CS 558: Computer Vision 12th Set of Notes

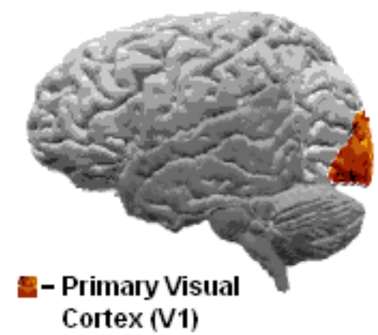
Instructor: Enrique Dunn
Webpage: www.cs.stevens.edu/~edunn
E-mail: edunn@stevens.edu
Office: NB 219

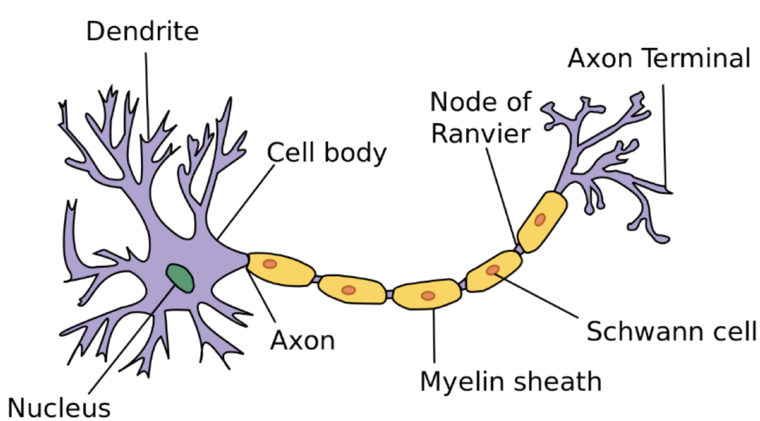
Overview

- Deep Learning for Computer Vision
  - Based on slides by M. Ranzato (mainly), S. Lazebnik, R. Fergus and Q. Zhang

Natural Neurons

- Human recognition of digits
  - visual cortices
  - neuron interaction



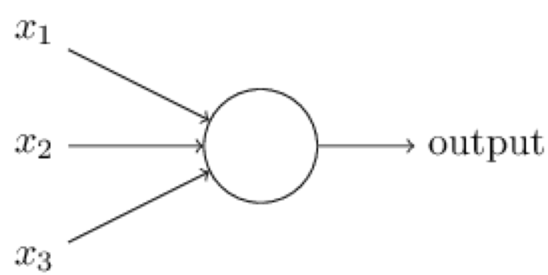


Recognizing Handwritten Digits

- How to describe a digit to a computer
  - "a 9 has a loop at the top, and a vertical stroke in the bottom right“
  - Algorithmically difficult to describe various 9s

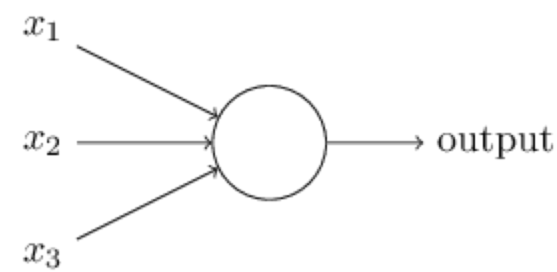
Perceptrons

- Perceptrons
  - 1950s ~ 1960s, Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts
- Standard model of artificial neurons



Binary Perceptrons

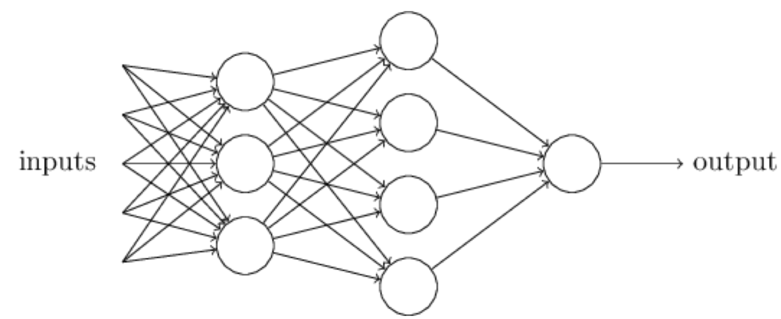
- Inputs
  - Multiple binary inputs
- Parameters
  - Thresholds & weights
- Outputs
  - Thresholded weighted linear combination



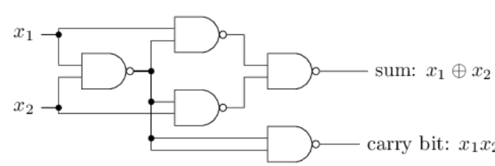
$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

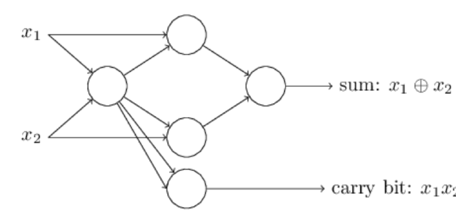
Layered Perceptrons

- Layered, complex model
  - 1st layer, 2nd layer of perceptrons
- Perceptron rule
  - Weights, thresholds
- Similarity to logical functions (NAND)



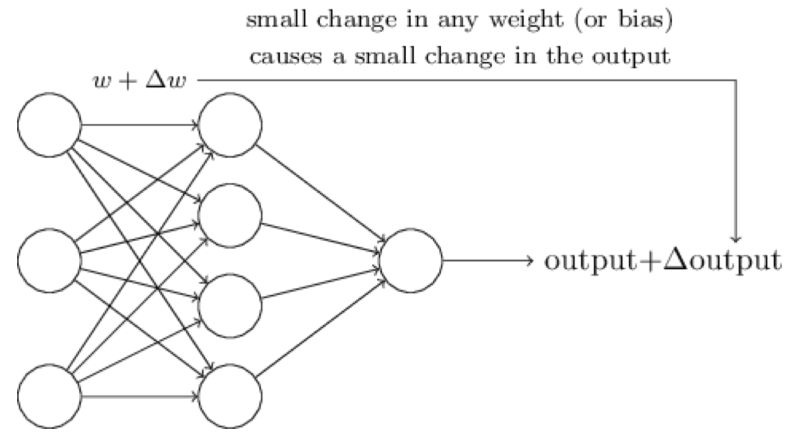
$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

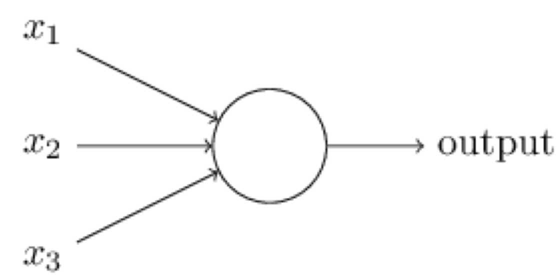




Sigmoid Neurons

- Sigmoid neurons
  - Stability
    - Small perturbation, small output change
  - Continuous inputs
  - Continuous outputs
  - Soft thresholds



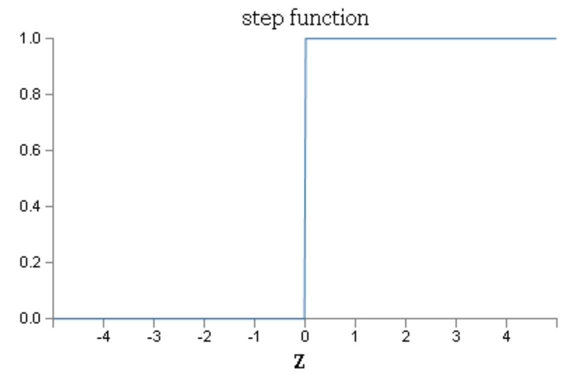


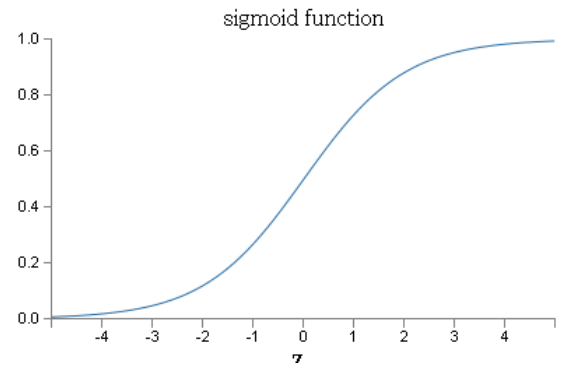
Output Functions

- Sigmoid neurons
- Output $\sigma(w \cdot x + b)$, $\sigma(z) \equiv \frac{1}{1+e^{-z}}$

$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}.$$

- Sigmoid vs conventional thresholds

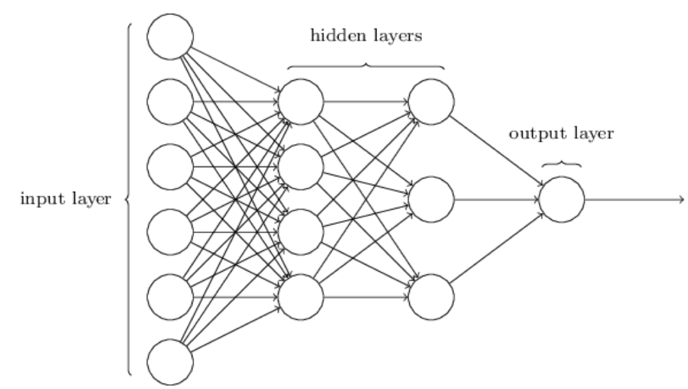




Smoothness & Differentiability

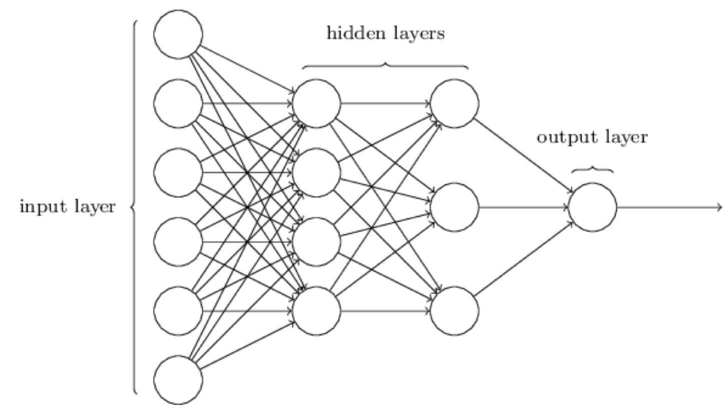
- Perturbations and Derivatives
  - Continuous function
  - Differentiable
- Layers
  - Input layers, output layers, hidden layers

$$\Delta \text{output} \approx \sum_j \frac{\partial\, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial\, \text{output}}{\partial b} \Delta b,$$



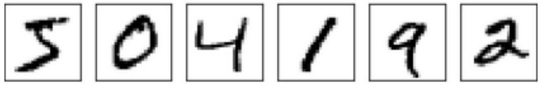
Layer Structure Design

- Design of hidden layer
  - Heuristic rules
  - Number of hidden layers vs. computational resources
  - Feedforward network
    - No loops involved

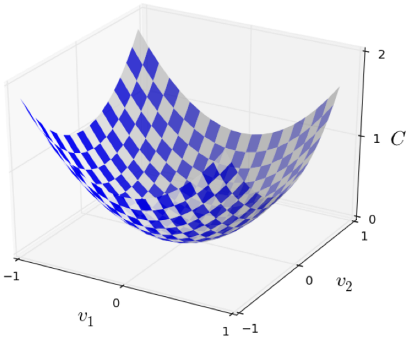


Cost Function & Optimization

- Learning with gradient descent
  - Cost function
  - Euclidean loss
  - Non-negative, smooth, differentiable



$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$



Cost Function & Optimization

- Gradient Descent

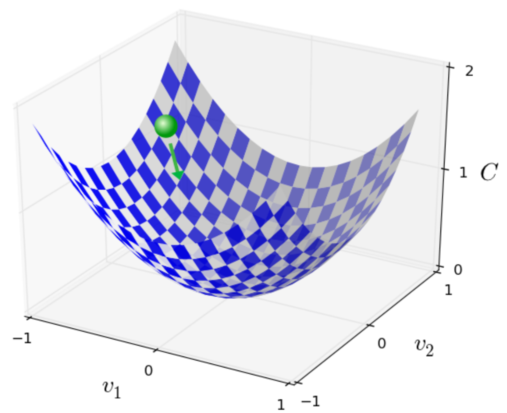
$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2.$$

- Gradient vector

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T.$$

$$\Delta C \approx \nabla C \cdot \Delta v.$$

$$v \rightarrow v' = v - \eta \nabla C.$$



Cost Function & Optimization

- Extension to multiple dimension
  - m variables $v_1, \ldots, v_m$
  - Small change in variable $\Delta v = (\Delta v_1, \ldots, \Delta v_m)^T$
  - Small change in cost $\Delta C \approx \nabla C \cdot \Delta v,$

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m}\right)^T$$

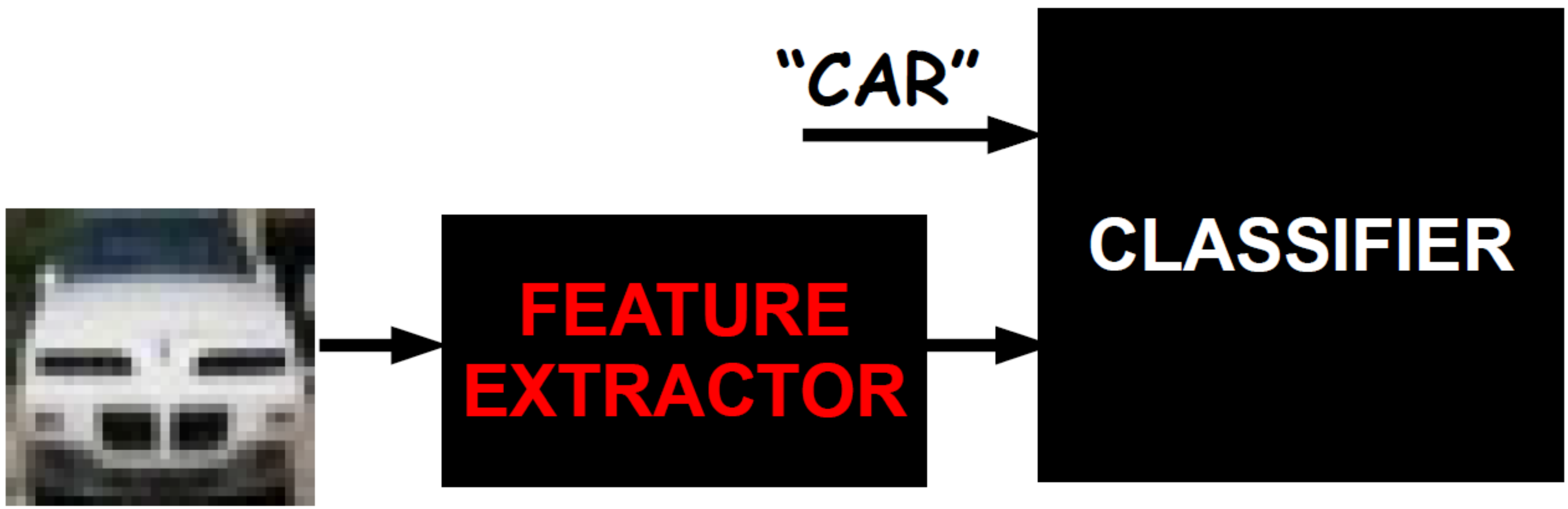
$$\Delta v = -\eta \nabla C \qquad v \rightarrow v' = v - \eta \nabla C.$$

Neural Nets for Vision

Based on Tutorials at CVPR 2012 and 2014 by

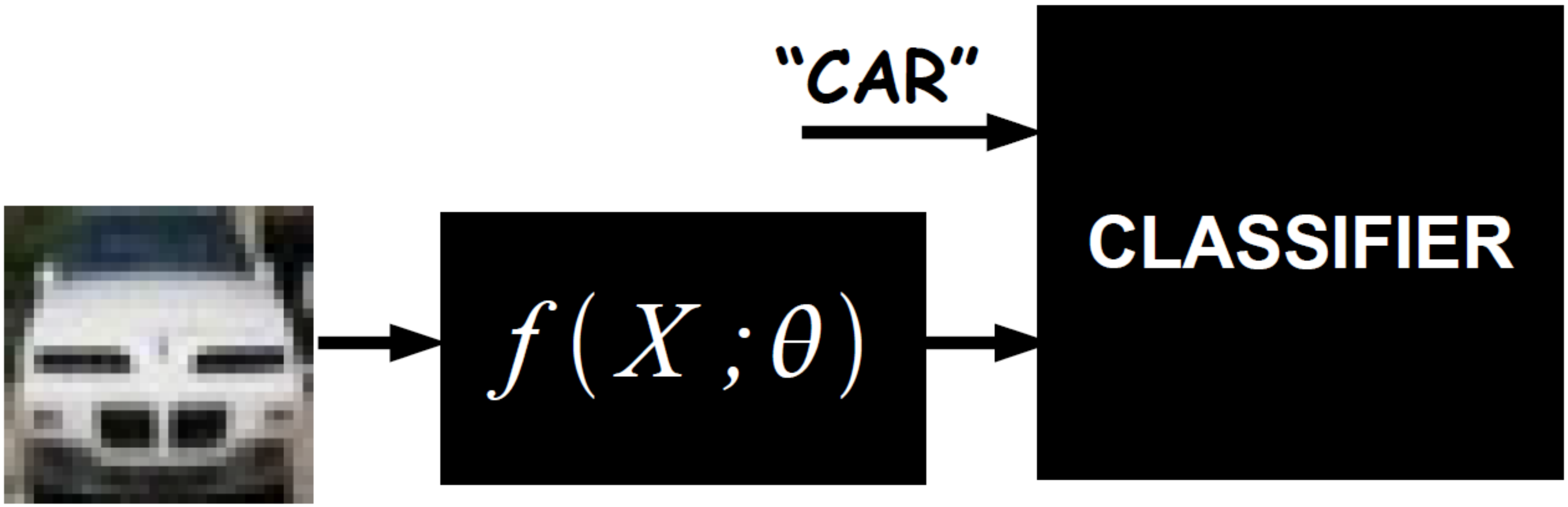
Marc'Aurelio Ranzato

Building an Object Recognition System



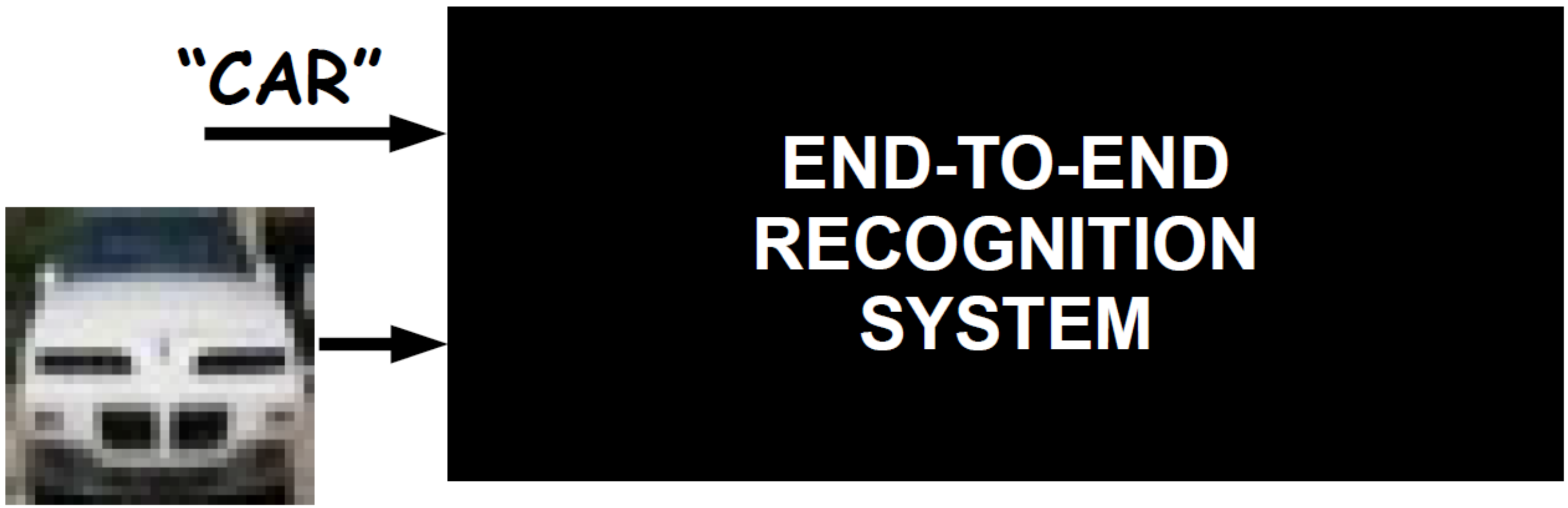
IDEA: Use data to optimize features for the given task

Building an Object Recognition System



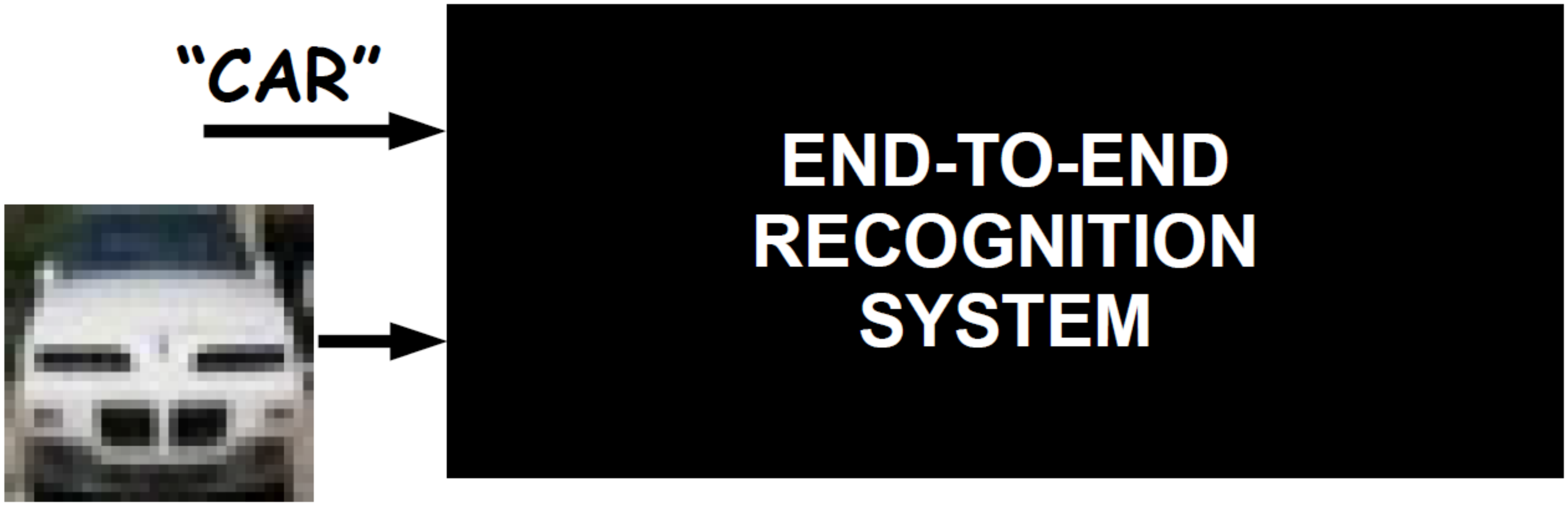
What we want: Use parameterized function such that
a) features are computed efficiently
b) features can be trained efficiently

