MIT CSAIL

6.869: Advances in Computer Vision

Antonio Torralba, 2016

MIT COMPUTER VISION

Lecture 6

Pyramids

Learned feedforward visual processing

The Gaussian pyramid

512×512

(original image)

256×256

128×128

64×64

32×32

Image down-sampling

Blur →

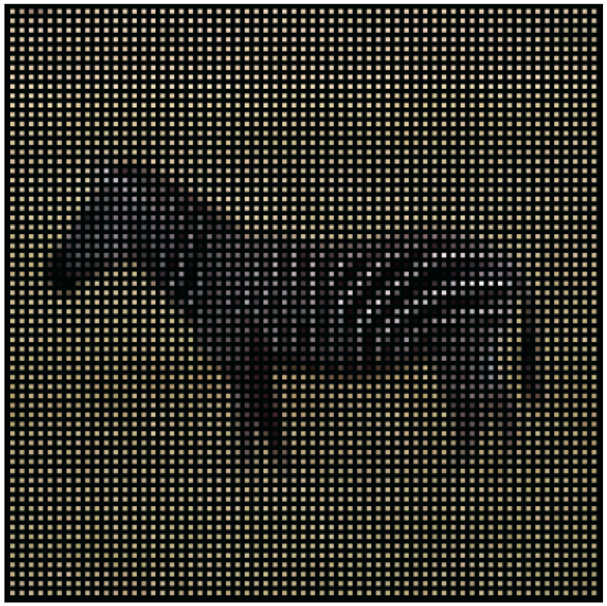
$\downarrow 2$ →

Image up-sampling

$\uparrow 2$ →

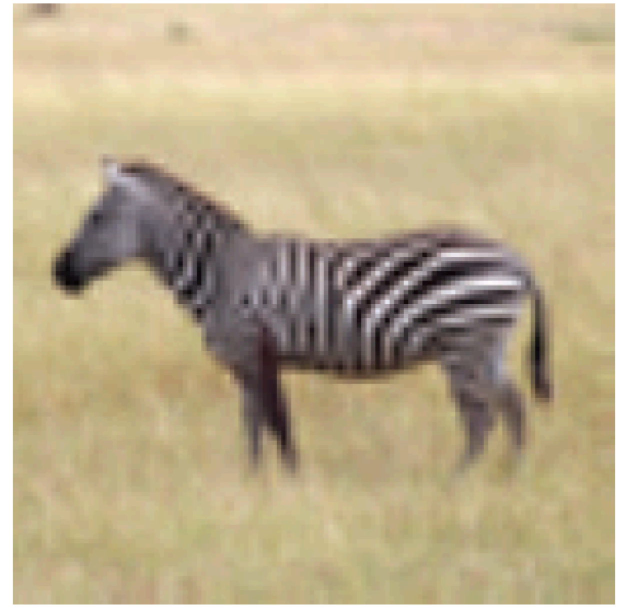
Image up-sampling

64×64



Start by inserting zeros

$$\circ \begin{matrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{matrix} =$$



128×128

Convolution and up-sampling as a matrix multiply (1D case)

$$y_2 = F_3 x_3$$

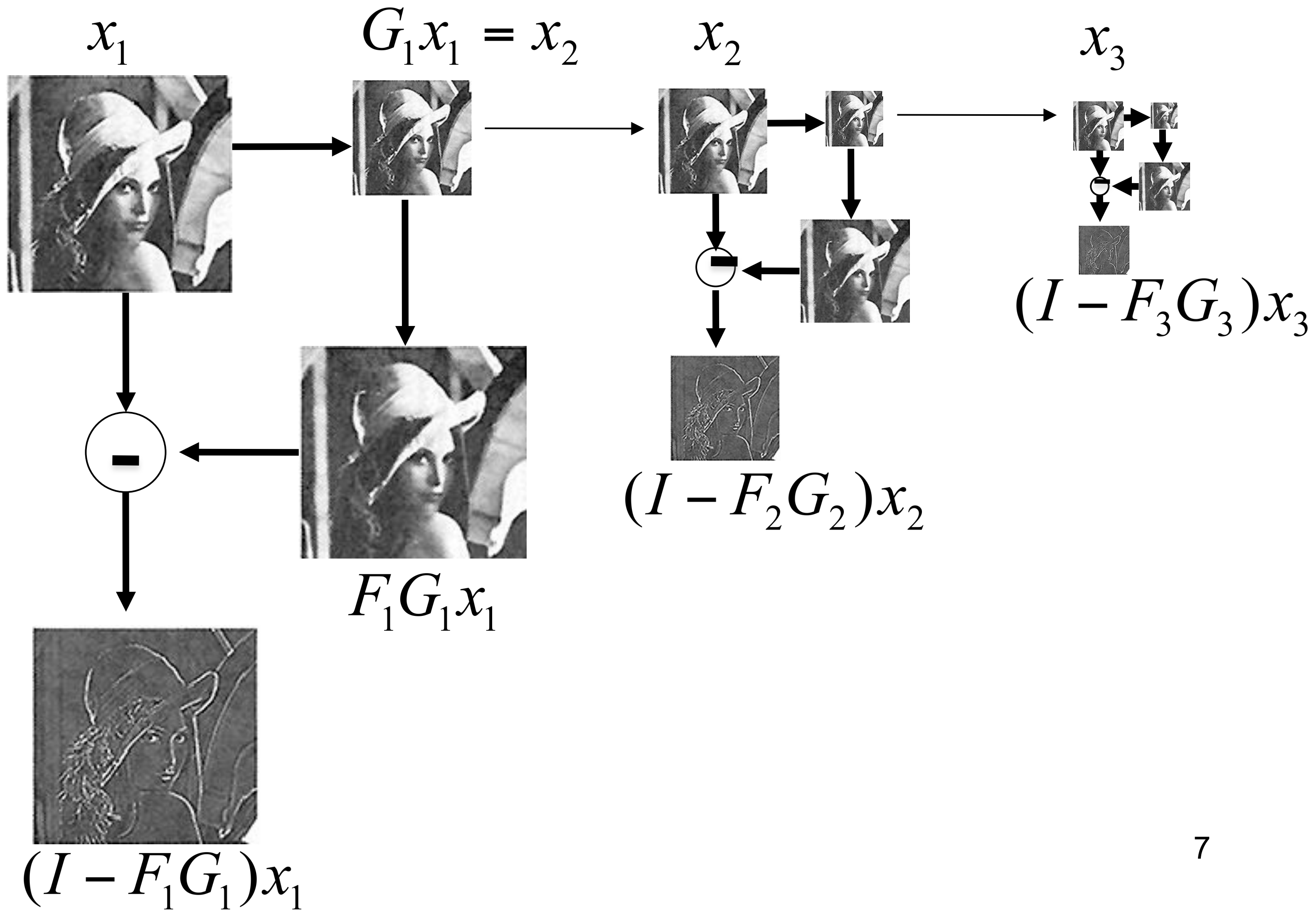
Insert zeros between pixels, then apply a low-pass filter, [1 4 6 4 1]

$$F_3 = \begin{matrix} 6 & 1 & 0 & 0 \\ 4 & 4 & 0 & 0 \\ 1 & 6 & 1 & 0 \\ 0 & 4 & 4 & 0 \\ 0 & 1 & 6 & 1 \\ 0 & 0 & 4 & 4 \\ 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 4 \end{matrix}$$

The Laplacian Pyramid

- Synthesis
  - Compute the difference between upsampled Gaussian pyramid level and Gaussian pyramid level.
  - band pass filter - each level represents spatial frequencies (largely) unrepresented at other level.

Laplacian pyramid algorithm



Showing, at full resolution, the information captured at each level of a Gaussian (top) and Laplacian (bottom) pyramid.

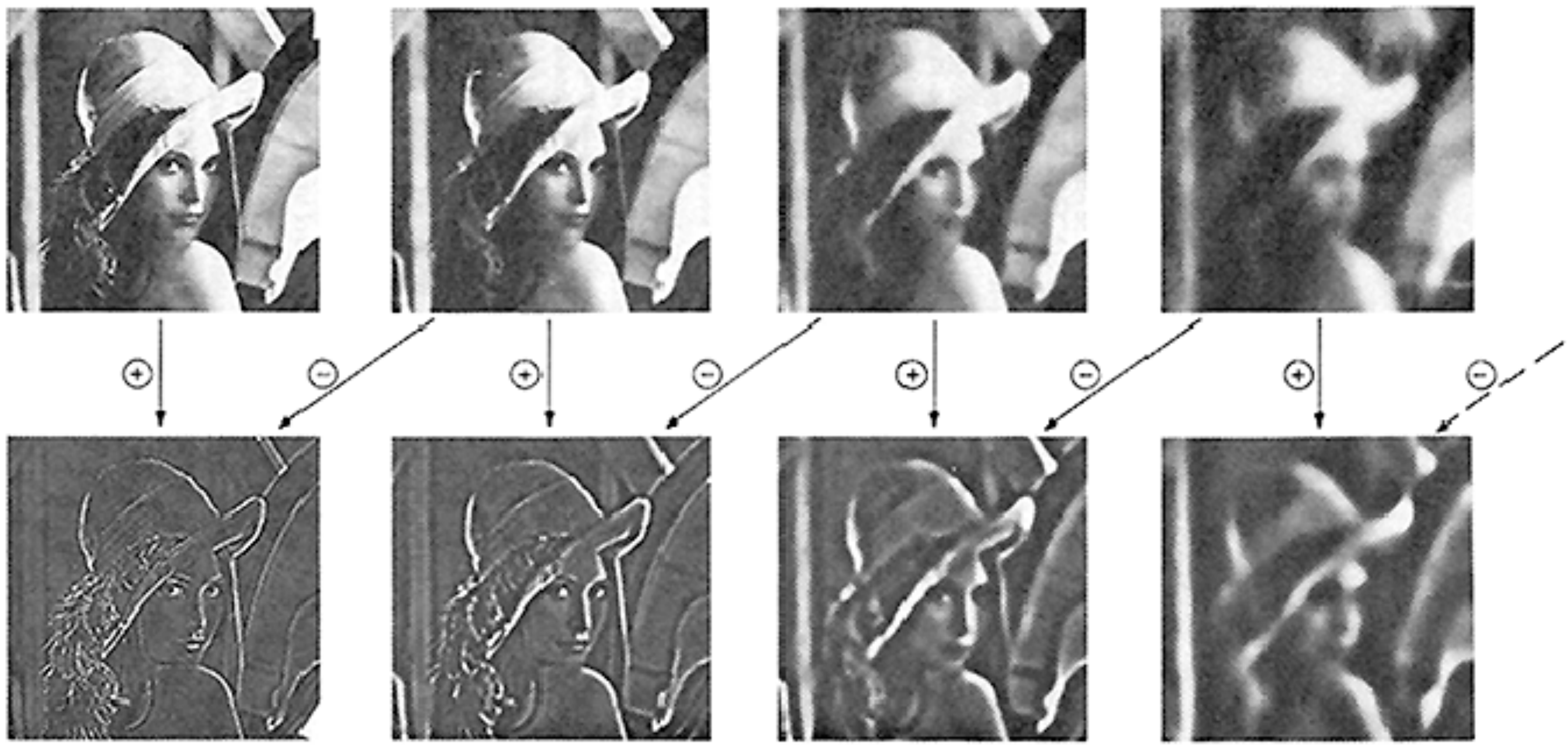


Fig 5. First four levels of the Gaussian and Laplacian pyramid. Gaussian images, upper row, were obtained by expanding pyramid arrays (Fig. 4) through Gaussian interpolation. Each level of the Laplacian pyramid is the difference between the corresponding and next higher levels of the Gaussian pyramid.

http://www-bcs.mit.edu/people/adelson/pub_pdfs/pyramid83.pdf

IEEE TRANSACTIONS ON COMMUNICATIONS, VOL. COM-31, NO. 4, APRIL 1983

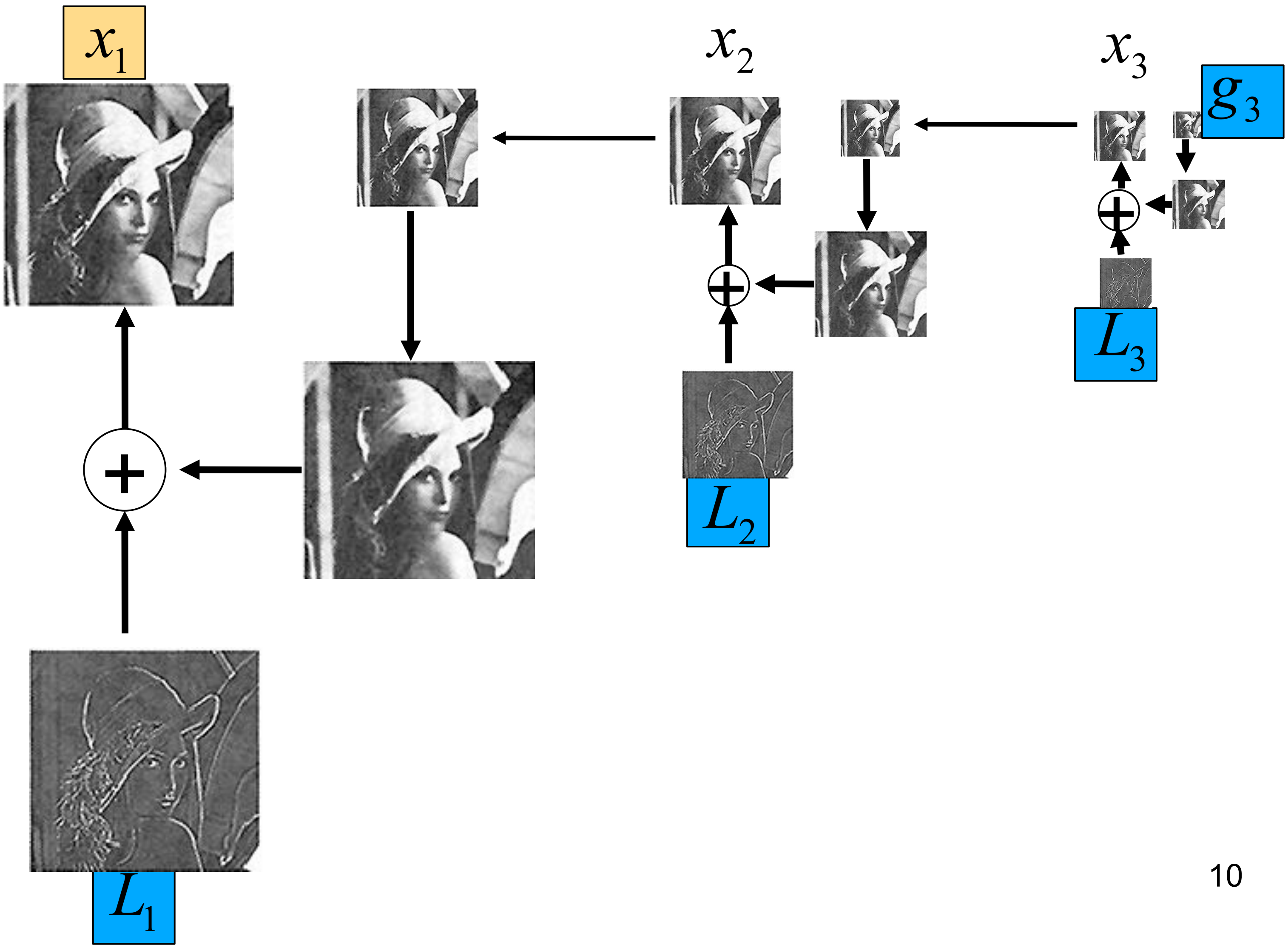
Laplacian pyramid reconstruction algorithm: recover x_1 from L_1, L_2, L_3 and x_4

G# is the blur-and-downsample operator at pyramid level #
F# is the blur-and-upsample operator at pyramid level #

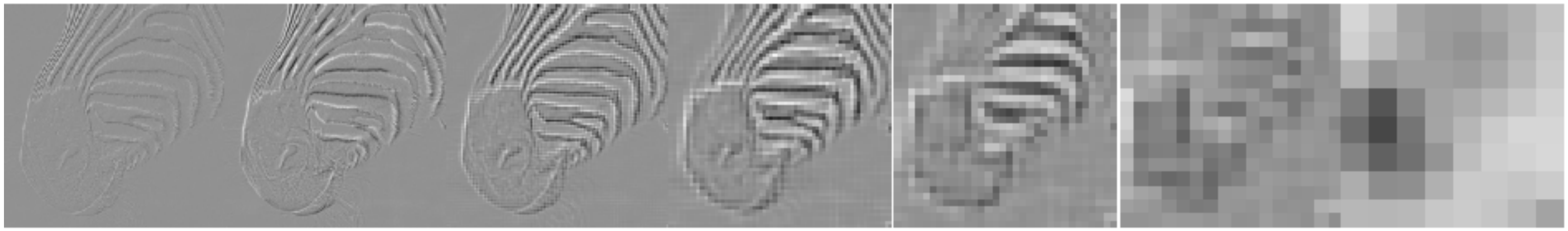
Laplacian pyramid elements:
L1 = (I – F1 G1) x1
L2 = (I – F2 G2) x2
L3 = (I – F3 G3) x3
x2 = G1 x1
x3 = G2 x2
x4 = G3 x3

Reconstruction of original image (x1) from Laplacian pyramid elements:
x3 = L3 + F3 x4
x2 = L2 + F2 x3
x1 = L1 + F1 x2

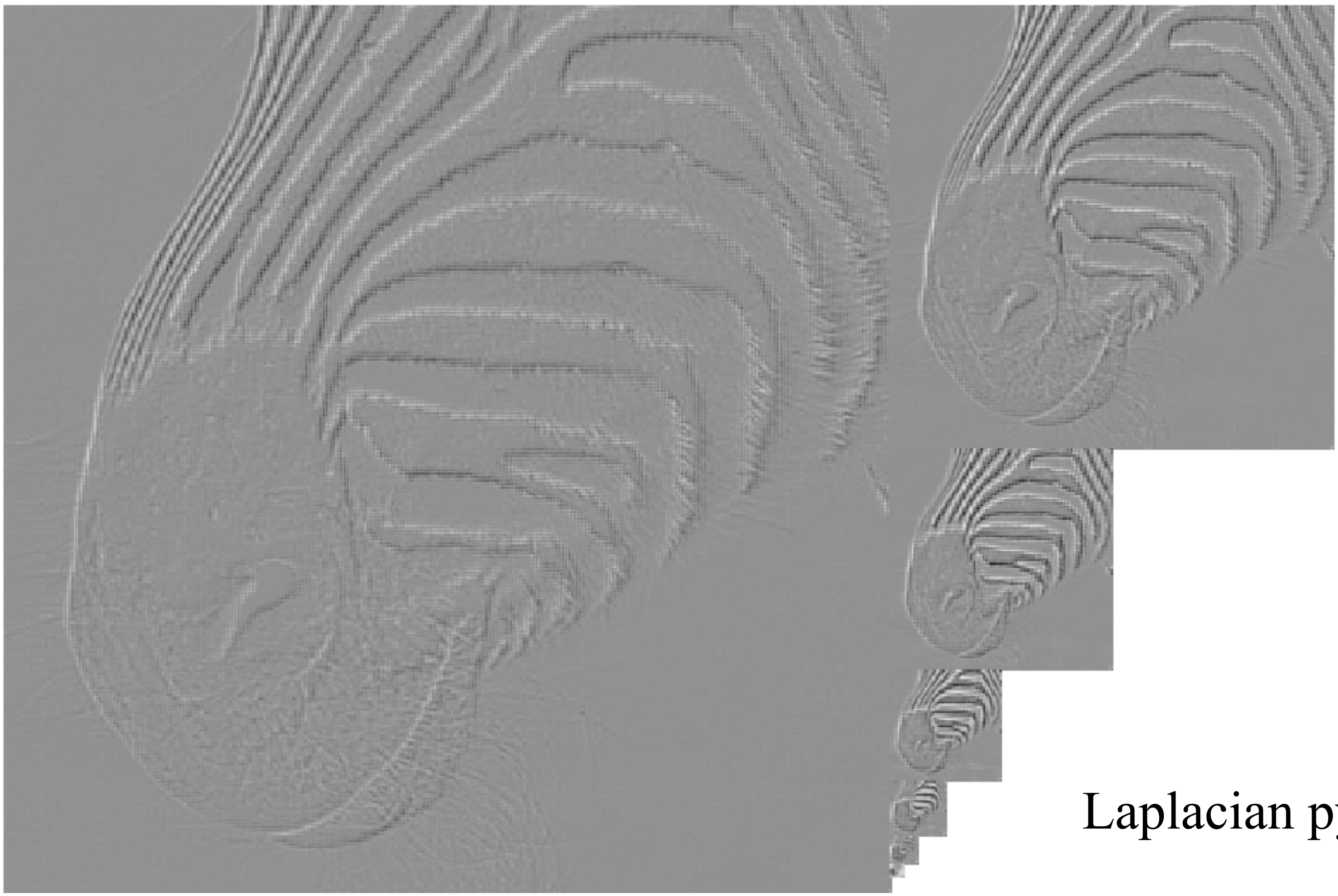
Laplacian pyramid reconstruction algorithm: recover x_1 from L_1, L_2, L_3 and g_3



Gaussian pyramid



512 256 128 64 32 16 8 (Low-pass residual)



