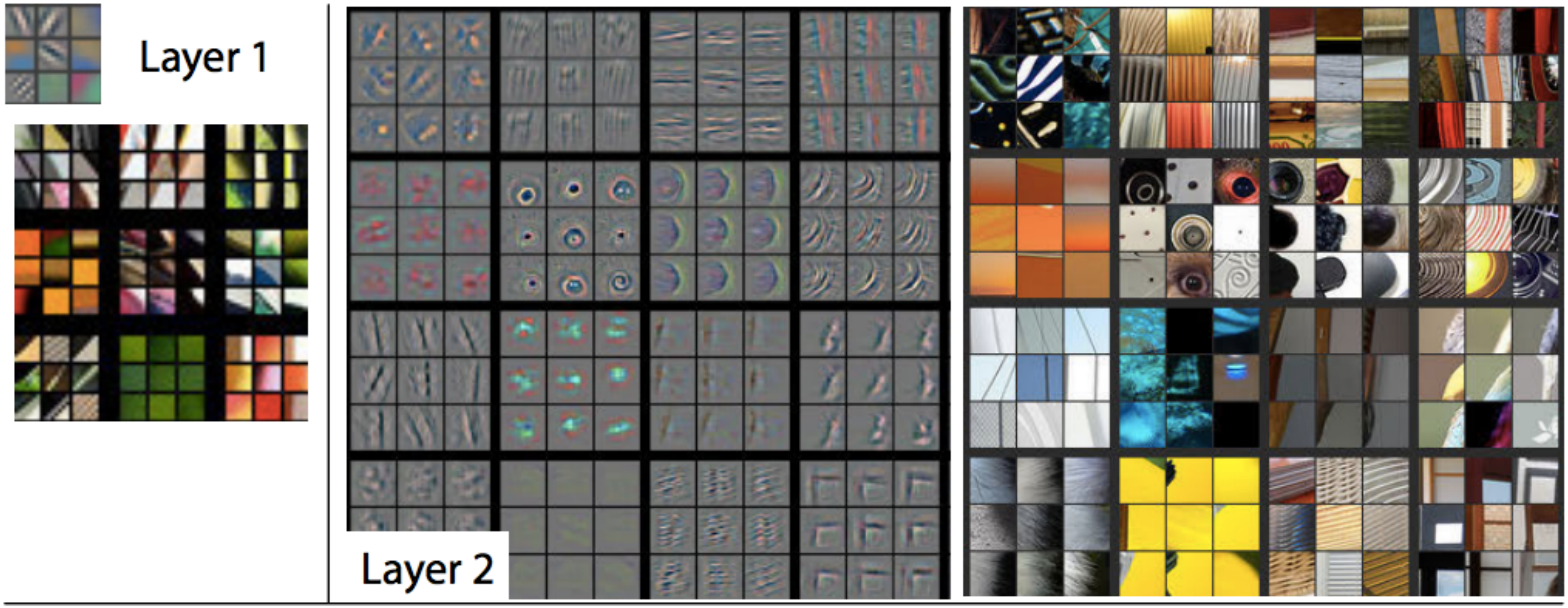


Character recognition example



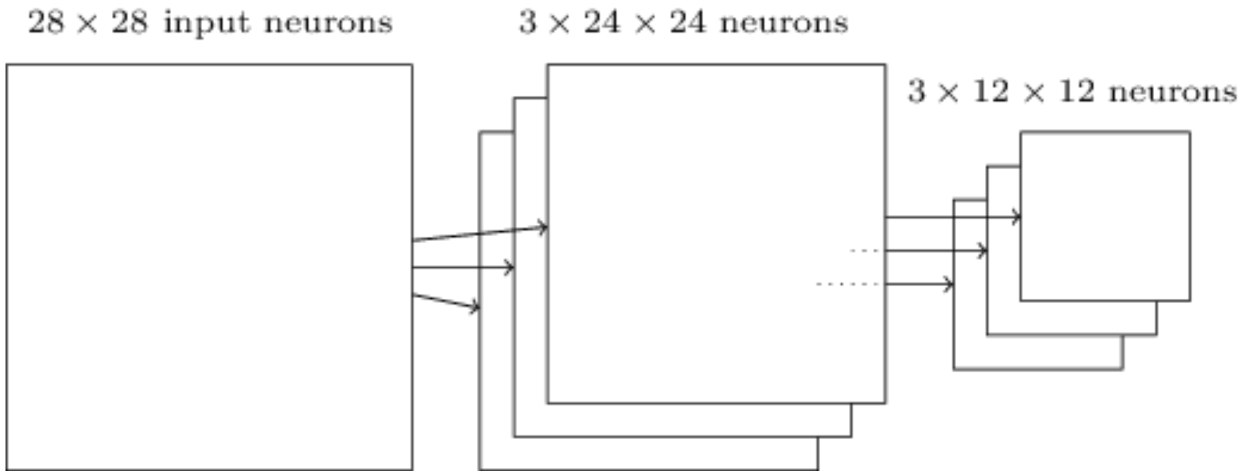
Darker blocks mean a larger weight, so the feature map responds more to the corresponding input pixels.
Some spatial correlations are "there"