Building an Object Recognition System



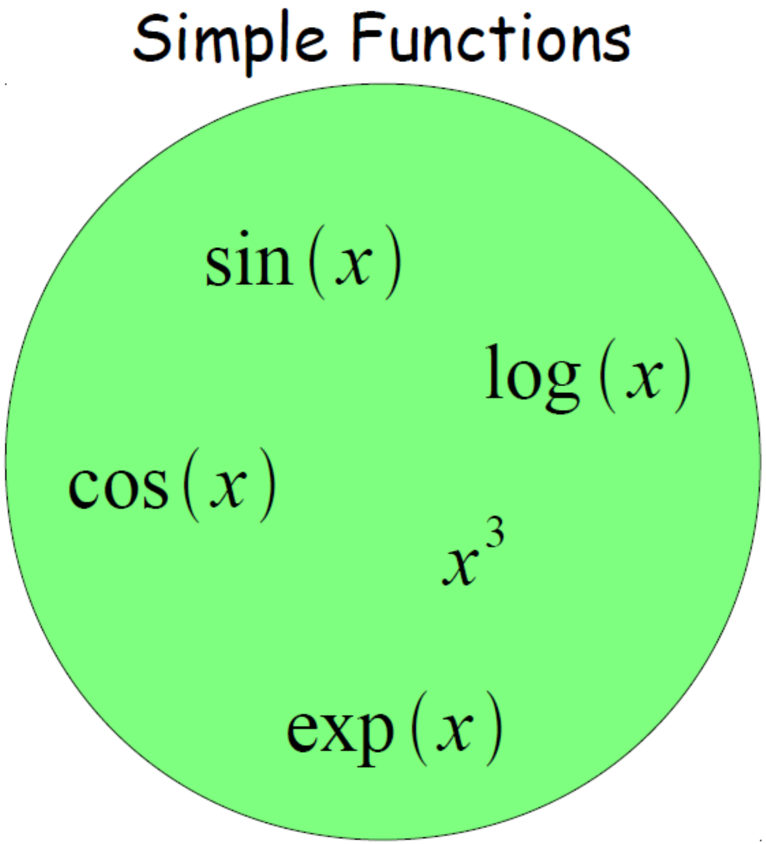
- Everything becomes adaptive
- No distinction between feature extractor and classifier
- Big non-linear system trained from raw pixels to labels

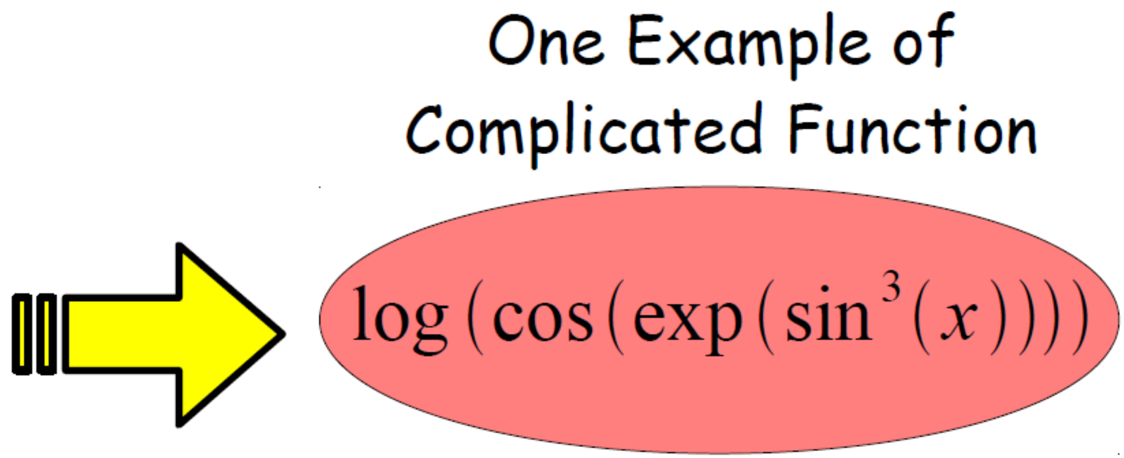
Building an Object Recognition System



Q: How can we build such a highly non-linear system?
A: By combining simple building blocks we can make more and more complex systems

Building a Complicated Function



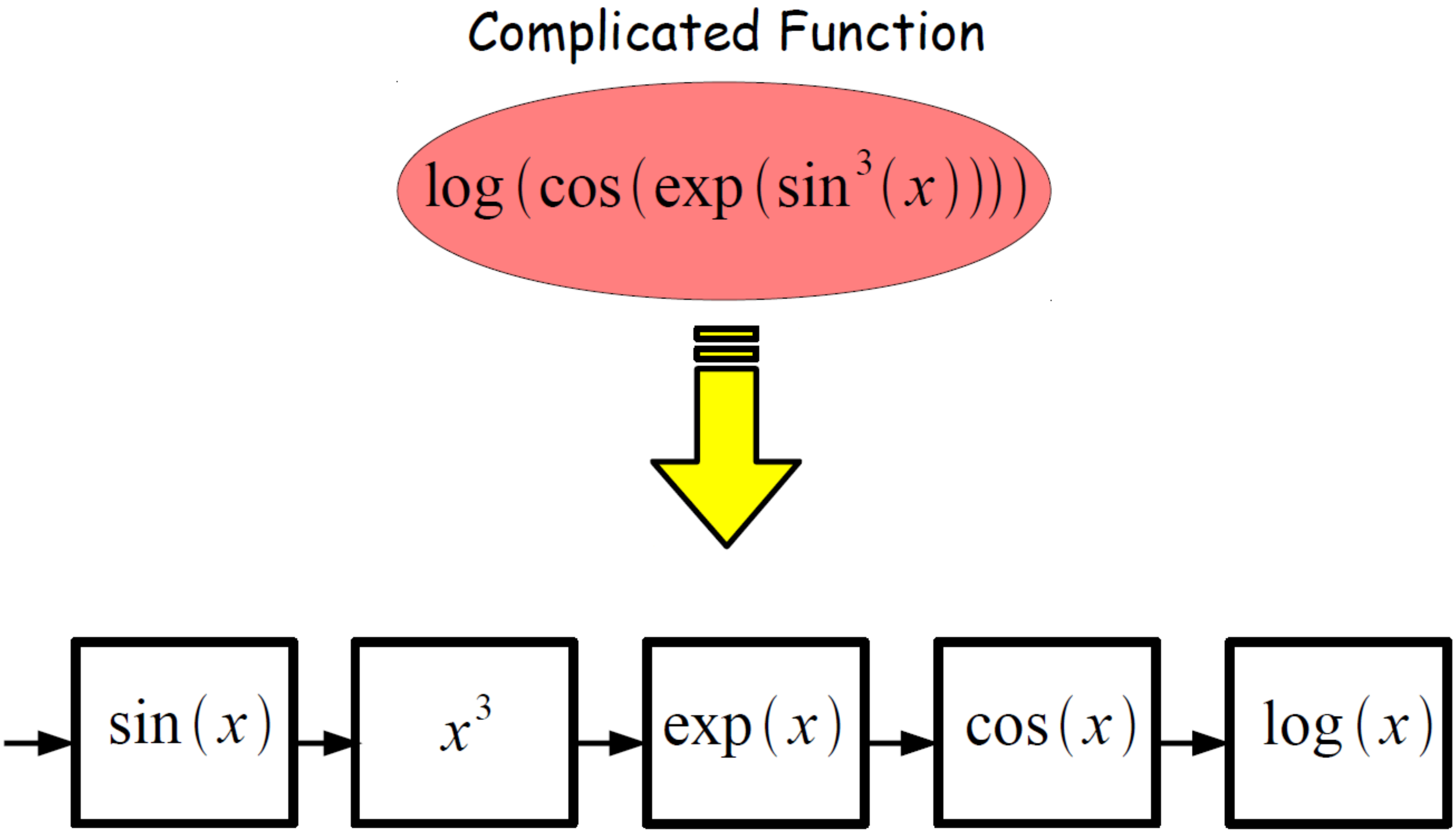


- Function composition is at the core of deep learning methods
- Each “simple function” will have parameters subject to training

Building a Complicated Function

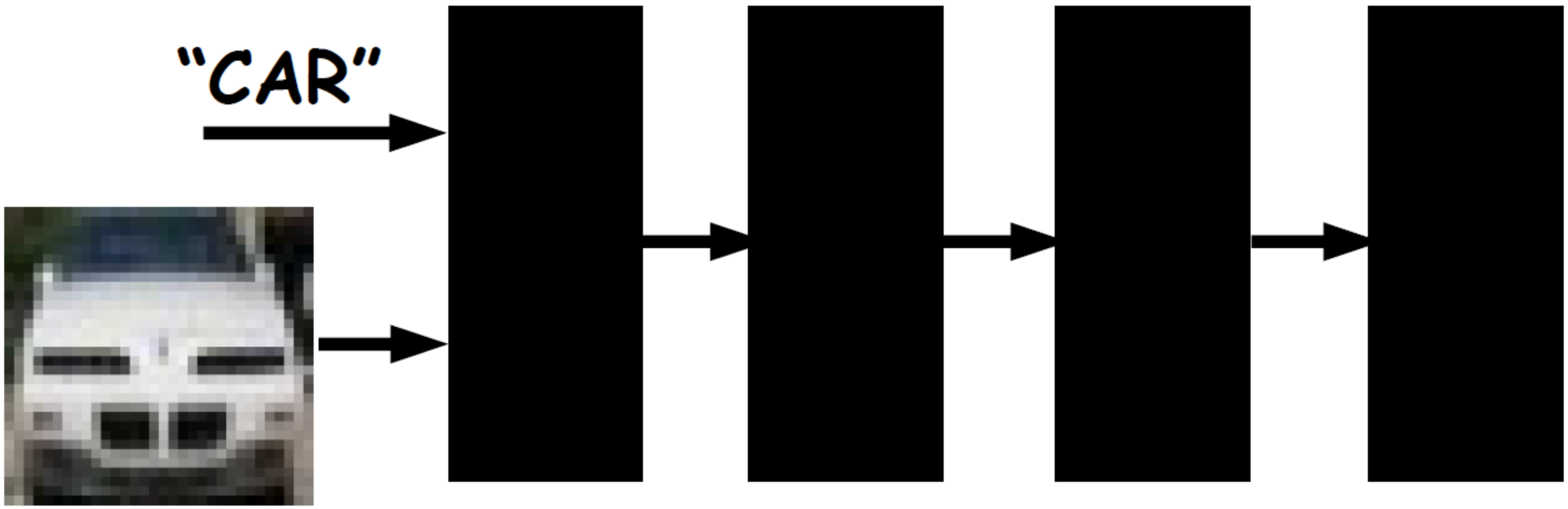
- Function composition is at the core of deep learning methods
- Each “simple function” will have parameters subject to training

Implementing a Complicated Function



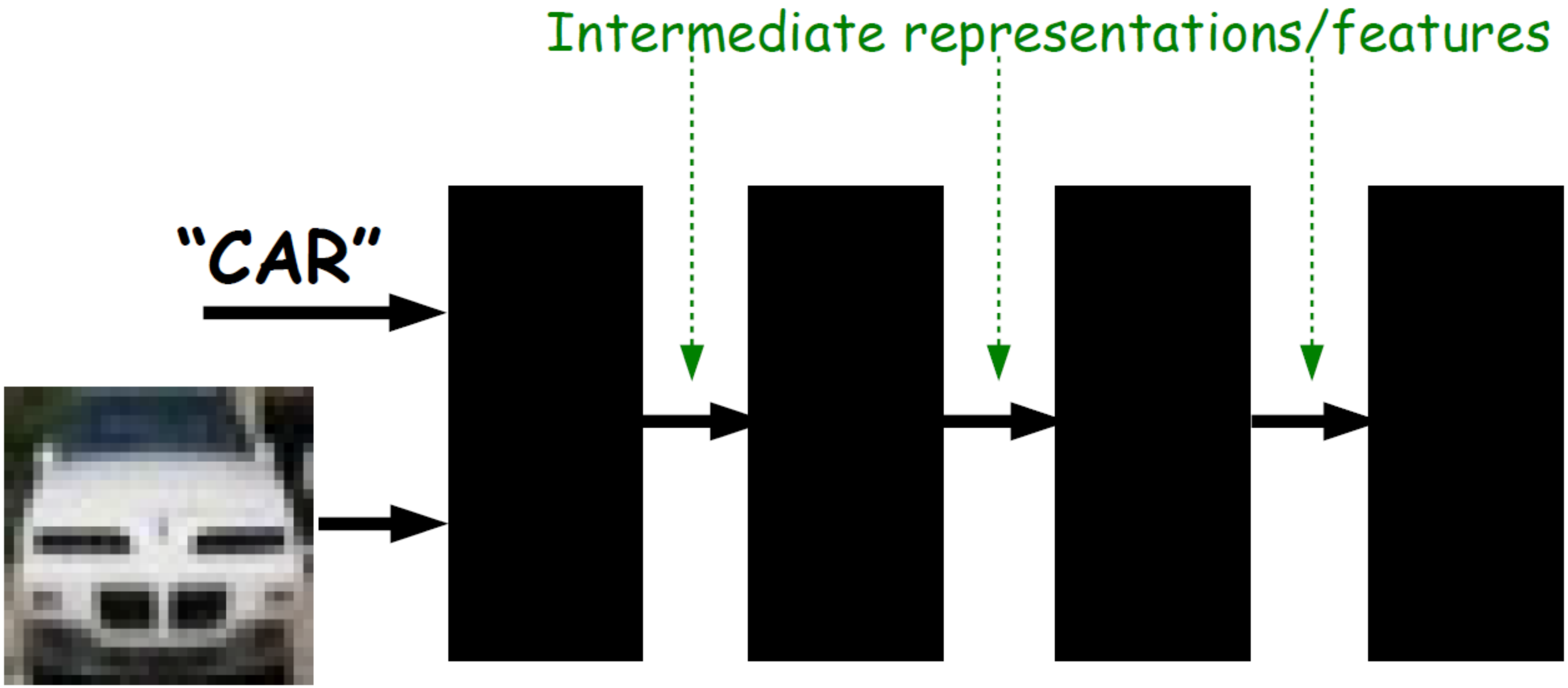
Intuition Behind Deep Neural Nets

Intuition Behind Deep Neural Nets



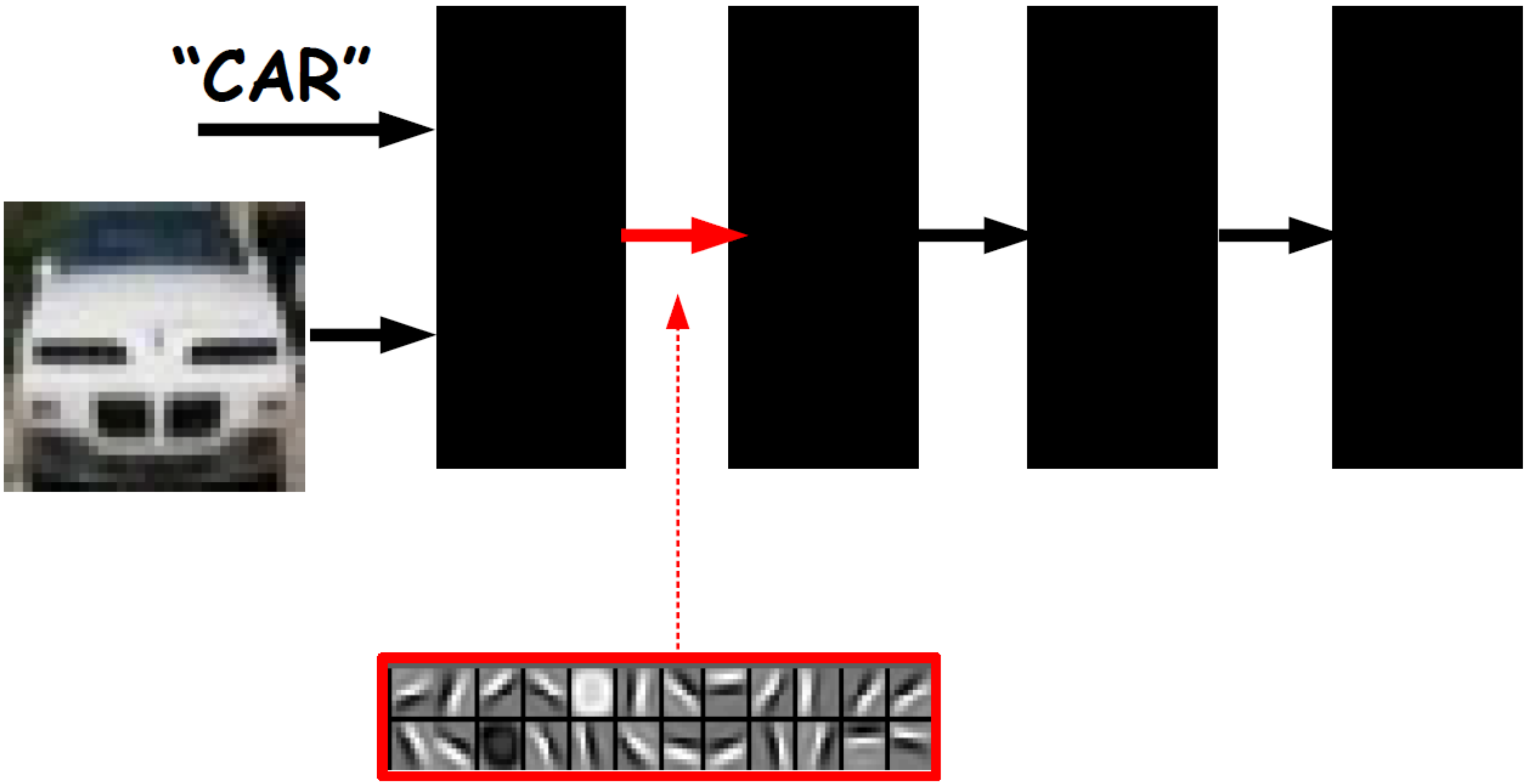
Each black box can have trainable parameters. Their composition makes a highly non-linear system.

Intuition Behind Deep Neural Nets

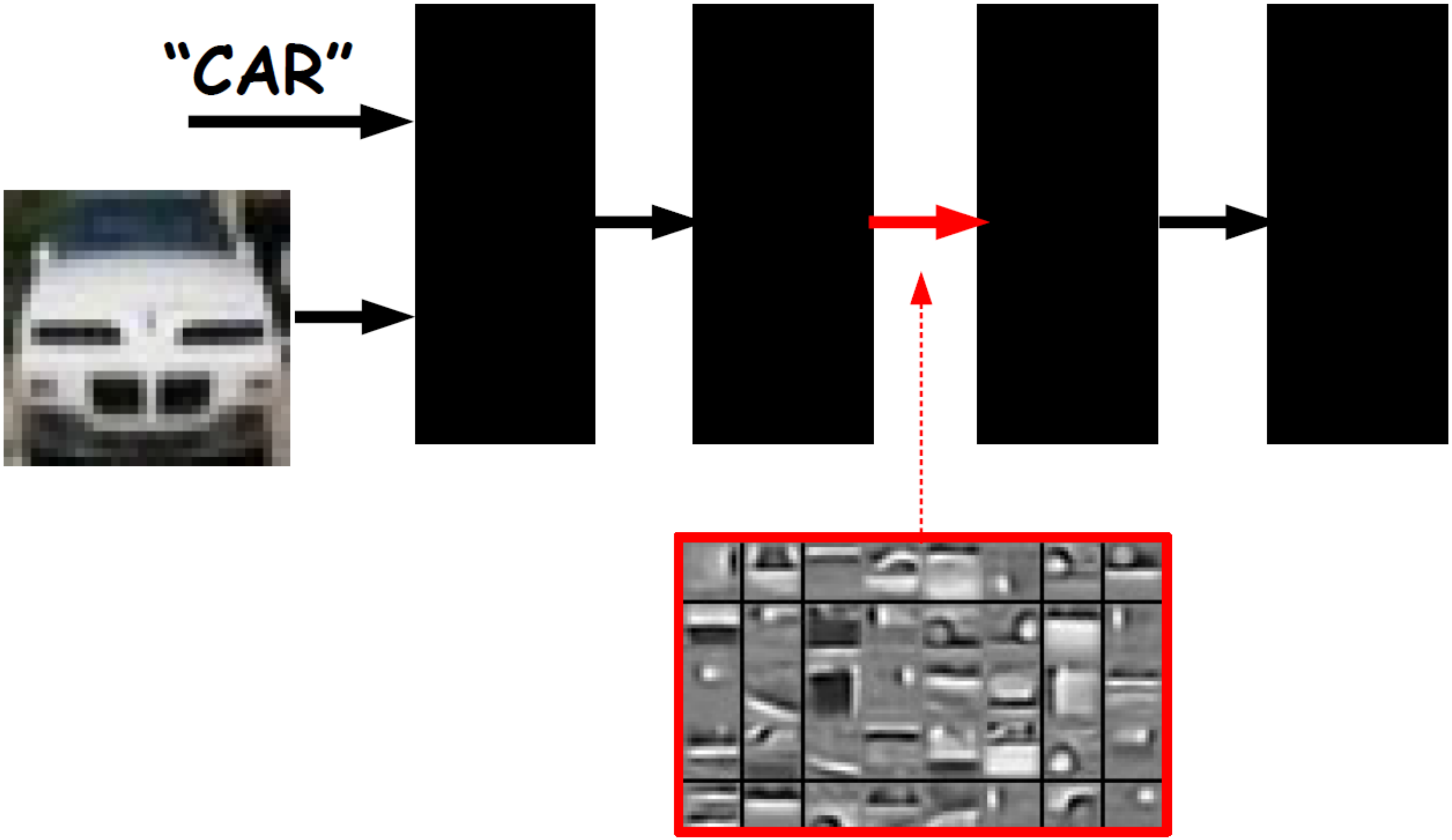


System produces hierarchy of features

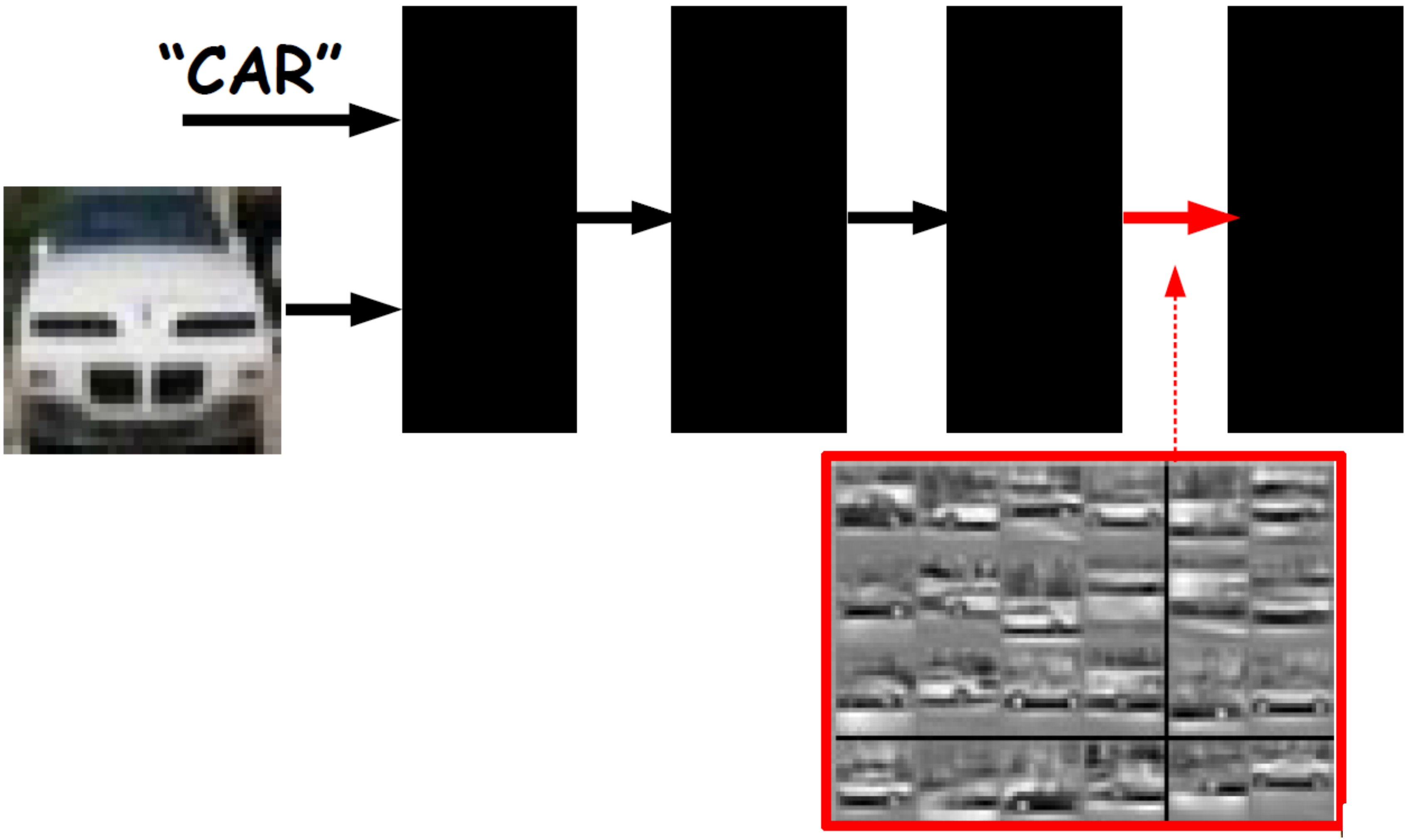
Intuition Behind Deep Neural Nets



Intuition Behind Deep Neural Nets



Intuition Behind Deep Neural Nets



Key Ideas of Neural Nets

IDEA # 1

Learn features from data

IDEA # 2

Use differentiable functions that produce features efficiently

IDEA # 3

End-to-end learning:
no distinction between feature extractor and classifier

IDEA # 4

“Deep” architectures:
cascade of simpler non-linear modules

Key Questions

- What is the input-output mapping?
- How are parameters trained?
- How computational expensive is it?
- How well does it work?