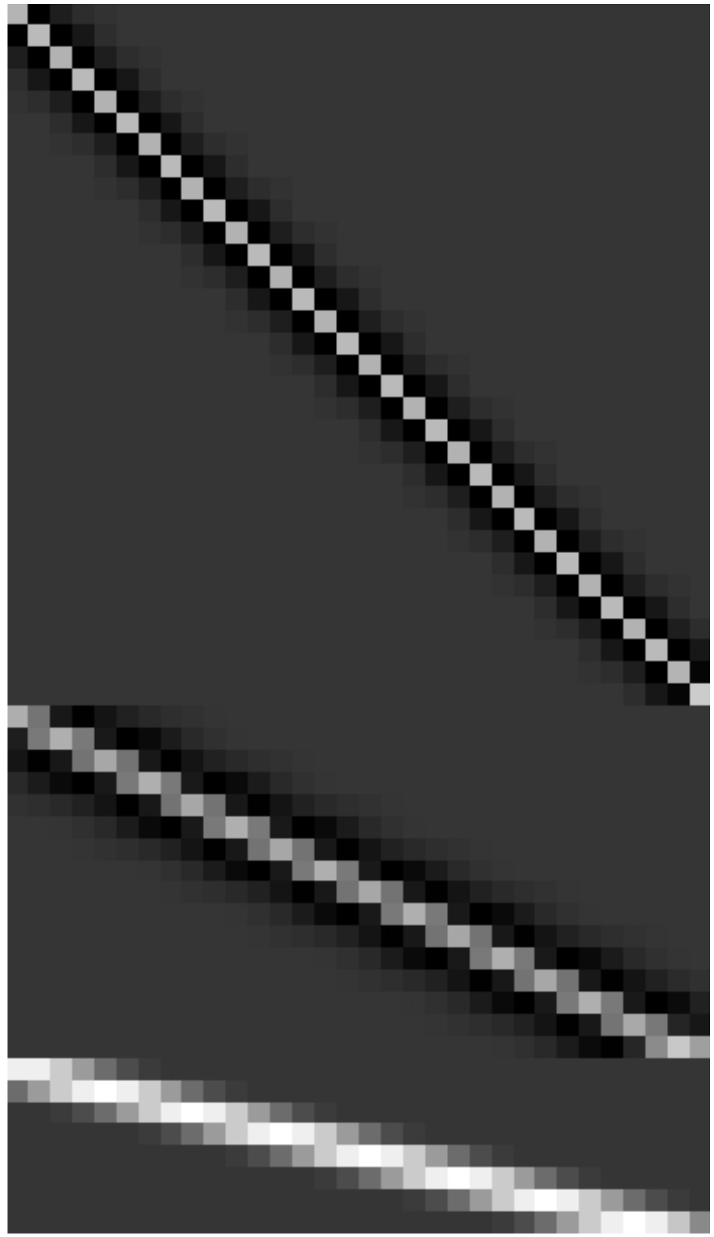
Laplacian pyramid

1-d Laplacian pyramid matrix, for [1 4 6 4 1] low-pass filter

high frequencies

mid-band frequencies

low frequencies

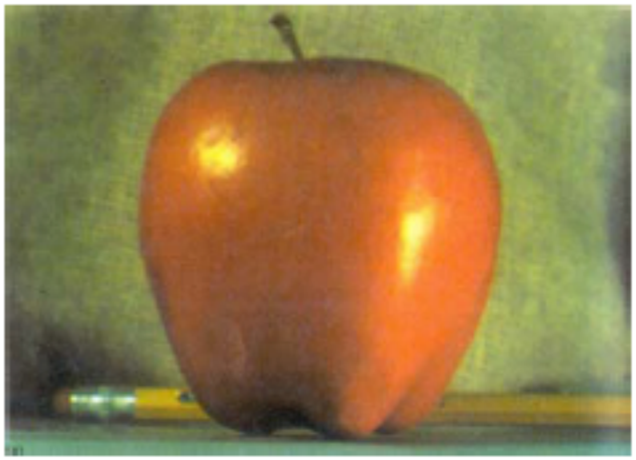


Laplacian pyramid applications

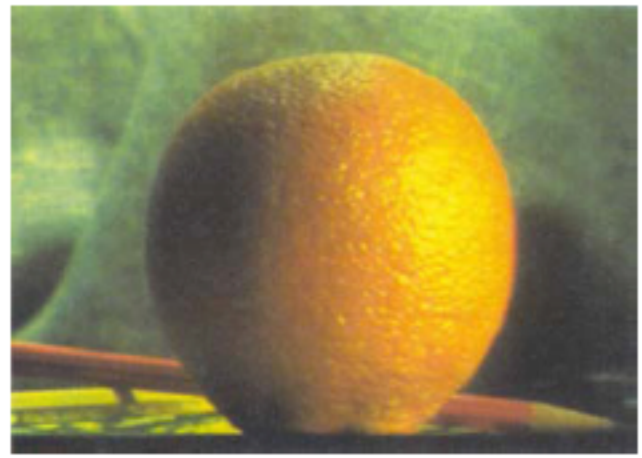
- Texture synthesis
- Image compression
- Noise removal

- Also related to SIFT

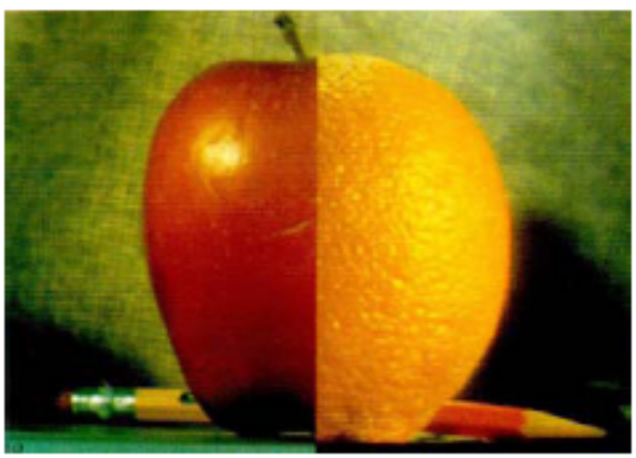
Image blending

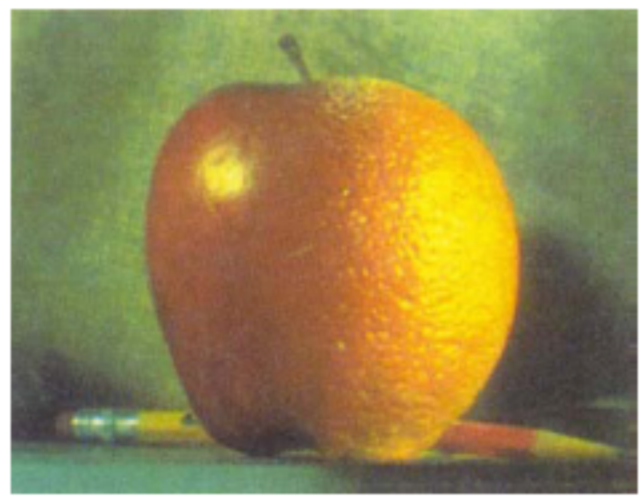


(a)



(b)





Szeliski, Computer Vision, 2010

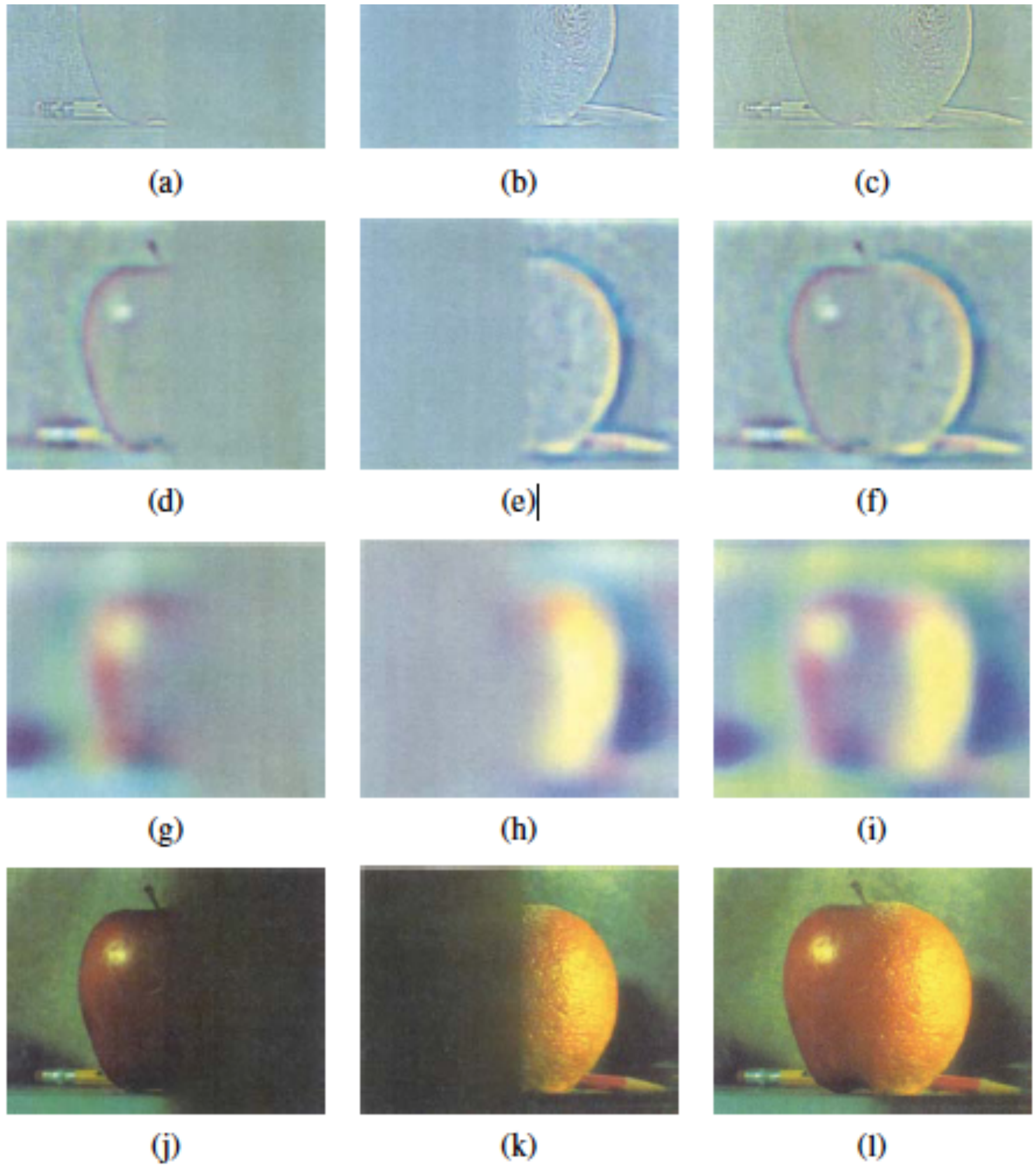
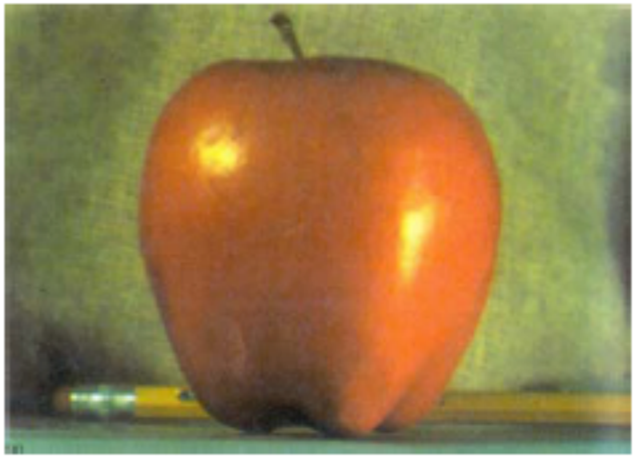


Figure 3.42 Laplacian pyramid blending details (Burt and Adelson 1983b) © 1983 ACM.
The first three rows show the high, medium, and low frequency parts of the Laplacian pyramid

Image blending



- Build Laplacian pyramid for both images: LA, LB
- Build Gaussian pyramid for mask: G
- Build a combined Laplacian pyramid: L(j) = G(j) LA(j) + (1-G(j)) LB(j)
- Collapse L to obtain the blended image

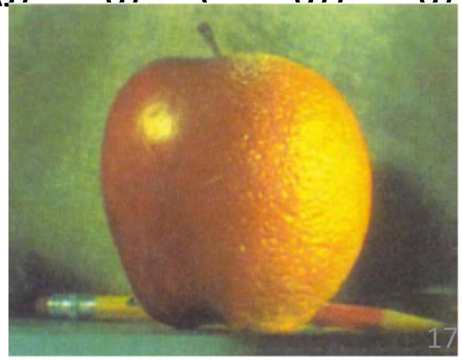
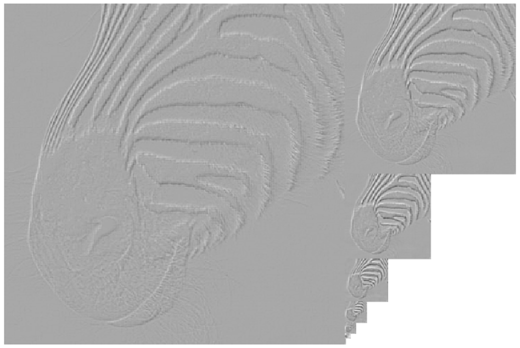


Image pyramids

- Gaussian

Progressively blurred and subsampled versions of the image. Adds scale invariance to fixed-size algorithms.

- Laplacian

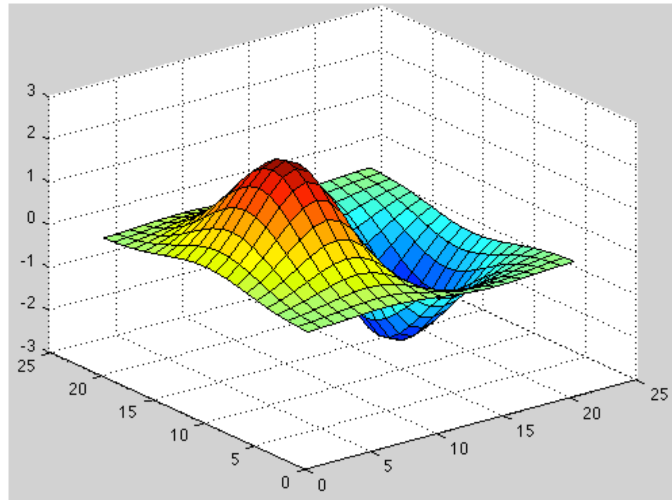


Shows the information added in Gaussian pyramid at each spatial scale. Useful for noise reduction & coding.

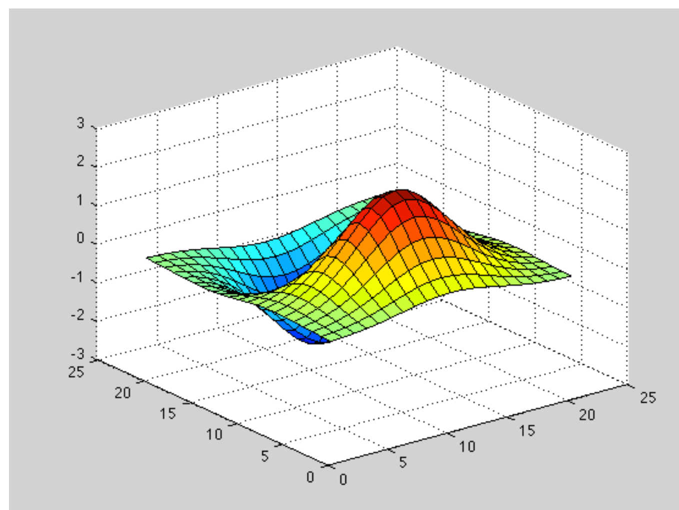
Those pyramids do not encode orientation

Gaussian derivatives: Steerability.

$$g_x(x,y) = \frac{\partial g(x,y)}{\partial x} = \frac{-x}{2\pi\sigma^4} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

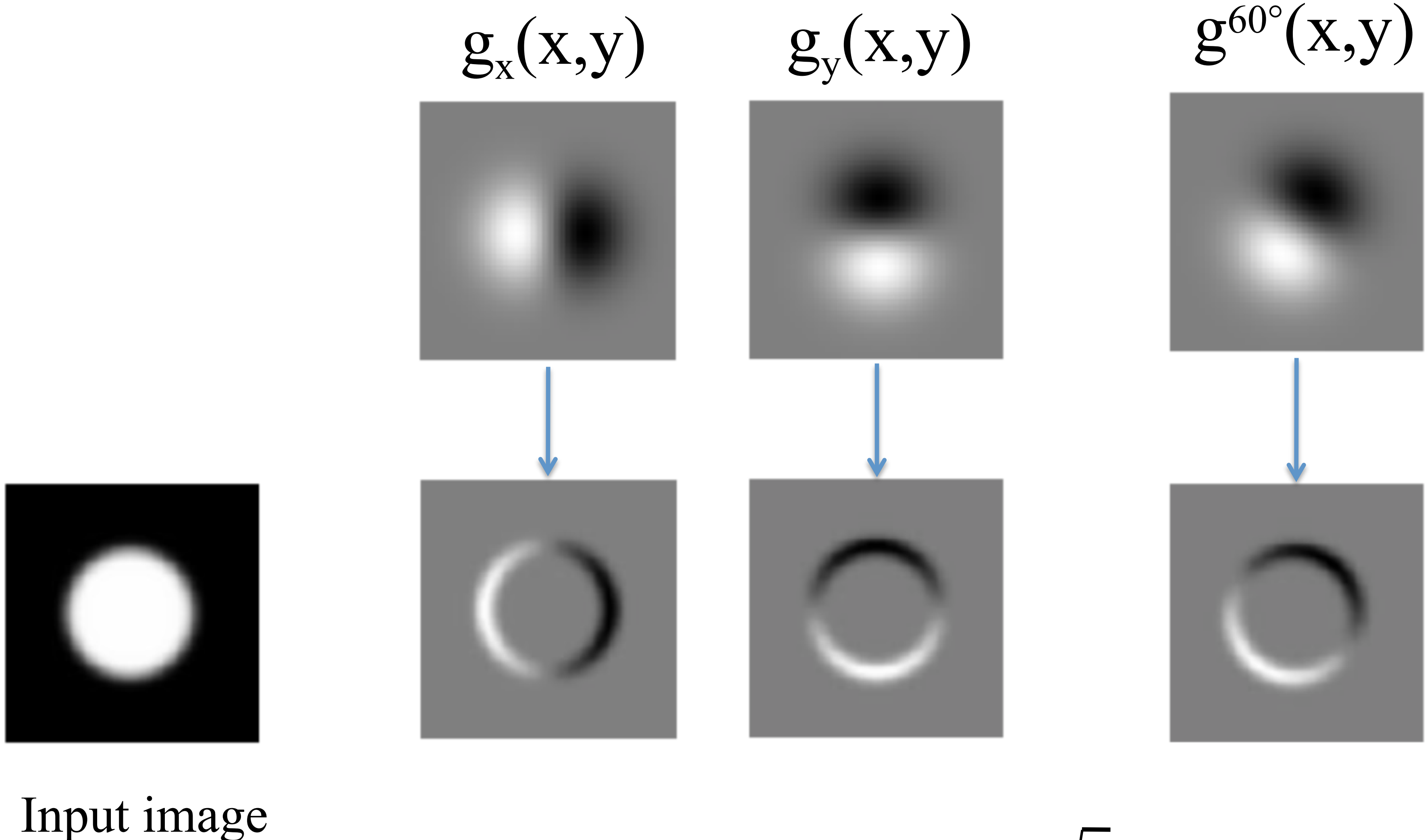


$$g_y(x,y) = \frac{\partial g(x,y)}{\partial y} = \frac{-y}{2\pi\sigma^4} e^{-\frac{x^2+y^2}{2\sigma^2}}$$



What about other orientations not axis aligned?

Gaussian derivatives: Steerability.



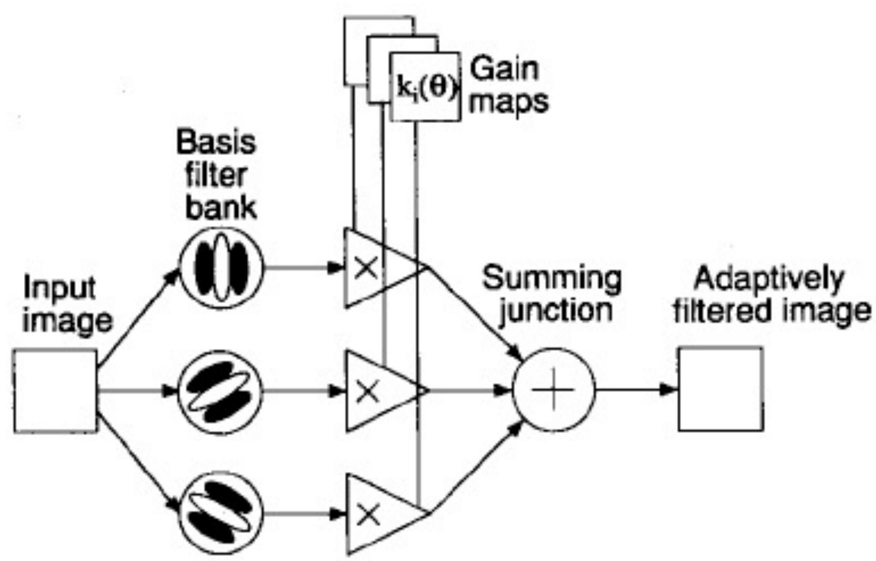
$$g^{60°}(x,y) = ½g_x(x,y) + \sqrt{3}/2g_y(x,y)$$

Gaussian derivatives: Steerability.

For the Gaussian derivatives, any orientation can be obtained as a linear combination of two basis functions:

$$g^{\alpha}(x,y) = \cos(\alpha)g_x(x,y) + \sin(\alpha)g_y(x,y)$$

In general, a kernel is steerable, if it can any rotation can be obtained as a linear combination of N basis functions.