A more exciting example

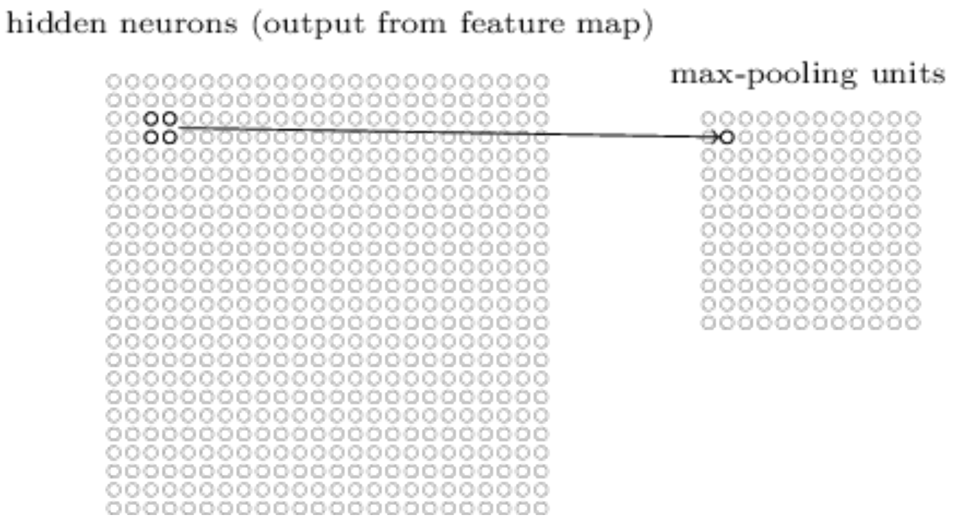


Pooling layers

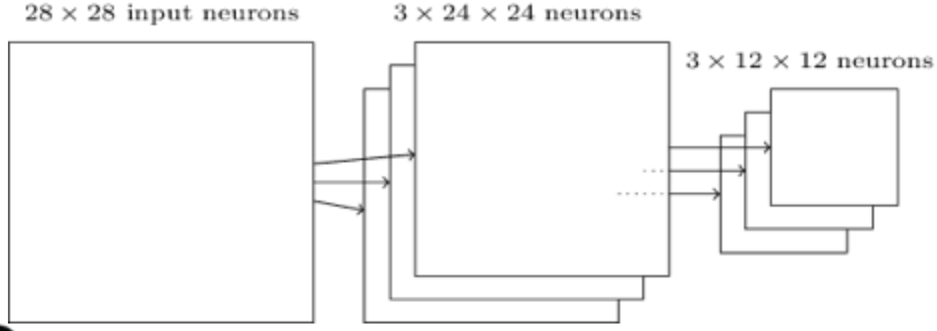
- **Pooling layers** are usually used immediately after convolutional layers.
- Pooling layers simplify / subsample / compress the information in the output from the convolutional layer
- A pooling layer takes each feature map output from the convolutional layer and prepares a condensed feature map



Pooling layers



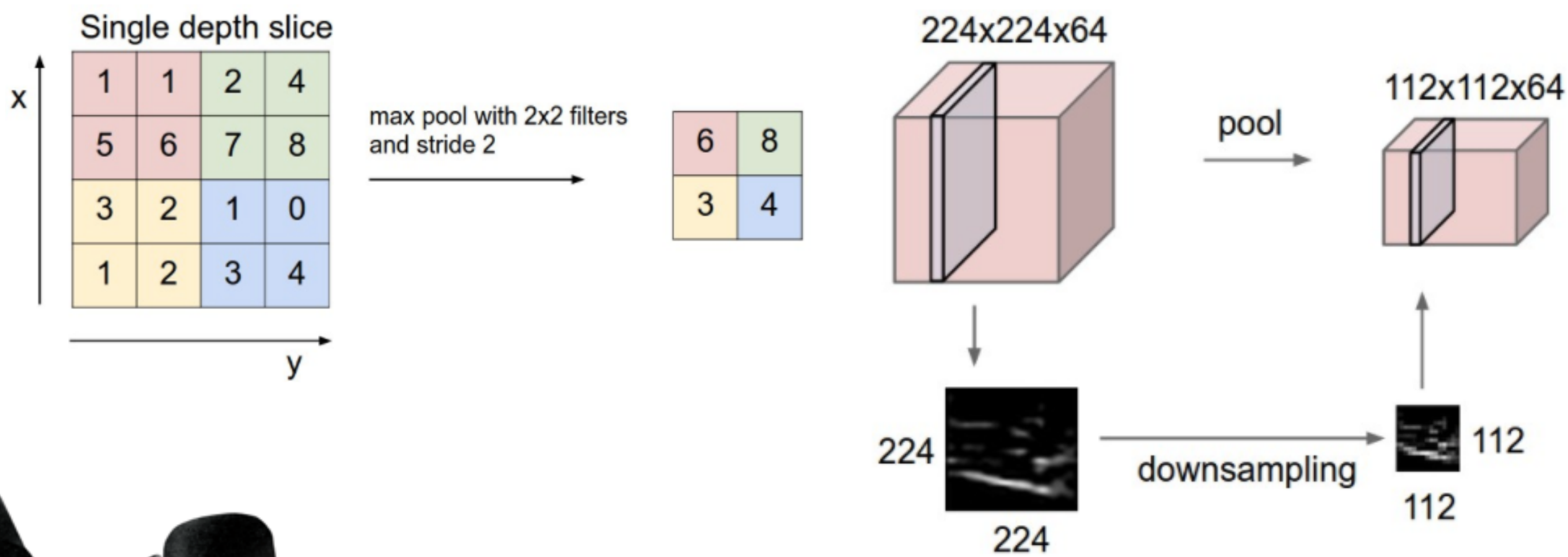
Each neuron in the pooling layer summarizes a region of $n \times n$ neurons in the previous hidden layer, which results in **subsampling**



Max-Pooling

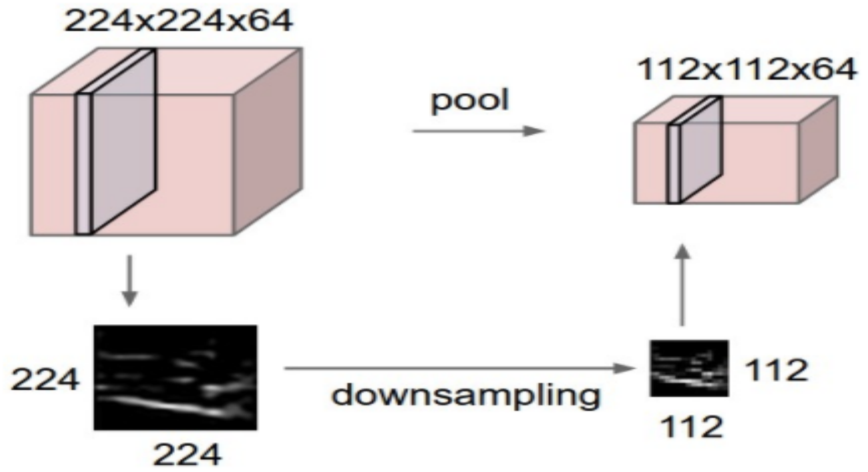
How to do pooling?