Supervised Deep Learning

Marc'Aurelio Ranzato

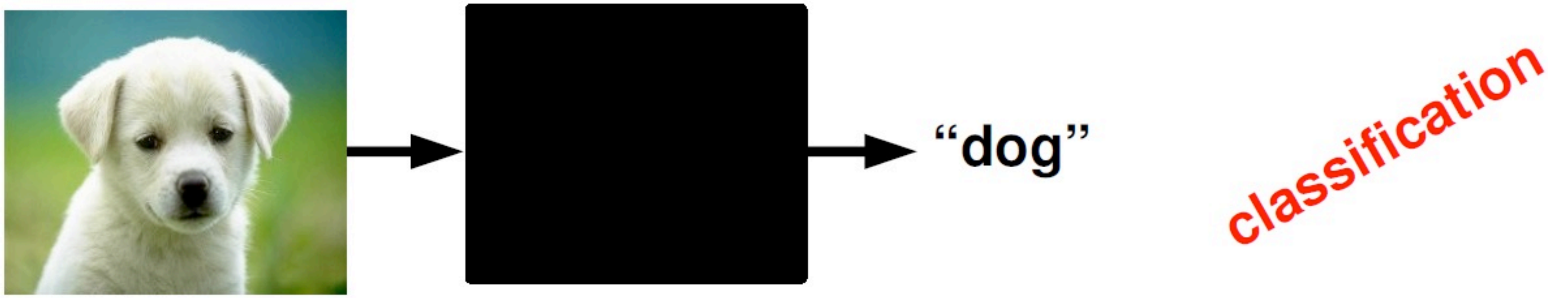
Supervised Learning

$\{(x_i, y_i), i=1... P\}$ training set
x_i i-th input training example
y_i i-th target label
P number of training examples

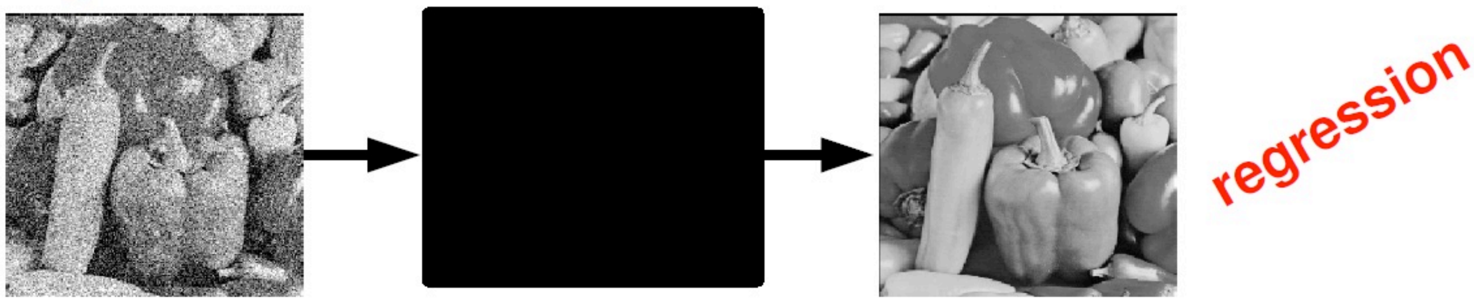
- Goal: predict the target label of unseen inputs

Supervised Learning Examples

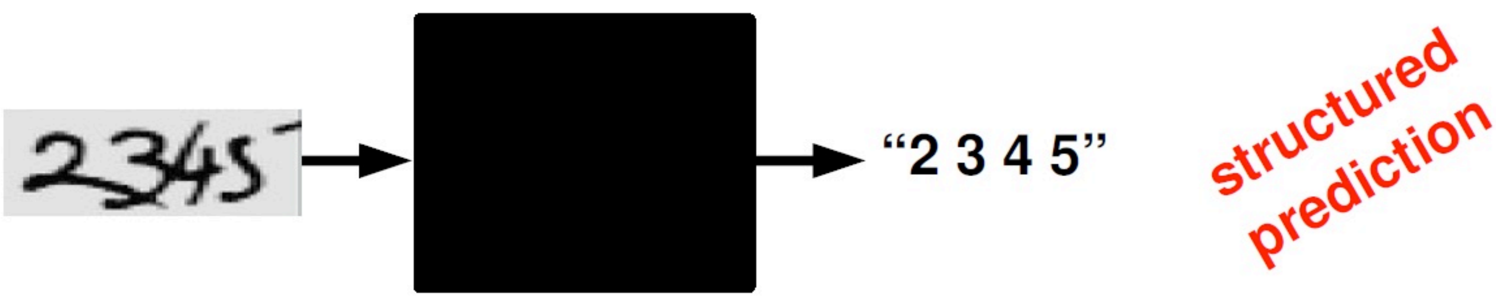
Classification



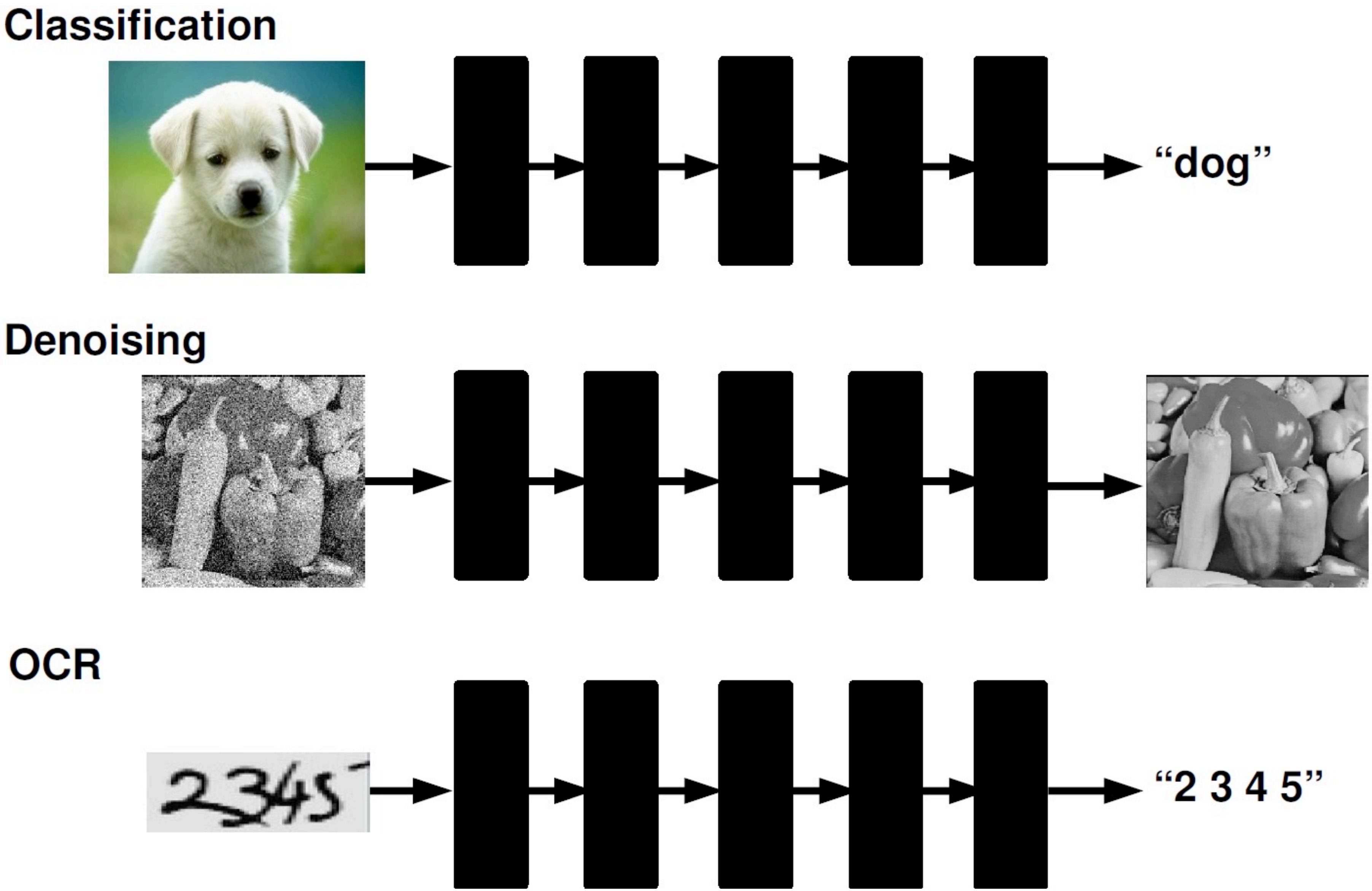
Denoising



OCR



Supervised Deep Learning



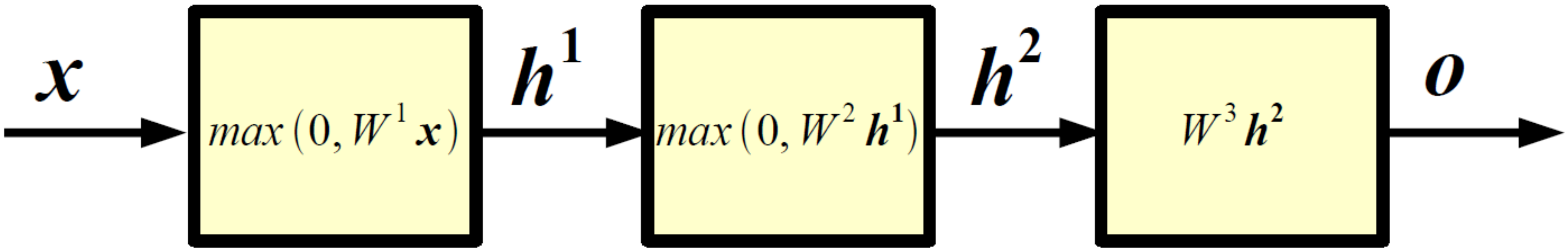
Neural Networks

Assumptions (for the next few slides):
- The input image is vectorized (disregard the spatial layout of pixels)
- The target label is discrete (classification)

Question: what class of functions shall we consider to map the input into the output?
Answer: composition of simpler functions.

Follow-up questions: Why not a linear combination? What are the "simpler" functions? What is the interpretation?
Answer: later...

Neural Networks: example



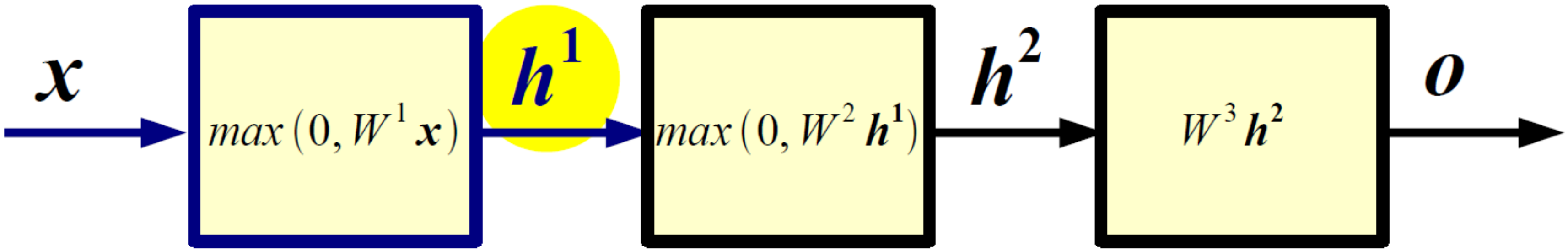
x input
h^1 1-st layer hidden units
h^2 2-nd layer hidden units
o output

Example of a 2 hidden layer neural network (or 4 layer network, counting also input and output)

Forward Propagation

Forward propagation is the process of computing the output of the network given its input

Forward Propagation



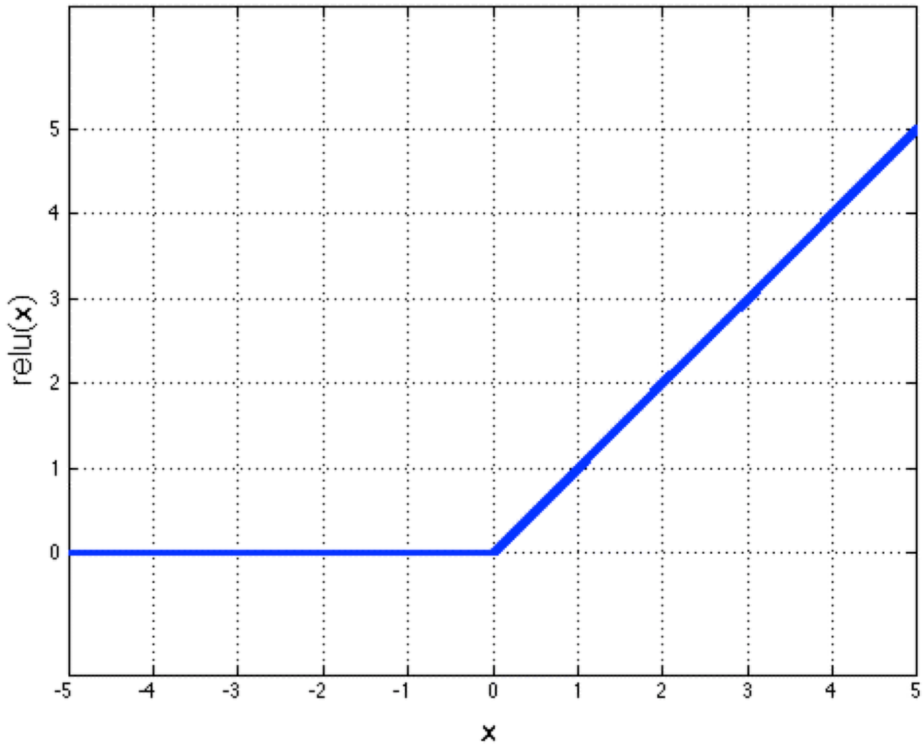
$\boldsymbol{x} \in R^D \quad W^1 \in R^{N_1 \times D} \quad \boldsymbol{b}^1 \in R^{N_1} \quad \boldsymbol{h}^1 \in R^{N_1}$

$$\boldsymbol{h}^1 = max(0, W^1 \boldsymbol{x} + \boldsymbol{b}^1)$$

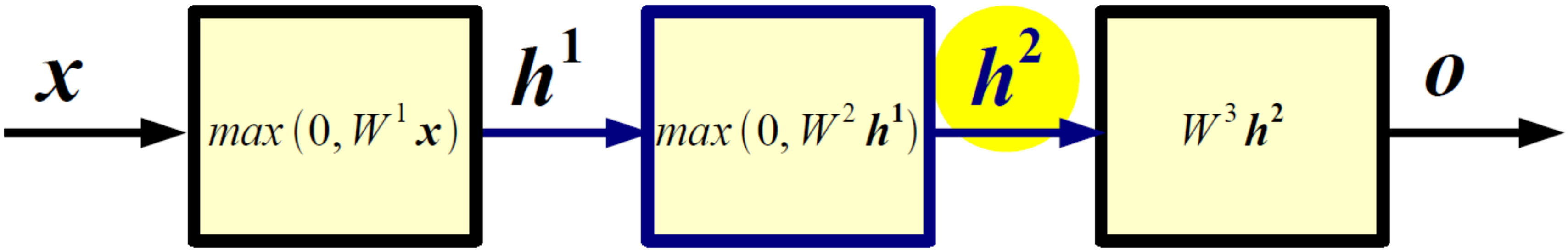
W^1 1^{st} layer weight matrix or weights
$\boldsymbol{b}^1$ 1^{st} layer biases

- The non-linearity u=max(0,v) is called ReLU in the DL literature.
- Each output hidden unit takes as input all the units at the previous layer: each such layer is called "fully connected"

Rectified Linear Unit (ReLU)



Forward Propagation



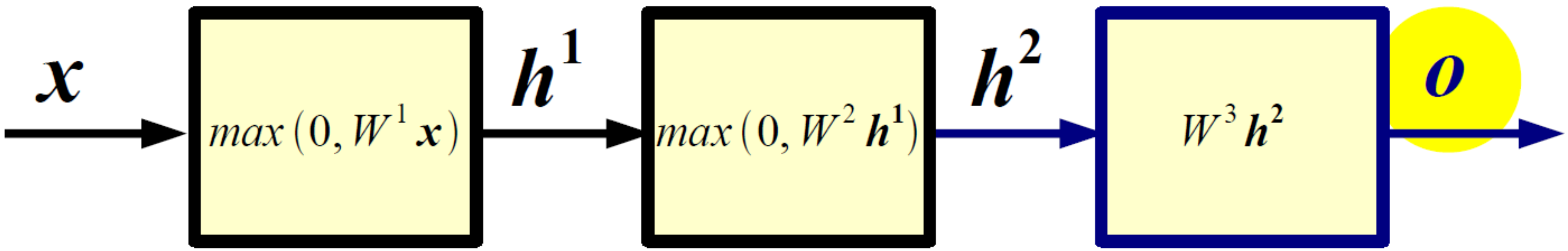
2 is index, not squared

$\boldsymbol{h}^1 \in R^{N_1}$ $\quad W^2 \in R^{N_2 \times N_1}$ $\quad \boldsymbol{b}^2 \in R^{N_2}$ $\quad \boldsymbol{h}^2 \in R^{N_2}$

$$\boldsymbol{h}^2 = max\,(0, W^2 \boldsymbol{h}^1 + \boldsymbol{b}^2)$$

W^2 2nd layer weight matrix or weights
$\boldsymbol{b}^2$ 2nd layer biases

Forward Propagation

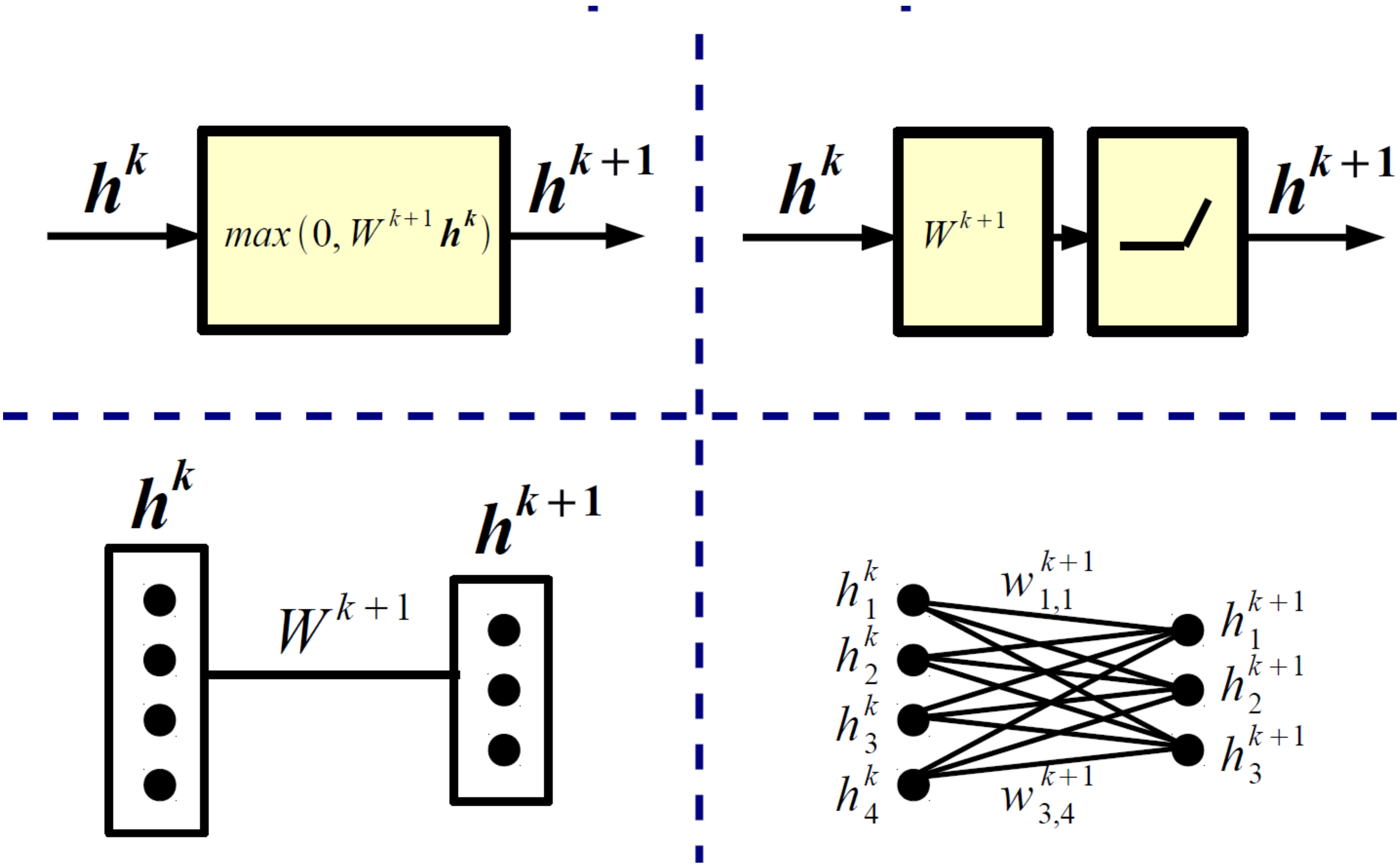


$\boldsymbol{h}^2 \in R^{N_2}$ $W^3 \in R^{N_3 \times N_2}$ $\boldsymbol{b}^3 \in R^{N_3}$ $\boldsymbol{o} \in R^{N_3}$

$$\boldsymbol{o} = max(0, W^3 \boldsymbol{h}^2 + \boldsymbol{b}^3)$$

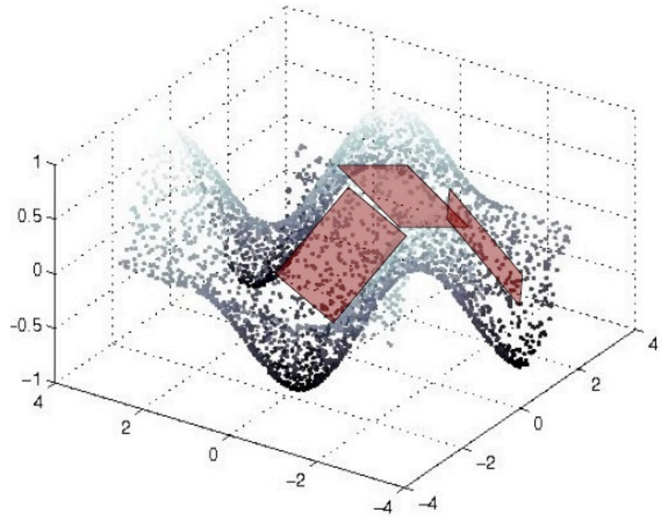
W^3 3rd layer weight matrix or weights
$\boldsymbol{b}^3$ 3rd layer biases

Alternative Graphical Representations



Interpretation

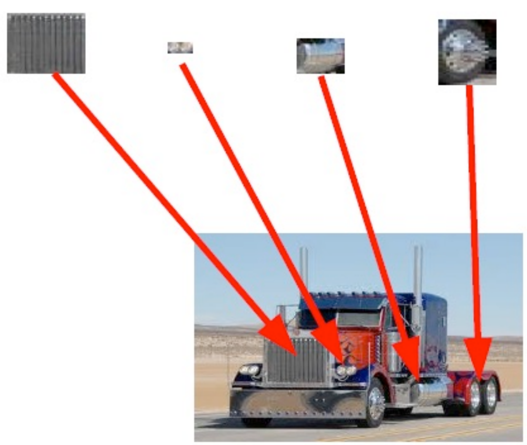
- Question: Why can't the mapping between layers be linear?
- Answer: Because composition of linear functions is a linear function. Neural network would reduce to (1 layer) logistic regression.
- Question: What do ReLU layers accomplish?
- Answer: Piece-wise linear tiling: mapping is locally linear.