Steereability of gaussian derivatives, Freeman & Adelson 92

Steerable filters

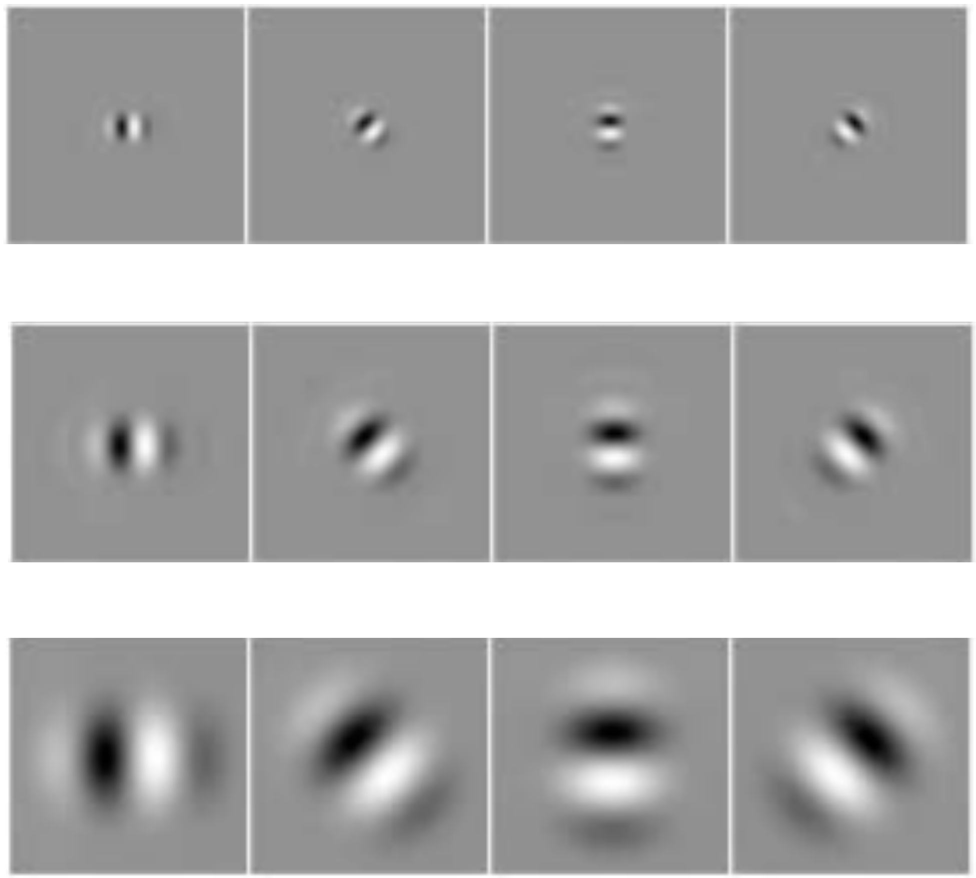
- N-th order derivatives of Gaussians are steerable with N basis functions.
- In general, if a function can be decomposed as a Fourier basis in polar coordinates with a finite number of polar terms, then the function is steerable.

$$f(r, \phi) = \sum_{n=-N}^{N} a_n(r) e^{in\phi}$$

Steereability of gaussian derivatives, Freeman & Adelson 92

Steerable Pyramids

We can extend the oriented filters into a multi-scale pyramid



Simoncelli, Freeman, Adelson

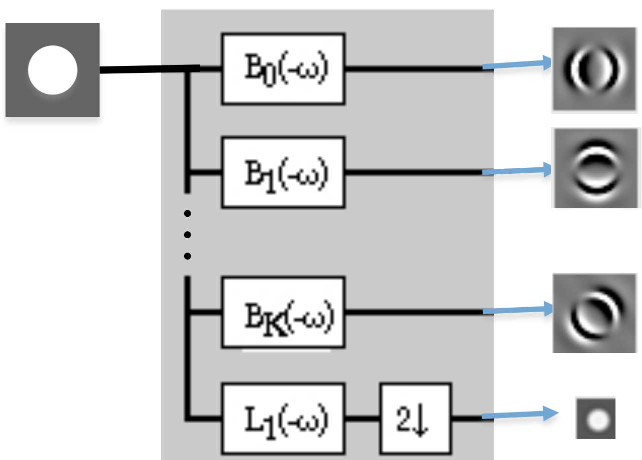
Steerable Pyramids

Simoncelli, Freeman, Adelson

Steerable Pyramid

We may combine Steerability with Pyramids to get a Steerable Laplacian Pyramid as shown below

Decomposition

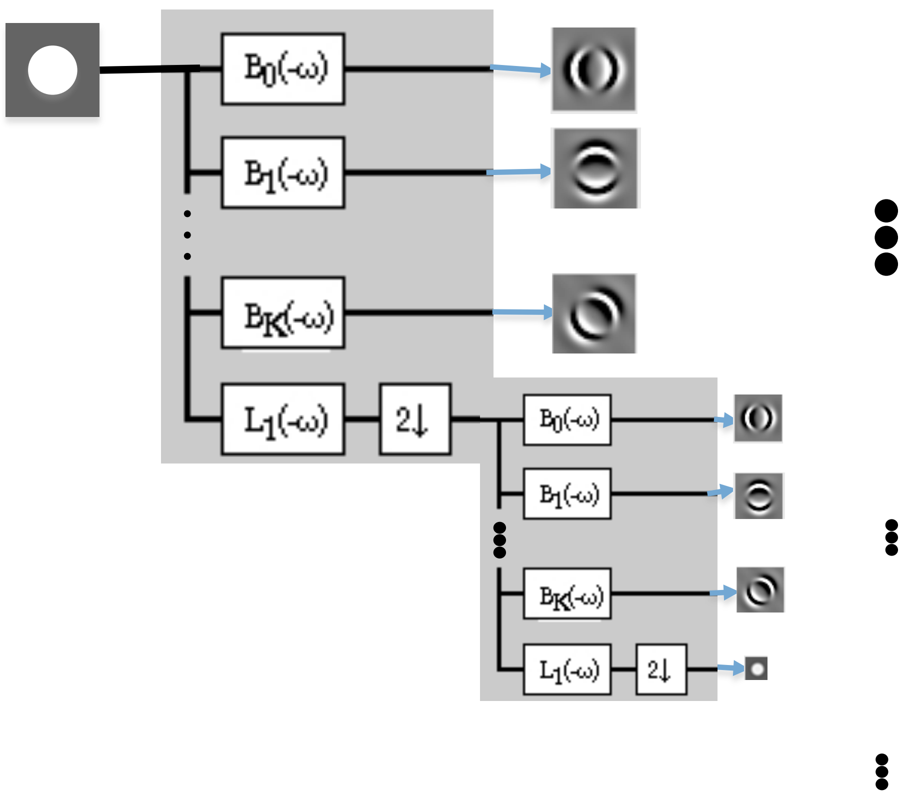


Images from: http://www.cis.upenn.edu/~eero/steerpyr.html

Steerable Pyramid

We may combine Steerability with Pyramids to get a Steerable Laplacian Pyramid as shown below

Decomposition

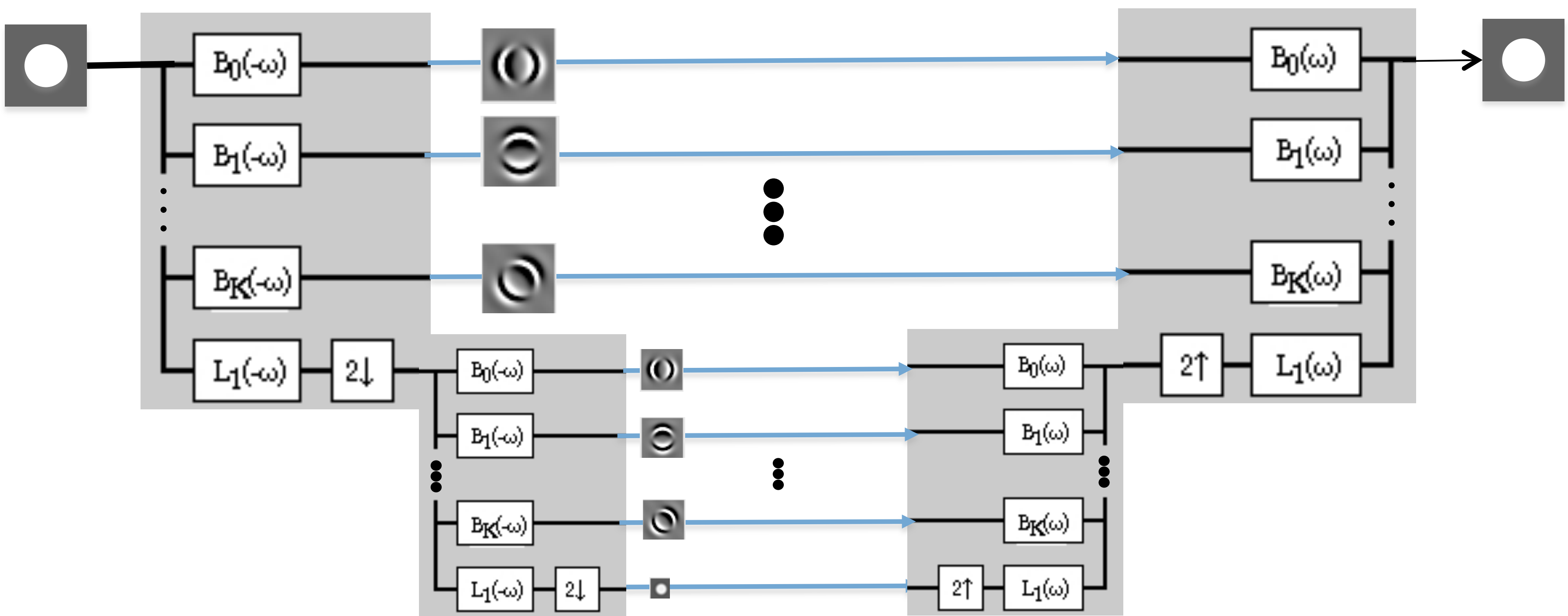


Images from: http://www.cis.upenn.edu/~eero/steerpyr.html

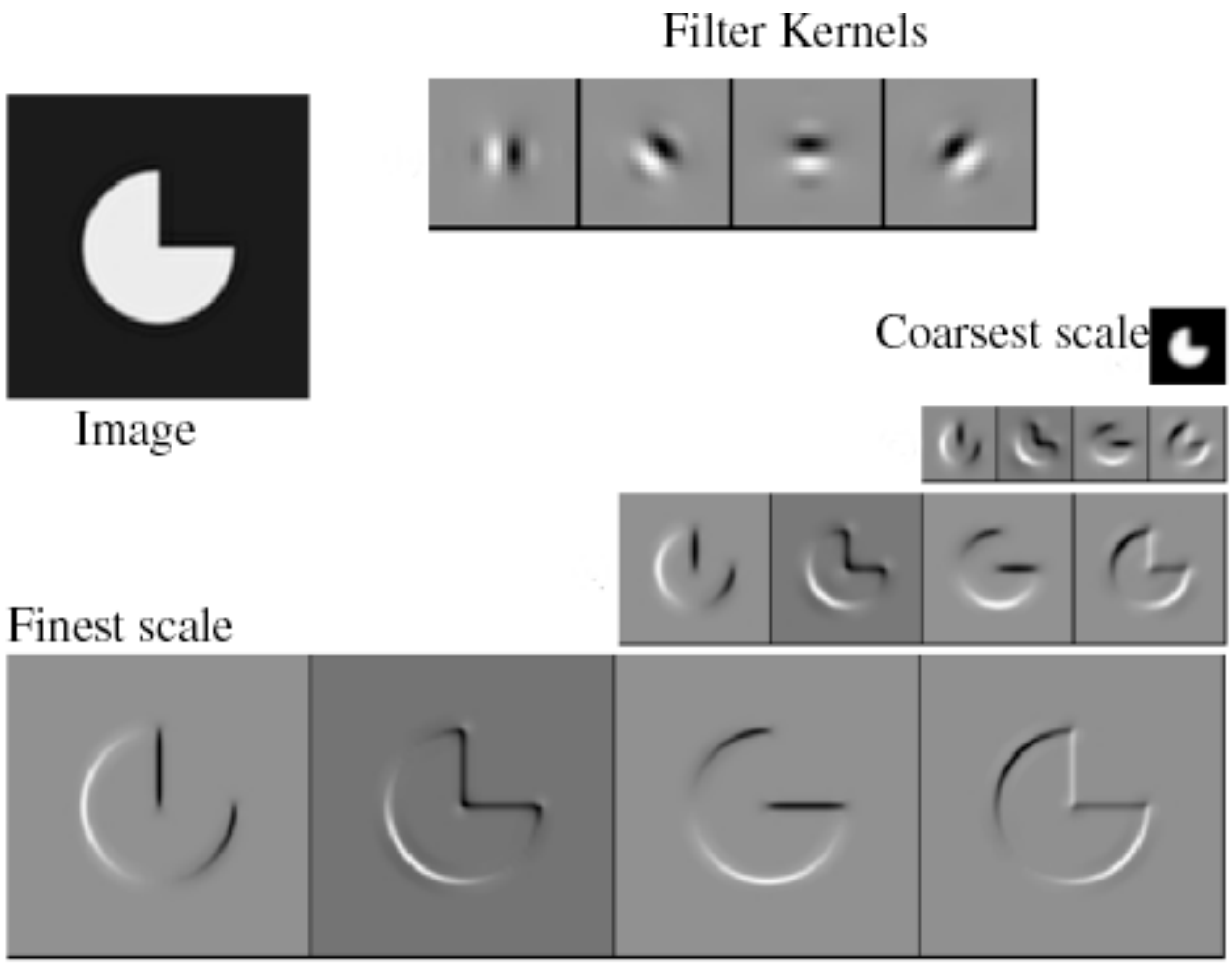
Steerable Pyramid

We may combine Steerability with Pyramids to get a Steerable Laplacian Pyramid as shown below

Decomposition **Reconstruction**



Images from: http://www.cis.upenn.edu/~eero/steerpyr.html



There is also a high pass residual…

"Shiftable MultiScale Transforms," by Simoncelli et al., IEEE Transactions on Information Theory, 1992

Steerable pyramid

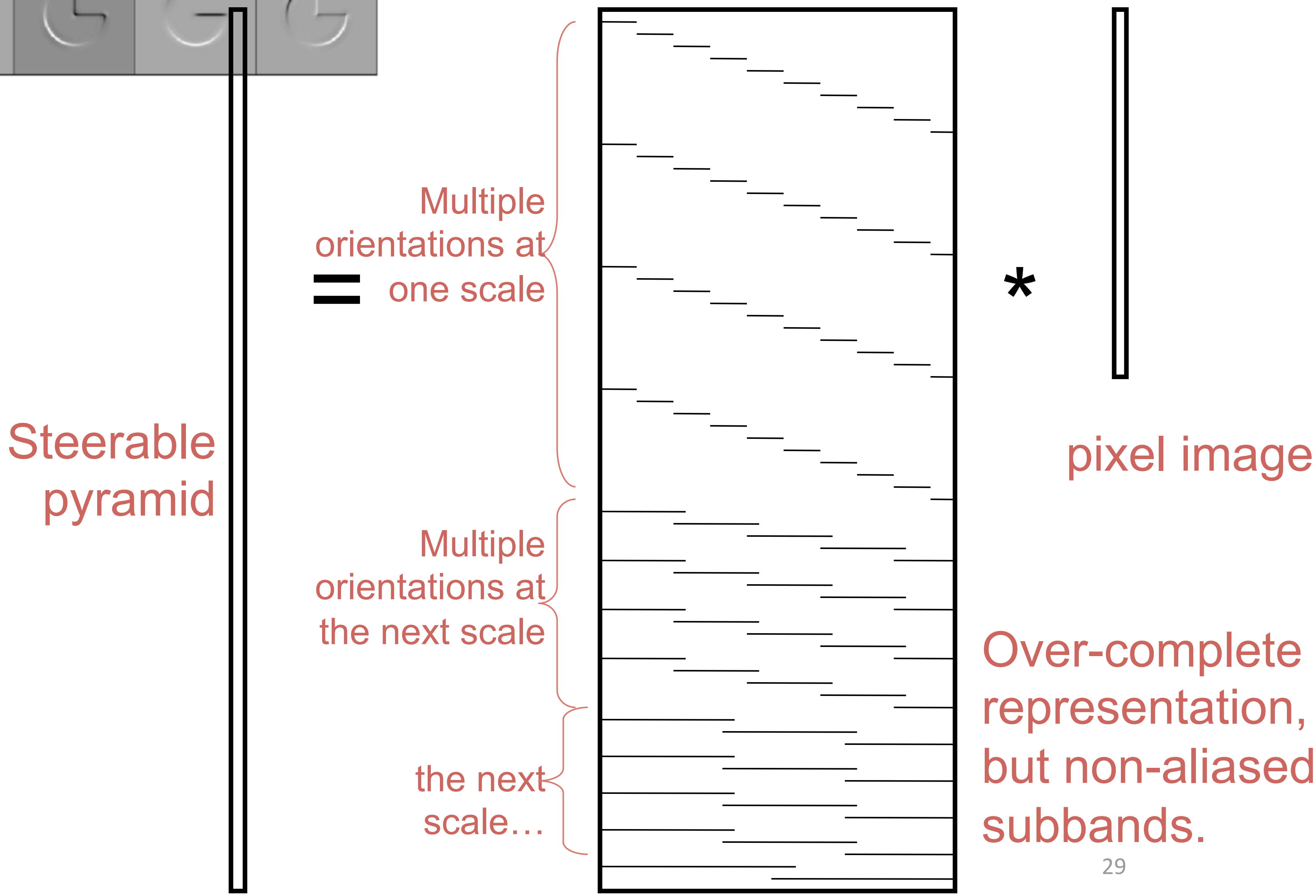
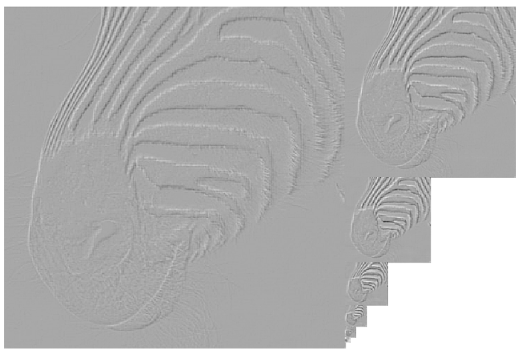


Image pyramids

- Gaussian

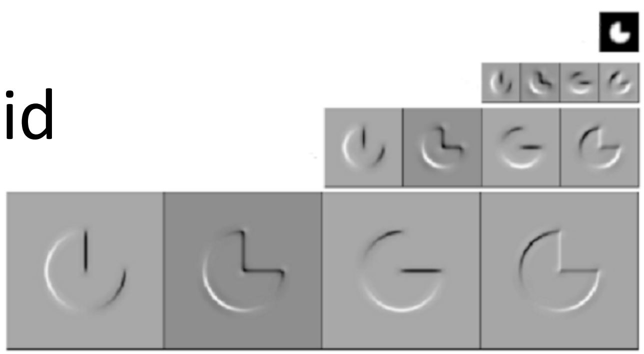
Progressively blurred and subsampled versions of the image. Adds scale invariance to fixed-size algorithms.

- Laplacian



Shows the information added in Gaussian pyramid at each spatial scale. Useful for noise reduction & coding.

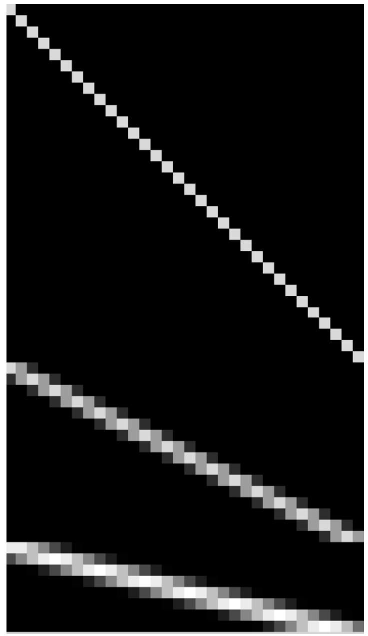
- Steerable pyramid



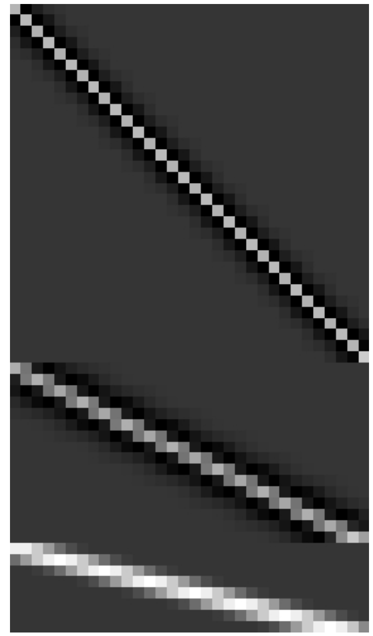
Shows components at each scale and orientation separately. Non-aliased subbands. Good for texture and feature analysis. But overcomplete and with HF residual.

