MIT CSAIL

6.869: Advances in Computer Vision

William T. Freeman, Antonio Torralba, 2017

MIT COMPUTER VISION

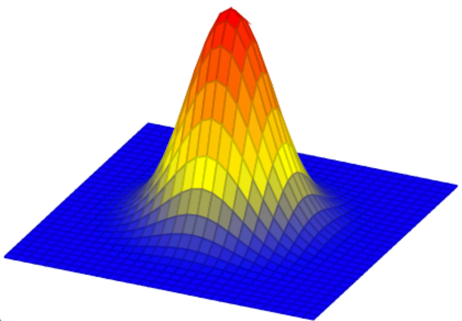
Lecture 6

Learned feedforward visual processing
Neural Networks, Deep learning, ConvNets

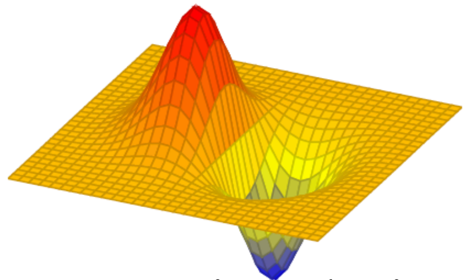
Some slides modified from R. Fergus

We need translation invariance

Lots of useful linear filters…

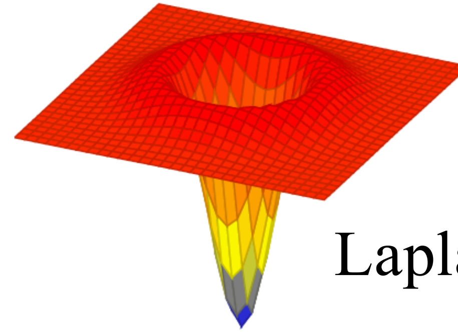


a) Gaussian



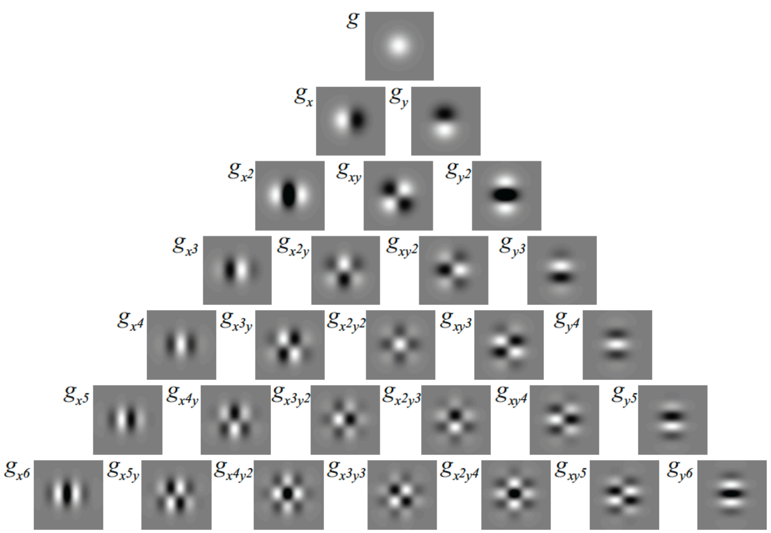
Gaussian derivative

b)

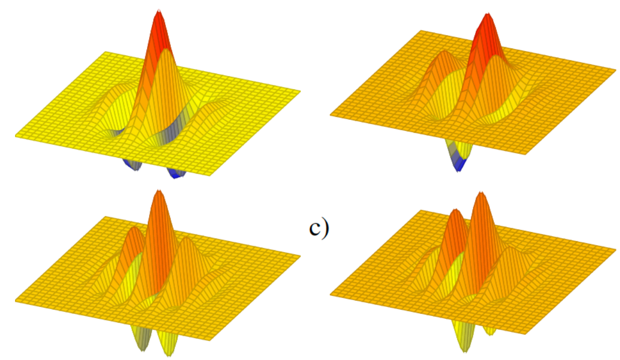


Laplacian

c)