Max-pooling: a pooling unit simply outputs the *maximum activation* in the input region

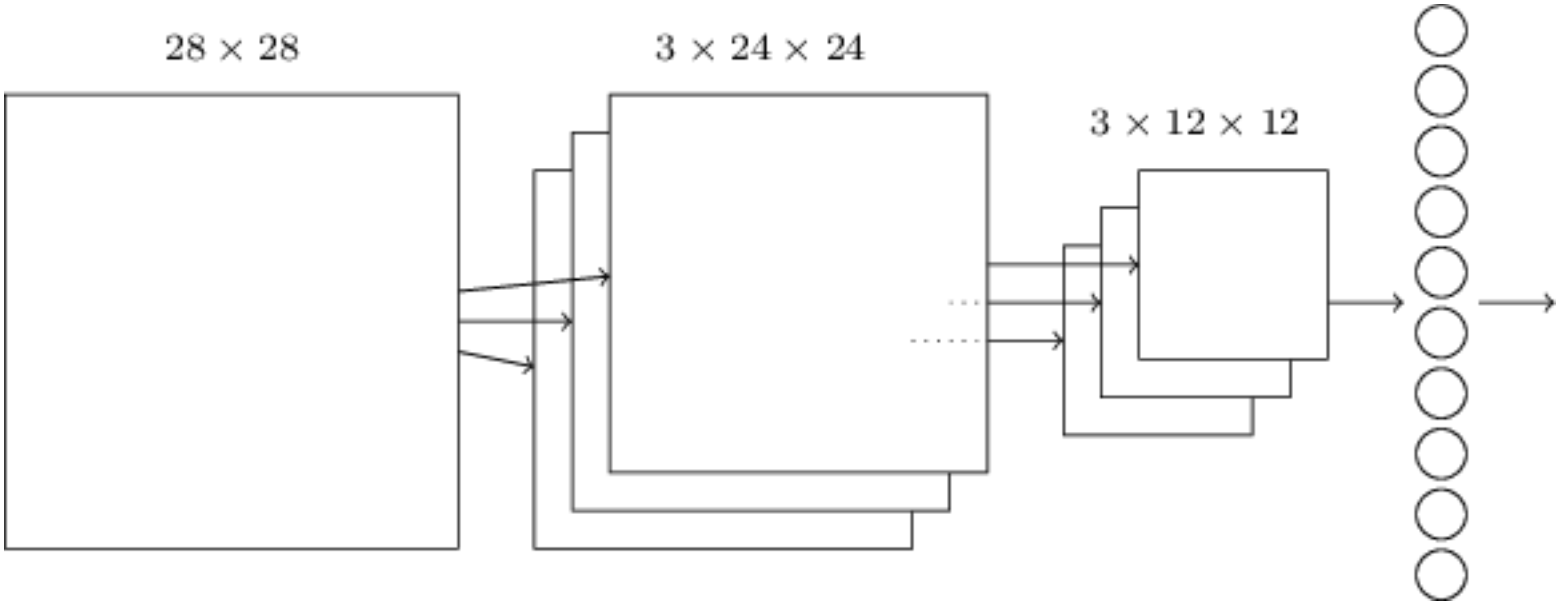


Max-Pooling

- Max-pooling as a way for the network to ask whether a given feature is found anywhere in a region of the image. It then **throws away the exact positional information**.
- Once a feature has been found, its exact location isn't as important as its rough location relative to other features.
- A big benefit is that there are many fewer pooled features, and so this helps **reduce the number of parameters needed in later layers**.



Putting altogether



- The final, output layer is a **fully connected** one
- The transfer function can be a *soft-max function*, to probabilistically weight each possible output (e.g., for a classification task)