Image transformations

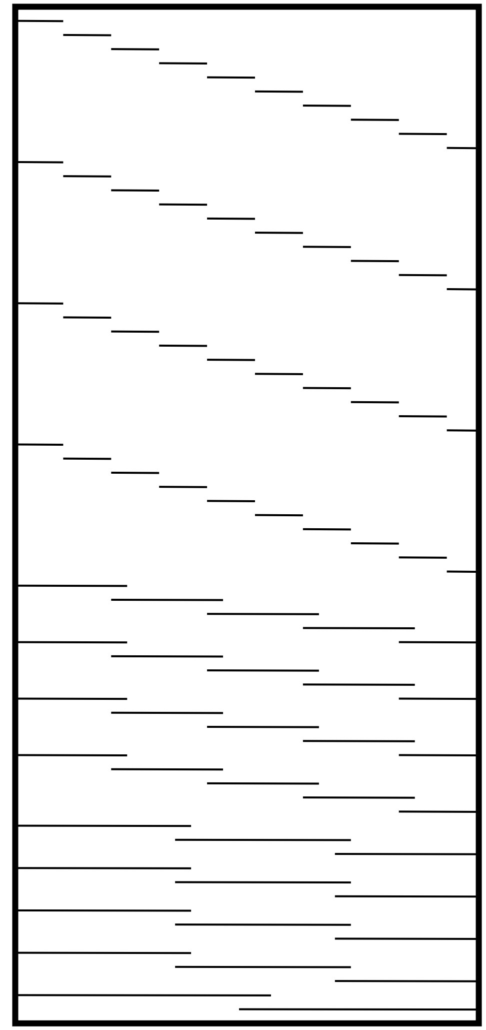
DFT



Gaussian pyramid



Laplacian pyramid



Steerable pyramid

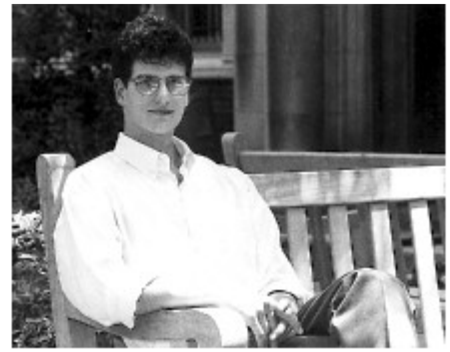
Matlab resources for pyramids (with tutorial)

http://www.cns.nyu.edu/~eero/software.html

Eero P. Simoncelli

Associate Investigator,
Howard Hughes Medical Institute

Associate Professor,
Neural Science and Mathematics,
New York University



Matlab resources for pyramids (with tutorial)

http://www.cns.nyu.edu/~eero/software.html

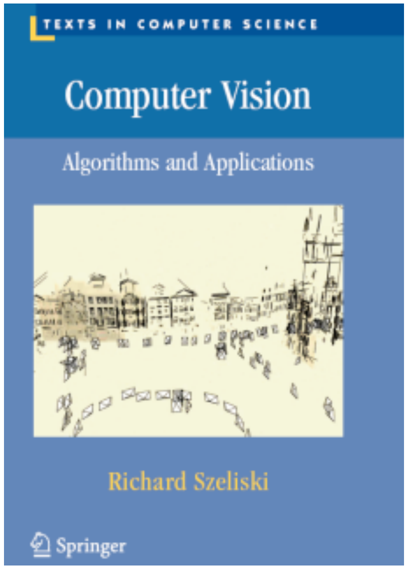
lcv

Laboratory for Computational Vision

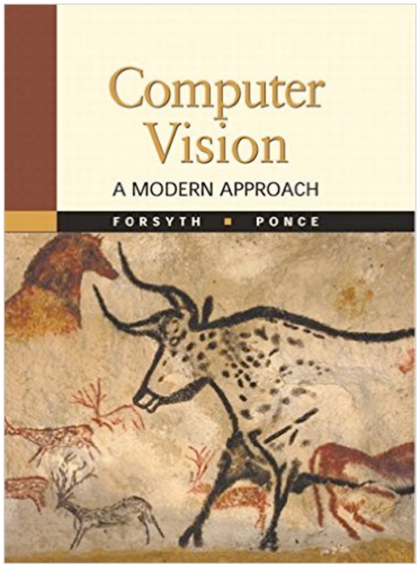
Home | People | Research | Publications | Software

Publicly Available Software Packages

- Texture Analysis/Synthesis - Matlab code is available for analyzing and synthesizing visual textures. README | Contents | ChangeLog | Source code (UNIX/PC, gzip'ed tar file)
- EPWIC - Embedded Progressive Wavelet Image Coder. C source code available.
- **matlabPyrTools** - Matlab source code for multi-scale image processing. Includes tools for building and manipulating Laplacian pyramids, QMF/Wavelets, and steerable pyramids. Data structures are compatible with the Matlab wavelet toolbox, but the convolution code (in C) is faster and has many boundary-handling options. README, Contents, Modification list, UNIX/PC source or Macintosh source.
- The Steerable Pyramid, an (approximately) translation- and rotation-invariant multi-scale image decomposition. MatLab (see above) and C implementations are available.
- Computational Models of cortical neurons. Macintosh program available.
- EPIC - Efficient Pyramid (Wavelet) Image Coder. C source code available.
- OBVIUS [Object-Based Vision & Image Understanding System]: README / ChangeLog / Doc (225k) / Source Code (2.25M).
- CL-SHELL [Gnu Emacs <-> Common Lisp Interface]: README / Change Log / Source Code (119k).



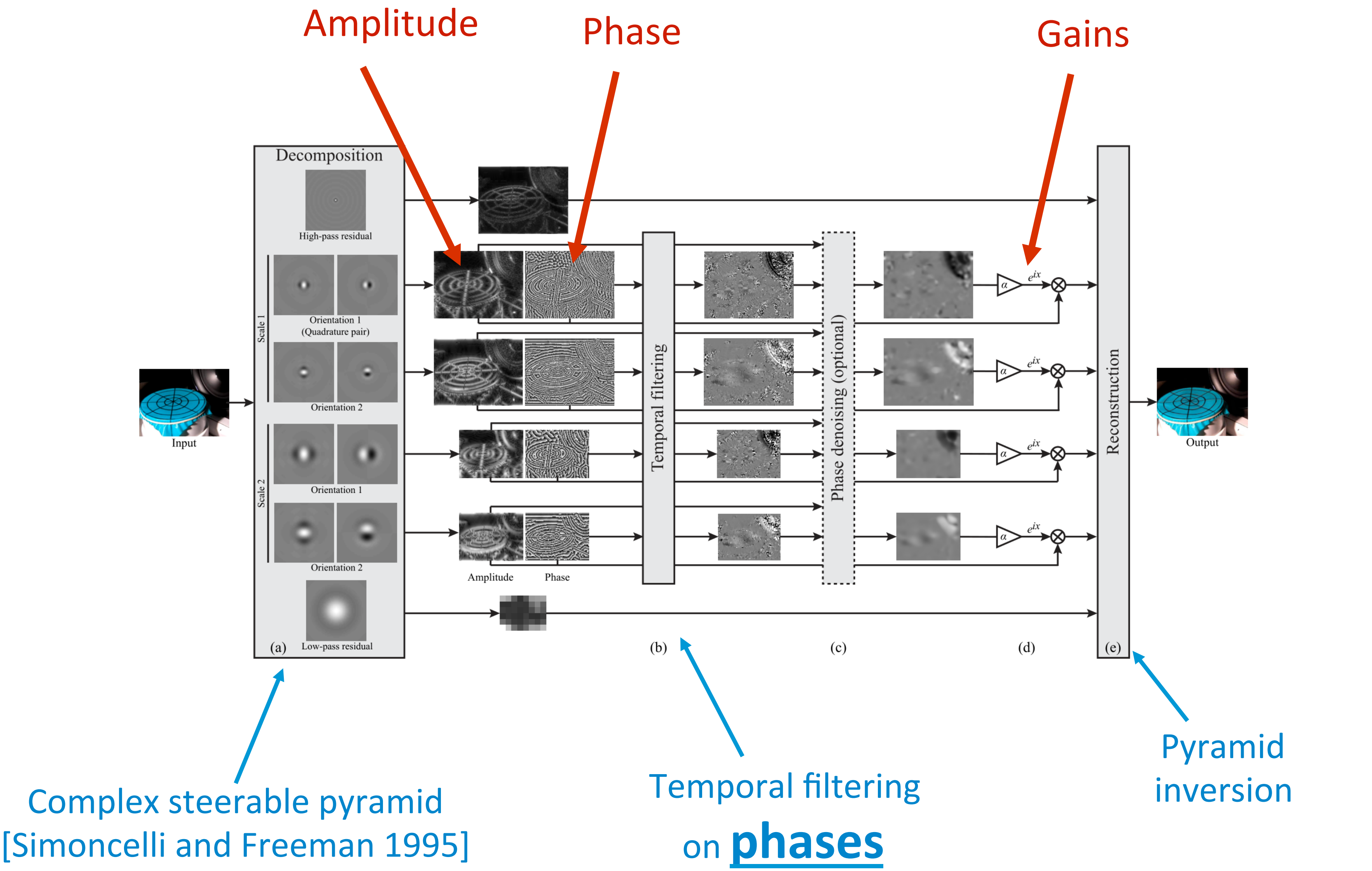
Chapter 3: Image Processing



Why use these representations?

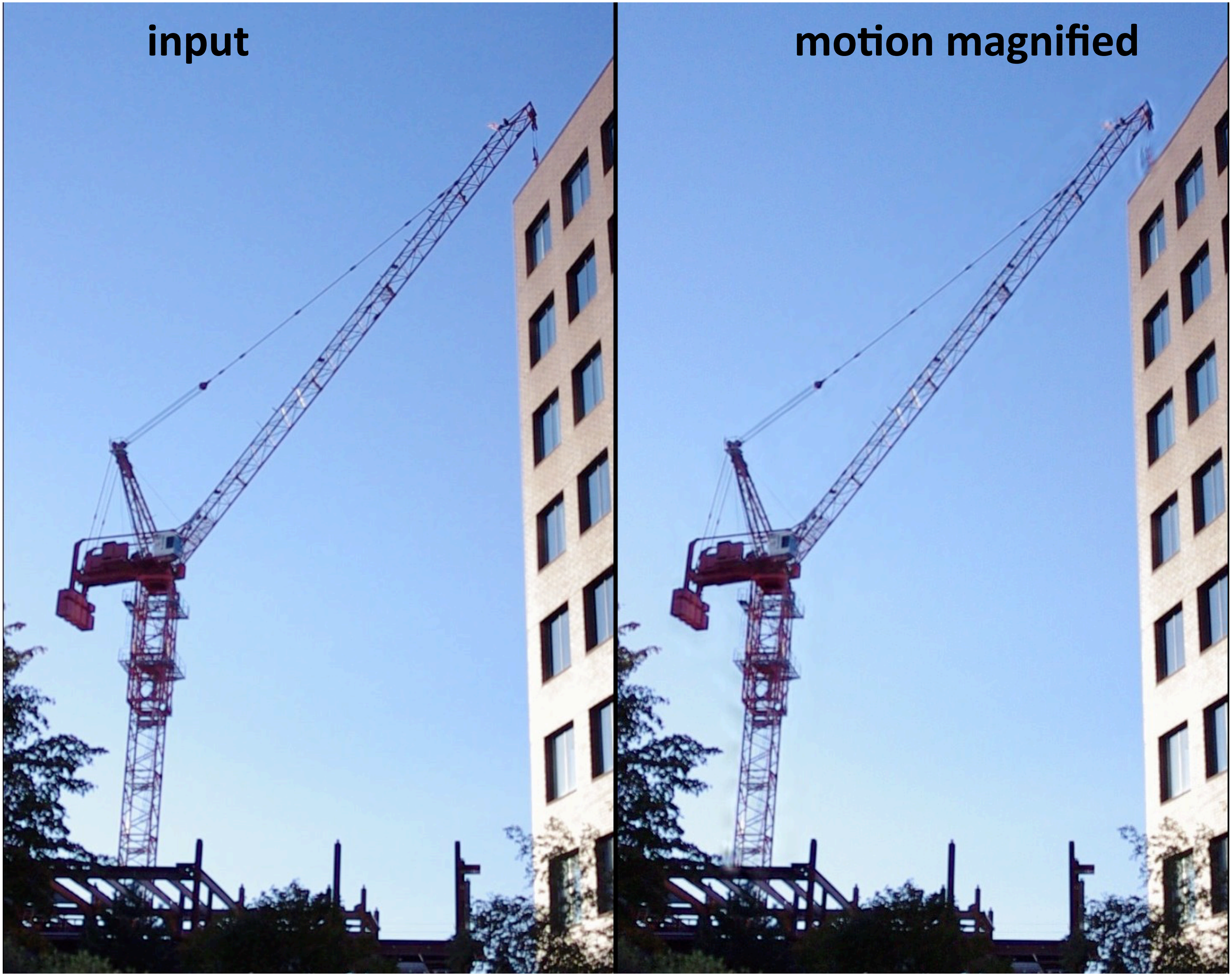
- Handle real-world size variations with a constant-size vision algorithm.
- Remove noise
- Analyze texture
- Recognize objects
- Label image features
- Image priors can be specified naturally in terms of wavelet pyramids.

Phase-based Pipeline (SIGGRAPH'13)



http://people.csail.mit.edu/mrub/vidmag/

input **motion magnified**


input
motion magnified

MIT CSAIL

6.869: Advances in Computer Vision

Antonio Torralba, 2016

MIT COMPUTER VISION

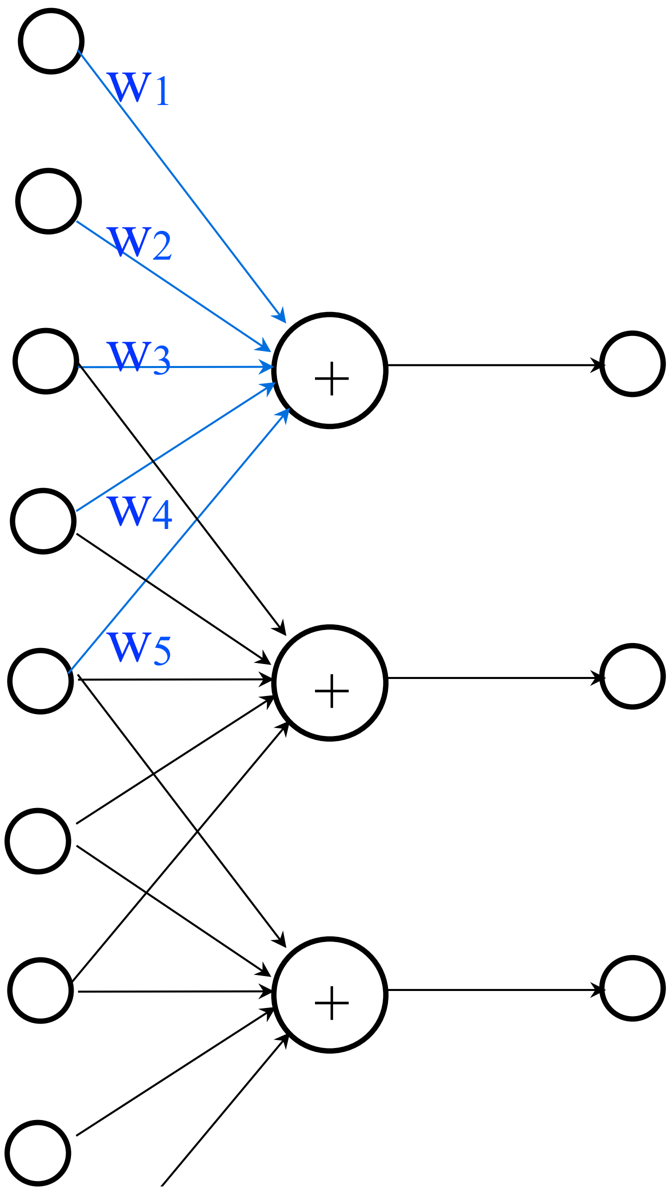
Lecture 6

Learned feedforward visual processing (Deep learning)

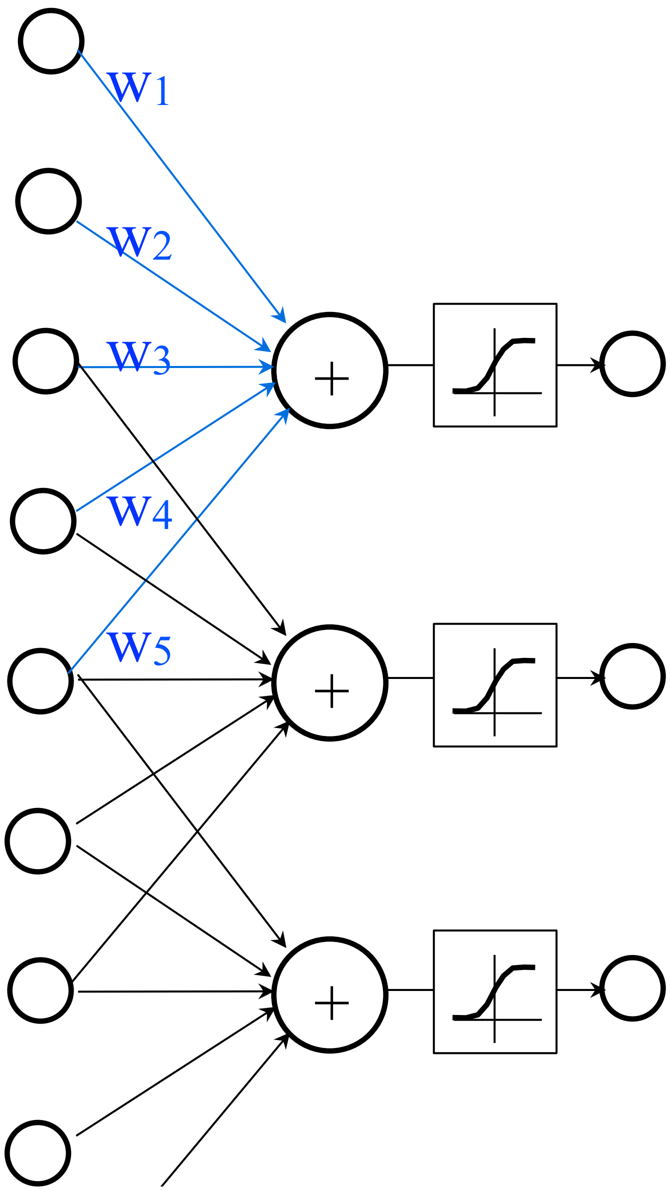
What is the best representation?

- All the previous representation are manually constructed.
- Could they be learnt from data?

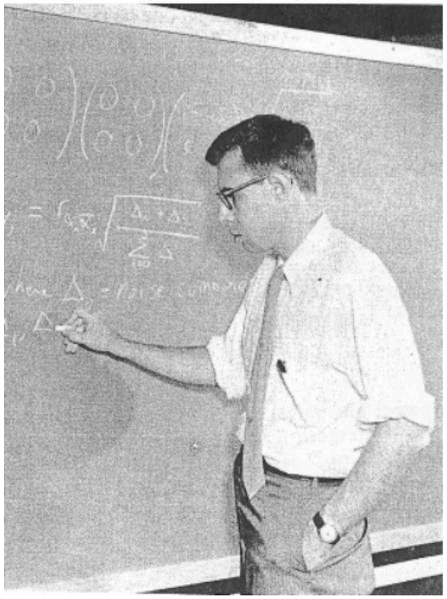
Linear filtering pyramid architecture



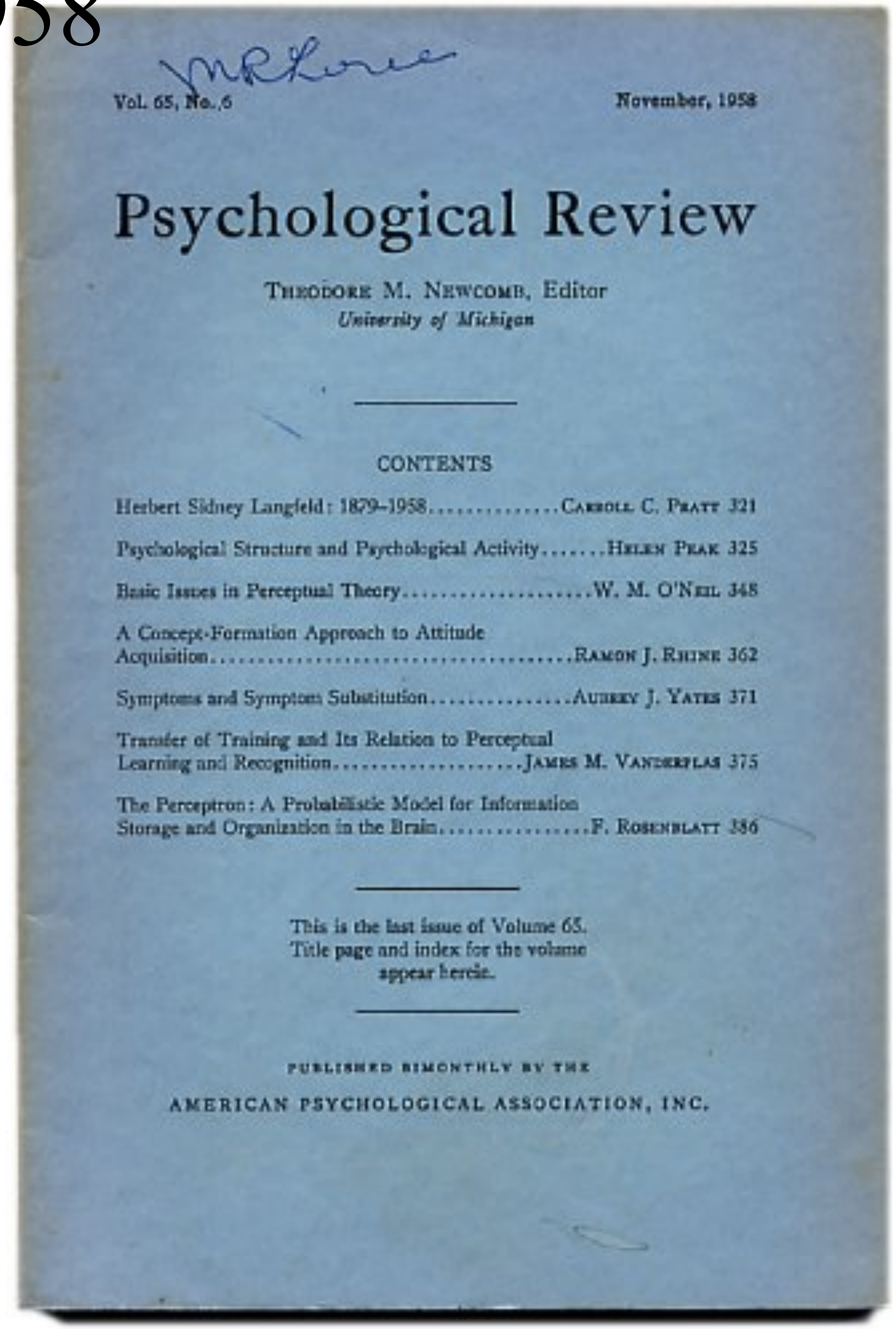
Convolutional neural network architecture



Perceptrons, 1958



http://www.ecse.rpi.edu/homepages/nagy/PDF_chrono/2011_Nagy_Pace_FR.pdf. Photo by George Nagy



Vol. 65, No. 6 November, 1958

Psychological Review

THEODORE M. NEWCOMB, Editor
University of Michigan

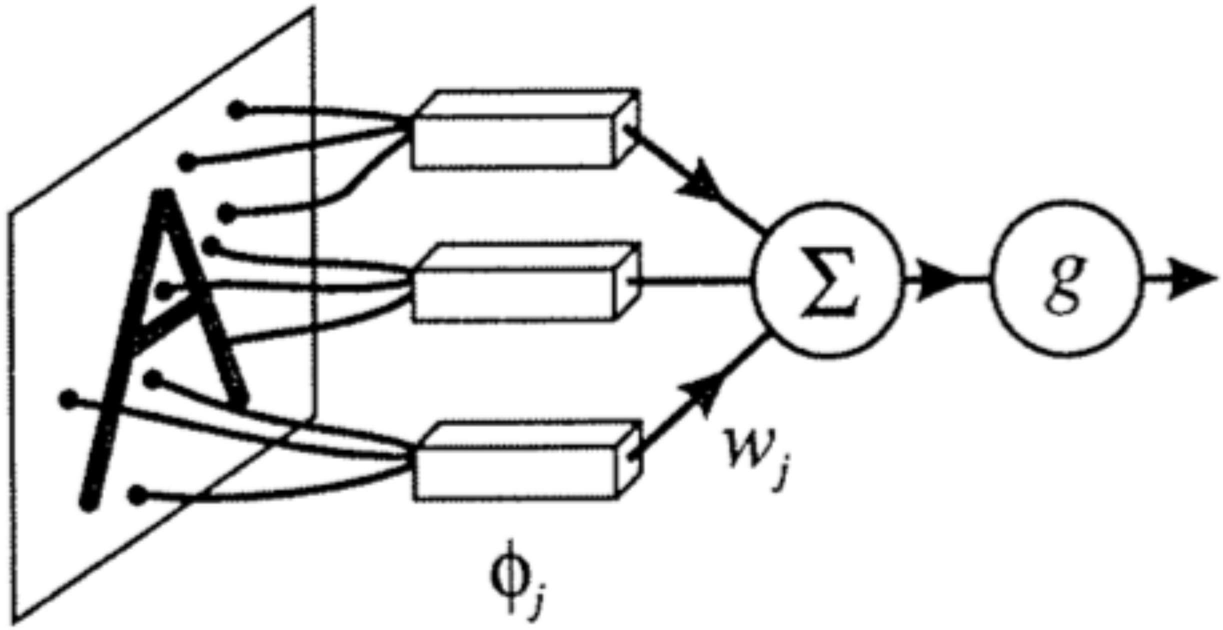
CONTENTS

This is the last issue of Volume 65. Title page and index for the volume appear herein.