Interpretation

- Question: Why do we need many layers?
- Answer: When input has hierarchical structure, the use of a hierarchical architecture is potentially more efficient because intermediate computations can be re-used. DL architectures are efficient also because they use distributed representations which are shared across classes.

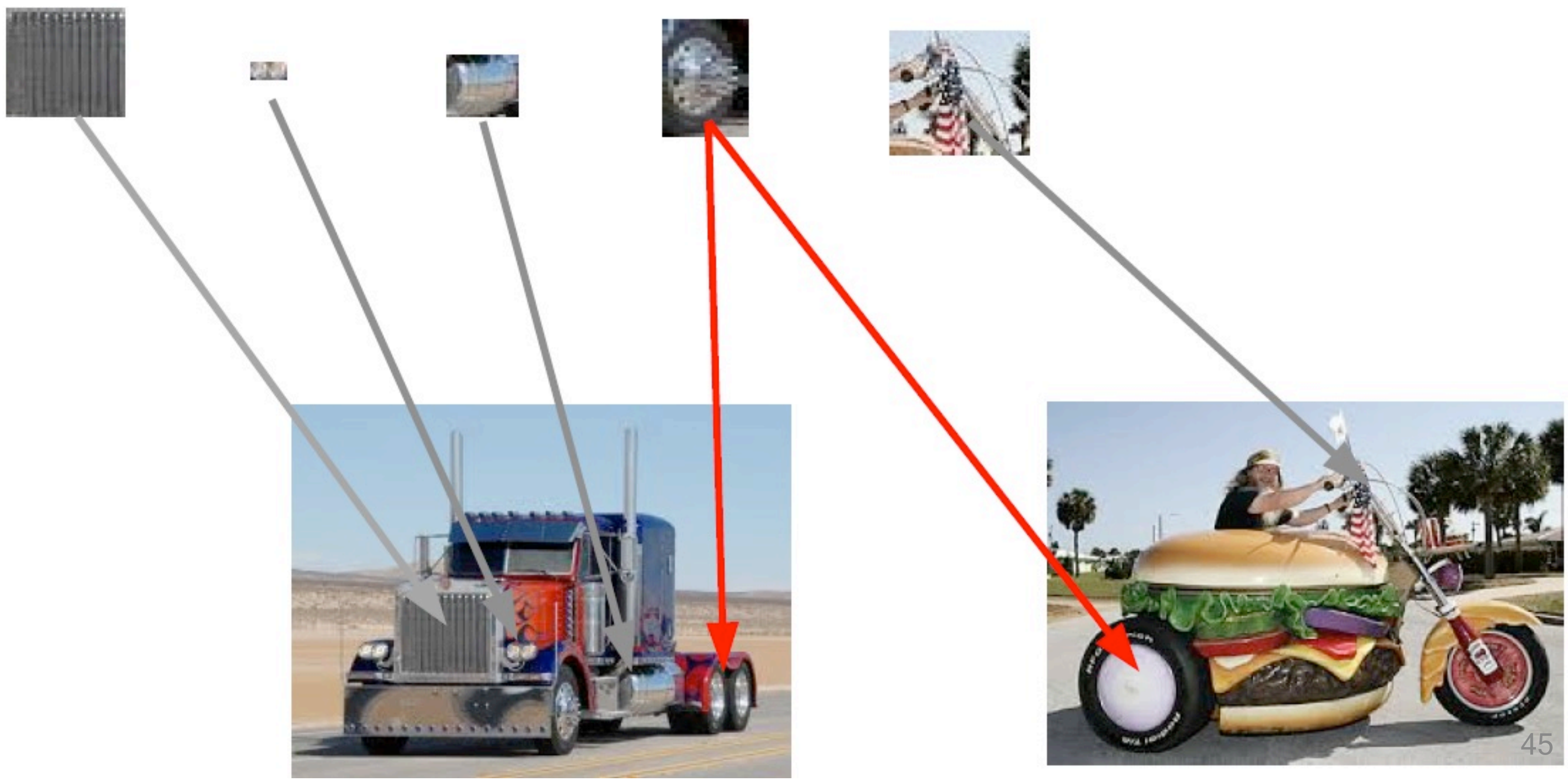
[0 0 **1** 0 0 0 0 **1** 0 0 **1** **1** 0 0 **1** 0 …] truck feature



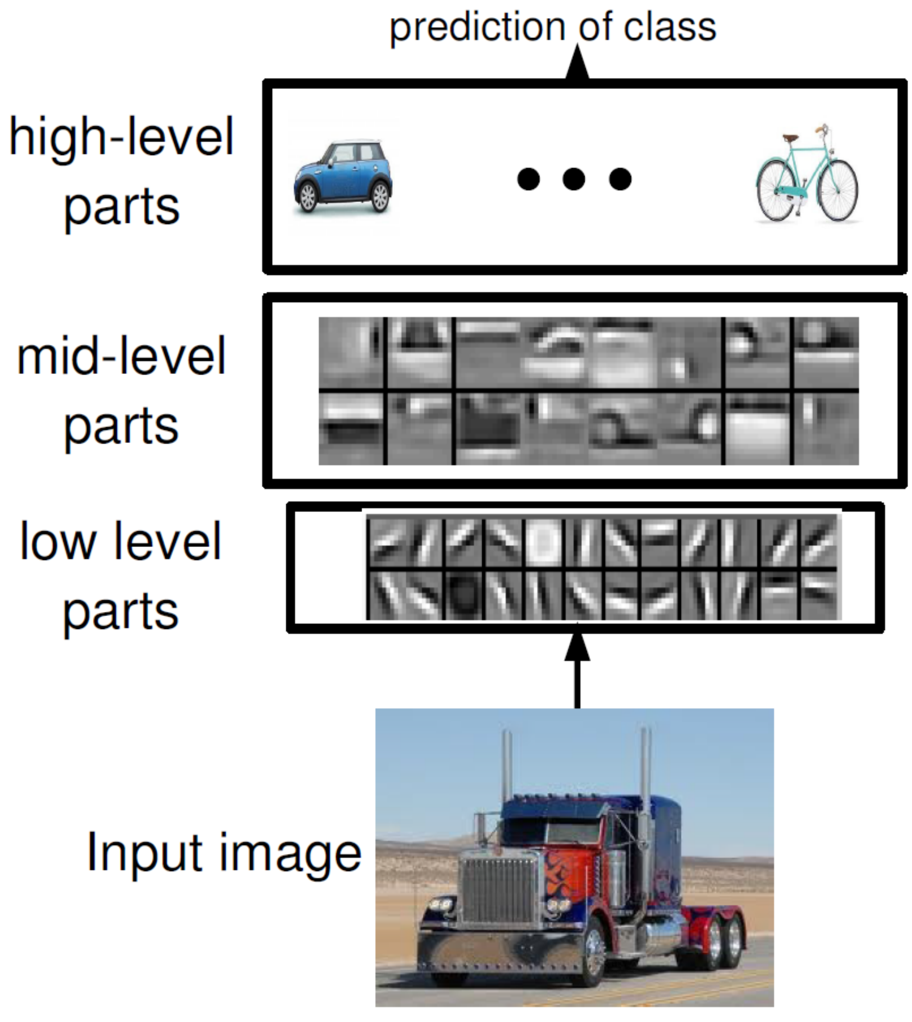
Interpretation

[1 1 0 0 0 1 0 1 0 0 0 0 1 1 0 1…] motorbike

[0 0 1 0 0 0 0 1 0 0 1 1 0 0 1 0 …] truck



Interpretation



- Distributed representations
- Feature sharing
- Compositionality

Interpretation

Question: What does a hidden unit do?

Answer: It can be thought of as a classifier or feature detector.

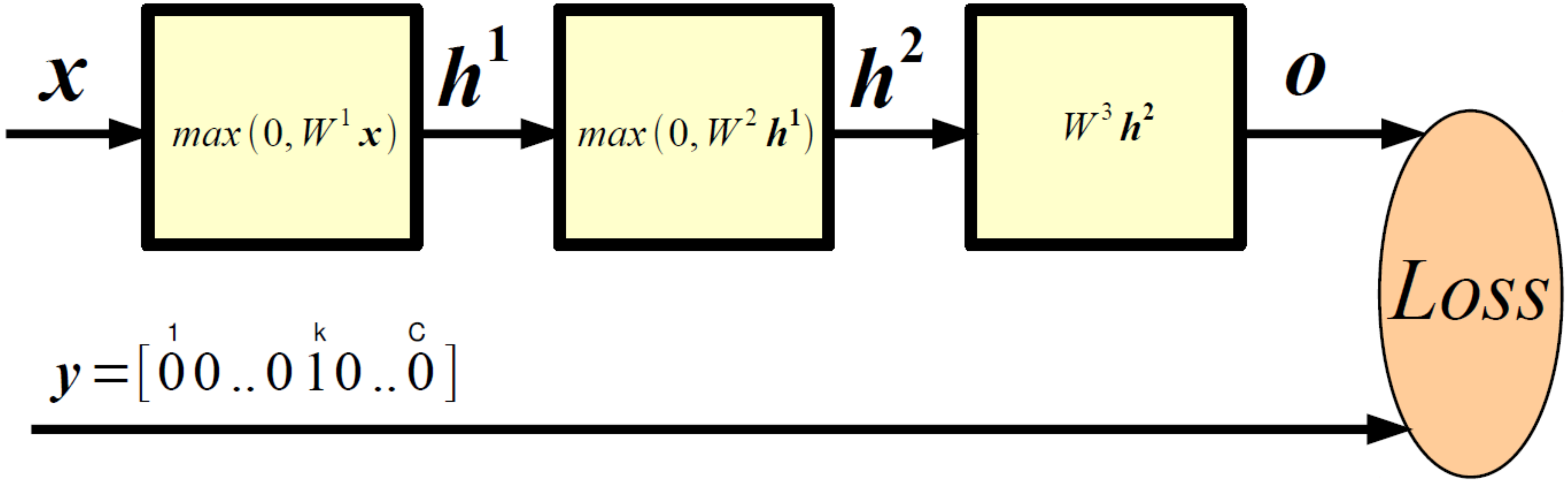
Question: How many layers? How many hidden units?

Answer: Cross-validation or hyper-parameter search methods are the answer. In general, the wider and the deeper the network the more complicated the mapping.

Question: How do I set the weight matrices?

Answer: Weight matrices and biases are learned. First, we need to define a measure of quality of the current mapping. Then, we need to define a procedure to adjust the parameters.

How Good is a Network



- Probability of class k given input (softmax):

$$p(c_k = 1 | \boldsymbol{x}) = \frac{e^{o_k}}{\sum_{j=1}^{C} e^{o_j}}$$

- (Per-sample) **Loss**; e.g., negative log-likelihood (good for classification of small number of classes):

$$L(\boldsymbol{x}, y; \boldsymbol{\theta}) = -\sum_j y_j \log p(c_j | \boldsymbol{x})$$

Training

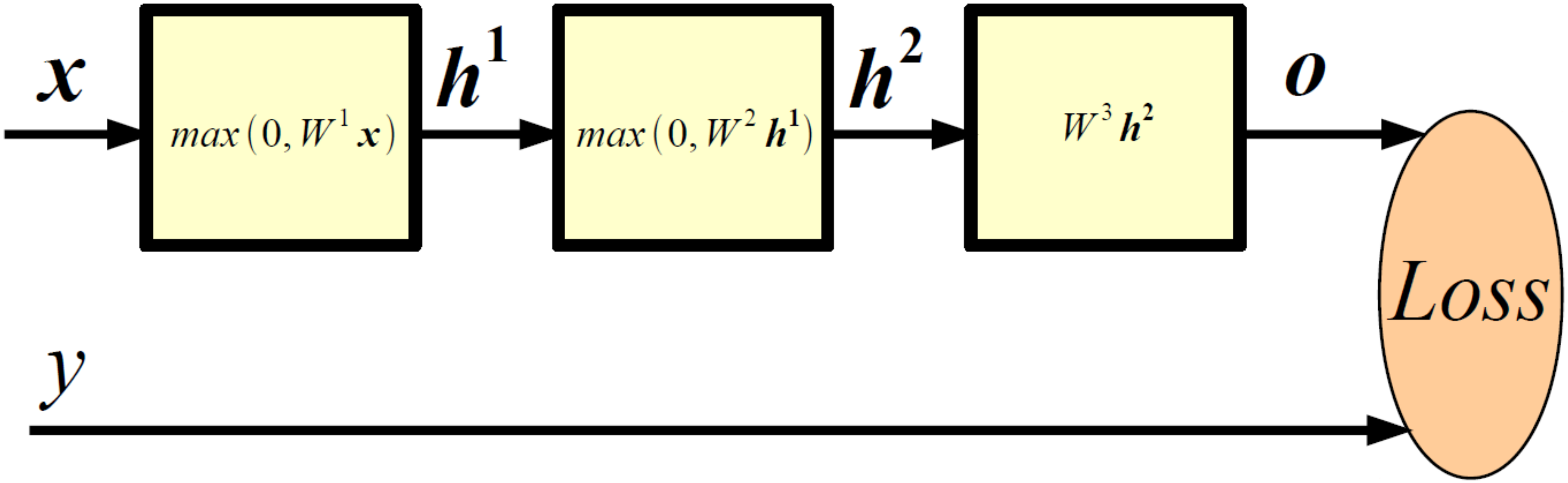
- Learning consists of minimizing the loss (plus some regularization term) w.r.t. parameters over the whole training set.

$$\boldsymbol{\theta}^* = arg\ min_{\boldsymbol{\theta}} \sum_{n=1}^{P} L(\boldsymbol{x}^n, y^n; \boldsymbol{\theta})$$

Question: How to minimize a complicated function of the parameters?

Answer: Chain rule, a.k.a. Backpropagation! That is the procedure to compute gradients of the loss w.r.t. parameters in a multi-layer neural network.

Key Idea: Wiggle to Decrease Loss



- Let's say we want to decrease the loss by adjusting $W^1_{i,j}$.
- We could consider a very small ϵ=1e-6 and compute:

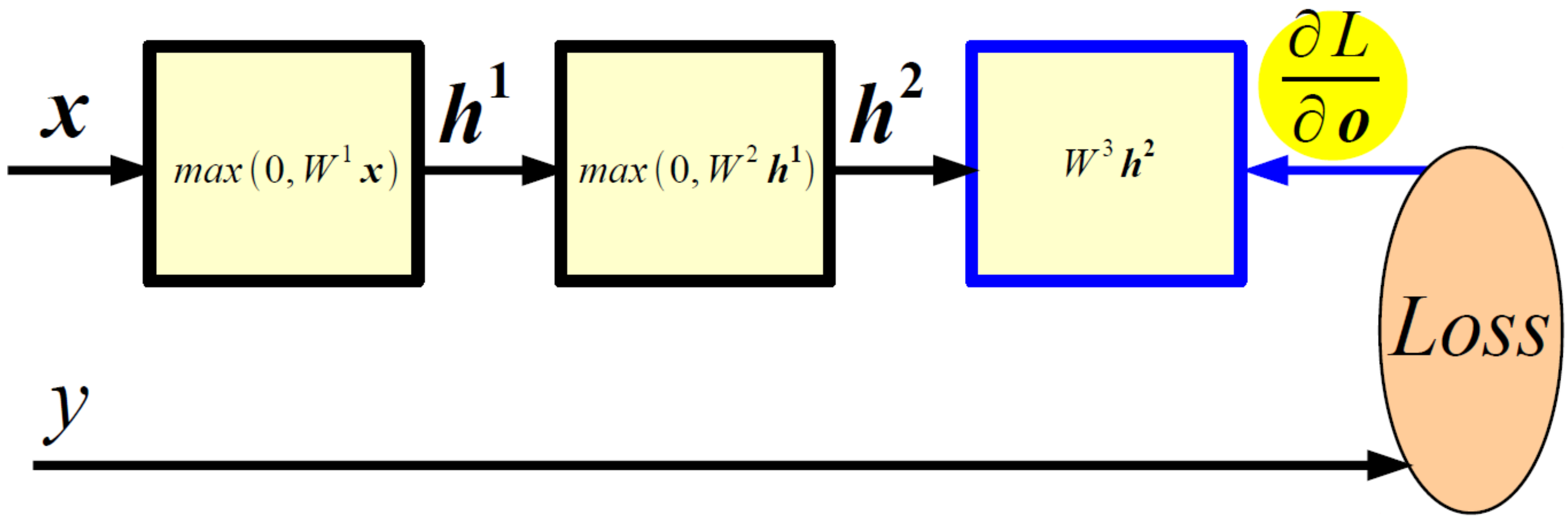
$$L(\boldsymbol{x}, y; \boldsymbol{\theta})$$

$$L(\boldsymbol{x}, y; \boldsymbol{\theta} \setminus W^1_{i,j}, W^1_{i,j} + \epsilon)$$

- Then update:

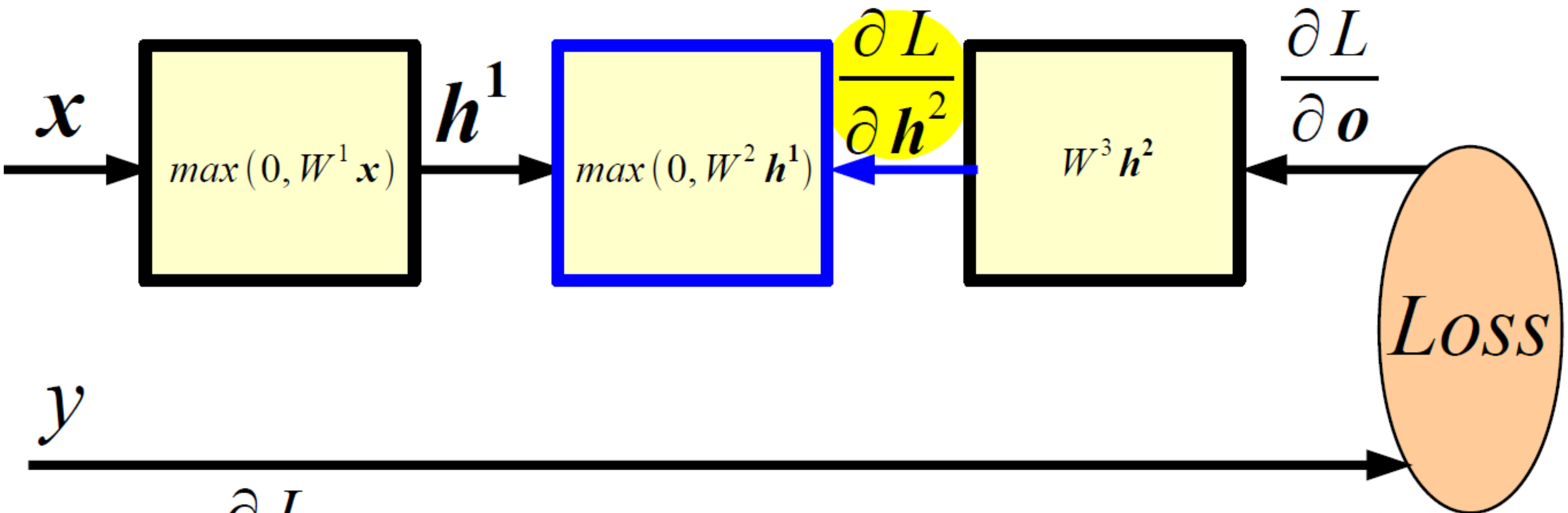
$$W^1_{i,j} \leftarrow W^1_{i,j} + \epsilon\, sgn(L(\boldsymbol{x}, y; \boldsymbol{\theta}) - L(\boldsymbol{x}, y; \boldsymbol{\theta} \setminus W^1_{i,j}, W^1_{i,j} + \epsilon))$$

Backward Propagation



$$\frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial W^3} \qquad \frac{\partial L}{\partial \boldsymbol{h}^2} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}^2}$$

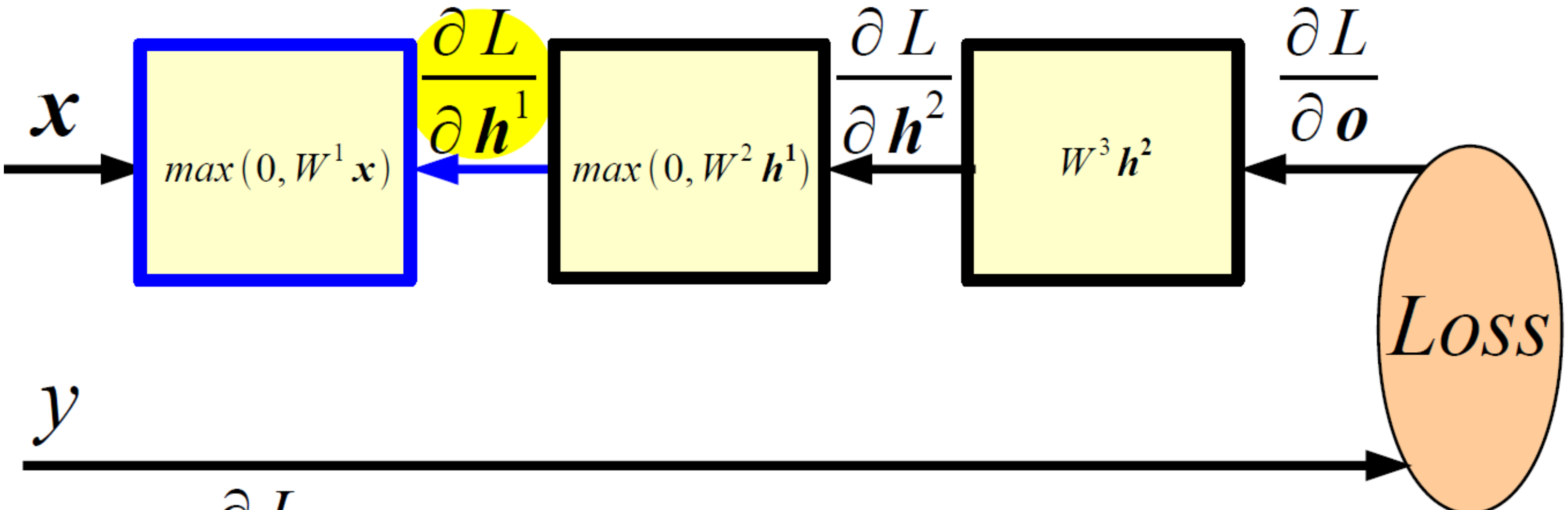
Backward Propagation



Given $\frac{\partial L}{\partial \boldsymbol{h}^2}$ we can compute now:

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial \boldsymbol{h}^2} \frac{\partial \boldsymbol{h}^2}{\partial W^2} \qquad \frac{\partial L}{\partial \boldsymbol{h}^1} = \frac{\partial L}{\partial \boldsymbol{h}^2} \frac{\partial \boldsymbol{h}^2}{\partial \boldsymbol{h}^1}$$

Backward Propagation



Given $\frac{\partial L}{\partial \boldsymbol{h}^1}$ we can compute now:

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial \boldsymbol{h}^1} \frac{\partial \boldsymbol{h}^1}{\partial W^1}$$

Optimization

Stochastic Gradient Descent

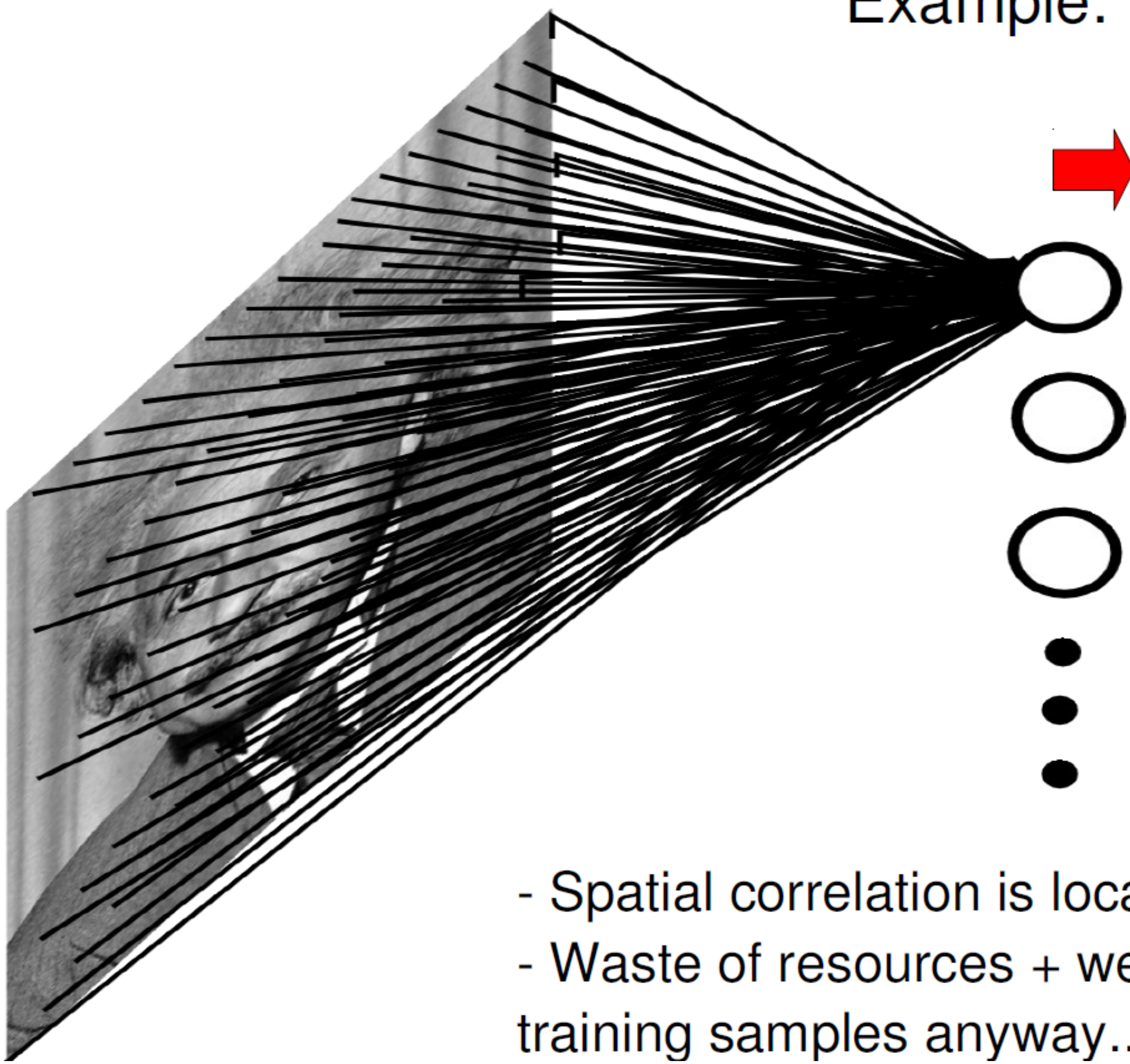
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\partial L}{\partial \boldsymbol{\theta}}, \eta \in (0, 1)$$

Or one of its many variants

Convolutional Neural Networks

Marc'Aurelio Ranzato

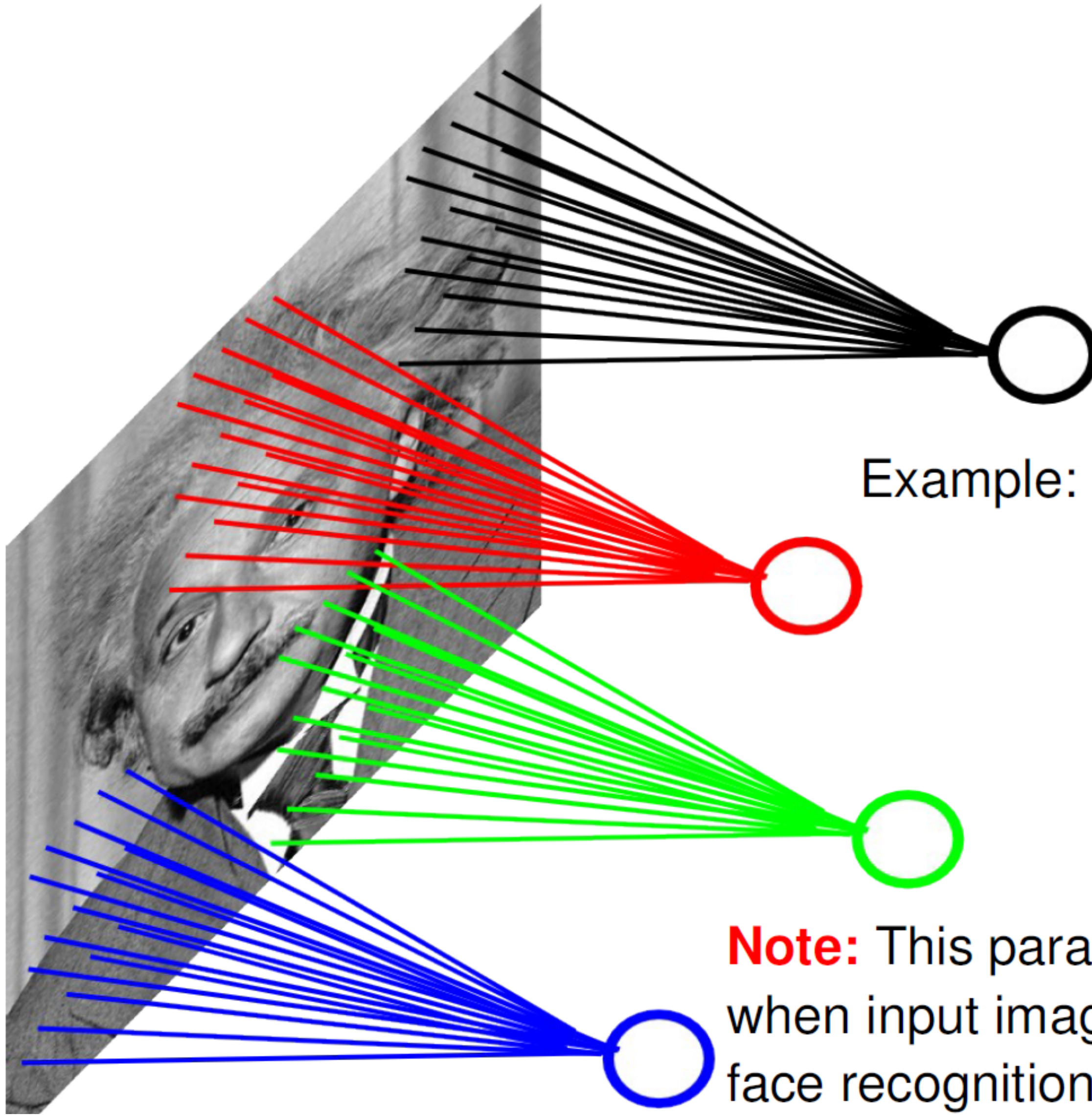
Fully Connected Layer



Example: 200x200 image
40K hidden units
~2B parameters!!!

- Spatial correlation is local
- Waste of resources + we have not enough training samples anyway..

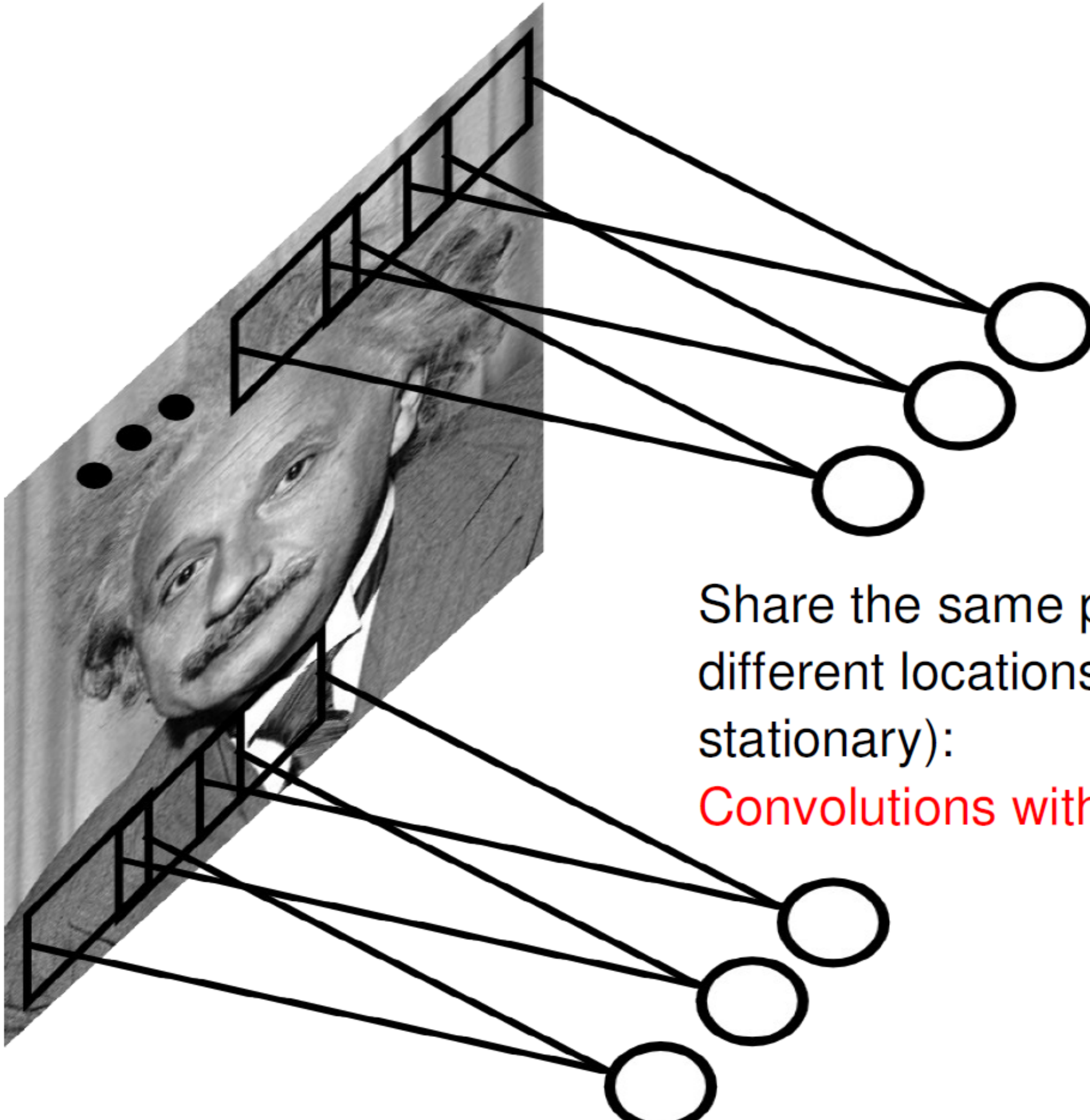
Locally Connected Layer



Example: 200x200 image
40K hidden units
Filter size: 10x10
4M parameters

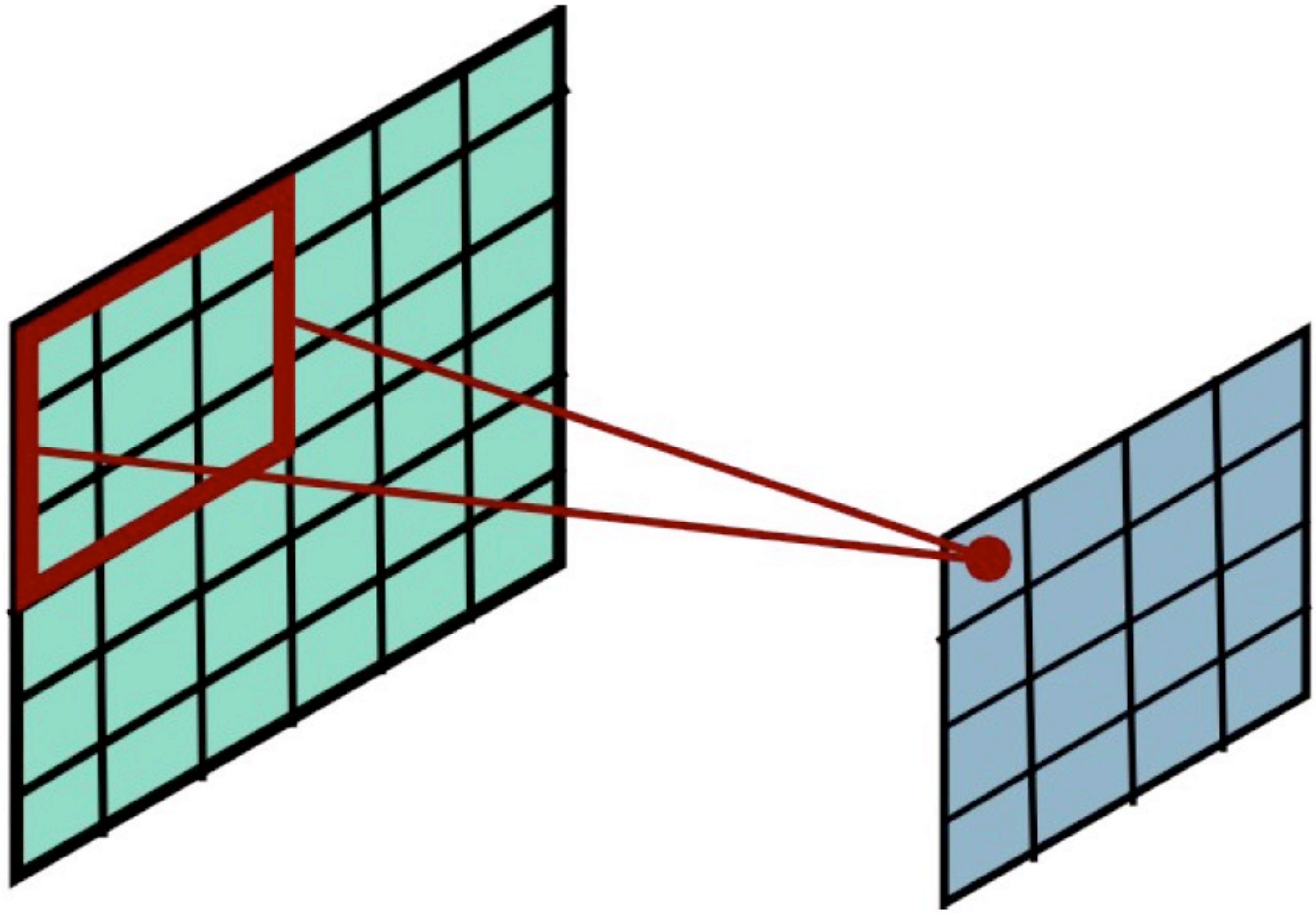
Note: This parameterization is good when input image is registered (e.g., face recognition).

Convolutional Layer

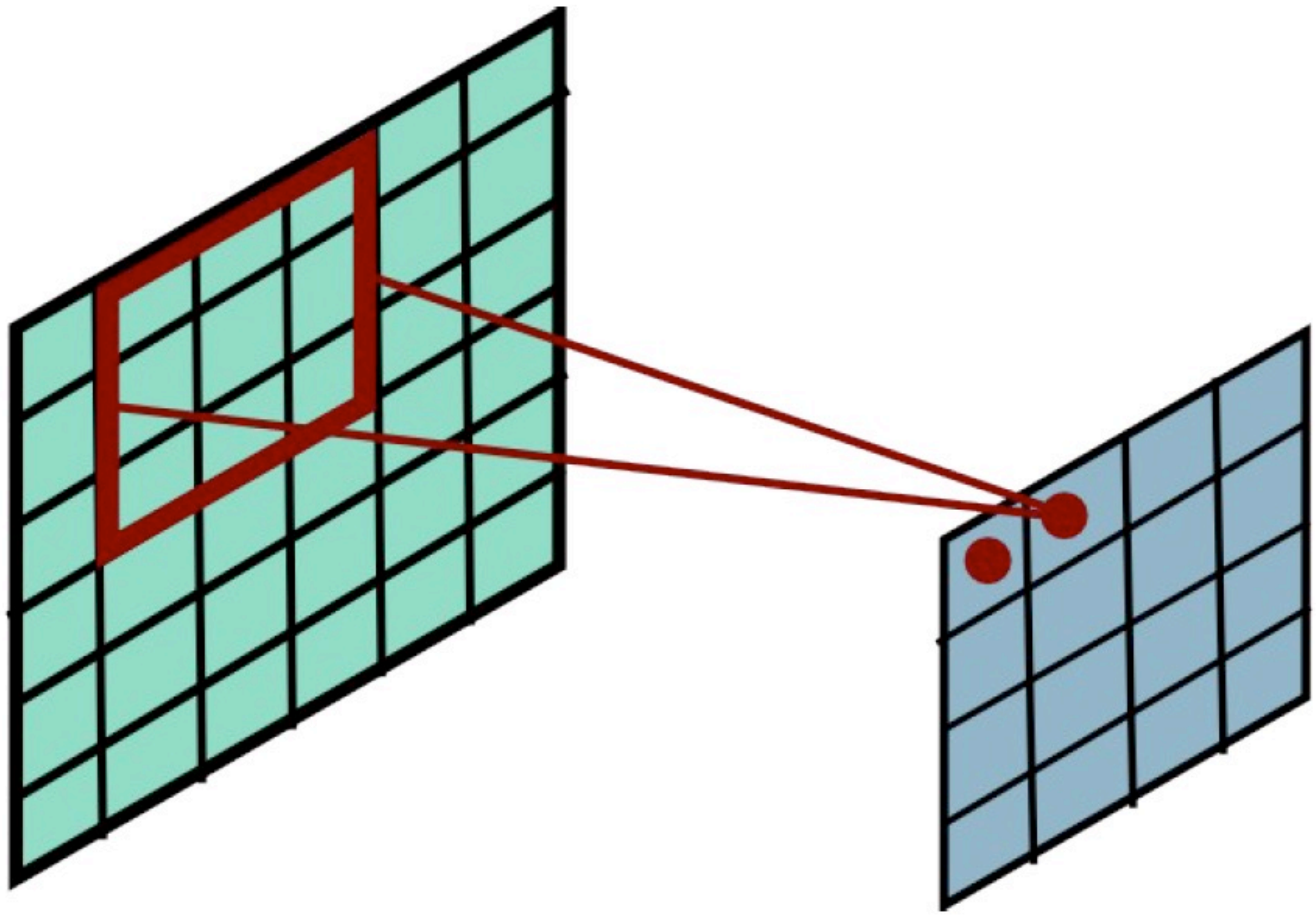


Share the same parameters across different locations (assuming input is stationary):
Convolutions with learned kernels