High order Gaussian derivatives

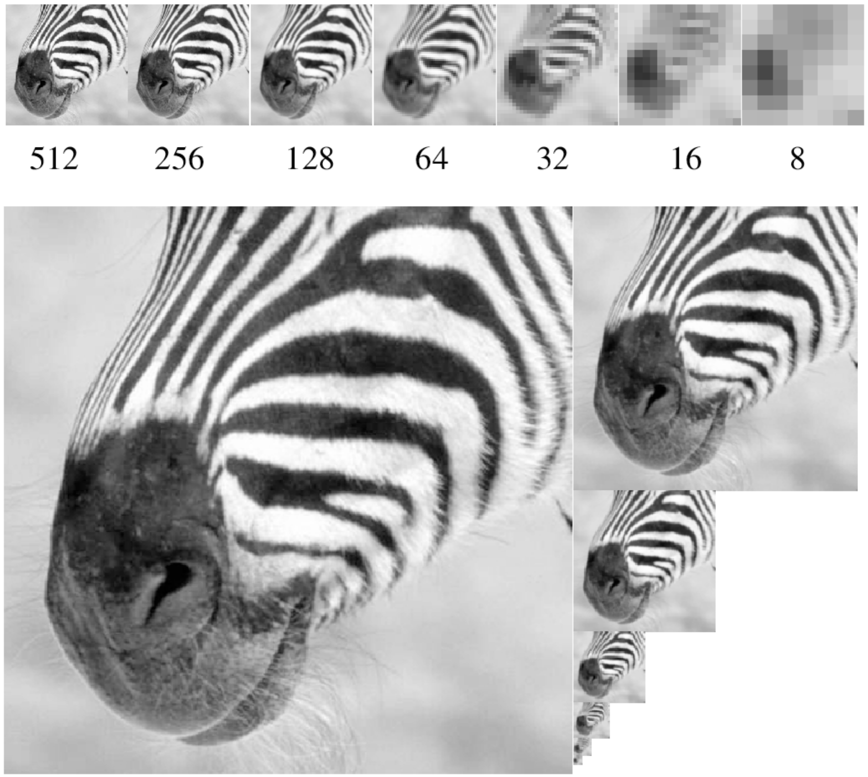


Gabor

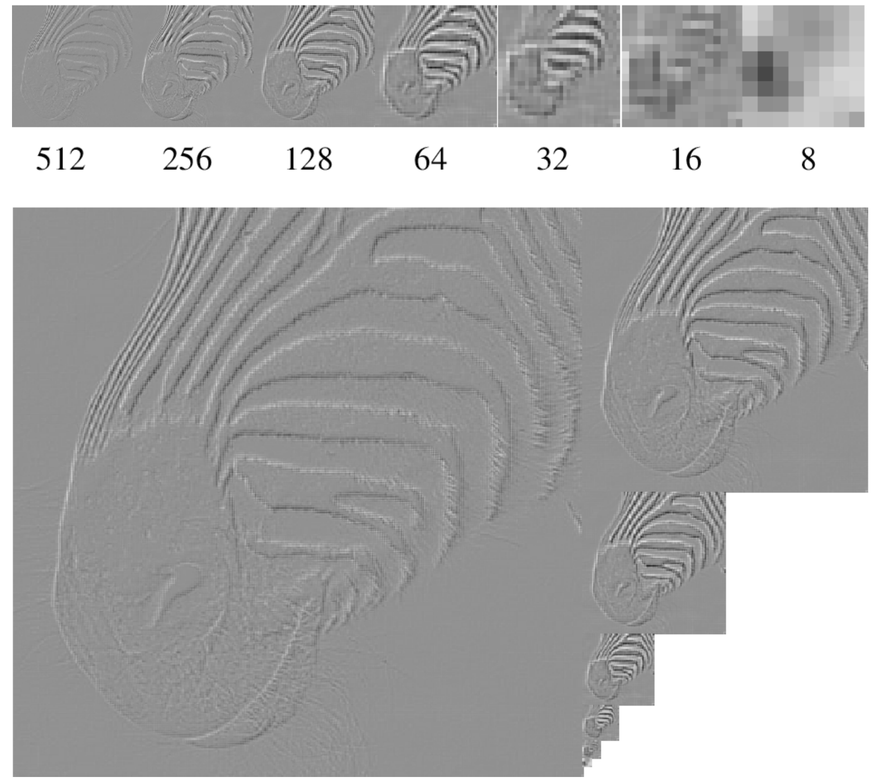
And many more…

We need translation and scale invariance

Lots of image pyramids…



Gaussian Pyr



Laplacian Pyr