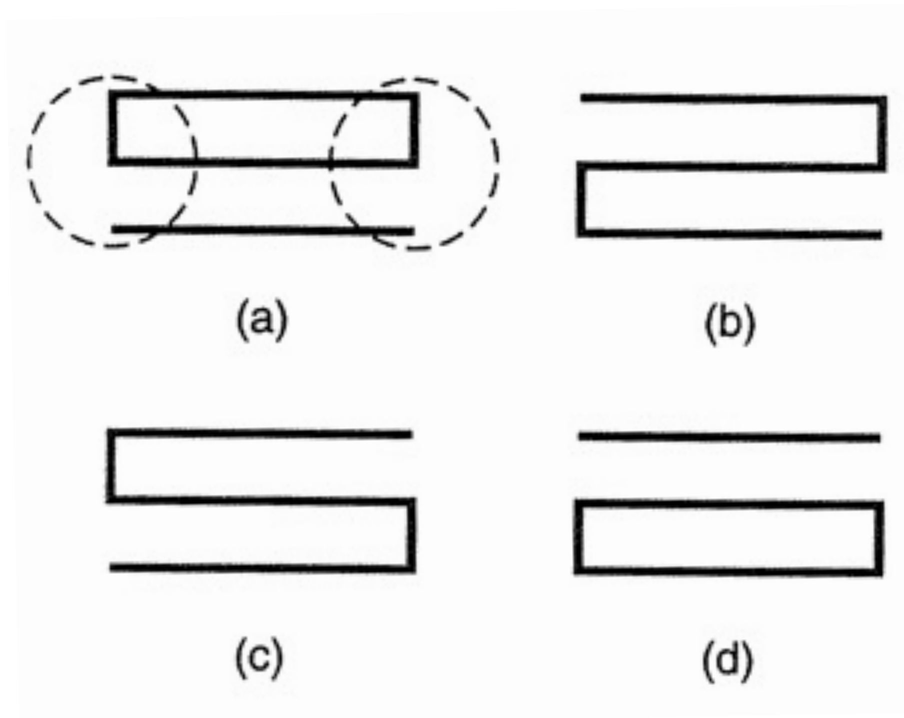
PUBLISHED BIMONTHLY BY THE
AMERICAN PSYCHOLOGICAL ASSOCIATION, INC.

http://www.manhattanrarebooks-science.com/rosenblatt.htm

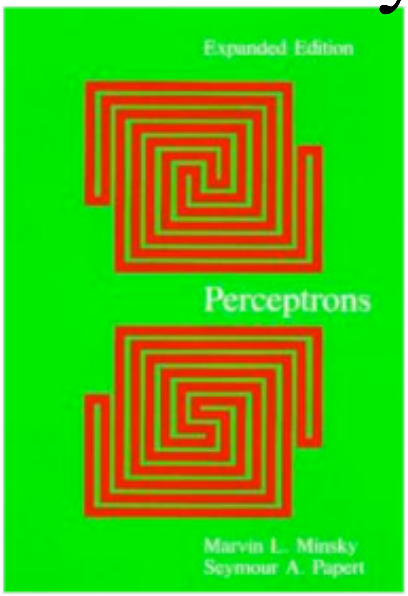
Perceptrons, 1958



Minsky and Papert pointed out many limitations of single-layer Perceptrons. Among them: very difficult to compute "connectedness"



Minsky and Papert, Perceptrons, 1972



Perceptrons, expanded edition

An Introduction to Computational Geometry

By Marvin Minsky and Seymour A. Papert

Overview

Perceptrons - the first systematic study of parallelism in computation - has remained a classical work on threshold automata networks for nearly two decades. It marked a historical turn in artificial intelligence, and it is required reading for anyone who wants to understand the connectionist counterrevolution that is going on today.

Artificial-intelligence research, which for a time concentrated on the programming of ton Neumann computers, is swinging back to the idea that intelligence might emerge from the activity of networks of neuronlike entities. Minsky and Papert's book was the first example of a mathematical analysis carried far enough to show the exact limitations of a class of computing machines that could seriously be considered as models of the brain. Now the new developments in mathematical tools, the recent interest of physicists in the theory of disordered matter, the new insights into and psychological models of how the brain works, and the evolution of fast computers that can simulate networks of automata have given *Perceptrons* new importance.

Witnessing the swing of the intellectual pendulum, Minsky and Papert have added a new chapter in which they discuss the current state of parallel computers, review developments since the appearance of the 1972 edition, and identify new research directions related to connectionism. They note a central theoretical challenge facing connectionism: the challenge to reach a deeper understanding of how "objects" or "agents" with individuality can emerge in a network. Progress in this area would link connectionism with what the authors have called "society theories of mind."

FOR BUYING OPTIONS, START HERE

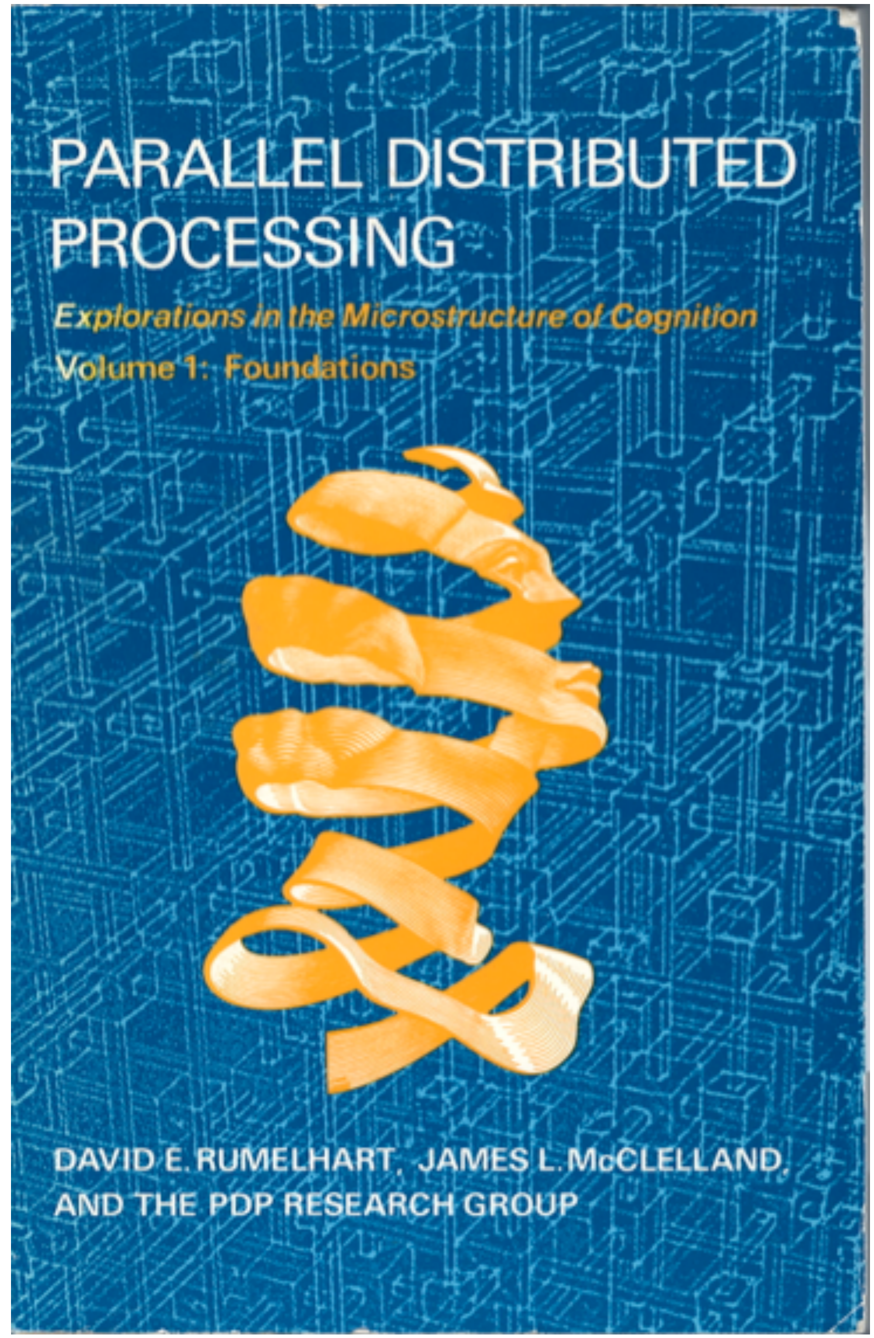
Select Shipping Destination

Paperback | **$35.00 Short** | **£24.95** | ISBN: 9780262631112 | 308 pp. | 6 x 8.9 in | December 1987

Perceptrons

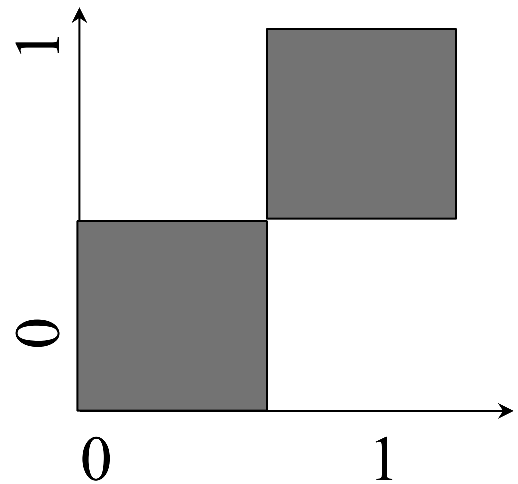
Minsky and Papert

Parallel Distributed Processing (PDP), 1986



XOR problem

Inputs		Output
0	0	0
1	0	1
0	1	1
1	1	0



PDP authors pointed to the backpropagation algorithm as a breakthrough, allowing multi-layer neural networks to be trained. Among the functions that a multi-layer network can represent but a single-layer network cannot: the XOR function.

LeCun conv nets, 1998

PROC. OF THE IEEE, NOVEMBER 1998 7

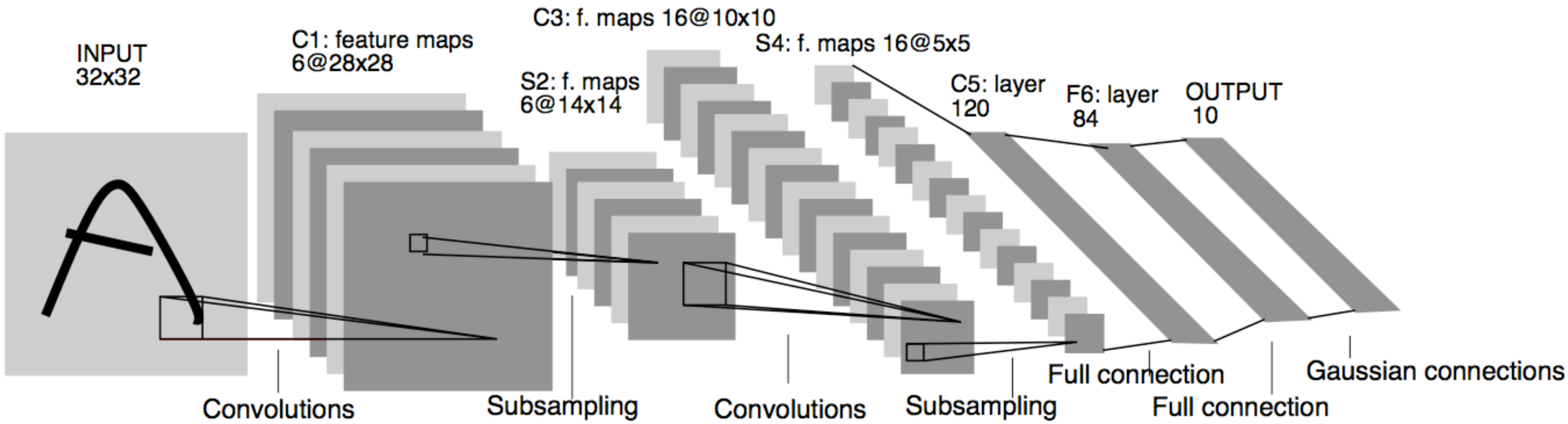


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Demos:

http://yann.lecun.com/exdb/lenet/index.html

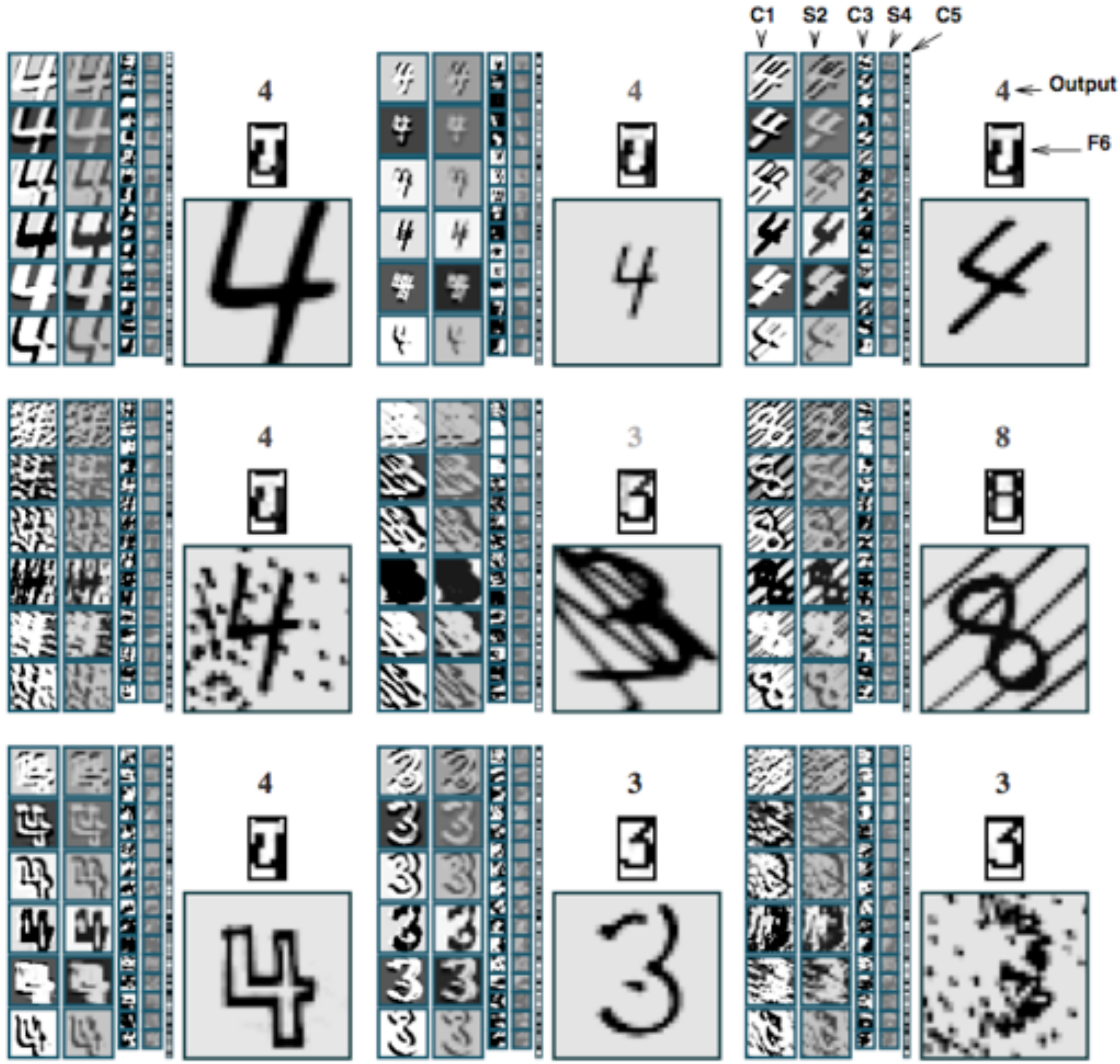
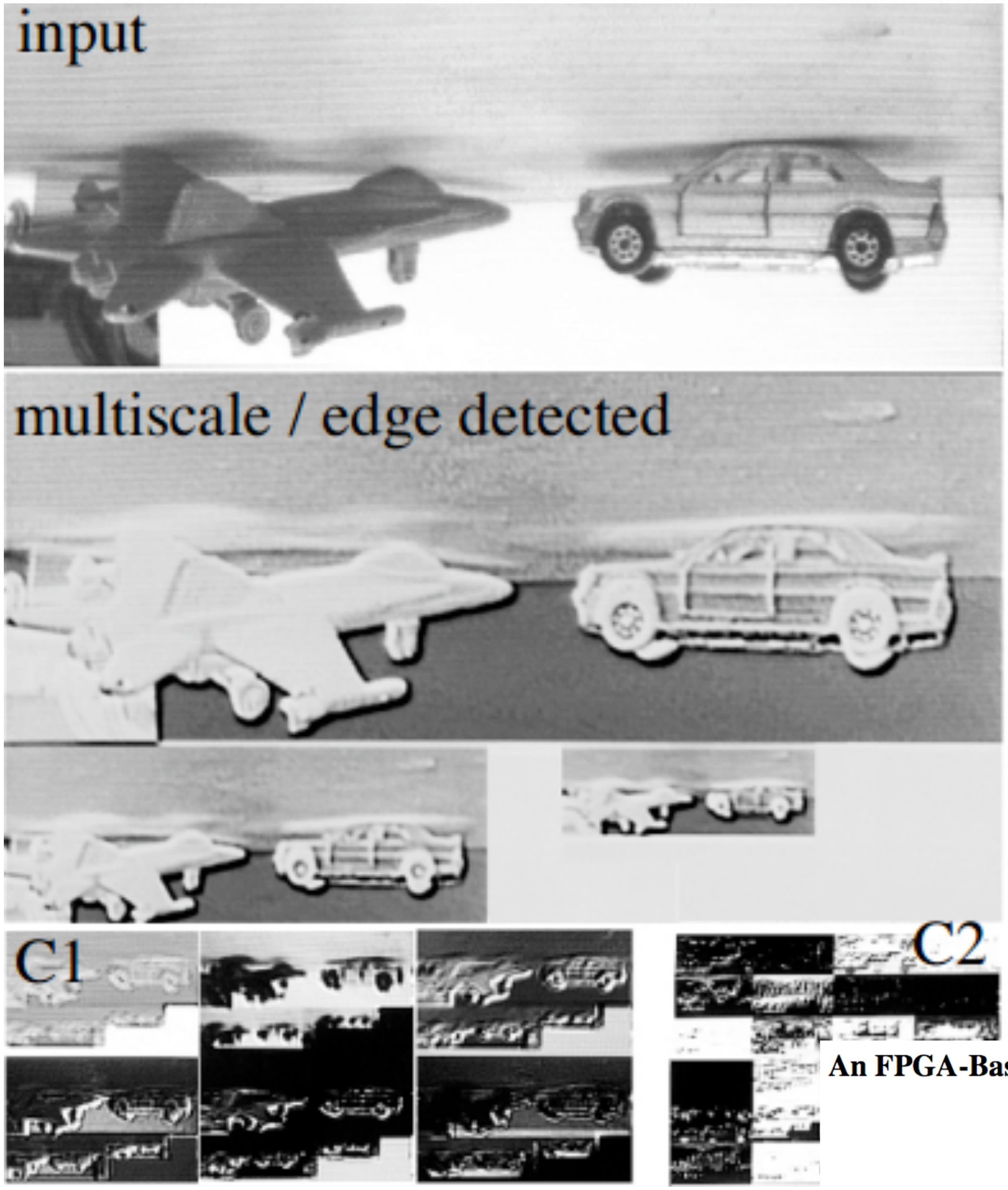


Fig. 13. Examples of unusual, distorted, and noisy characters correctly recognized by LeNet-5. The grey-level of the output label represents the penalty (lighter for higher penalties).



Neural networks to recognize handwritten digits? yes

Neural networks for tougher problems? not really

C Farabet, C Poulet, Y LeCun

Computer Vision Workshops

(ICCV Workshops), 2009 IEEE

12th International ..