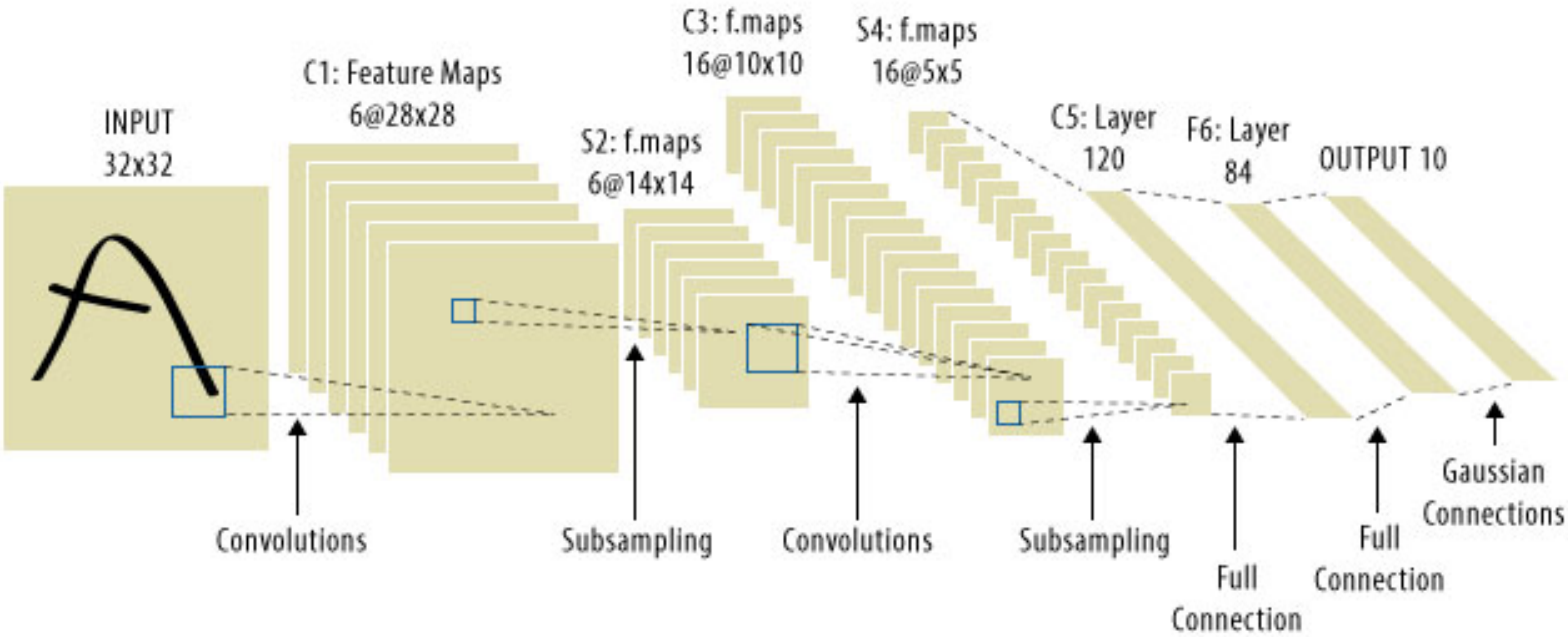
Soft-Max function

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \dots, K.$$

- The soft-max function σ "squashes" a K-dimensional real-valued vector $\mathbf{z}$ to a K-dimensional [0,1] normalized vector
- In the final, fully connected layer, σ can be used to express the probability of the j-th component of the output $\boldsymbol{y}$ (e.g, the probability that the digit in the image sample $\mathbf{x}$ is "7")

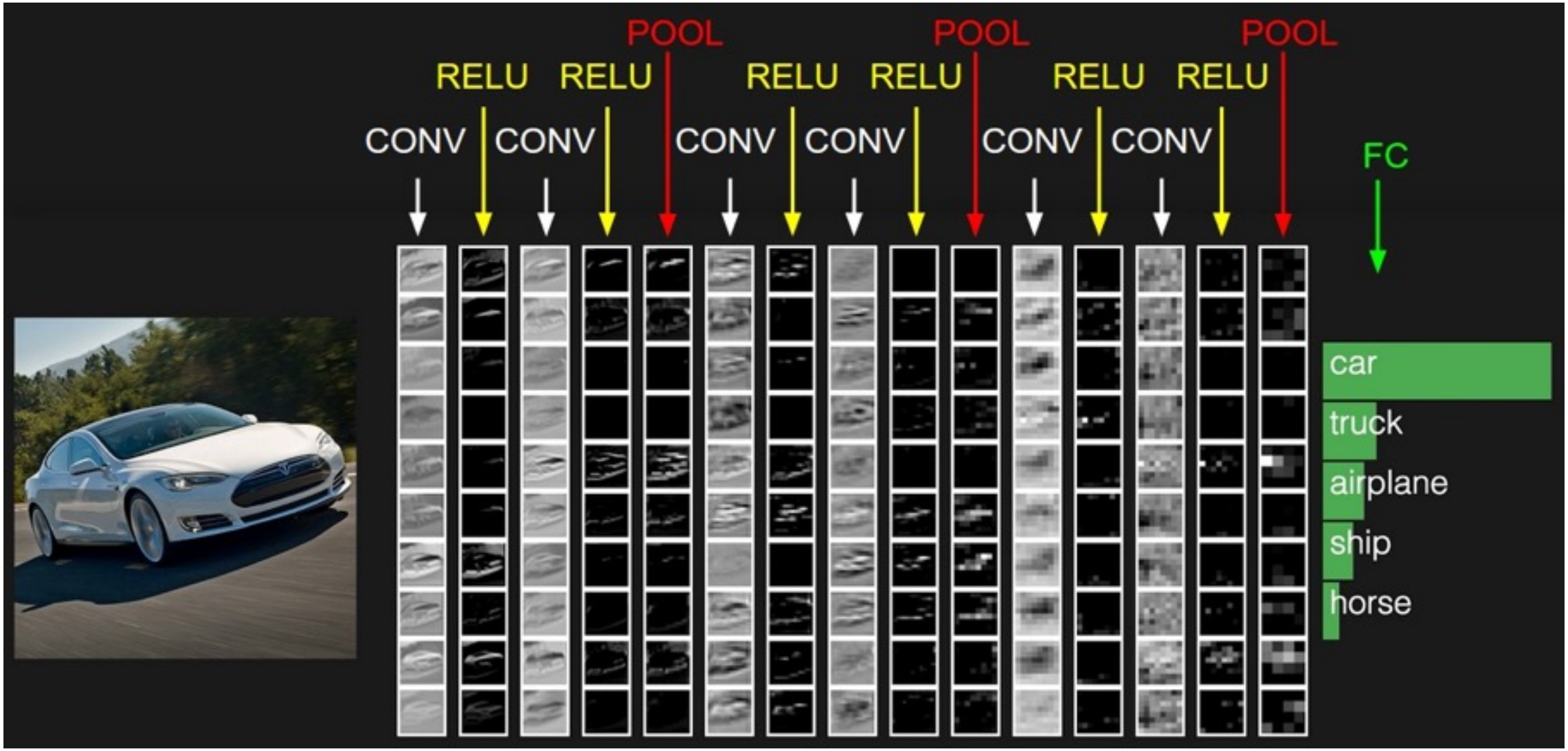
$$P(y = j|\mathbf{x}) = \frac{e^{\mathbf{x}^\mathsf{T}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\mathsf{T}\mathbf{w}_k}}$$