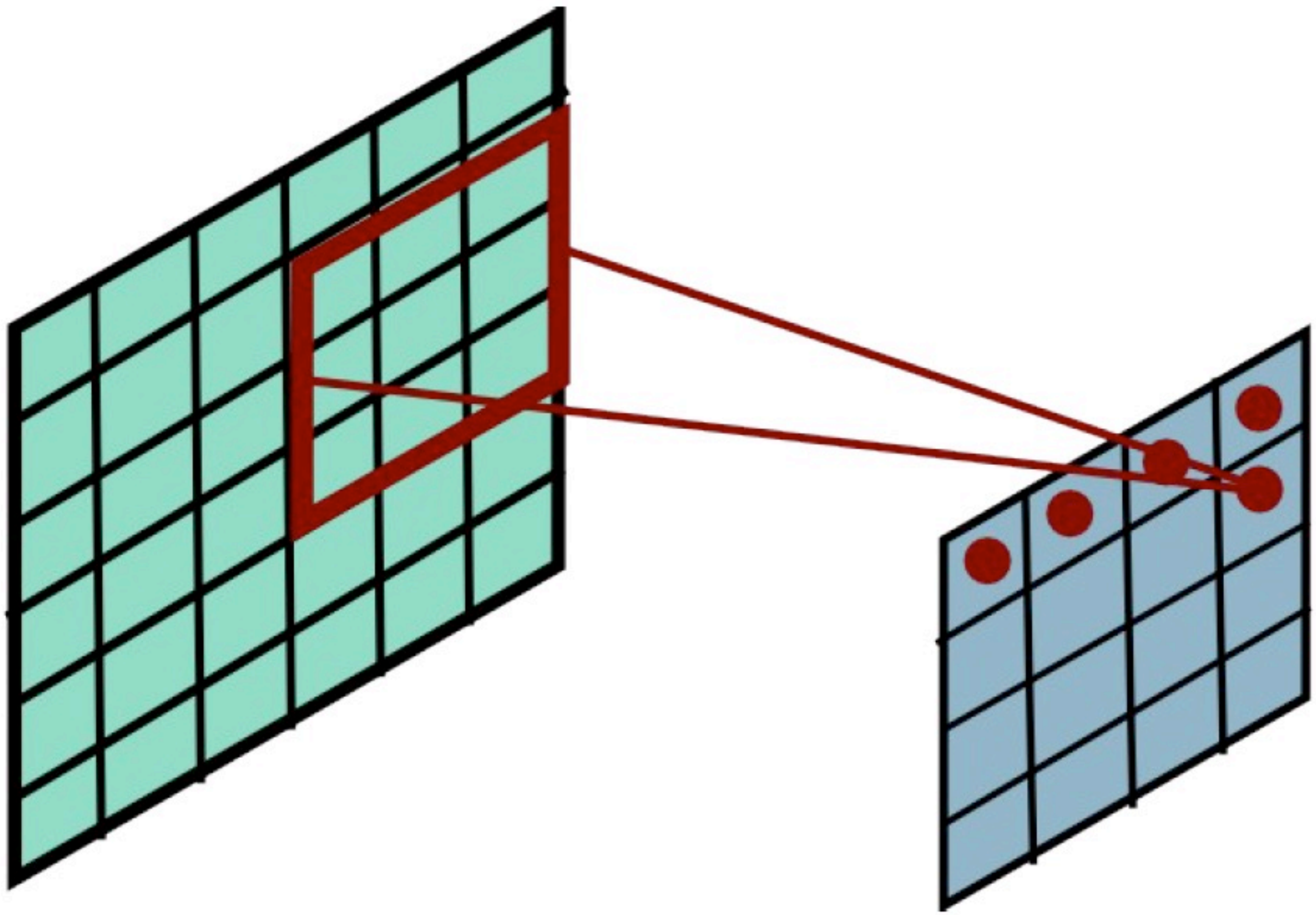
Convolutional Layer



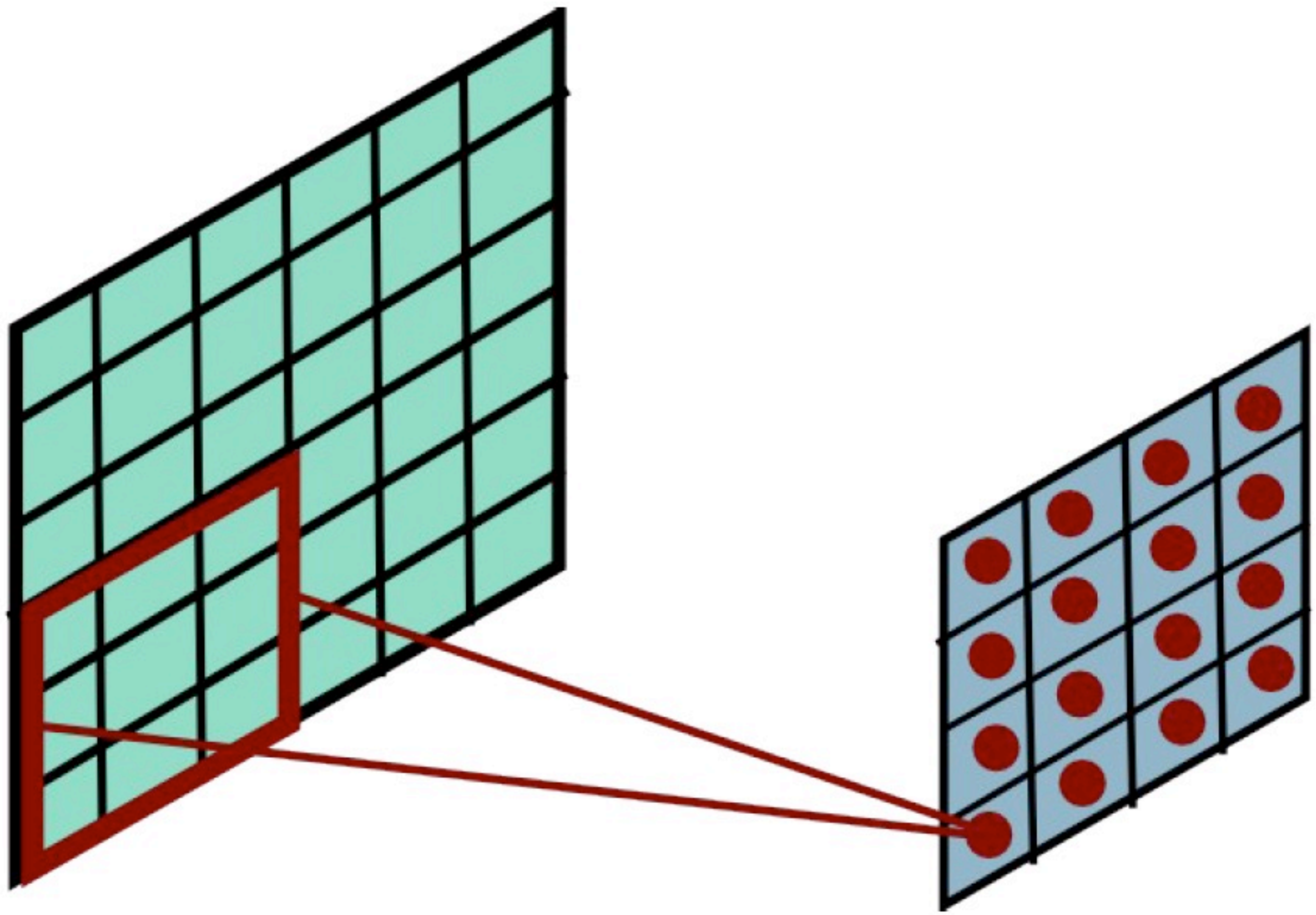
Convolutional Layer



Convolutional Layer

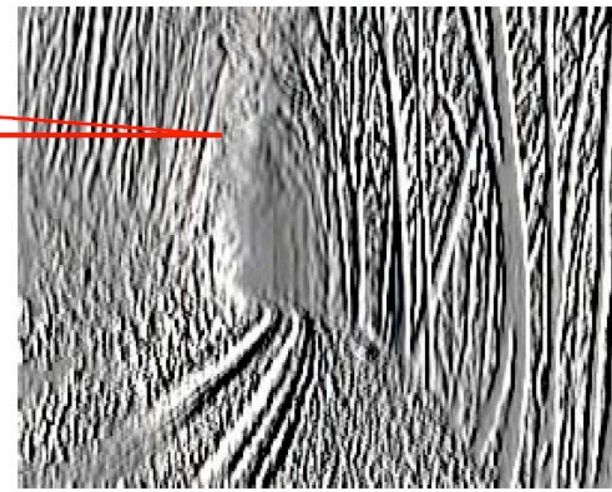


Convolutional Layer

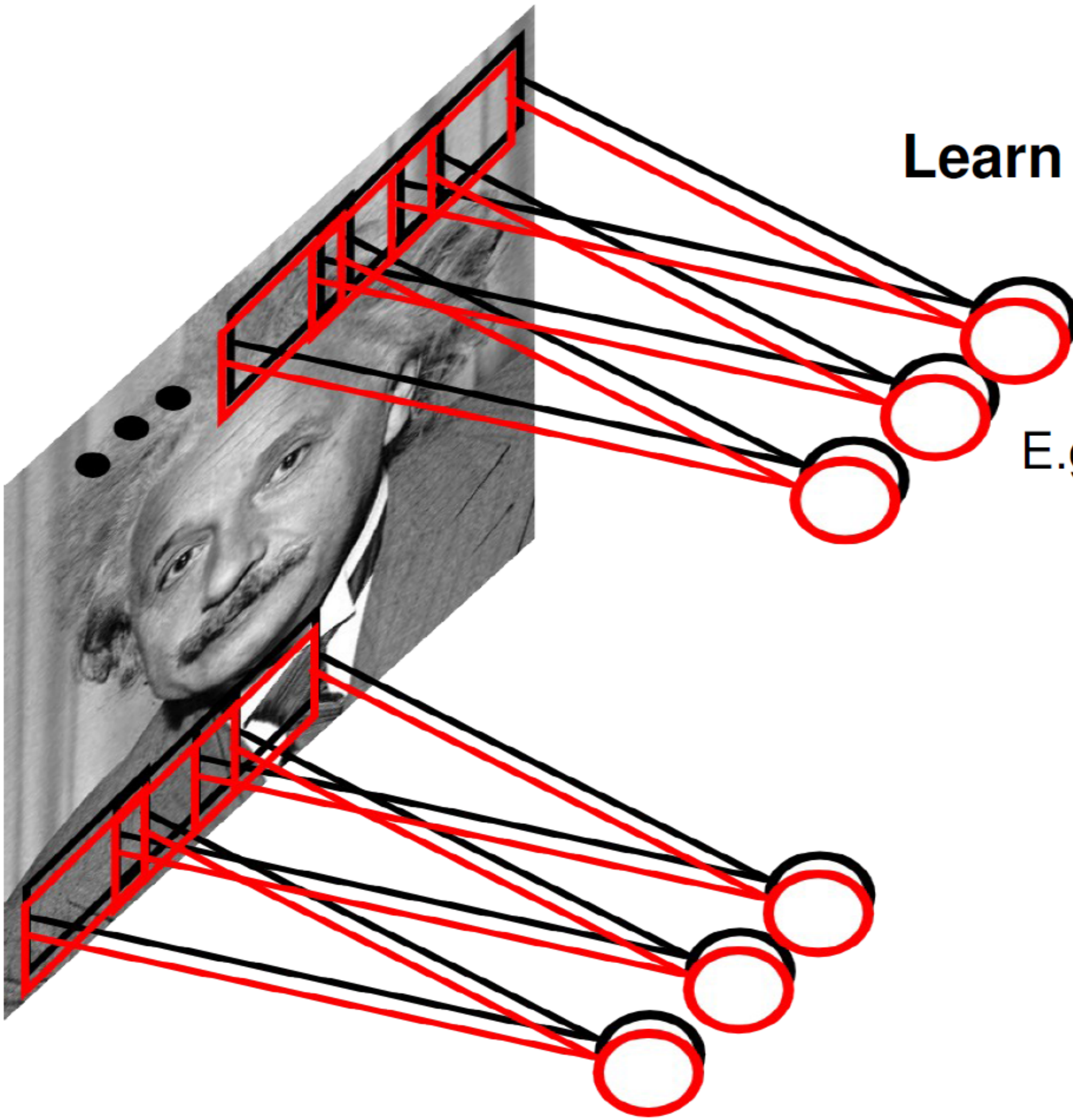


Convolutional Layer

$$* \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} =$$



Convolutional Layer



Learn multiple filters.

E.g.: 200x200 image
100 Filters
Filter size: 10x10
10K parameters

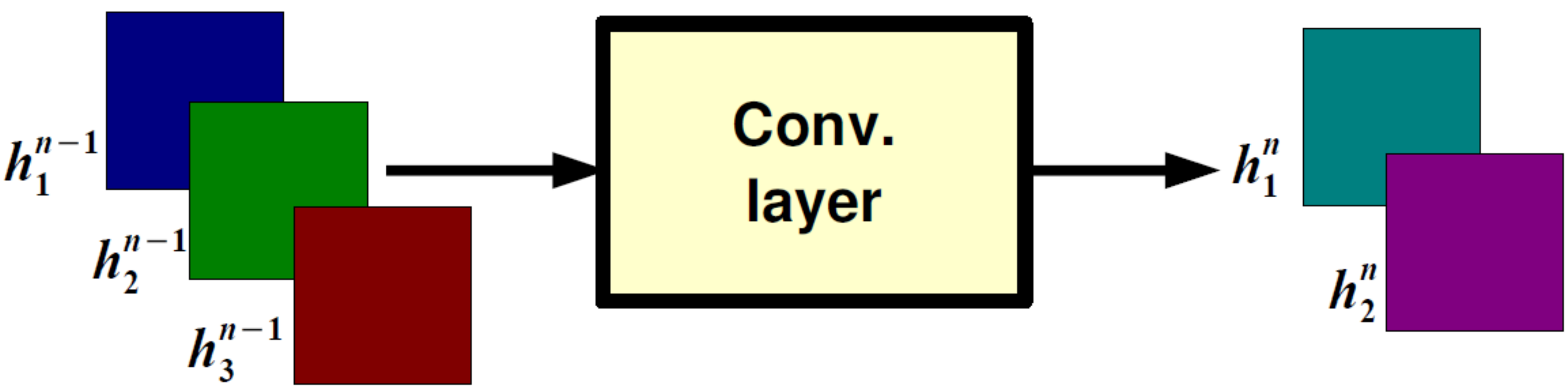
Convolutional Layer

$$h_j^n = max\left(0, \sum_{k=1}^{K} h_k^{n-1} * w_{kj}^n\right)$$

output feature map

input feature map

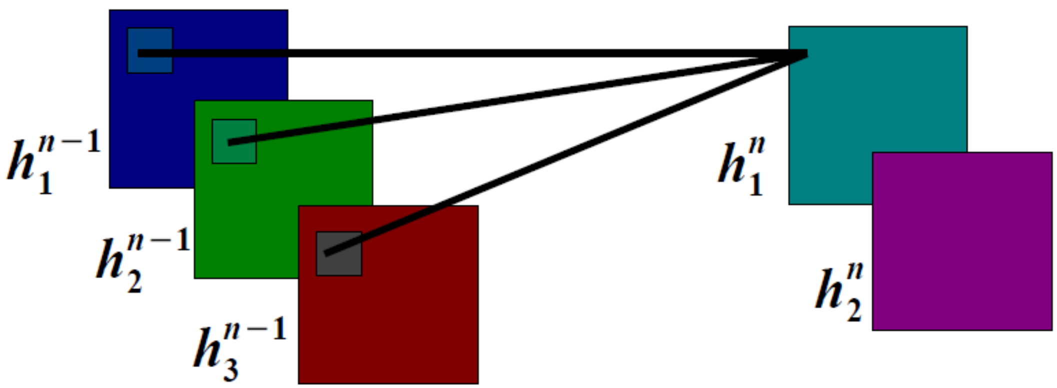
kernel



Convolutional Layer

$$h_j^n = max\,(0, \sum_{k=1}^{K} h_k^{n-1} * w_{kj}^n)$$

output feature map | input feature map | kernel



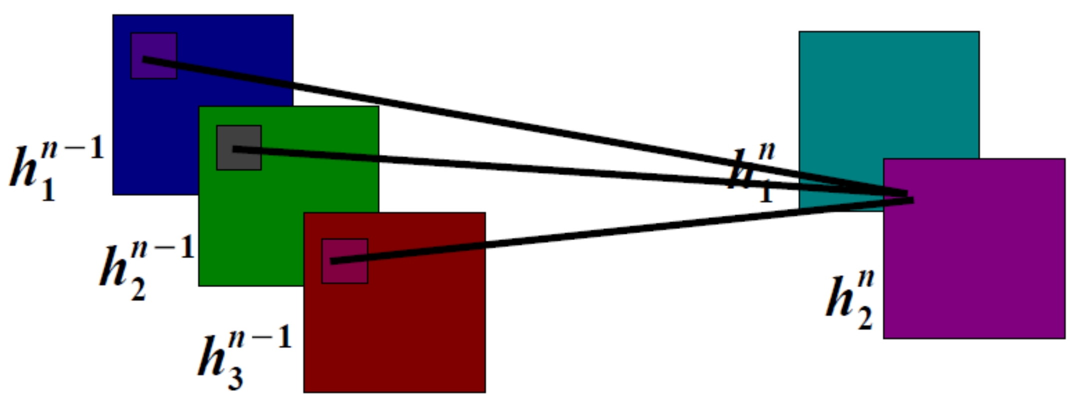
Convolutional Layer

$$h_j^n = max\left(0, \sum_{k=1}^{K} h_k^{n-1} * w_{kj}^n\right)$$

output feature map

input feature map

kernel



Convolutional Layer

Question: What is the size of the output? What's the computational cost?

Answer: It is proportional to the number of filters and depends on the stride. If kernels have size $K \times K$, input has size $D \times D$, stride is 1, and there are M input feature maps and N output feature maps then:

- the input has size $M \times D \times D$
- the output has size $N \times (D-K+1) \times (D-K+1)$
- the kernels have $M \times N \times K \times K$ coefficients (which have to be learned)
- cost: $M \times K \times K \times N \times (D-K+1) \times (D-K+1)$

Question: How many feature maps? What's the size of the filters?

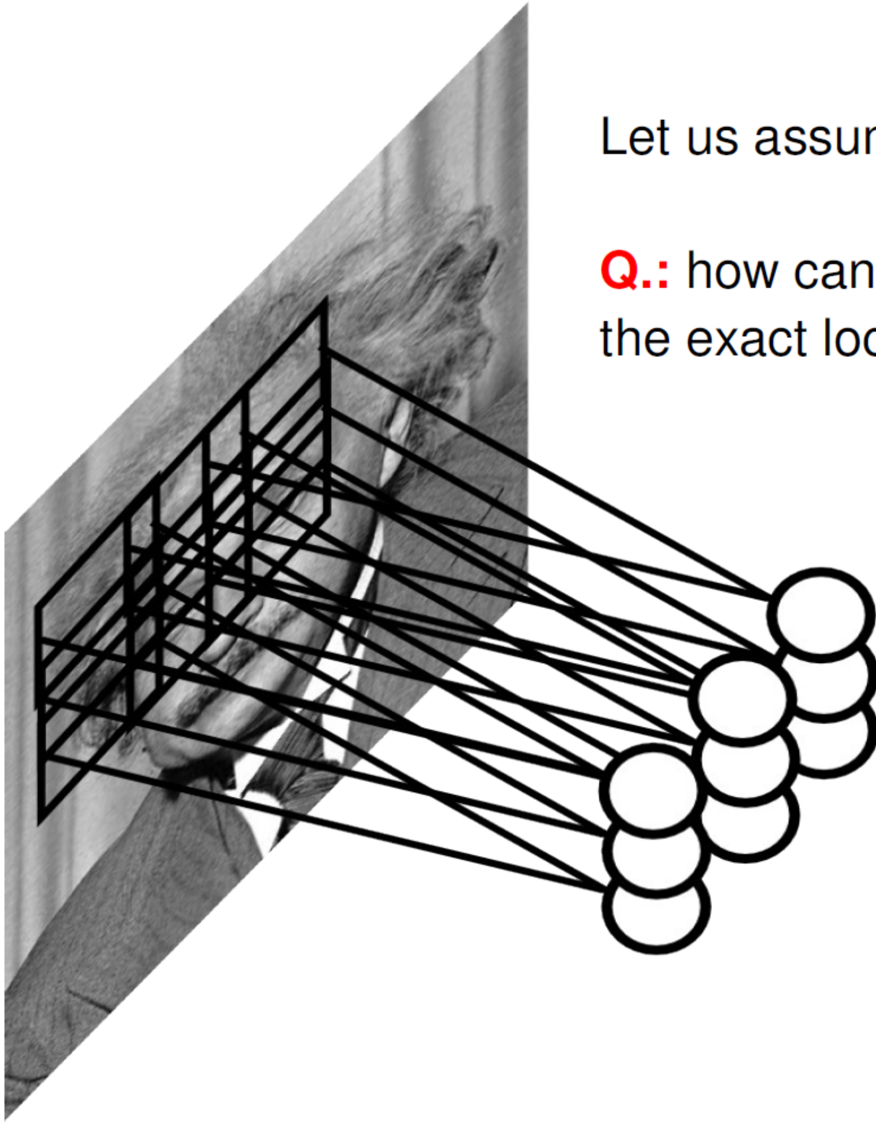
Answer: Usually, there are more output feature maps than input feature maps. Convolutional layers can increase the number of hidden units by big factors (and are expensive to compute). The size of the filters has to match the size/scale of the patterns we want to detect (task dependent).

Key Ideas

- A standard neural net applied to images:
  - scales quadratically with the size of the input
  - does not leverage stationarity

- Solution:
  - connect each hidden unit to a small patch of the input
  - share the weight across space
- This is called: convolutional layer
- A network with convolutional layers is called convolutional network

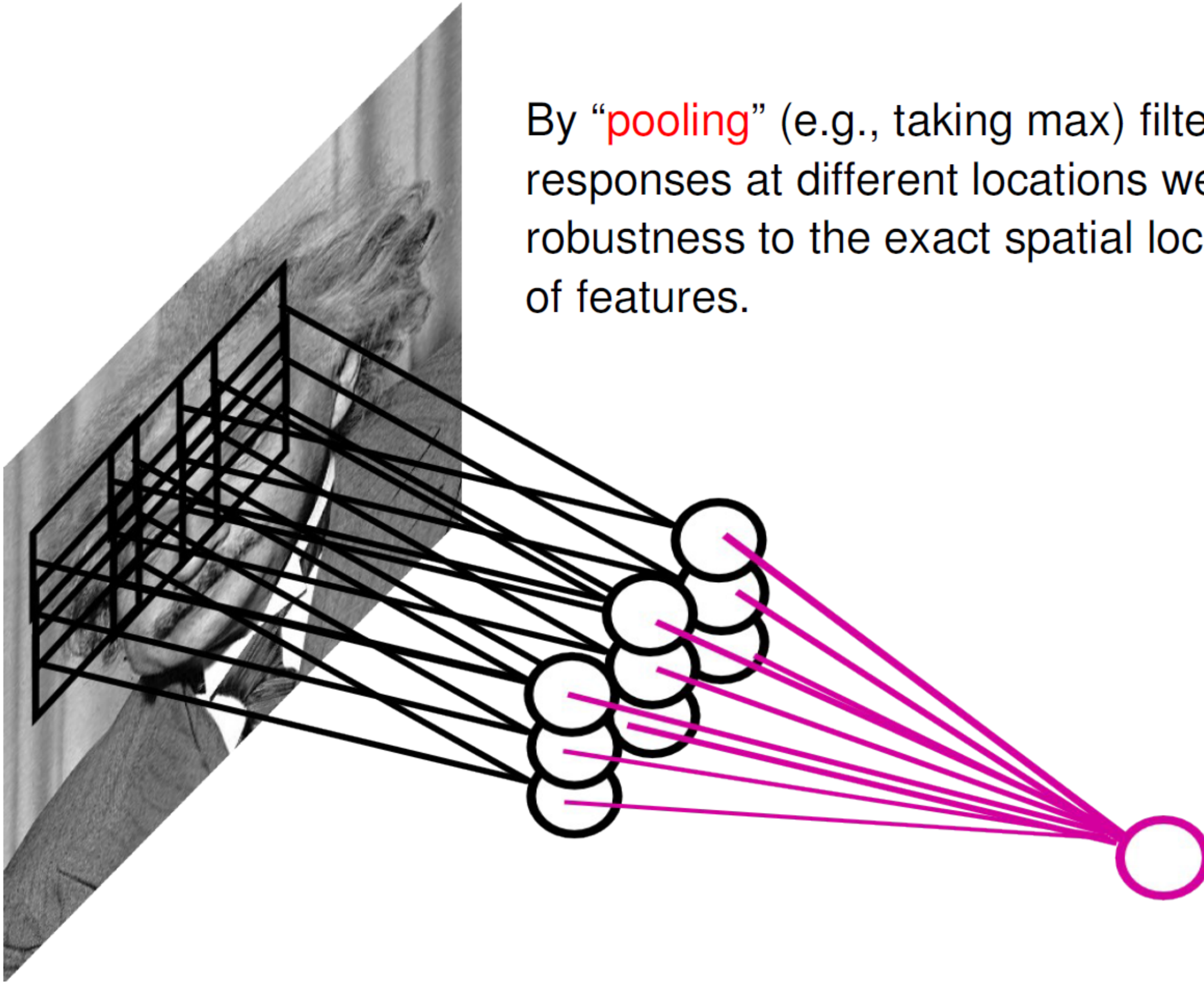
Pooling Layer



Let us assume filter is an "eye" detector.

Q.: how can we make the detection robust to the exact location of the eye?

Pooling Layer



By "pooling" (e.g., taking max) filter responses at different locations we gain robustness to the exact spatial location of features.