An FPGA-Based Stream Processor for Embedded Real-Time Vision
Convolutional Networks

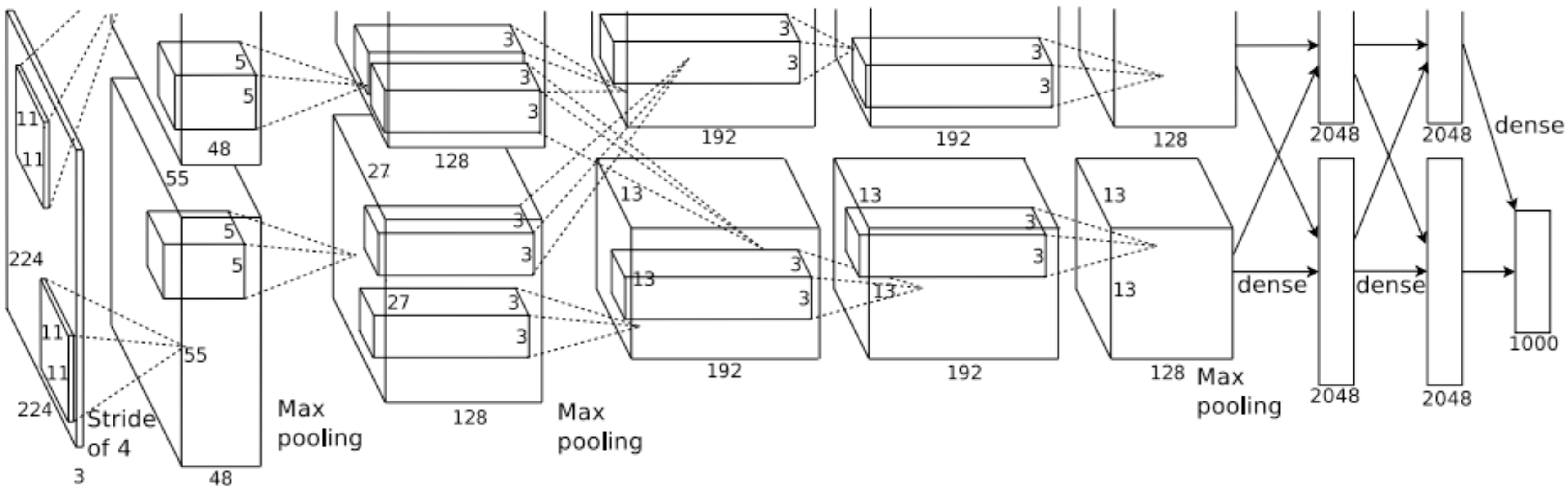
Clément Farabet, Cyril Poulet and Yann LeCun
Courant Institute of Mathematical Sciences, New York University
{cfarabet,yann}@cs.nyu.edu
http://www.cs.nyu.edu/~yann

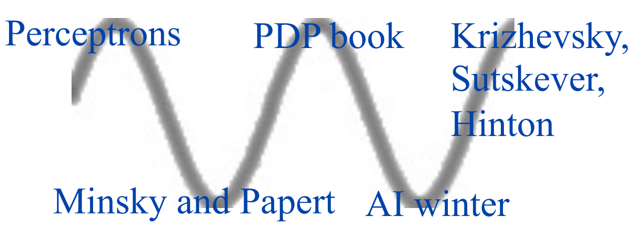
http://pub.clement.farabet.net/ecvw09.pdf

NIPS 2000

- NIPS, Neural Information Processing Systems, is the premier conference on machine learning. Evolved from an interdisciplinary conference to a machine learning conference.
- For the NIPS 2000 conference:
  - title words predictive of paper acceptance: “Belief Propagation” and “Gaussian”.
  - title words predictive of paper rejection:

Krizhevsky, Sutskever, and Hinton, NIPS 2012

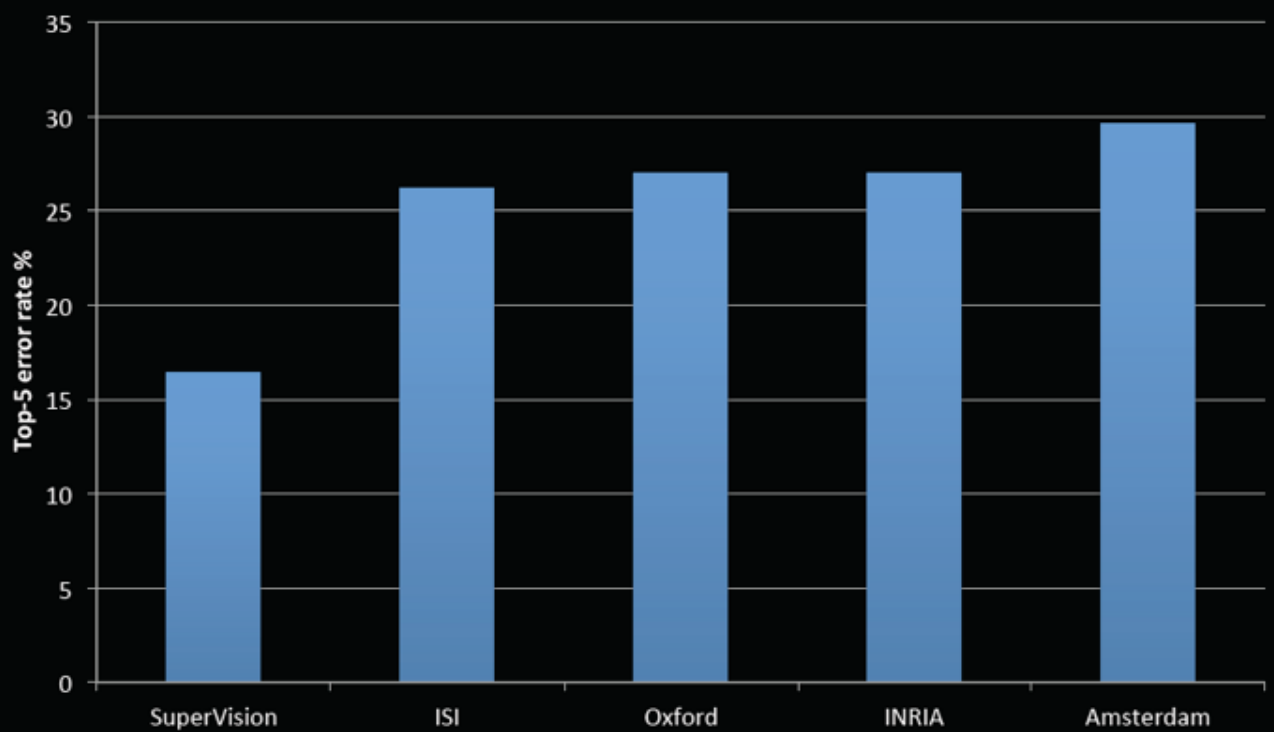




Slide from Rob Fergus, NYU

ImageNet Classification 2012

- Krizhevsky et al. -- 16.4% error (top-5)
- Next best (non-convnet) – 26.2% error

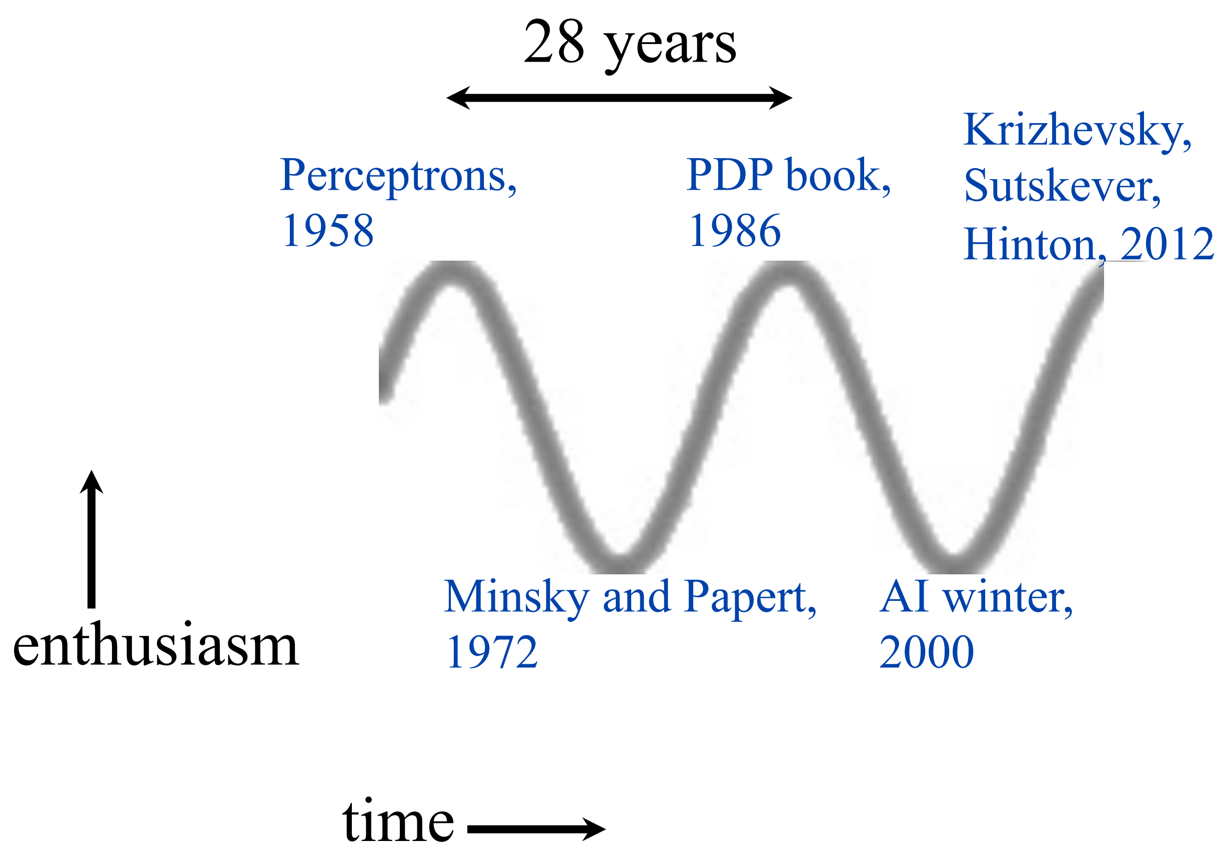


Krizhevsky, Sutskever, and Hinton, NIPS 2012

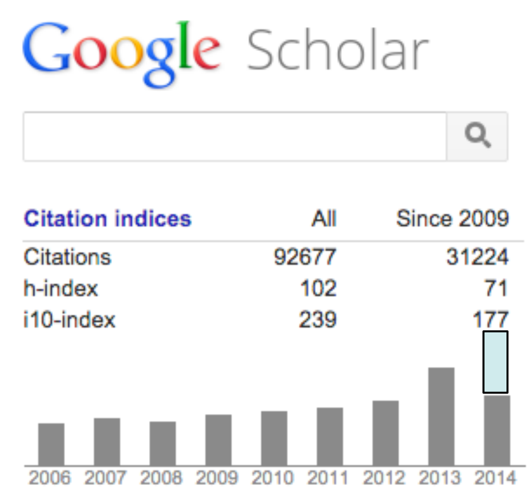
Test Nearby images, according to NN features

Krizhevsky, Sutskever, and Hinton, NIPS 2012

Research enthusiasm for artificial neural networks

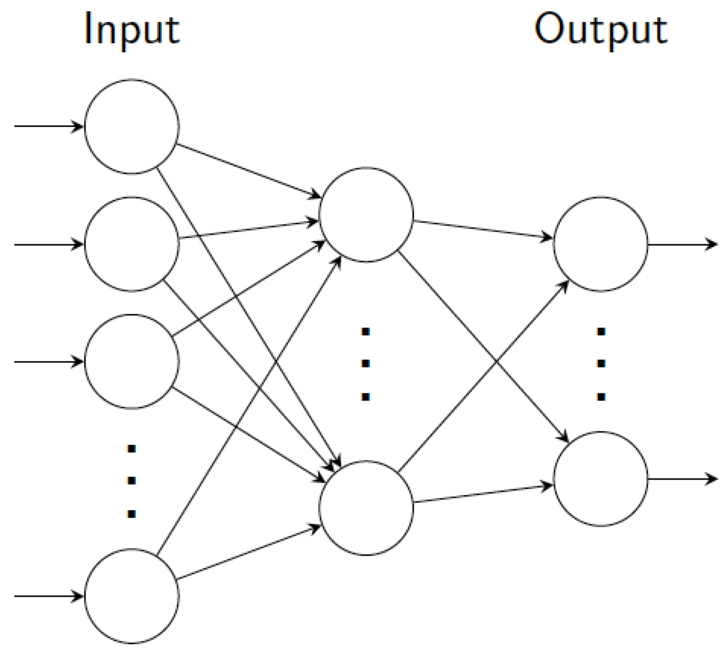


Geoff Hinton's citations



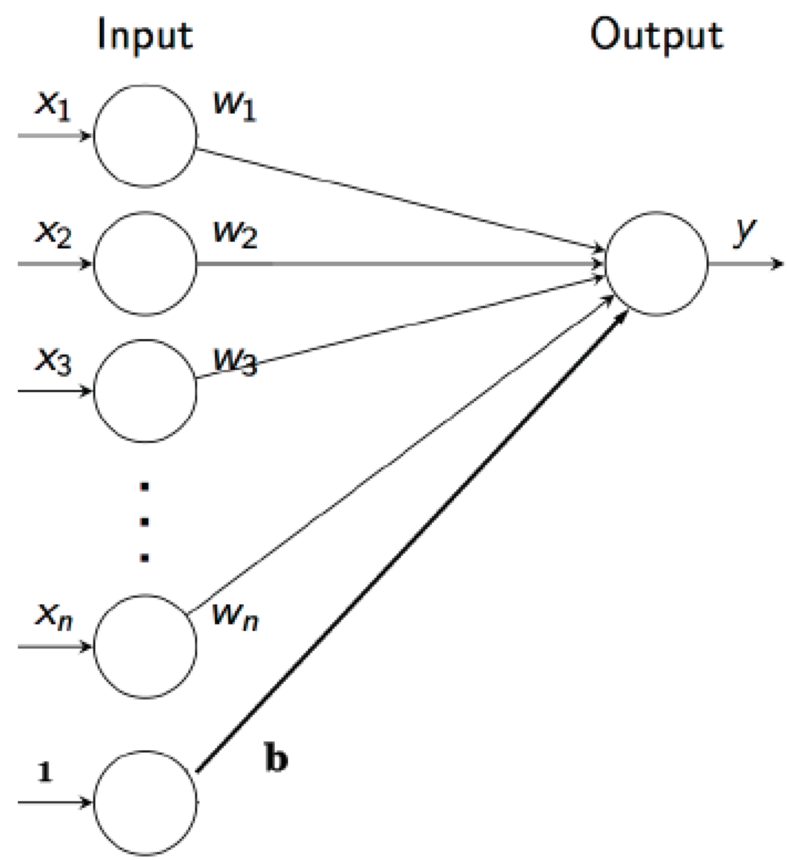
Neural networks

• Neural nets composed of layers of artificial neurons.
• Each layer computes some function of layer beneath.
• Inputs mapped in feed-forward fashion to output.
• Consider only feed-forward neural models at the moment, i.e. no cycles



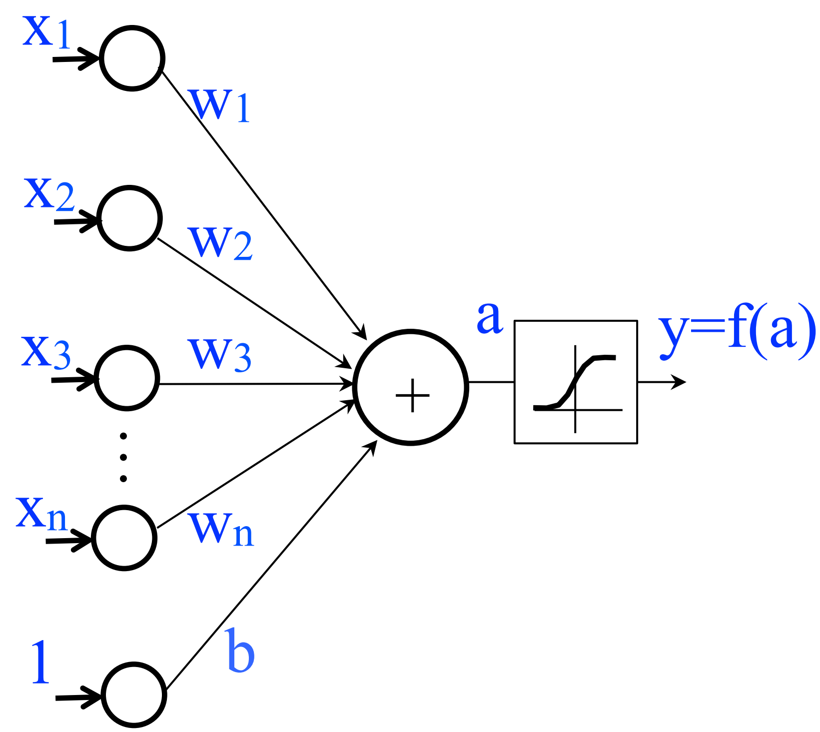
An individual neuron

- Input: x (n×1 vector)
- Parameters: weights w (n×1 vector), bias b (scalar)
- Activation: a = **Pn** i=1 x iwi + b .
- Note a is a scalar.
- Multiplicative interaction
- between weights and input.
- Point-wise non-linear function:
- **(:), e.g. (:) = tanh (:).**
- Output:
- y = f (a) = (
- **Pn**
- i=1 xiwi + b)
- Can think of bias as weight w0 ,
- connected to constant input 1:
- y = f (~ wT [1**; x]).**



An individual neuron (unit)

- Input: vector x (size n×1)
- Unit parameters: vector w (size n×1)
  bias b (scalar)
- Unit activation: $a = \sum_{i=1}^{n} x_i w_i + b$
- Output: $y = f(a) = f\left(\sum_{i=1}^{n} x_i w_i + b\right)$



f(.) is a point-wise non-linear function. E.g.:

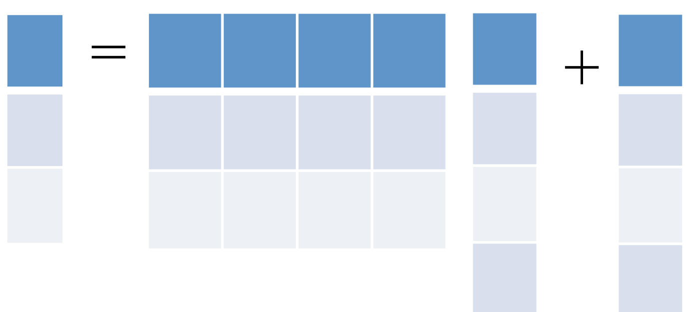
$$f(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$

Can think of bias as weight w_0, connected to constant input 1: $y = f([w_0, w]^T [1; x])$.

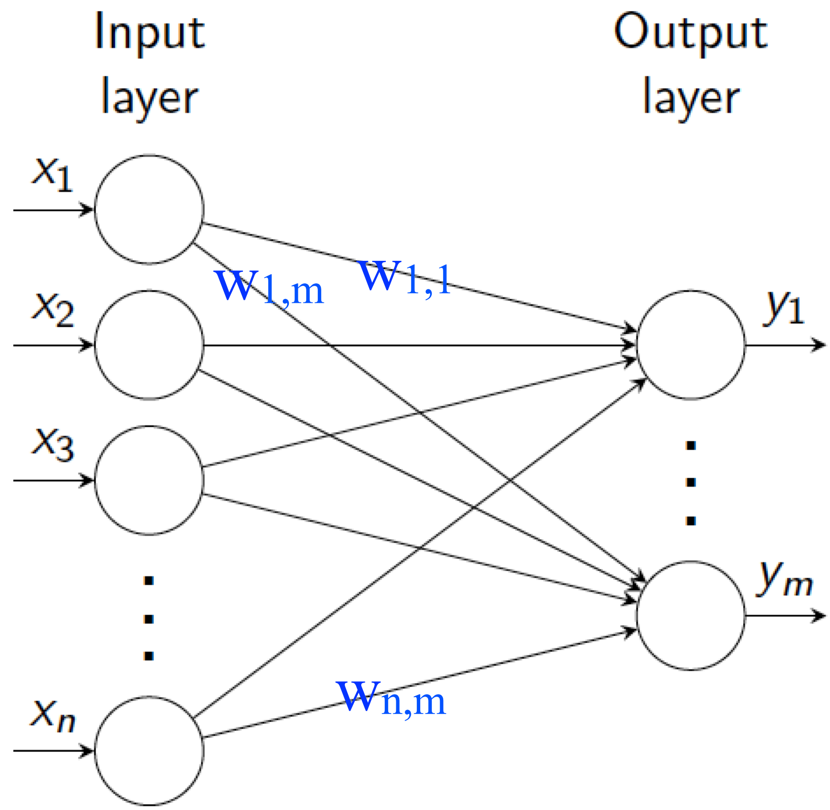
Single layer network

- Input: column vector x (size n×1)
- Output: column vector y (size m×1)
- Layer parameters:
  - weight matrix W (size n×m)
  - bias vector b (m×1)
- Units activation: $a = Wx + b$

  ex. 4 inputs, 3 outputs

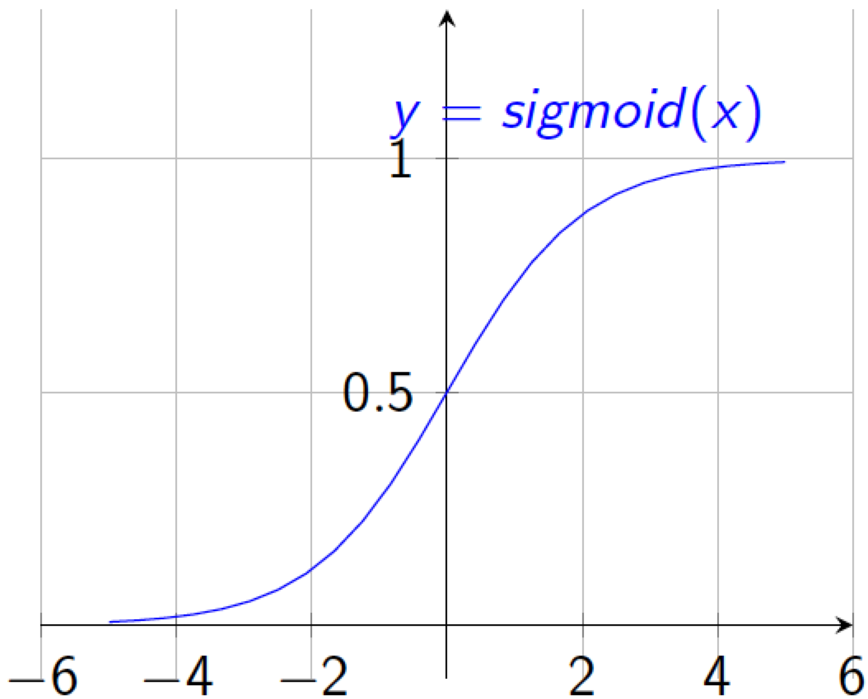


- Output: $y = f(a) = f\left(Wx + b\right)$



Non-linearities: sigmoid

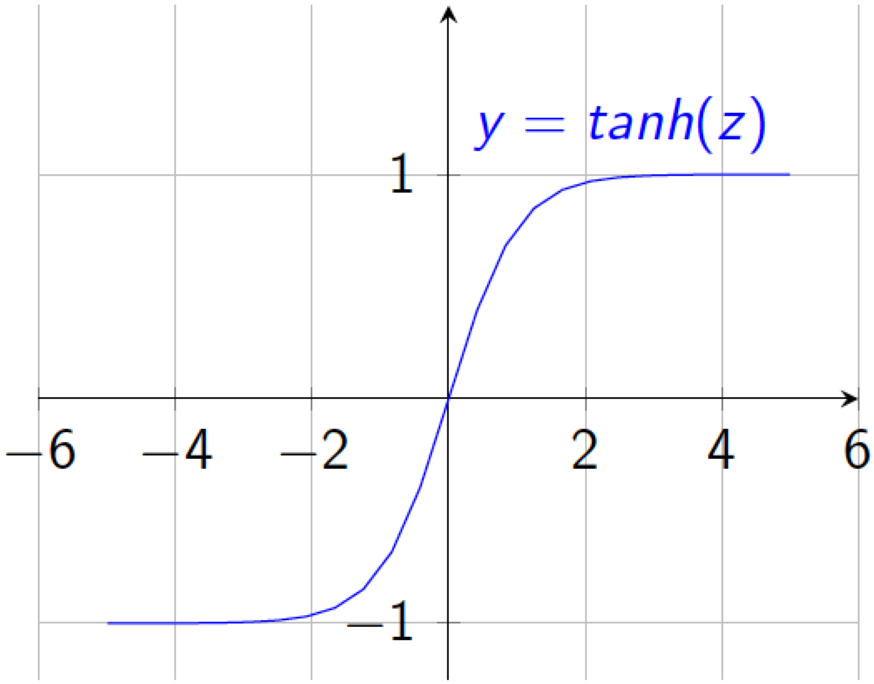
$$f(a) = sigmoid(a) = \frac{1}{1+e^{-a}}$$