Convolutional NN



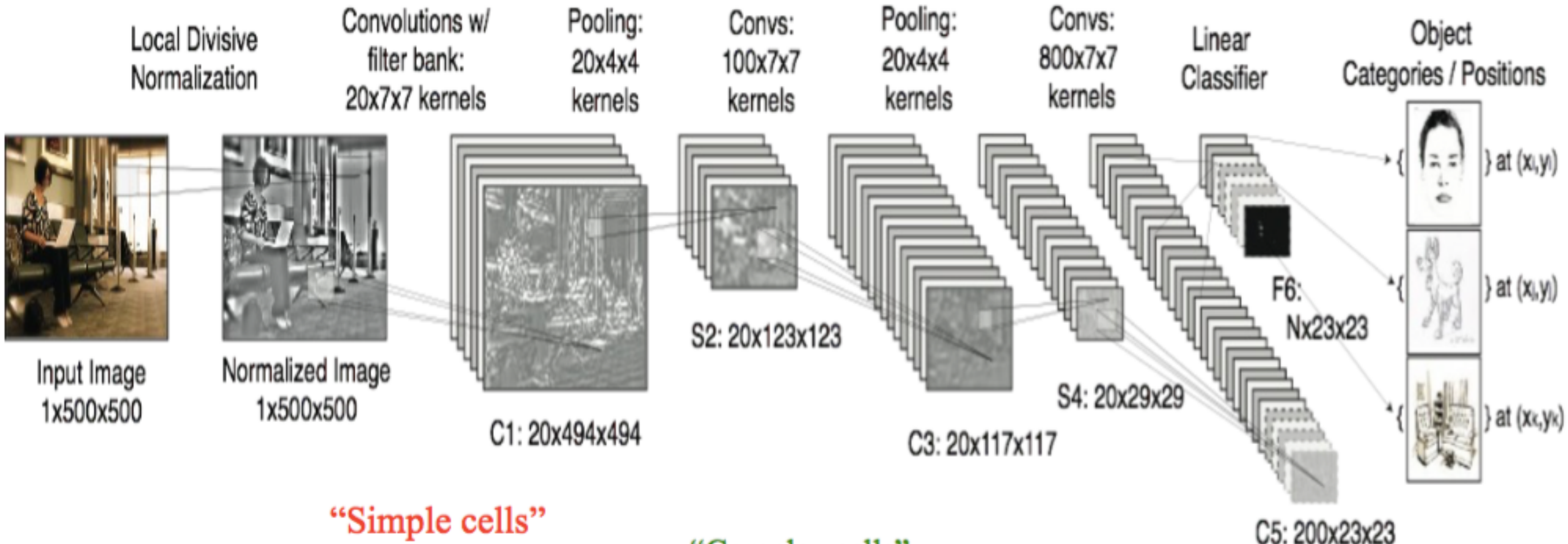
340,098 connections, but *only* 60,000 free, trainable parameters thanks to weight sharing