And many more: QMF, steerable, …

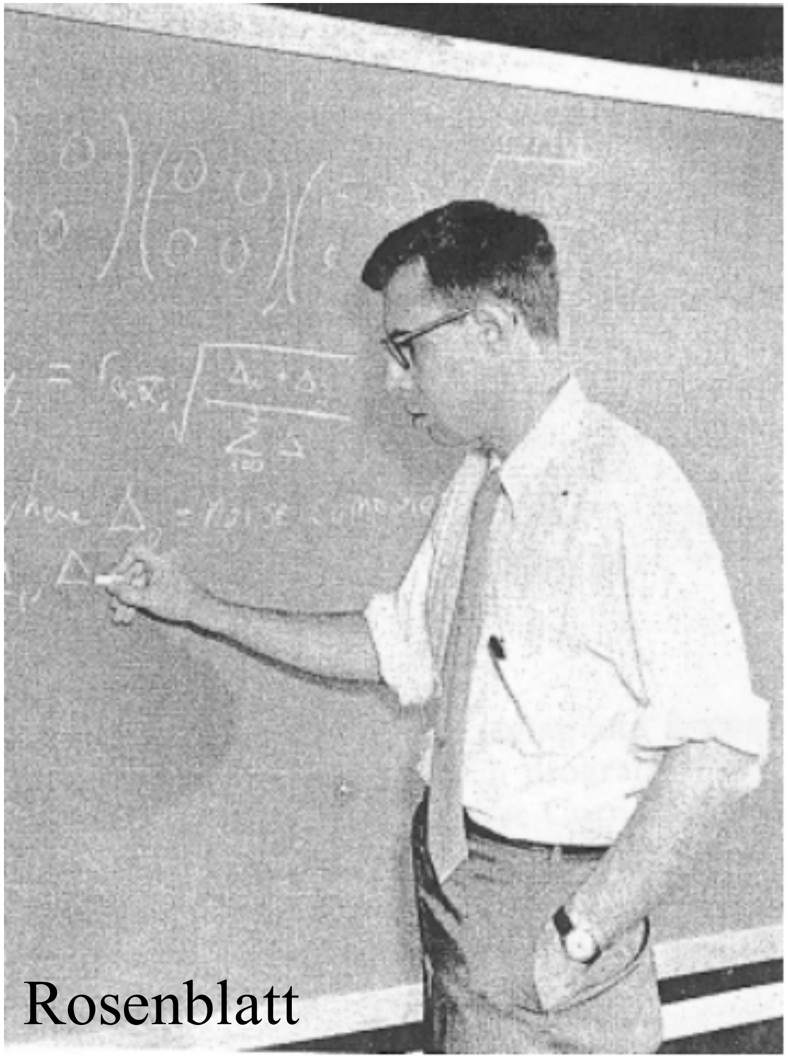
We need …

What is the best representation?

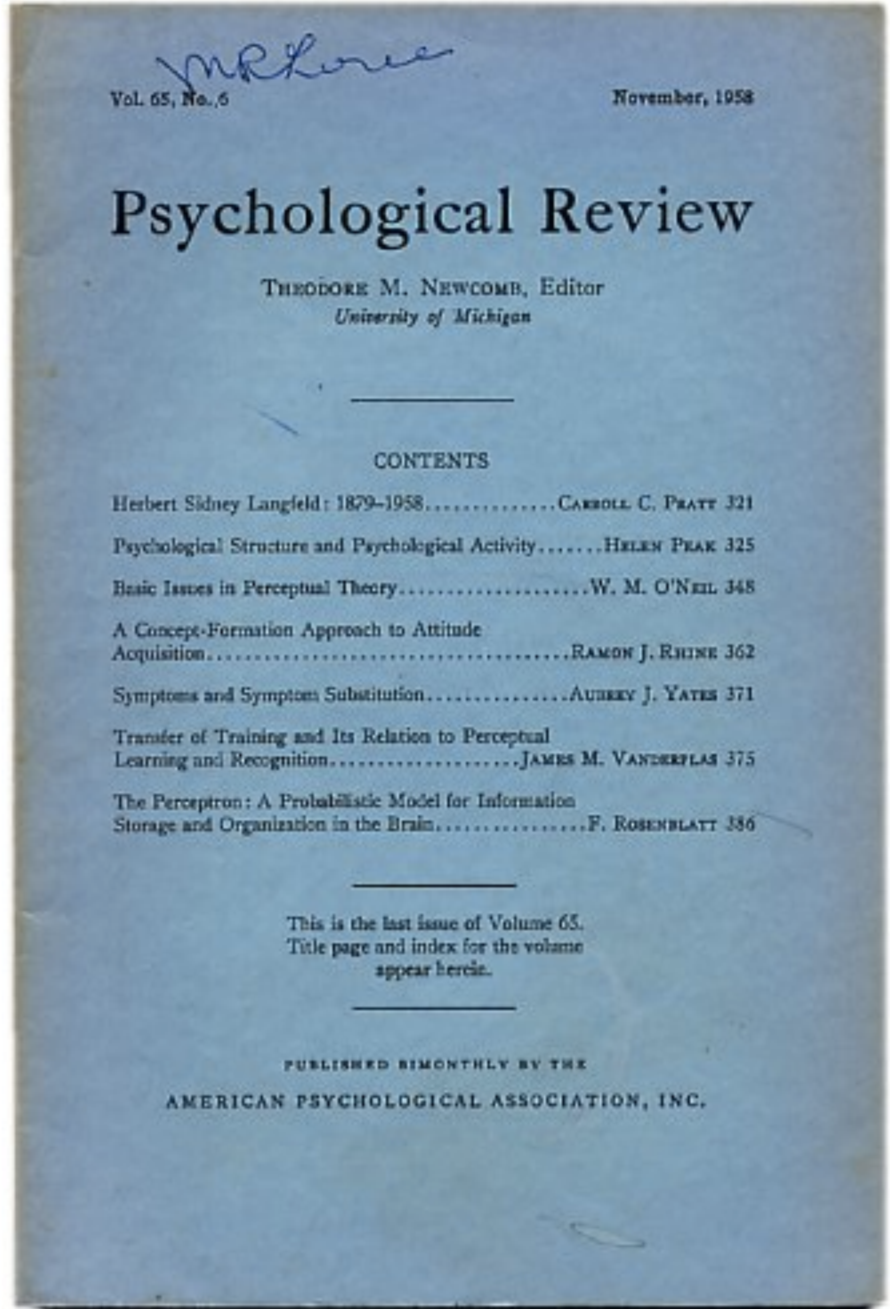
- All the previous representation are manually constructed.
- Could they be learnt from data?