- Interpretation as ring rate of neuron
- Bounded between [0,1]
- Saturation for large +ve,-ve inputs
- Gradients go to zero
- Outputs centered at 0.5 (poor conditioning)
- Not used in practice

Non-linearities: tanh

$$f(a) = \tanh(a) = \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}}$$

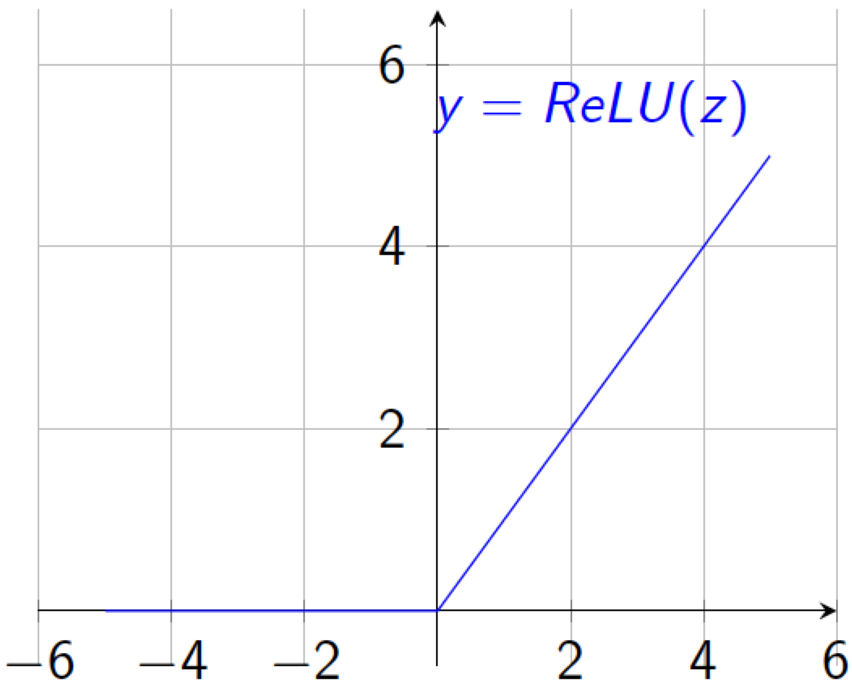


- Bounded between [-1,+1]
- Saturation for large +ve,-ve inputs
- Gradients go to zero
- Outputs centered at 0
- Preferable to sigmoid

$$\tanh(x) = 2\ \mathrm{sigmoid}(2x) - 1$$

Non-linearities: rectified linear (ReLU)

$$f(a) = \max(a, 0)$$



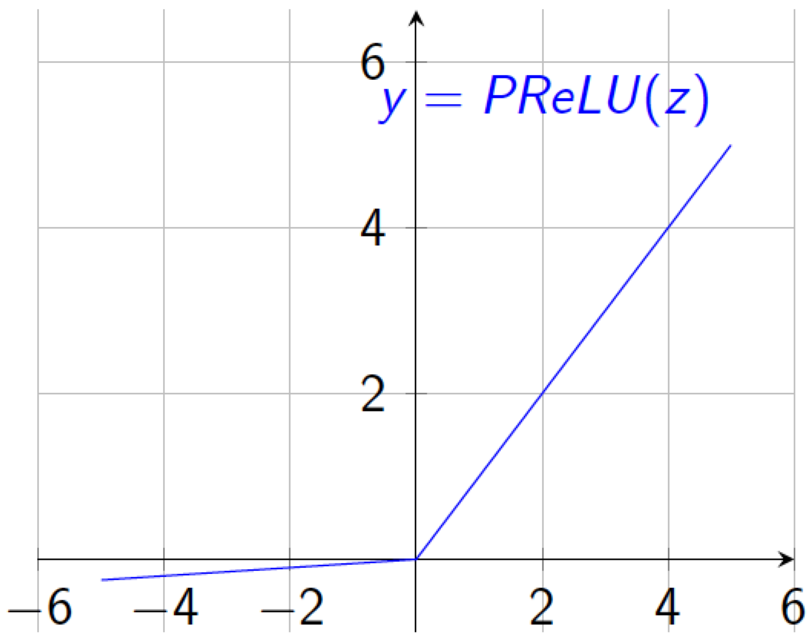
- Unbounded output (on positive side)
- Efficient to implement:

$$f'(a) = \frac{df}{da} = \begin{cases} 0 & a < 0 \\ 1 & a \geq 0 \end{cases}$$

- Also seems to help convergence (see 6x speedup vs tanh in Krizhevsky et al.)
- Drawback: if strongly in negative region, unit is dead forever (no gradient).
- Default choice: widely used in current models.

Non-linearities: Leaky ReLU

$$f(a) = \begin{cases} \max(0,a) & a > 0 \\ \alpha \min(0,a) & a < 0 \end{cases}$$



- where α is small (e.g. 0.02)
- Efficient to implement:

$$f'(a) = \frac{df}{da} = \begin{cases} -\alpha & a < 0 \\ 1 & a > 0 \end{cases}$$

- Also known as probabilistic ReLU (PReLU)
- Has non-zero gradients everywhere (unlike ReLU)
- α can also be learned (see Kaiming He et al. 2015).

Multiple layers

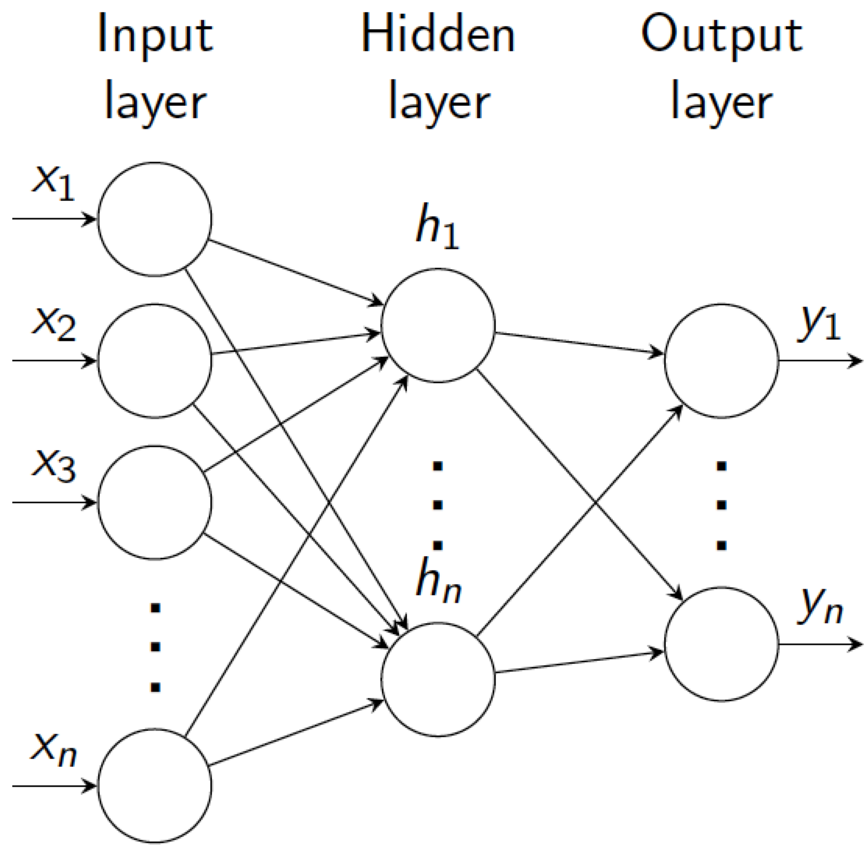
- Neural networks are composed of multiple layers of neurons.

- Acyclic structure. Basic model assumes full connections between layers.

- Layers between input and output are called hidden.

- Various names used:
    - Articial Neural Nets (ANN)
    - Multi-layer Perceptron (MLP)
    - Fully-connected network

- Neurons typically called units.

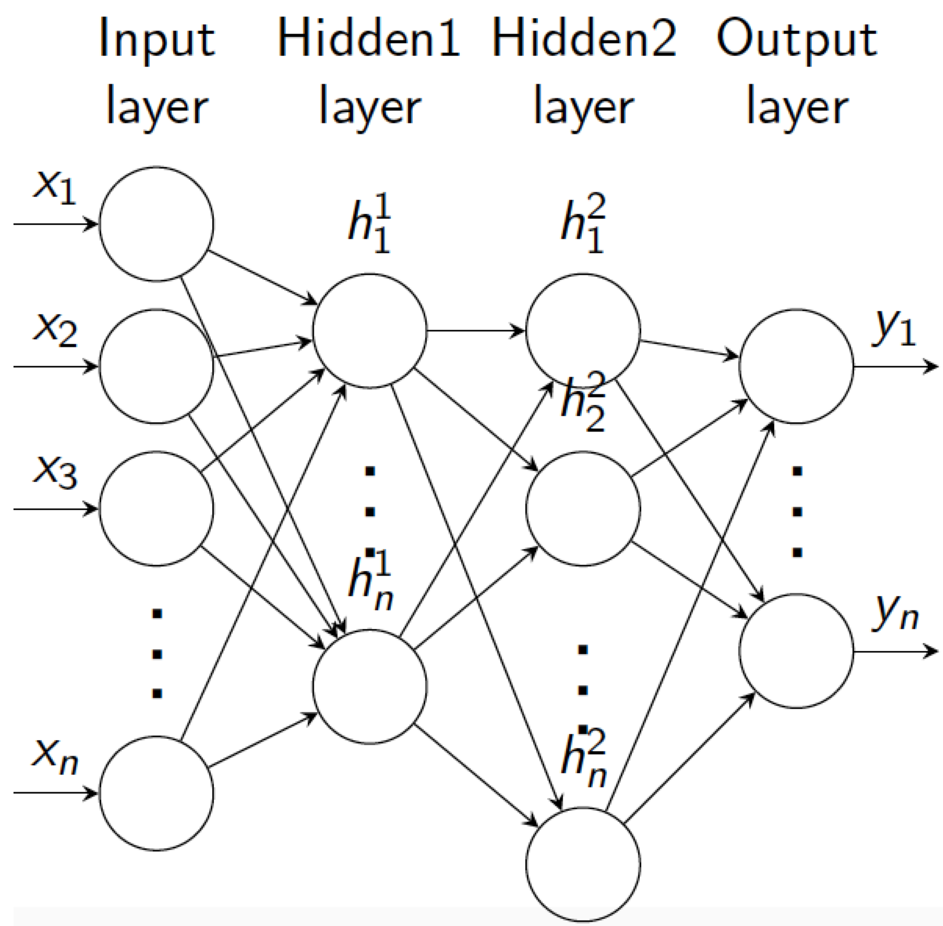


Example: 3 layer MLP

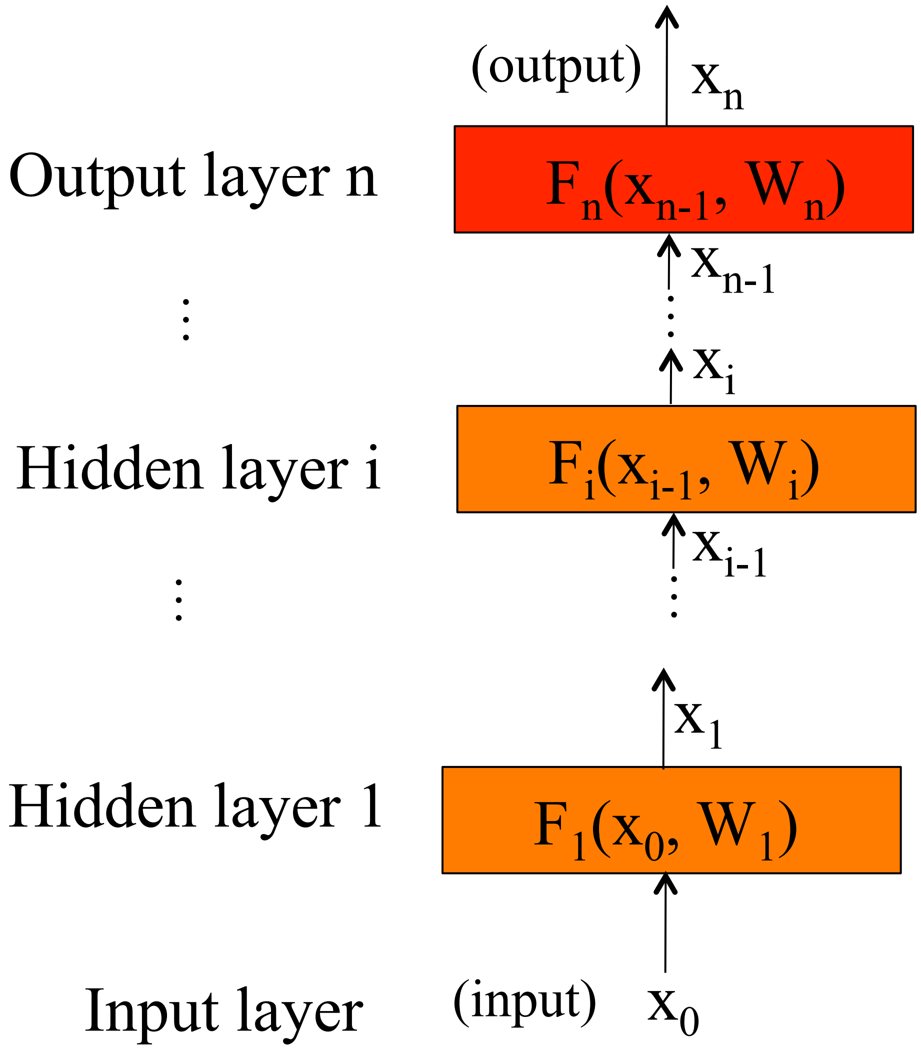
- By convention, number of layers is hidden + output (i.e. does not include input).

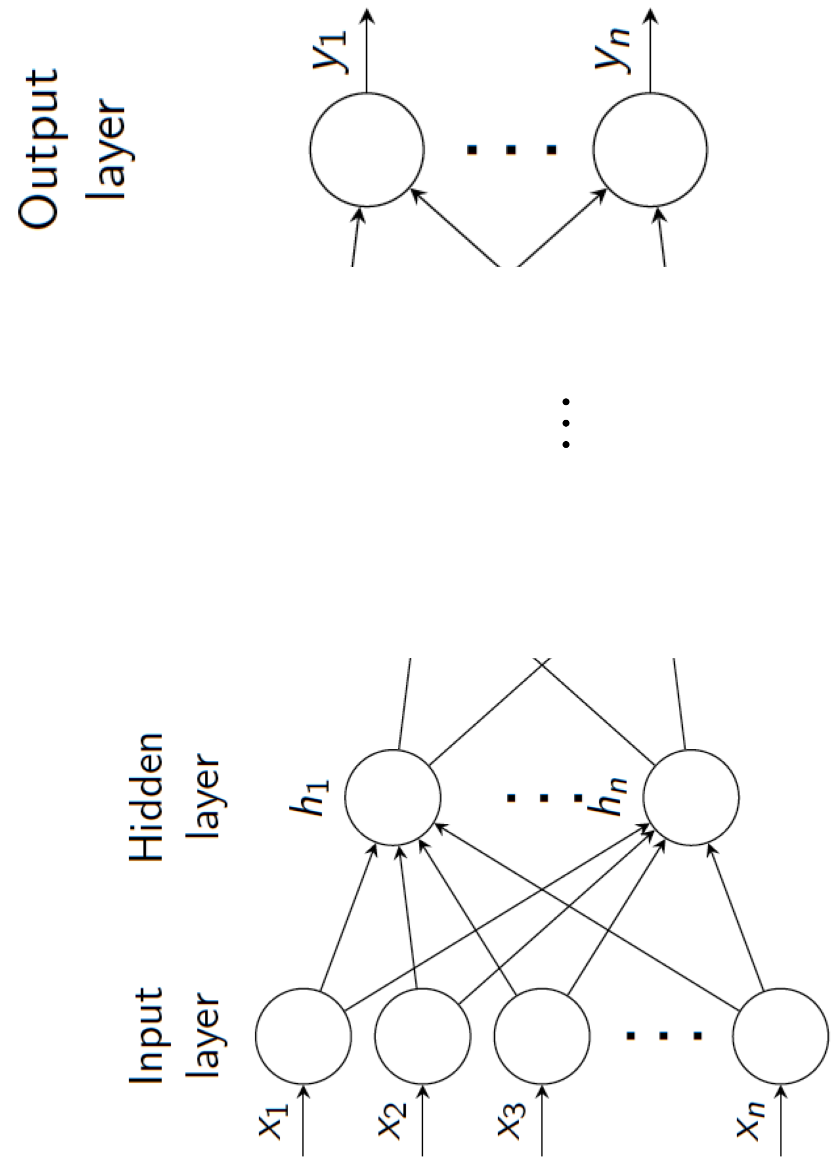
- So 3-layer model has 2 hidden layers.

- Parameters:
  weight matrices $W_1;W_2;W_3$
  bias vectors $b_1; b_2; b_3$.



Multiple layers





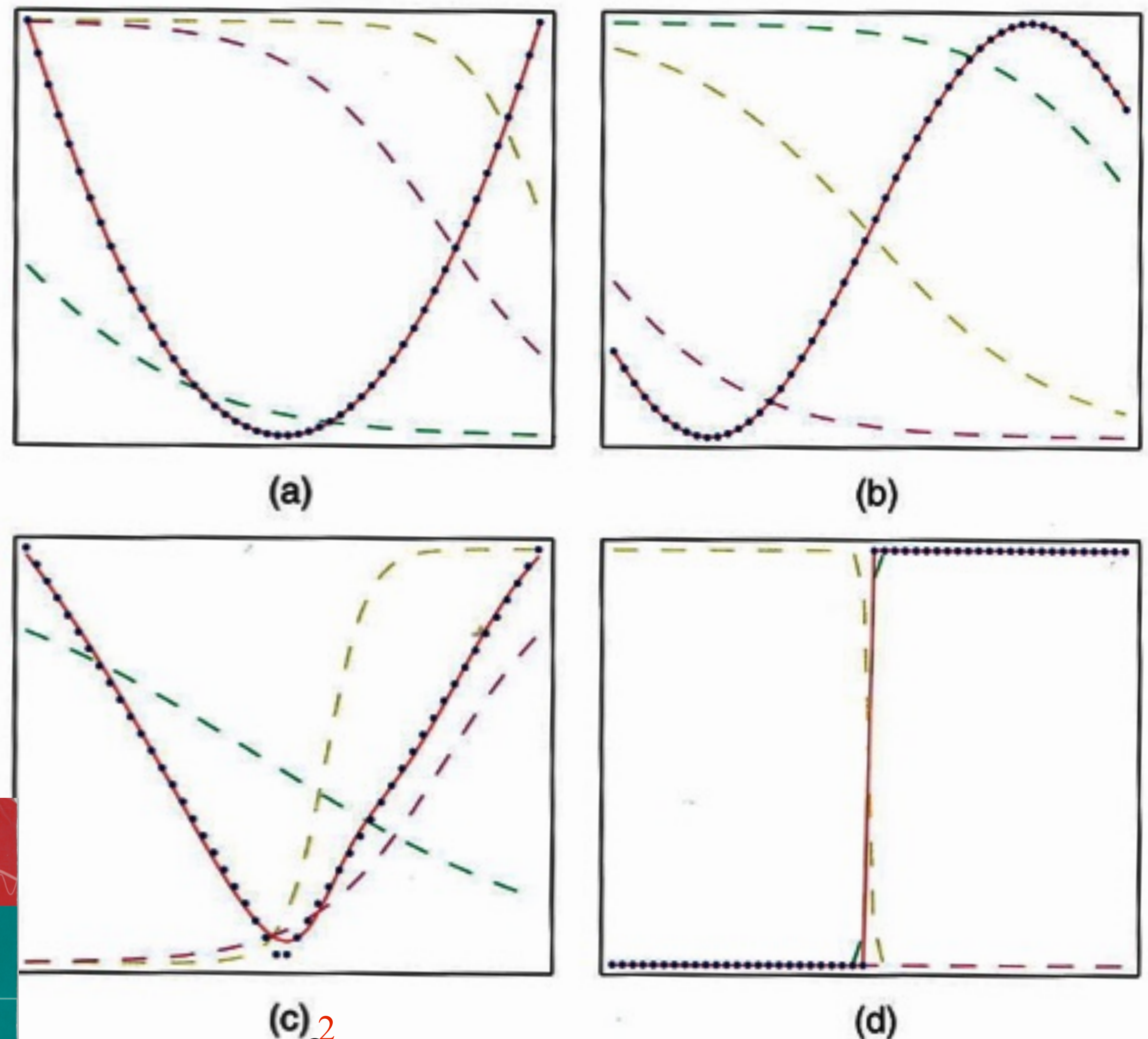
Architecture selection

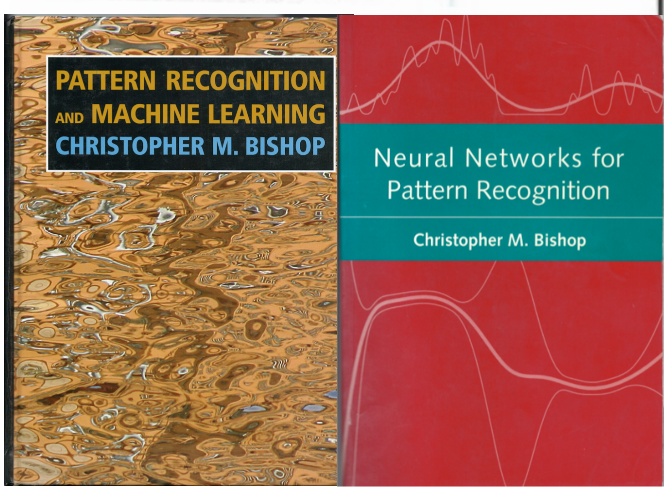
How to pick number of layers and units/layer?