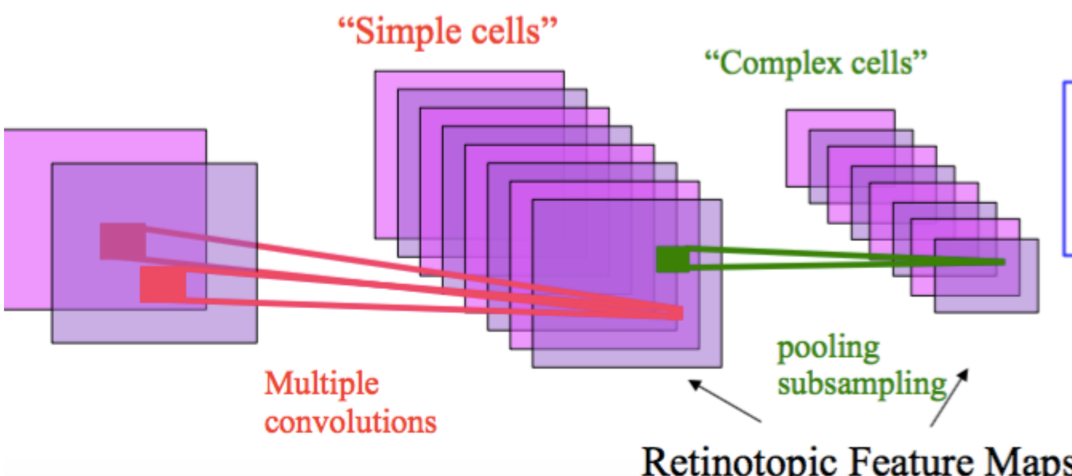
Convolutional NN



http://cs231n.github.io/

Convolutional NN

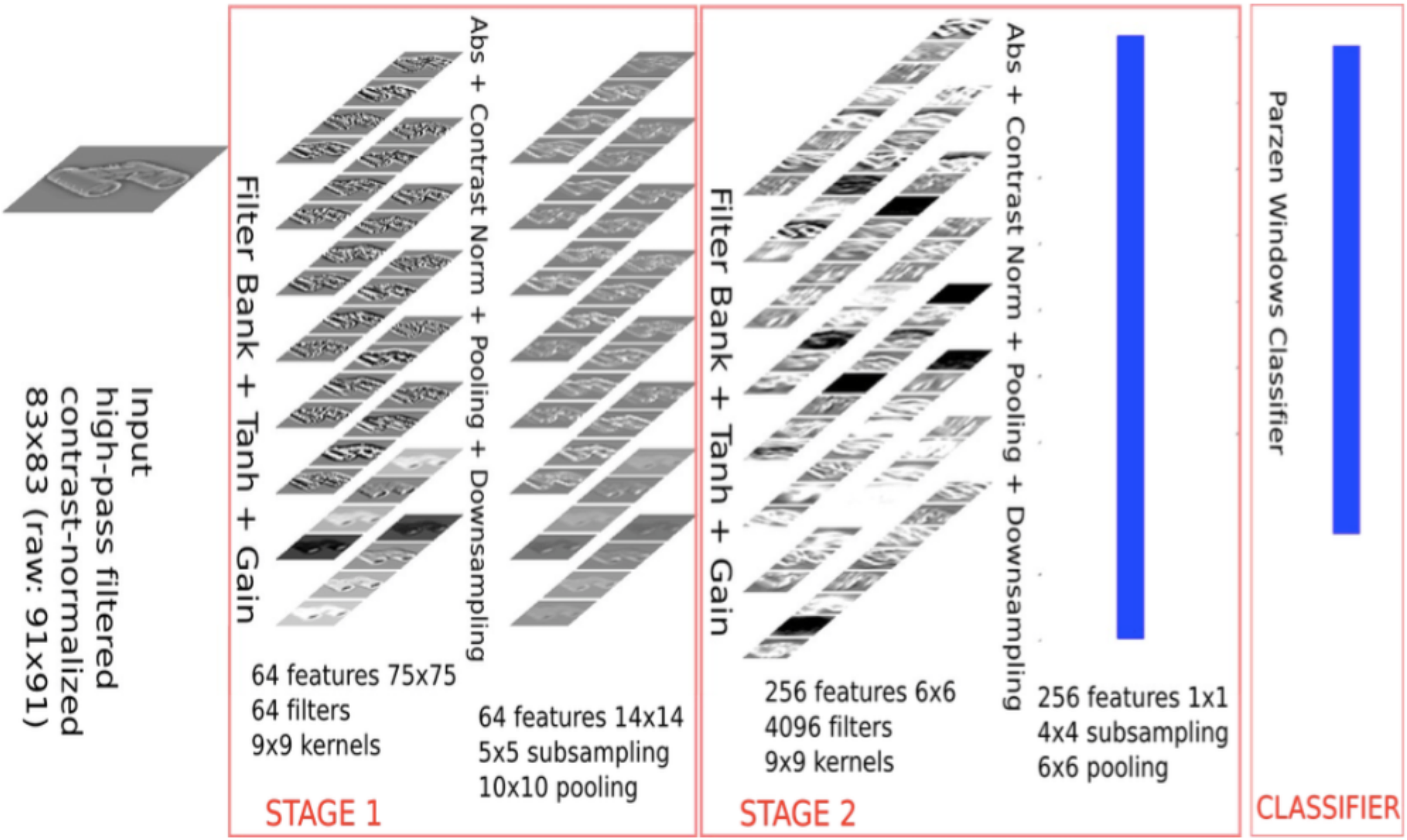




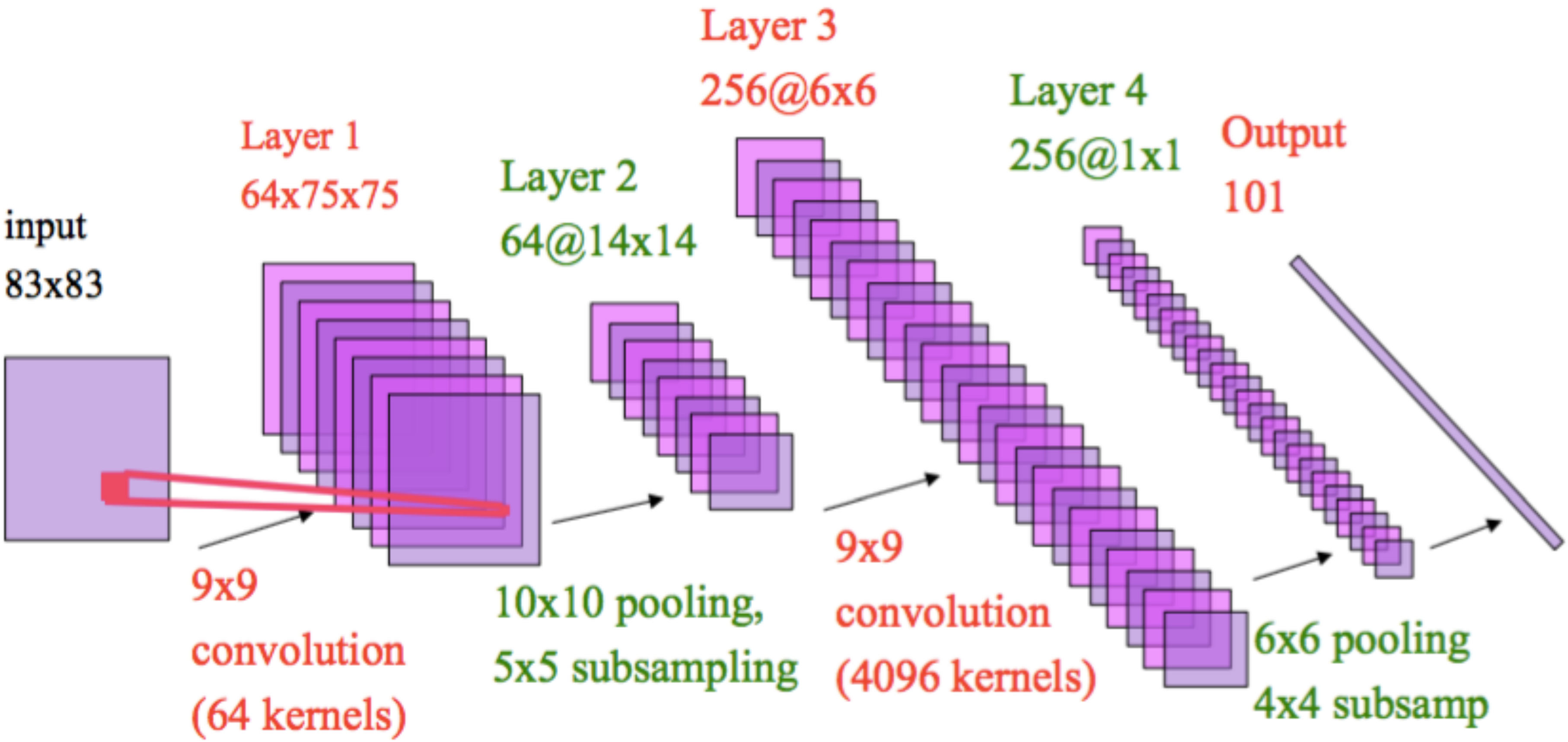
- Training is supervised
- With stochastic gradient descent