Pooling Layer

Question: What is the size of the output? What's the computational cost?

Answer: The size of the output depends on the stride between the pools. For instance, if pools do not overlap and have size $K \times K$, and the input has size $D \times D$ with M input feature maps, then:

- output is $M@(D/K) \times (D/K)$

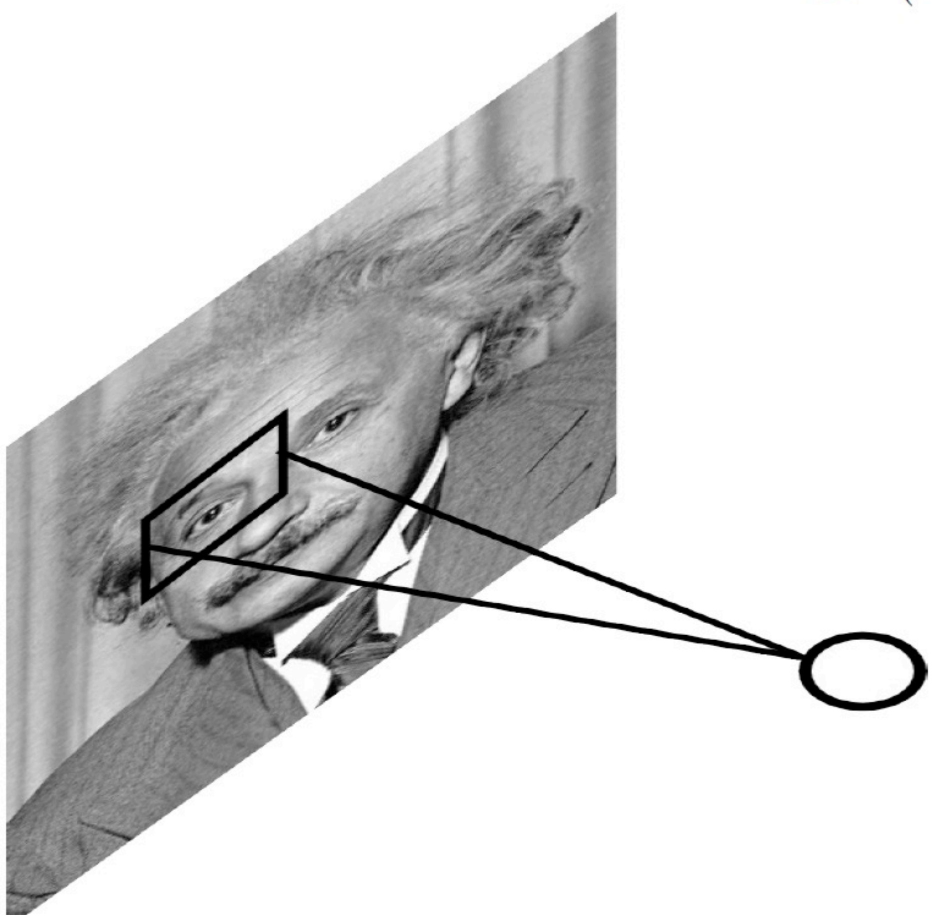
- the computational cost is proportional to the size of the input (negligible compared to a convolutional layer)

Question: How should I set the size of the pools?

Answer: It depends on how much "invariant" or robust to distortions we want the representation to be. It is best to pool slowly (via a few stacks of conv-pooling layers).

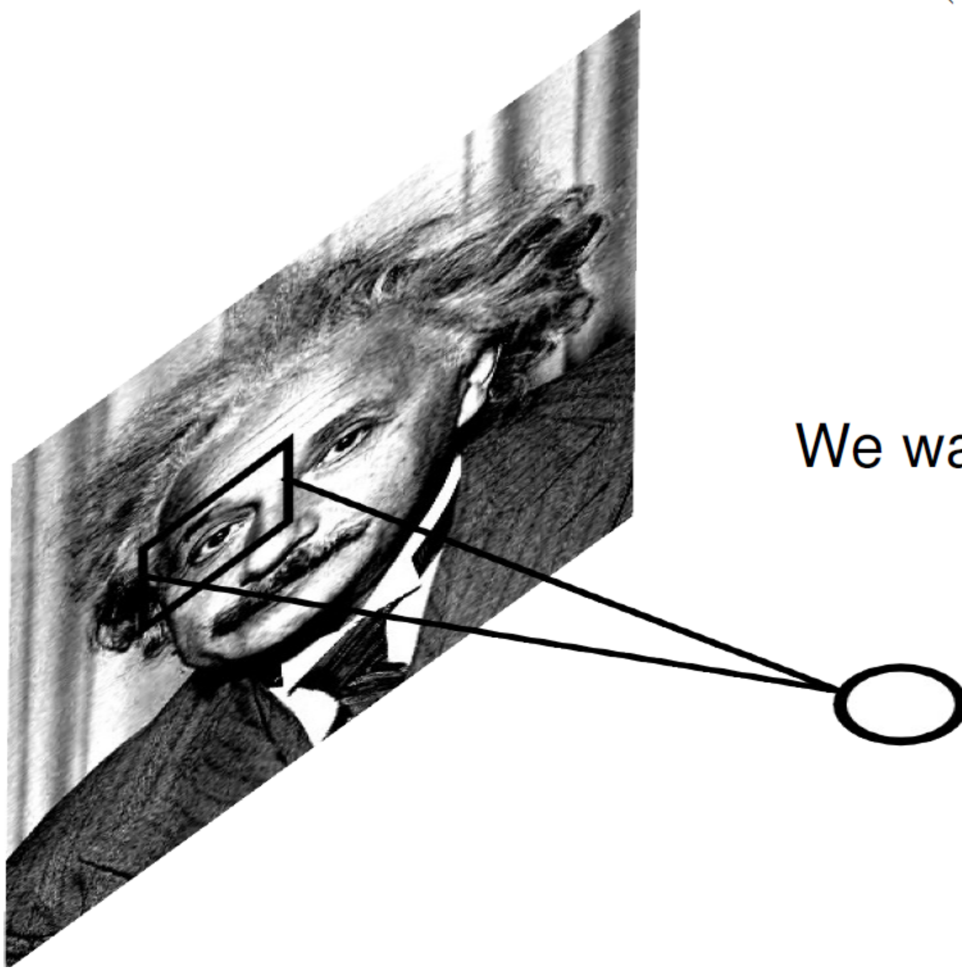
Local Contrast Normalization

$$h^{i+1}(x,y) = \frac{h^{i}(x,y) - m^{i}(N(x,y))}{\sigma^{i}(N(x,y))}$$



Local Contrast Normalization

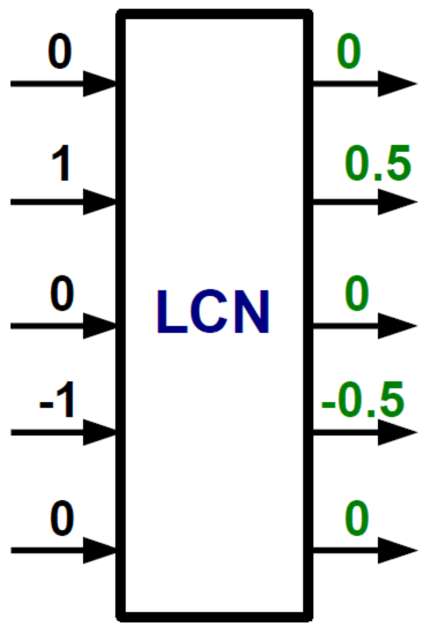
$$h^{i+1}(x,y) = \frac{h^i(x,y) - m^i(N(x,y))}{\sigma^i(N(x,y))}$$



We want the same response.

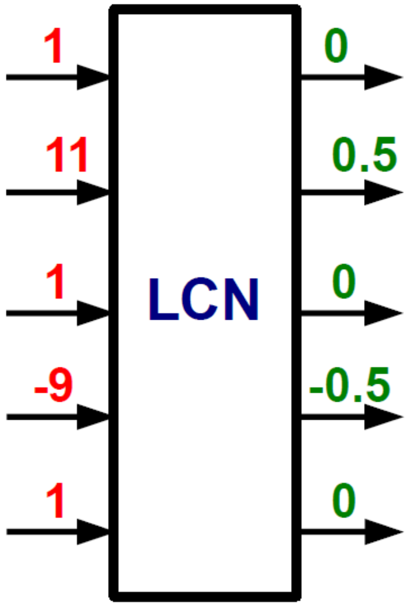
Local Contrast Normalization

$$h_{i+1,x,y} = \frac{h_{i,x,y} - m_{i,N(x,y)}}{\sigma_{i,N(x,y)}}$$

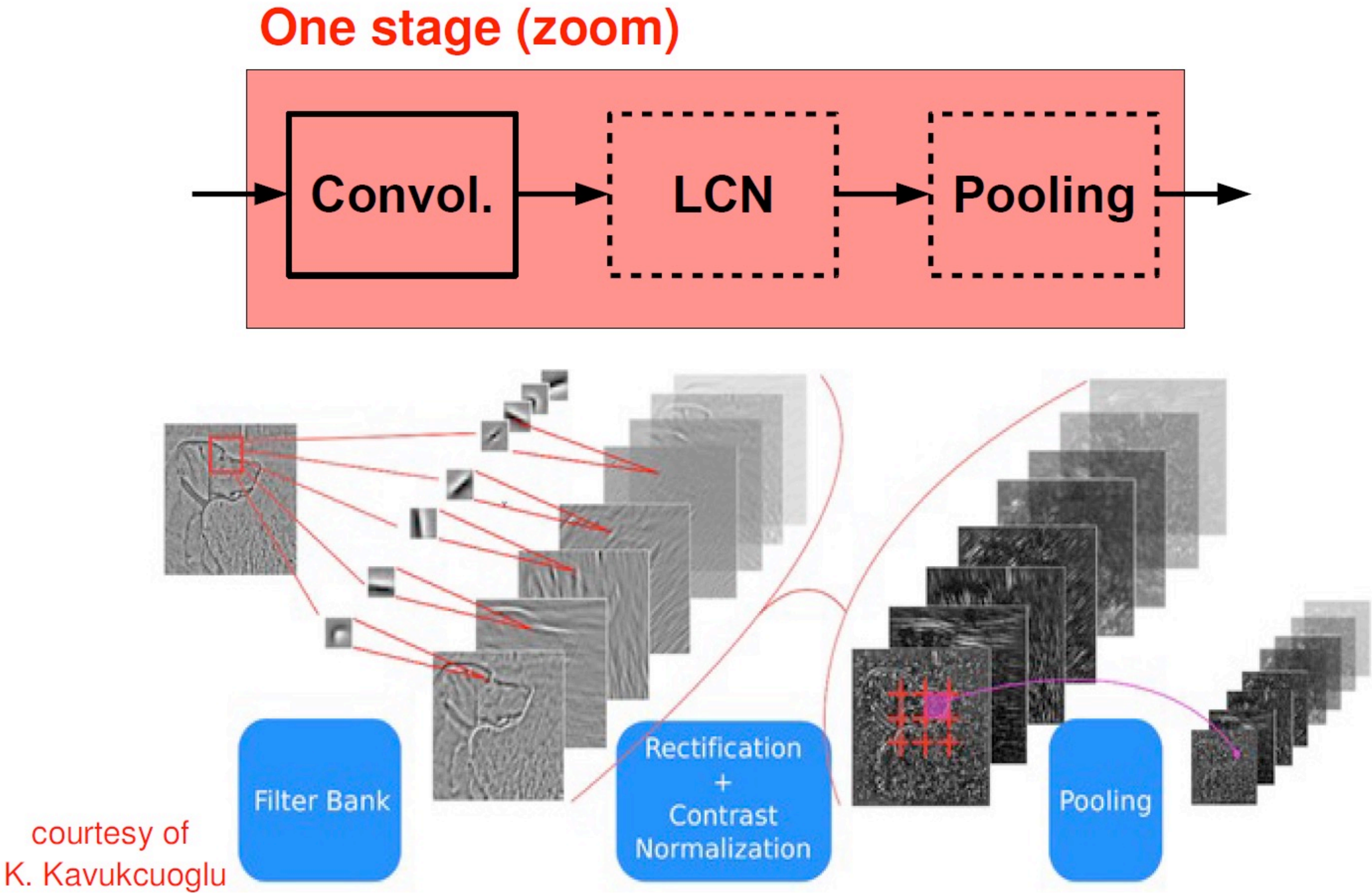


Local Contrast Normalization

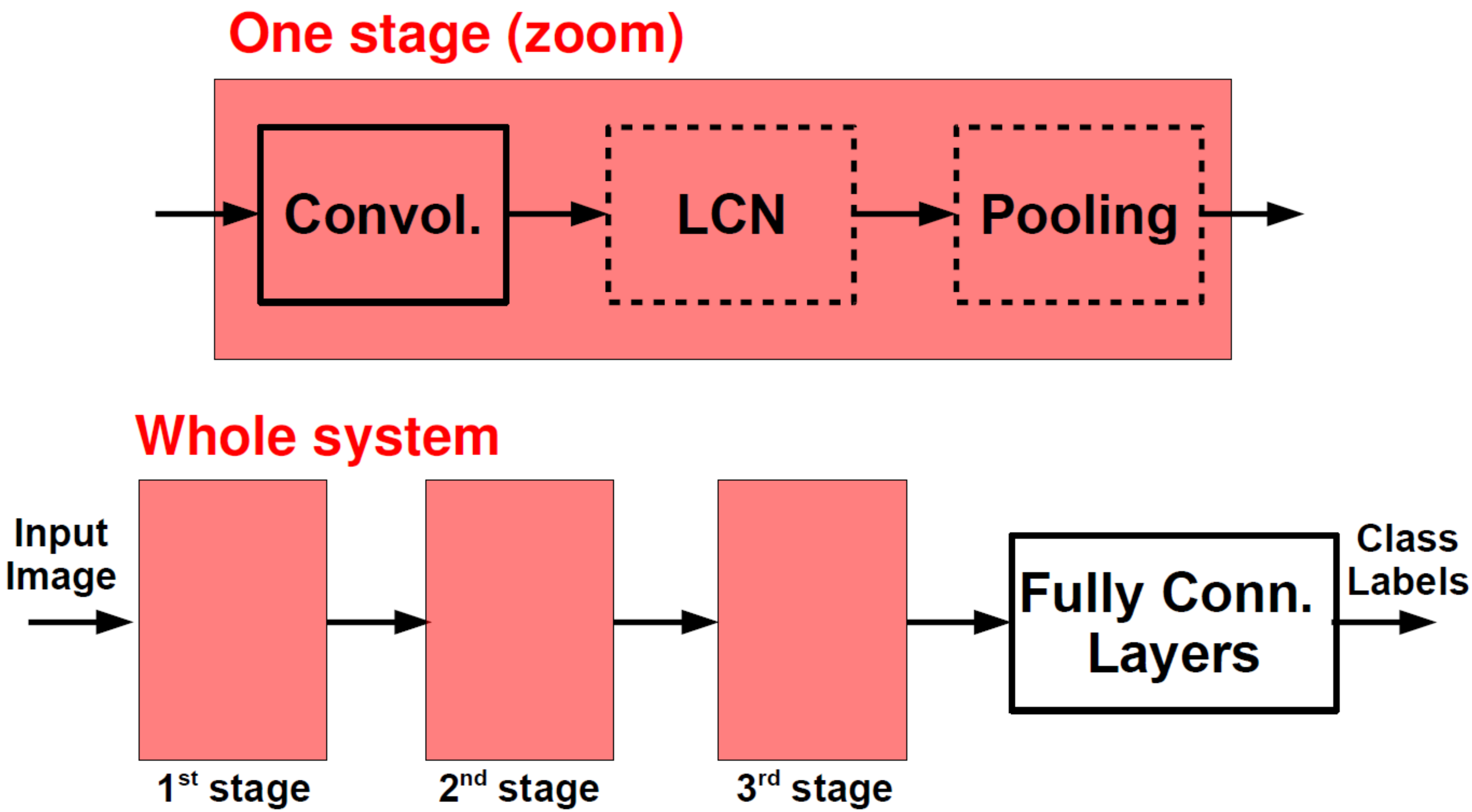
$$h_{i+1,x,y} = \frac{h_{i,x,y} - m_{i,N(x,y)}}{\sigma_{i,N(x,y)}}$$



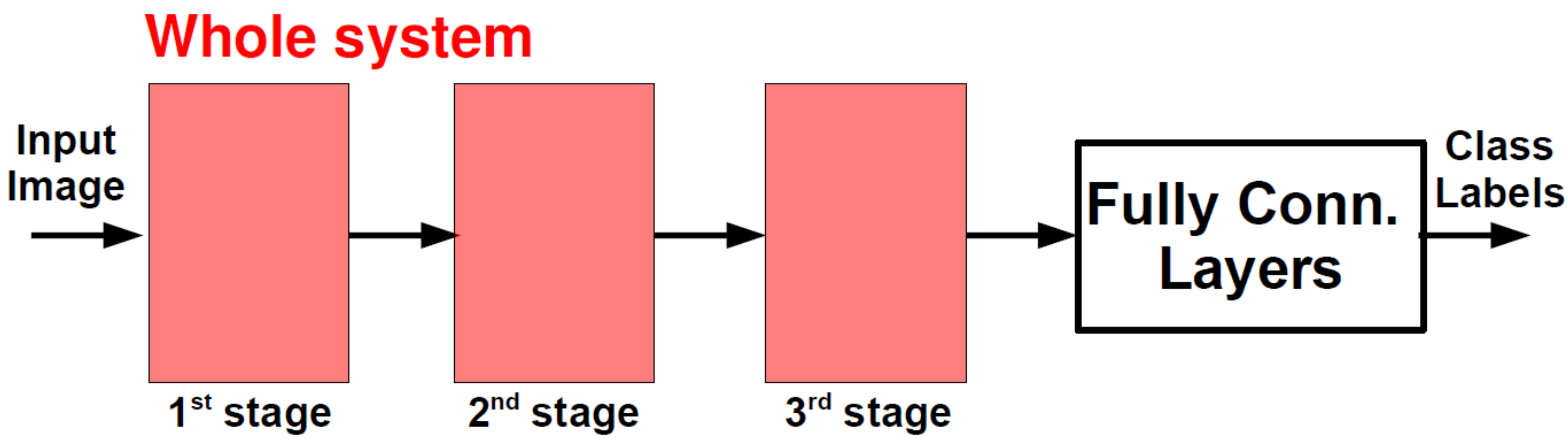
ConvNets: Typical Stage



ConvNets: Typical Architecture



ConvNets: Typical Architecture

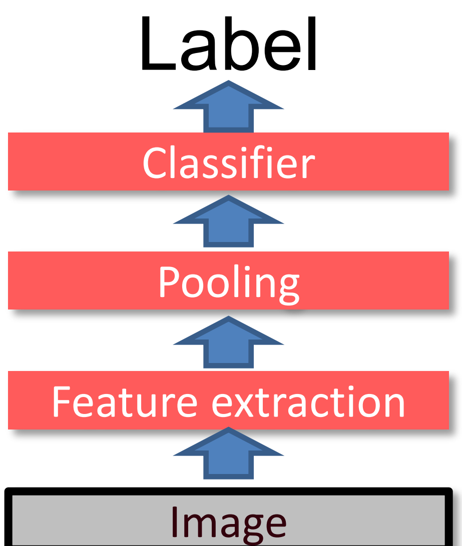


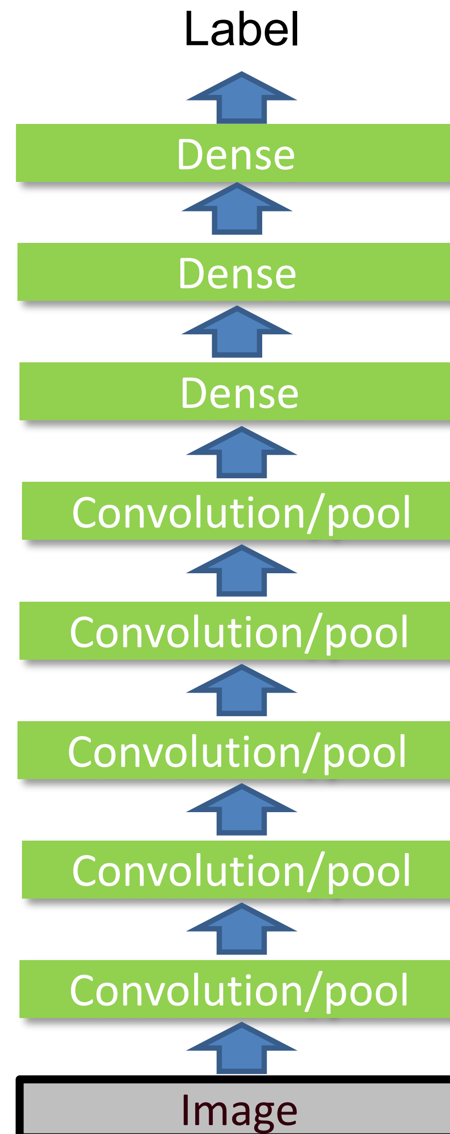
Conceptually similar to:

SIFT → k-means → Pyramid Pooling → SVM

Engineered vs. learned features

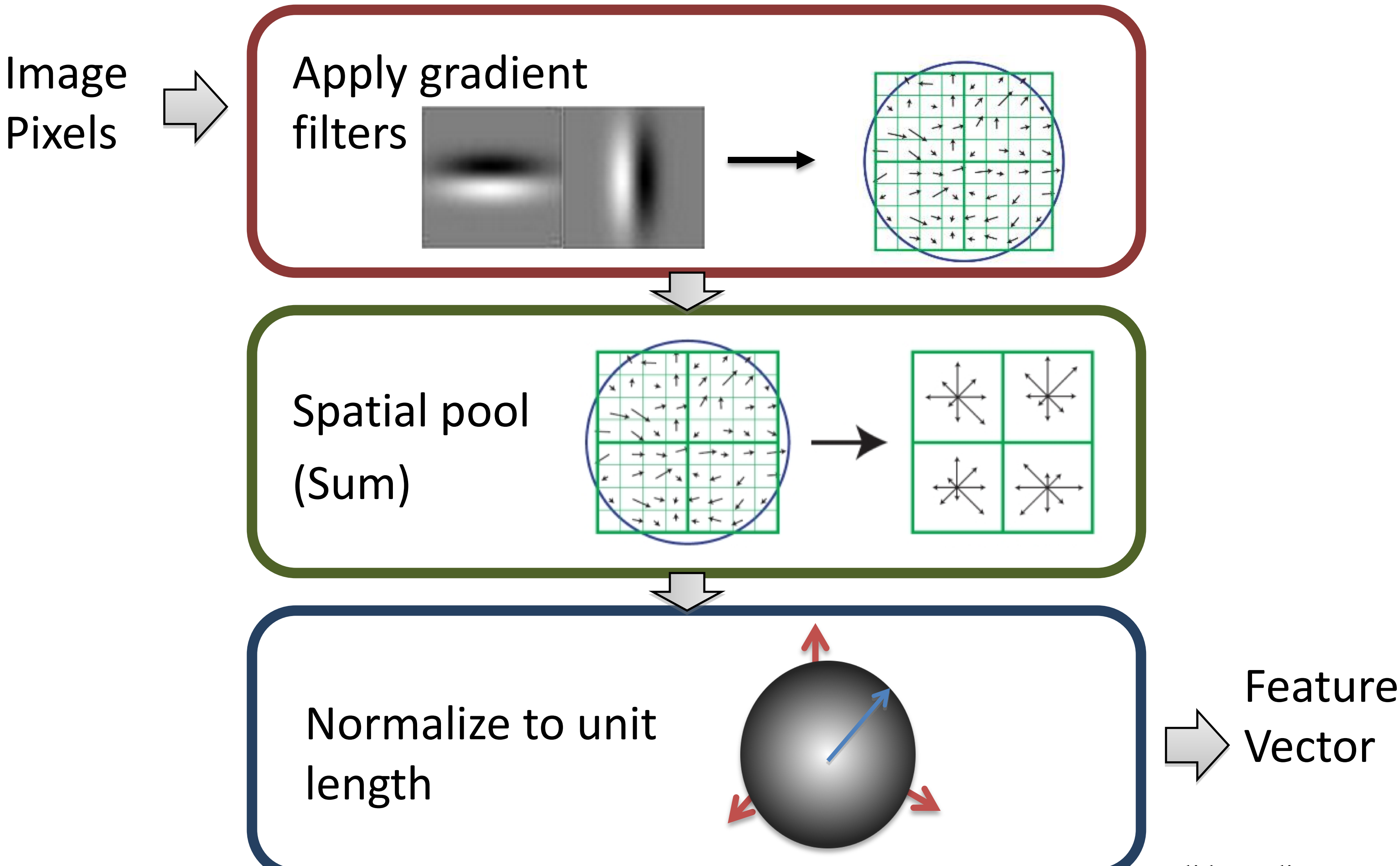
Convolutional filters are trained in a supervised manner by back-propagating classification error





slide credit: S. Lazebnik

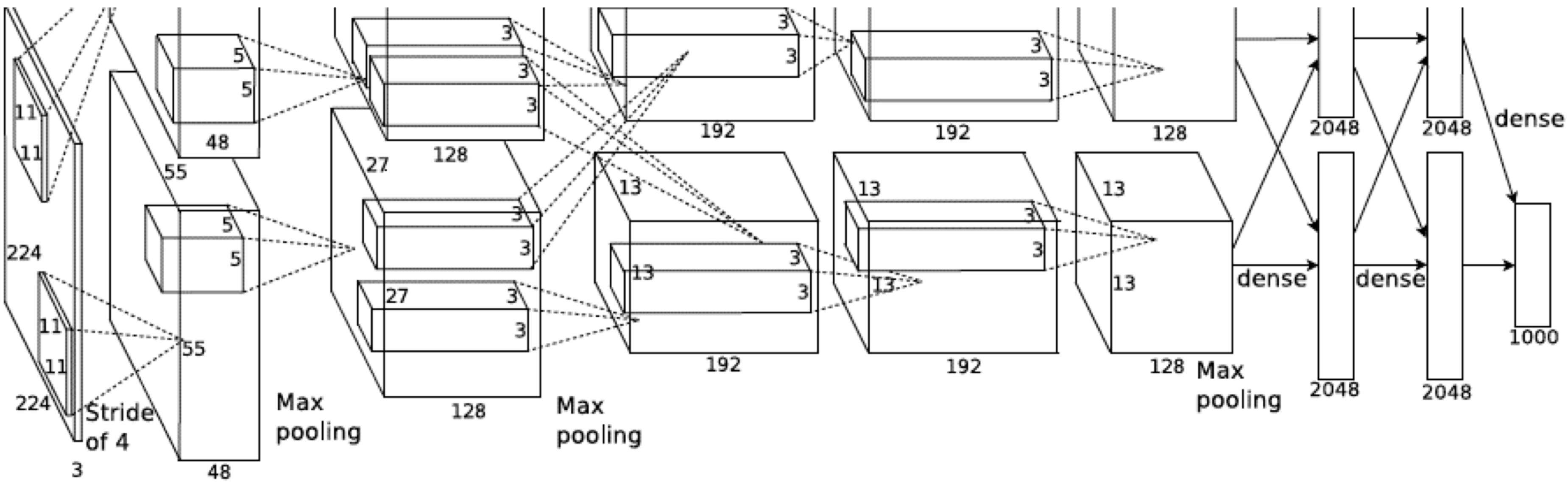
SIFT Descriptor



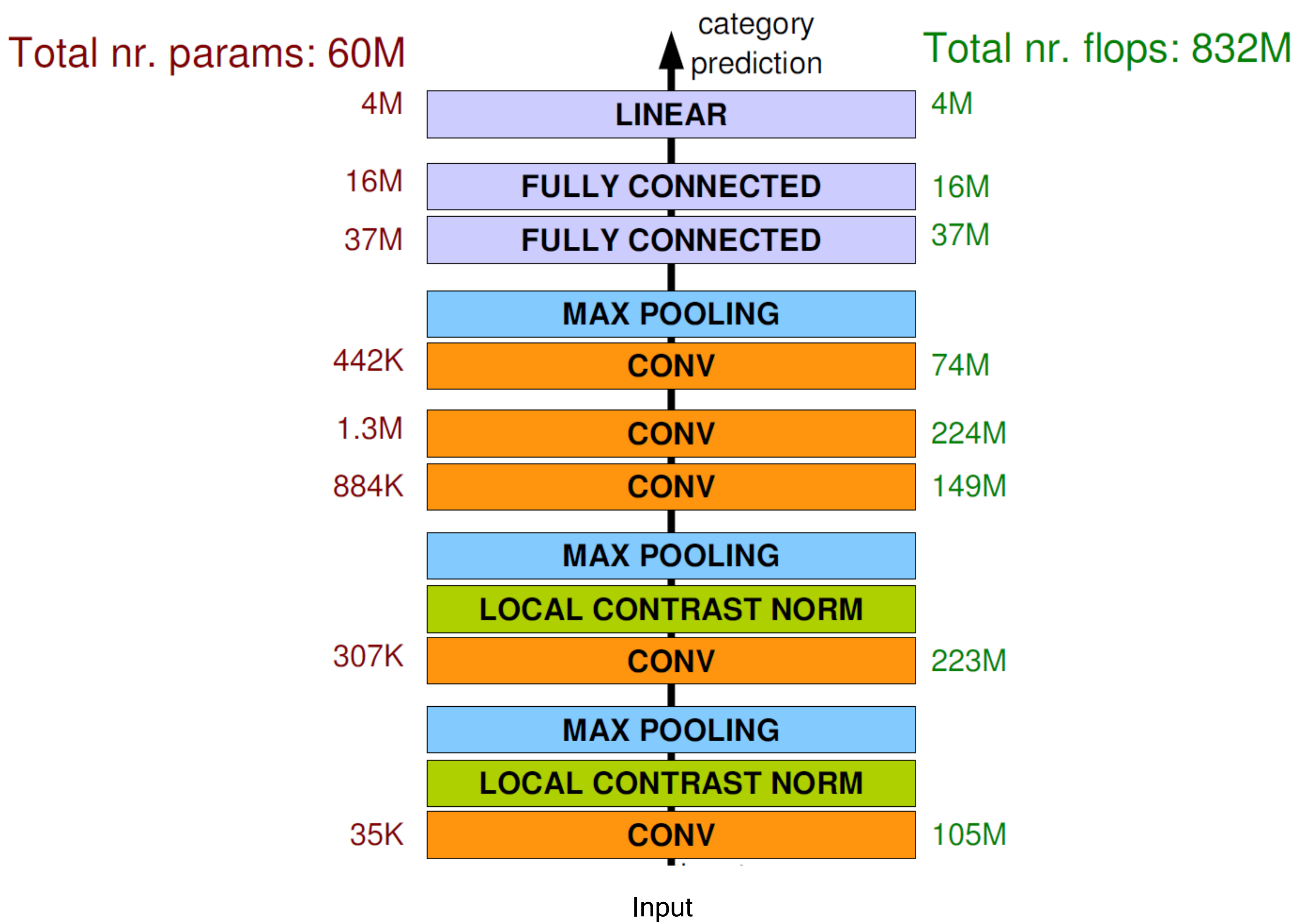
slide credit: R. Fergus

AlexNet

- Similar framework to LeCun'98 but:
  - Bigger model (7 hidden layers, 650,000 units, 60,000,000 params)
  - More data (10^6 vs. 10^3 images)
  - GPU implementation (50x speedup over CPU)
    - Trained on two GPUs for a week

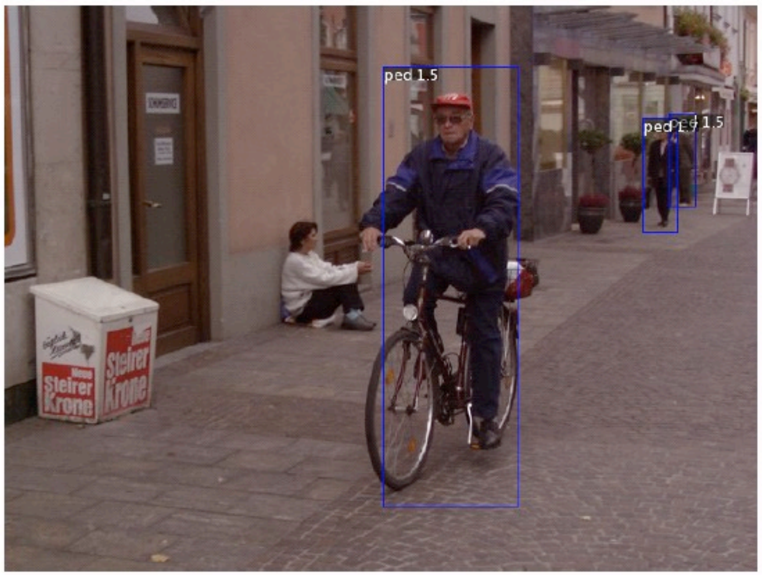


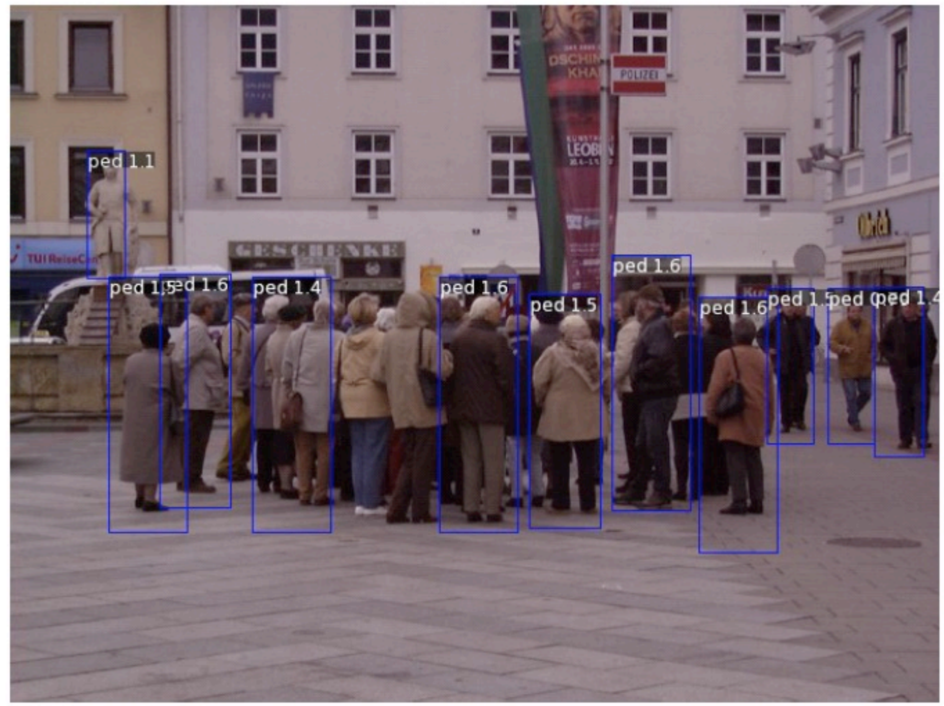
A. Krizhevsky, I. Sutskever, and G. Hinton,
[ImageNet Classification with Deep Convolutional Neural Networks](), NIPS 2012


Total nr. params: 60M
category prediction
Total nr. flops: 832M
4M
LINEAR
4M
16M
FULLY CONNECTED
16M
37M
FULLY CONNECTED
37M
MAX POOLING
442K
CONV
74M
1.3M
CONV
224M
884K
CONV
149M
MAX POOLING
LOCAL CONTRAST NORM
307K
CONV
223M
MAX POOLING
LOCAL CONTRAST NORM
35K
CONV
105M
Input

Conv Nets: Examples

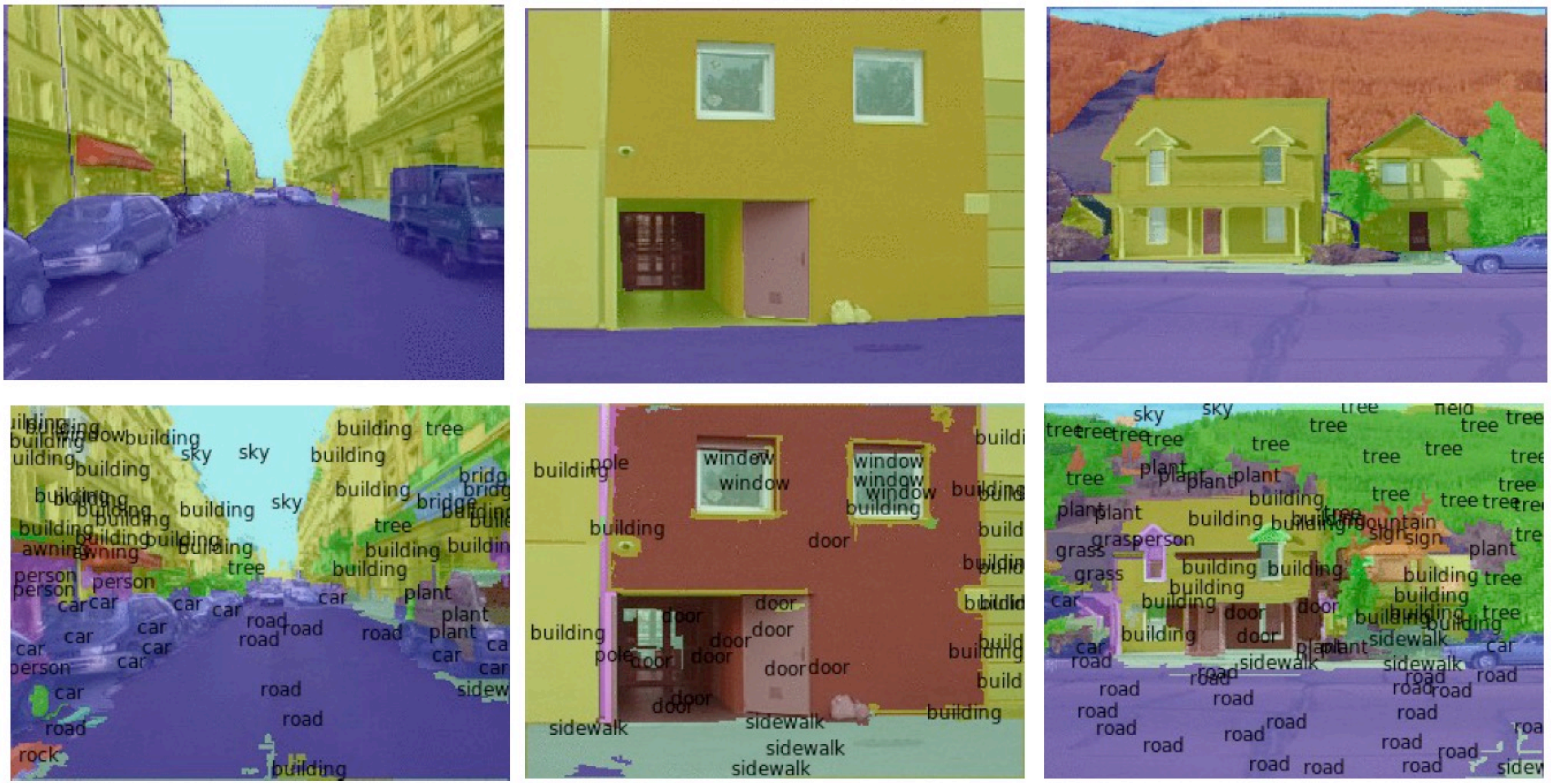
- Pedestrian detection





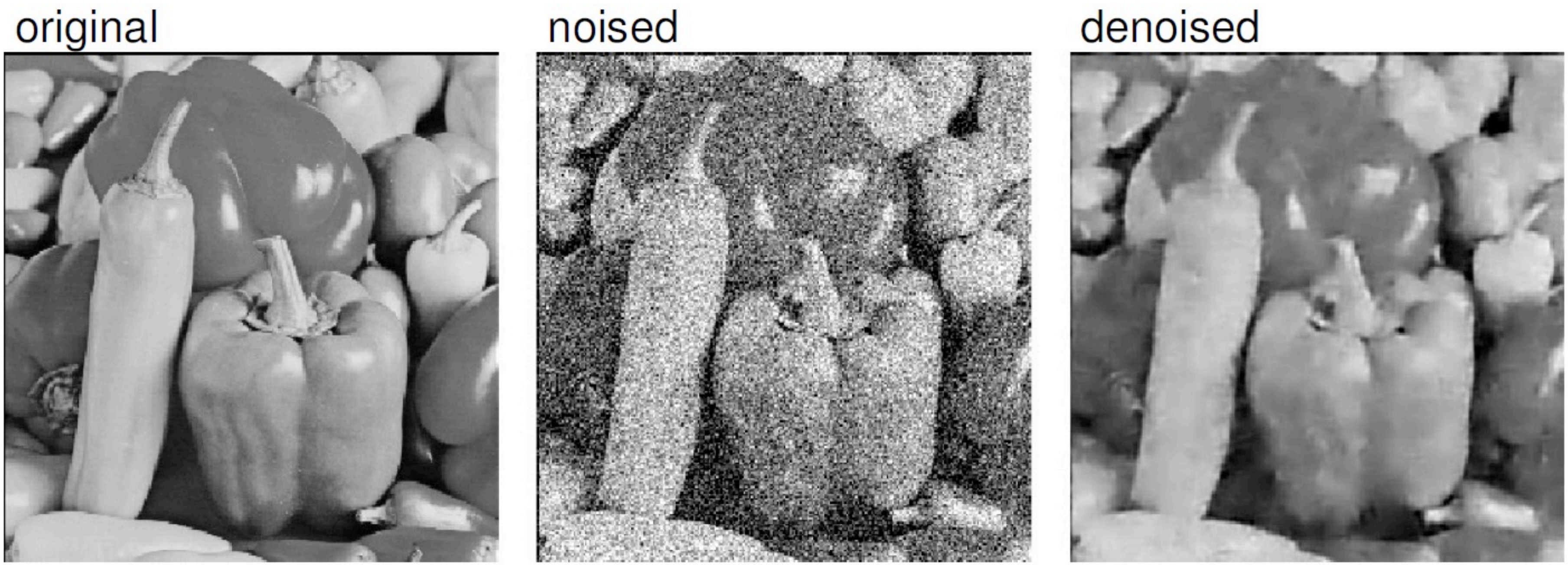
Conv Nets: Examples

- Scene Parsing



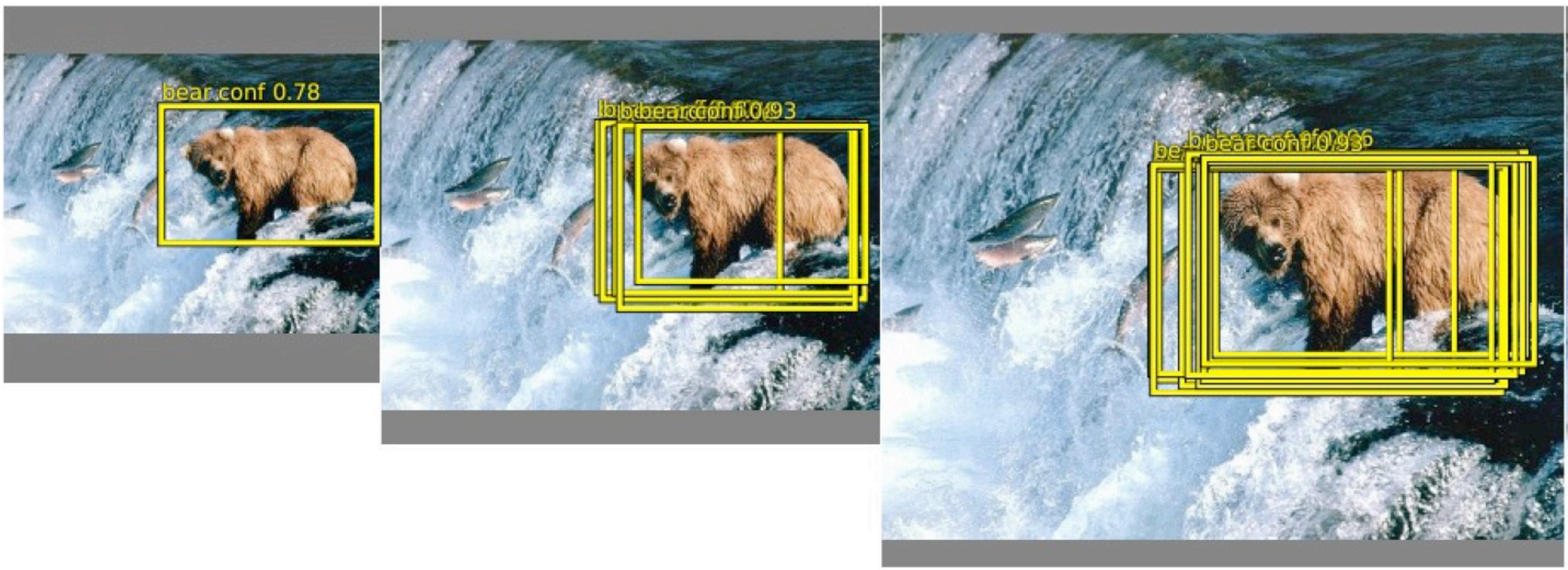
Conv Nets: Examples

- Denoising



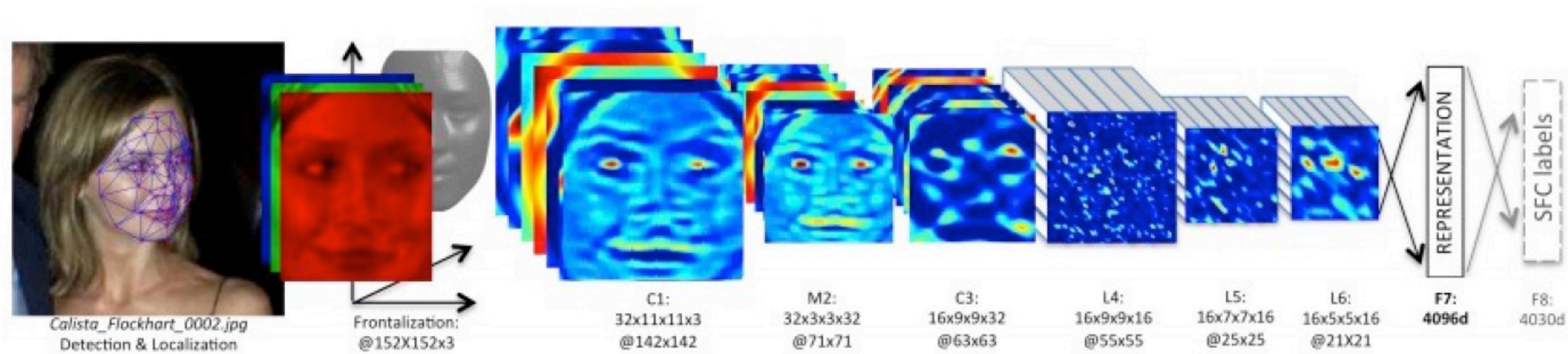
Conv Nets: Examples

- Object Detection



Conv Nets: Examples

- Face Verification and Identification (DeepFace)



Conv Nets: Examples

- Regression (DeepPose)