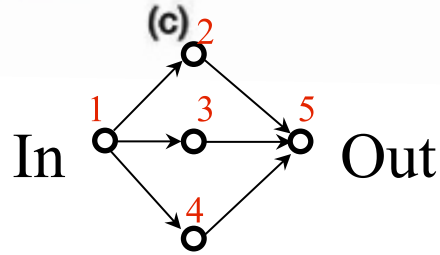
- Active area of research
- For fully connected models 2 or 3 layers seems the most that can be effectively trained (more later).
- Regarding number of units/layer:
  - Parameters grows with $(\text{units/layer})^2$.
  - With large units/layer, can easily overt.

Representational power of two-layer network

Figure 5.3 Illustration of the capability of a multilayer perceptron to approximate four different functions comprising (a) $f(x) = x^2$, (b) $f(x) = \sin(x)$, (c), $f(x) = |x|$, and (d) $f(x) = H(x)$ where $H(x)$ is the Heaviside step function. In each case, $N = 50$ data points, shown as blue dots, have been sampled uniformly in x over the interval $(-1, 1)$ and the corresponding values of $f(x)$ evaluated. These data points are then used to train a two-layer network having 3 hidden units with '$\tanh$' activation functions and linear output units. The resulting network functions are shown by the red curves, and the outputs of the three hidden units are shown by the three dashed curves.







$$z_5 = \sum_{i=2}^{i=4} w_{i5} \tanh(w_{1i} z_1 + \underbrace{w_{0i}}_{\text{bias}})$$

Representational power

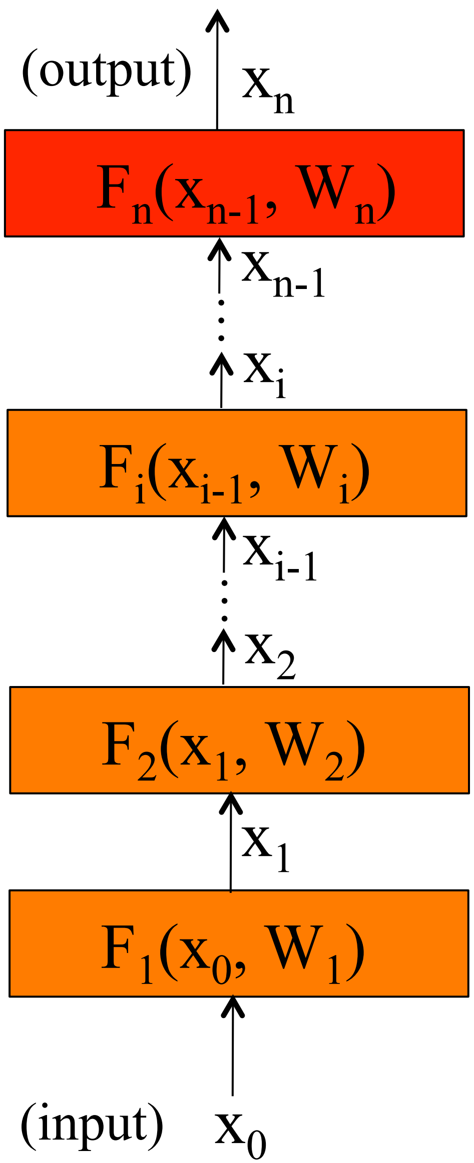
- 1 layer? Linear decision surface.
- 2+ layers? In theory, can represent any function. Assuming non-trivial non-linearity.
  - Bengio 2009,
    http://www.iro.umontreal.ca/~bengioy/papers/ftml.pdf
  - Bengio, Courville, Goodfellow book
    http://www.deeplearningbook.org/contents/mlp.html
  - Simple proof by M. Neilsen
    http://neuralnetworksanddeeplearning.com/chap4.html
  - D. Mackay book
    http://www.inference.phy.cam.ac.uk/mackay/itprnn/ps/482.491.pdf
- But issue is efficiency: very wide two layers vs narrow deep model? In practice, more layers helps.

Training a model: overview

- Given dataset {x; y}, pick appropriate cost function C.
- Forward-pass (f-prop) training examples through the model to get network output.
- Get error using cost function C to compare output to targets y
- Use Stochastic Gradient Descent (SGD) to update weights adjusting parameters to minimize loss/energy E (sum of the costs for each training example)

Cost function

- Consider model with N layers. Layer i has vector of weights Wi.
- Forward pass: takes input x and passes it through each layer F_i:
  $x_i = F_i\ (x_{i-1}, W_i)$
- Output of layer i is x_i. Network output (top layer) is x_n.

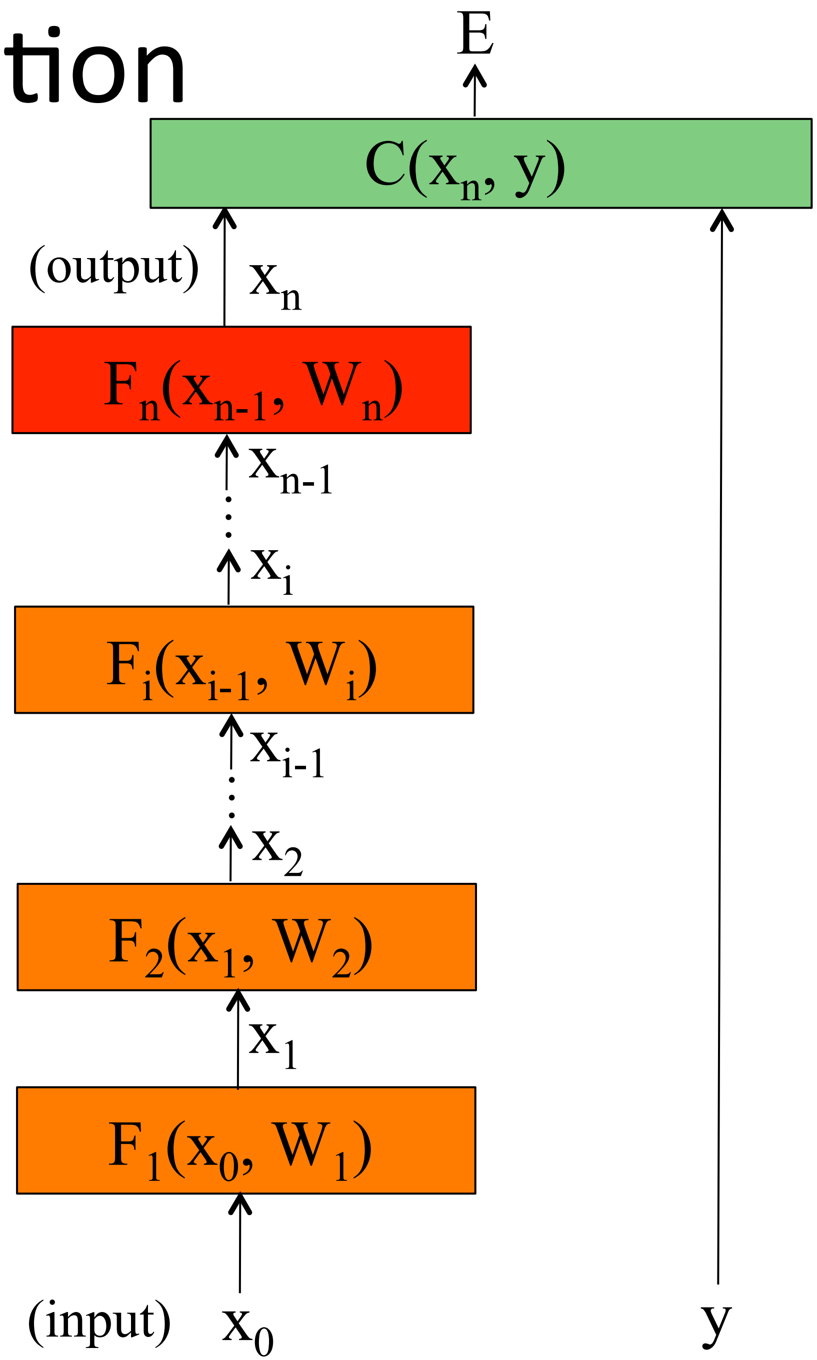


Cost function

- Consider model with N layers. Layer i has vector of weights Wi.
- Forward pass: takes input x and passes it through each layer F_i:

  $x_i = F_i\ (x_{i-1}, W_i)$
- Output of layer i is x_i. Network output (top layer) is x_n.
- Cost function C compares x_n to y
- Overall energy is the sum of the cost over all training examples:

$$E = \sum_{m=1}^{M} C\left(x_n^m, y^m\right)$$



Stochastic gradient descend

- Want to minimize overall loss function **E.** Loss is sum of individual losses over each example.
- In gradient descent, we start with some initial set of parameters θ
- Update parameters: $\theta^{k+1} \leftarrow \theta^k + \eta \nabla \theta$

  k is iteration index, η is learning rate (scalar; set semi-manually).
- Gradients $\nabla \theta = \frac{\partial E}{\partial \theta}$ computed by b-prop.
- In Stochastic gradient descent, compute gradient on sub-set (batch) of data.

  If batchsize=1 then θ is updated after each example.

  If batchsize=N (full set) then this is standard gradient descent.
- Gradient direction is noisy, relative to average over all examples (standard gradient descent).

Stochastic gradient descend

- We need to compute gradients of the cost with respect to model parameters w_i

- Back-propagation is essentially chain rule of derivatives back through the model.

- Each layer is differentiable with respect to parameters and input.

Computing gradients

- Training will be an iterative procedure, and at each iteration we will update the network parameters $\theta^{k+1} \leftarrow \theta^k + \eta \nabla \theta$

- We want to compute the gradients

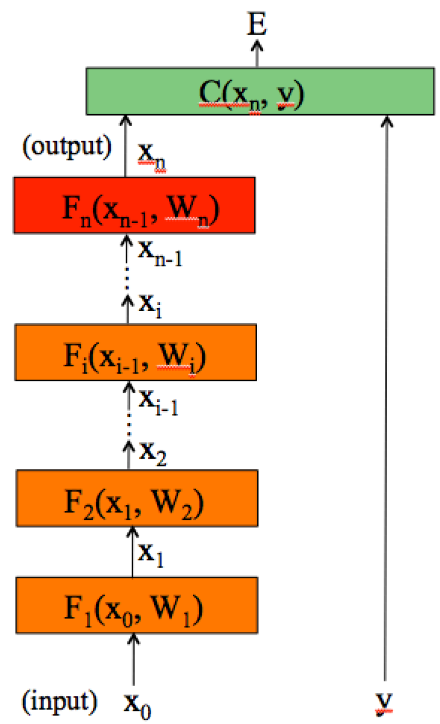
$$\nabla \theta = \frac{\partial E}{\partial \theta}$$

Where $\theta = \left\{ w_1, w_2, \ldots, w_n \right\}$

Computing gradients

To compute the gradients, we could start by wring the full energy E as a function of the network parameters.

$$E(\theta) = \sum_{m=1}^{M} C\left(F_n\left(F_{n-1}\left(F_2\left(F_1\left(x_0^m, w_1\right), w_2\right), w_{n-1}\right), w_n\right), y^m\right)$$



And then compute the partial derivatives… instead, we can use the chain rule to derive a compact algorithm: **back-propagation**

(

Matrix calculus

- x column vector of size [n×1]

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

- We now define a function on vector x: y = F(x)
- If y is a scalar, then

$$\partial y / \partial x = \begin{bmatrix} \partial y / \partial x_1 & \partial y / \partial x_2 & \cdots & \partial y / \partial x_n \end{bmatrix}$$

The derivative of y is a row vector of size [1×n]

- If y is a vector [1×m], then (*Jacobian formulation*):

$$\partial y / \partial x = \begin{bmatrix} \partial y_1 / \partial x_1 & \partial y_1 / \partial x_2 & \cdots & \partial y_1 / \partial x_n \\ \vdots & \vdots & & \vdots \\ \partial y_m / \partial x_1 & \partial y_m / \partial x_2 & \cdots & \partial y_n / \partial x_m \end{bmatrix}$$

The derivative of y is a matrix of size [m×n]

(m rows and n columns)

Matrix calculus

- Chain rule:

For the function: $z = h(x) = f(g(x))$

Its derivative is: $h'(x) = f'(g(x))\, g'(x)$

and writing $z=f(u)$, and $u=g(x)$:

$$\left.\frac{dz}{dx}\right|_{x=a} = \left.\frac{dz}{du}\right|_{u=g(a)} \cdot \left.\frac{du}{dx}\right|_{x=a}$$

$[m \times n]$ $\quad$ $[m \times p]$ $\quad$ $[p \times n]$

with p = length vector $u = |u|$, $m = |z|$, and $n = |x|$

Example, if $|z|=1$, $|u| = 2$, $|x|=4$

$h'(x) =$ 

Matrix calculus

- Chain rule:

For the function: $h(x) = f_n(f_{n-1}(\ldots(f_1(x))\,))$

With $u_1 = f_1(x)$
$u_i = f_i(u_{i-1})$
$z = u_n = f_n(u_{n-1})$

The derivative becomes a product of matrices:

$$\left.\frac{dz}{dx}\right|_{x=a} = \left.\frac{dz}{du_{n-1}}\right|_{u_{n-1}=f_{n-1}(u_{n-2})} \cdot \left.\frac{du_{n-1}}{du_{n-2}}\right|_{u_{n-2}=f_{n-2}(u_{n-3})} \cdot \ldots \cdot \left.\frac{du_2}{du_1}\right|_{u_1=f_1(a)} \cdot \left.\frac{du_1}{dx}\right|_{x=a}$$

(exercise: check that all the matrix dimensions work fine)

)

Computing gradients

The energy E is the sum of the costs associated to each training example x^m, y^m

$$E(\theta) = \sum_{m=1}^{M} C\left(x_n^m, y^m; \theta\right)$$

Its gradient with respect to the networks parameters is:

$$\frac{\partial E}{\partial \theta_i} = \sum_{m=1}^{M} \frac{C\left(x_n^m, y^m; \theta\right)}{\partial \theta_i}$$

is how much E varies when the parameter θ_i is varied.

Computing gradients

We could write the cost function to get the gradients:

$$C(x_n, y; \theta) = C(F_n(x_{n-1}, w_n), y)$$

with $\theta = [w_1, w_2, \cdots, w_n]$

If we compute the gradient with respect to the parameters of the last layer (output layer) w_n, using the chain rule:

$$\frac{\partial C}{\partial w_n} = \frac{\partial C}{\partial x_n} \cdot \frac{\partial x_n}{\partial w_n} = \frac{\partial C}{\partial x_n} \cdot \frac{\partial F_n(x_{n-1}, w_n)}{\partial w_n}$$

(how much the cost changes when we change wn, is the product between how much the cost changes when we change the output of the last layer, times how much the output changes when we change the layer parameters.)

Computing gradients: cost layer

If we compute the gradient with respect to the parameters of the last layer (output layer) w_n, using the chain rule:

$$\frac{\partial C}{\partial w_n} = \frac{\partial C}{\partial x_n} \cdot \frac{\partial x_n}{\partial w_n} = \frac{\partial C}{\partial x_n} \cdot \frac{\partial F_n\left(x_{n-1}, w_n\right)}{\partial w_n}$$

Will depend on the layer structure and non-linearity.

For example, for an Euclidean loss:

$$C(x_n, y) = \frac{1}{2}\left\| x_n - y \right\|^2$$

The gradient is:

$$\frac{\partial C}{\partial x_n} = x_n - y$$

Computing gradients: layer i

We could write the full cost function to get the gradients:

$$C(x_n, y; \theta) = C\Big(F_n\Big(F_{n-1}\Big(F_2\Big(F_1\big(x_0, w_1\big), w_2\Big), w_{n-1}\Big), w_n\Big), y\Big)$$

If we compute the gradient with respect to w_i, using the chain rule:

$$\frac{\partial C}{\partial w_i} = \underbrace{\frac{\partial C}{\partial x_n} \cdot \frac{\partial x_n}{\partial x_{n-1}} \cdot \frac{\partial x_{n-1}}{\partial x_{n-2}} \cdot \ldots \cdot \frac{\partial x_{i+1}}{\partial x_i}}_{\frac{\partial C}{\partial x_i}} \cdot \frac{\partial x_i}{\partial w_i}$$

$$\frac{\partial C}{\partial x_i}$$

And this can be computed iteratively!

$$\frac{\partial F_i(x_{i-1}, w_i)}{\partial w_i}$$

This is easy.

Backpropagation

$$\frac{\partial C}{\partial w_i} = \underbrace{\frac{\partial C}{\partial x_n} \cdot \frac{\partial x_n}{\partial x_{n-1}} \cdot \frac{\partial x_{n-1}}{\partial x_{n-2}} \cdot \ldots \cdot \frac{\partial x_{i+1}}{\partial x_i}}_{\frac{\partial C}{\partial x_i}} \cdot \frac{\partial x_i}{\partial w_i}$$

$$\frac{\partial F_i(x_{i-1}, w_i)}{\partial w_i}$$

If we have the value of $\frac{\partial C}{\partial x_i}$ we can compute the gradient at the layer bellow as:

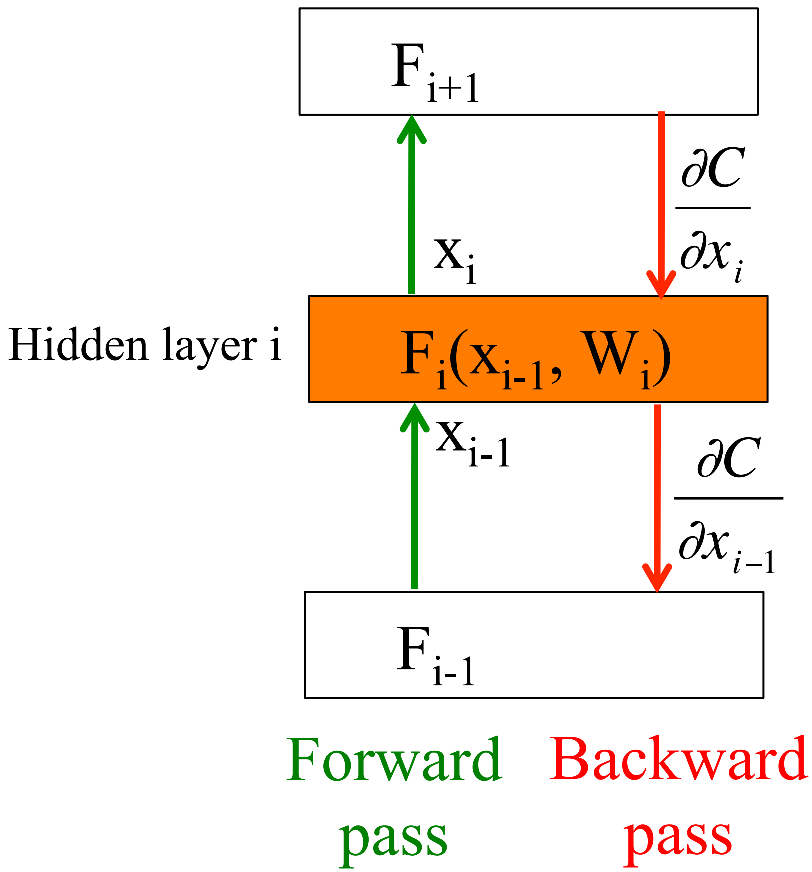
$$\frac{\partial C}{\partial x_{i-1}} = \frac{\partial C}{\partial x_i} \cdot \frac{\partial x_i}{\partial x_{i-1}}$$

Gradient layer i-1

Gradient layer i

$$\frac{\partial F_i(x_{i-1}, w_i)}{\partial x_{i-1}}$$

Backpropagation: layer i



- Layer i has two inputs (during training)

$$x_{i-1} \qquad \frac{\partial C}{\partial x_i}$$

- For layer i, we need the derivatives:

$$\frac{\partial F_i(x_{i-1}, w_i)}{\partial x_{i-1}} \qquad \frac{\partial F_i(x_{i-1}, w_i)}{\partial w_i}$$

- We compute the outputs

$$x_i = F_i(x_{i-1}, w_i)$$

$$\frac{\partial C}{\partial x_{i-1}} = \frac{\partial C}{\partial x_i} \cdot \frac{\partial F_i(x_{i-1}, w_i)}{\partial x_{i-1}}$$

- The weight update equation is:

$$\frac{\partial C}{\partial w_i} = \frac{\partial C}{\partial x_i} \cdot \frac{\partial F_i(x_{i-1}, w_i)}{\partial w_i}$$

$$w_i^{k+1} \leftarrow w_i^k + \eta_t \frac{\partial E}{\partial w_i}$$

(sum over all training examples to get E)

Backpropagation: summary

- Forward pass: for each training example. Compute the outputs for all layers

$$x_i = F_i(x_{i-1}, w_i)$$

- Backwards pass: compute cost derivatives iteratively from top to bottom:

$$\frac{\partial C}{\partial x_{i-1}} = \frac{\partial C}{\partial x_i} \cdot \frac{\partial F_i(x_{i-1}, w_i)}{\partial x_{i-1}}$$

- Compute gradients and update weights.