[LeCun et al. 89]
[LeCun et al. 98]

Convolutional NN



Convolutional NN



CONVOLUTIONAL NN

Object Recognition [Krizhevsky, Sutskever, Hinton 2012] Y LeCun MA Ranzato

- Method: large convolutional net
  - 650K neurons, 832M synapses, 60M parameters
  - Trained with backprop on GPU
  - Trained "with all the tricks Yann came up with in the last 20 years, plus dropout" (Hinton, NIPS 2012)
  - Rectification, contrast normalization,...
- Error rate: 15% (whenever correct class isn't in top 5)
- Previous state of the art: 25% error

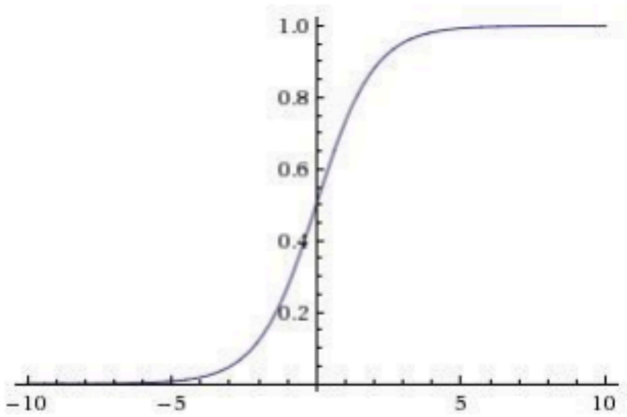
- A REVOLUTION IN COMPUTER VISION

- Acquired by Google in Jan 2013
- Deployed in Google+ Photo Tagging in May 2013

Learning / Optimization?

- **Modified back propagation**
- CNNs use *weight sharing* as opposed to feed-forward networks. During both forward and back-propagation convolutions have to be used where the weights and the activations are the functions in the convolution equation.
- *Pooling layers* do not do any learning themselves hence during forward pass, the “winning unit” has its index noted and consequently the gradient is passed back to this unit during the backward pass in the case of max-pooling

Which activation functions?



Sigmoid

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

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

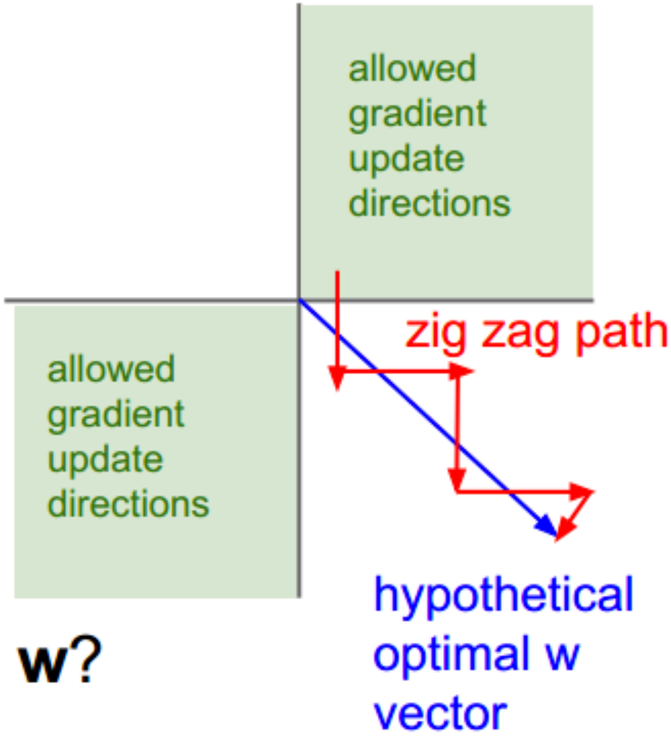
2 BIG problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered

Which activation function?

Consider what happens when the input to a neuron is always positive...

$$f\left(\sum_i w_i x_i + b\right)$$

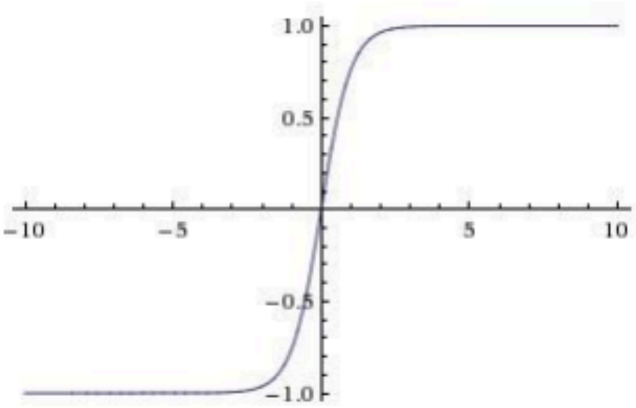


What can we say about the gradients on **w**?

Always all positive or all negative :(

(this is also why you want zero-mean data!)

Which activation function?

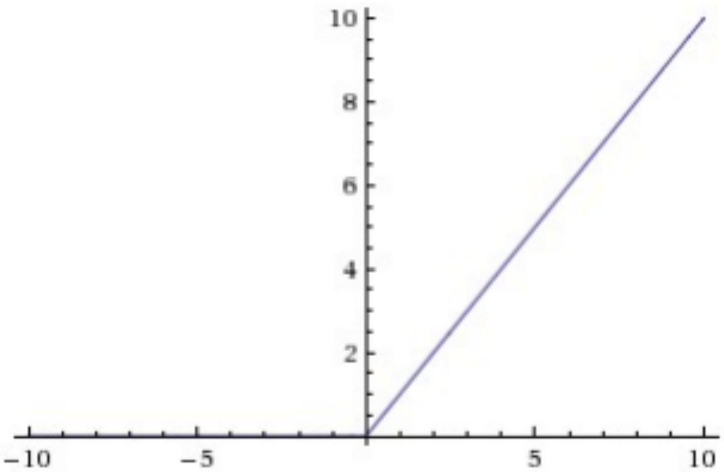


tanh(x)

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

Which activation function?

The currently most popular choice!



$$f(x) = \max\{0, x\}$$

ReLU

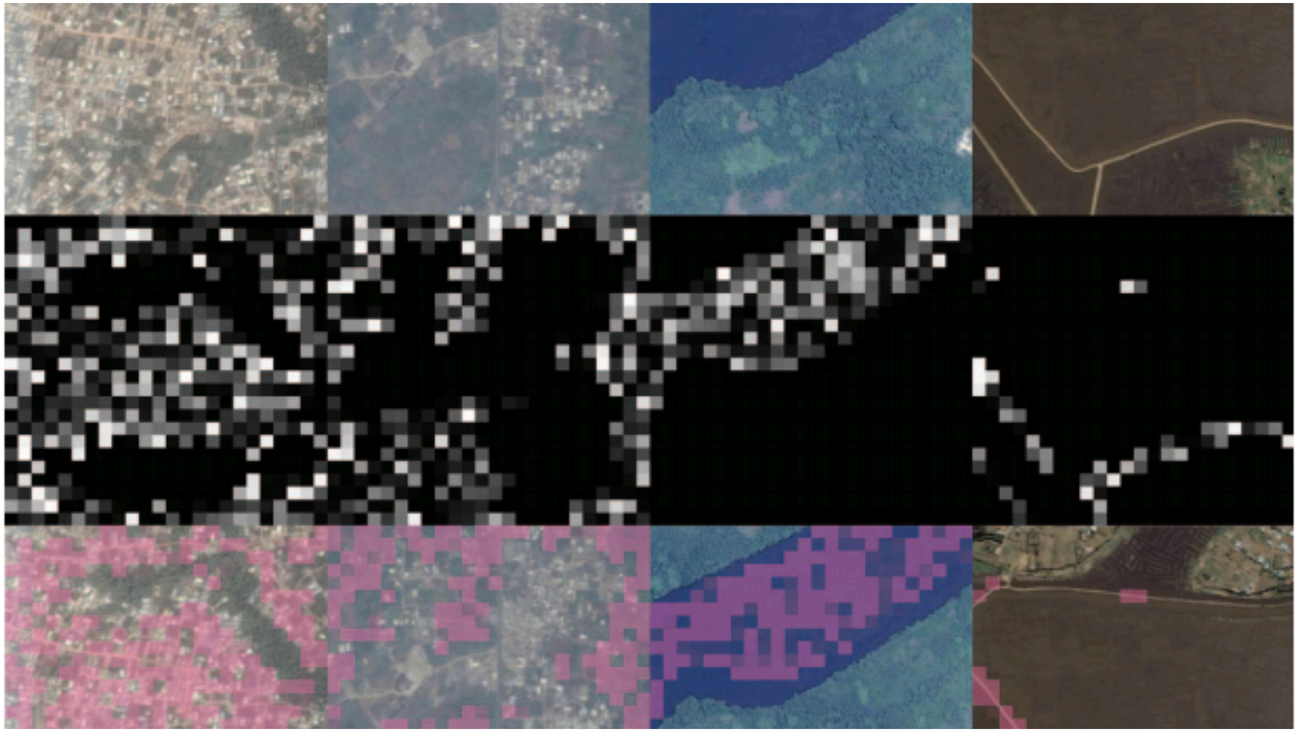
What if I don't have much data?

- In practice, very few people train an entire Convolutional Network from scratch (with random initialization)
- It is (usually) hard to have a dataset of sufficient size!
- It is common to *pretrain* (maybe for days/weeks) a CNN on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the trained CNN either as an **initialization** or a **fixed feature extractor** for the task of interest.
- → Transfer learning!

Transfer learning scenarios

- **Pretrained CNN as fixed feature extractor**: remove the last fully-connected layer, treat the rest of the CNN as a fixed feature extractor for the new dataset, add the last classification layer, and *retrain the final classifier* on top of the CNN
- **Fine-tuning the pretrained CNN**: As in the previous scenario, but in addition *fine-tune* the weights of the pretrained network by continuing the back-propagation. It is possible to fine-tune all the layers, or to keep some of the earlier layers fixed (e.g., to avoid overfitting) and only fine-tune some higher-level portions of the network (that usually learn features that are more specific to the training dataset)

Predicting Poverty Using Deep Transfer Learning (Ermon & colleagues)



What You Should Know

- Neural networks & nodes as features
  - Internal nodes can be viewed as features
  - Make more complicate function mapping input to output
- Benefits of deep over shallow
  - Number of parameters need to express complicated function may be way smaller
  - Important in terms of amount of data to train / fit classifier
- Nonlinearity: choices, implications for learning
  - Sigmoid (bad), ReLu (good)
  - Increases ezpressive power (1 hidden layer, universal approximator)
  - Optimization harder (not convex, many local optima)
- How to train/fit/learn
  - Gradient descent, backpropagation
  - Be able to derive gradient for simple case and use to update w
- New ideas for tackling vision applications
  - Convolutional networks
  - Reduce # parameters, exploit nodes as filters
  - How many parameters are involved?
  - Define common node types: conv, pooling, fully connected
- What if we don't have much data?
  - Transfer learning!
  - Learn features using big data, then use for other applications