A brief history of Neural Networks

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

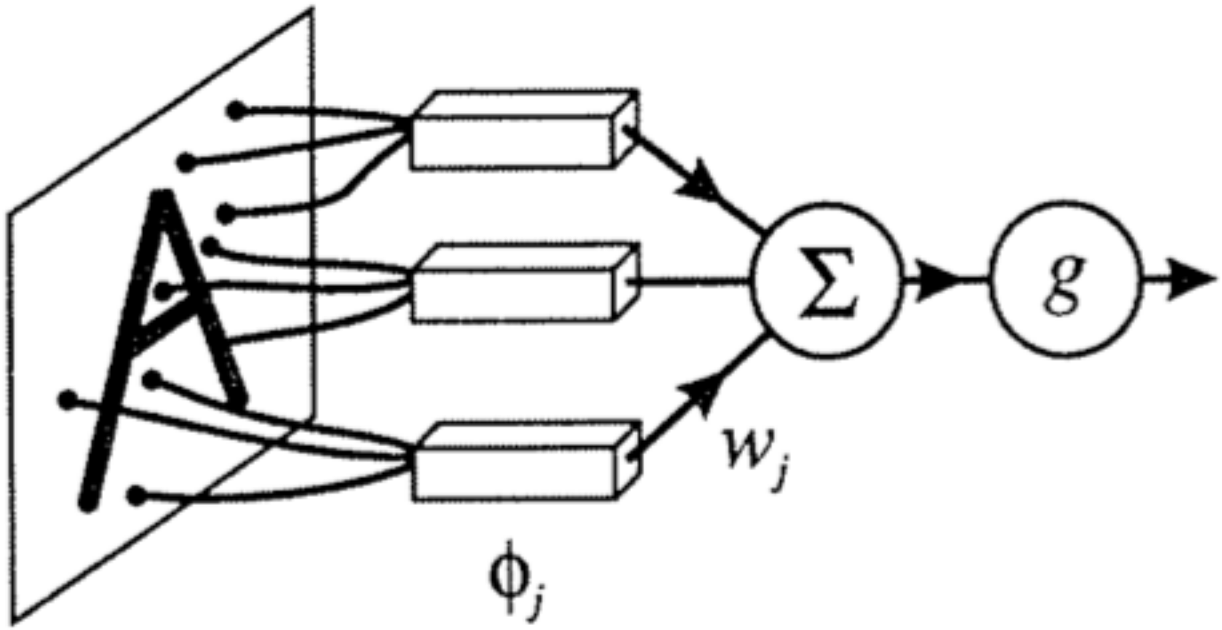
CONTENTS

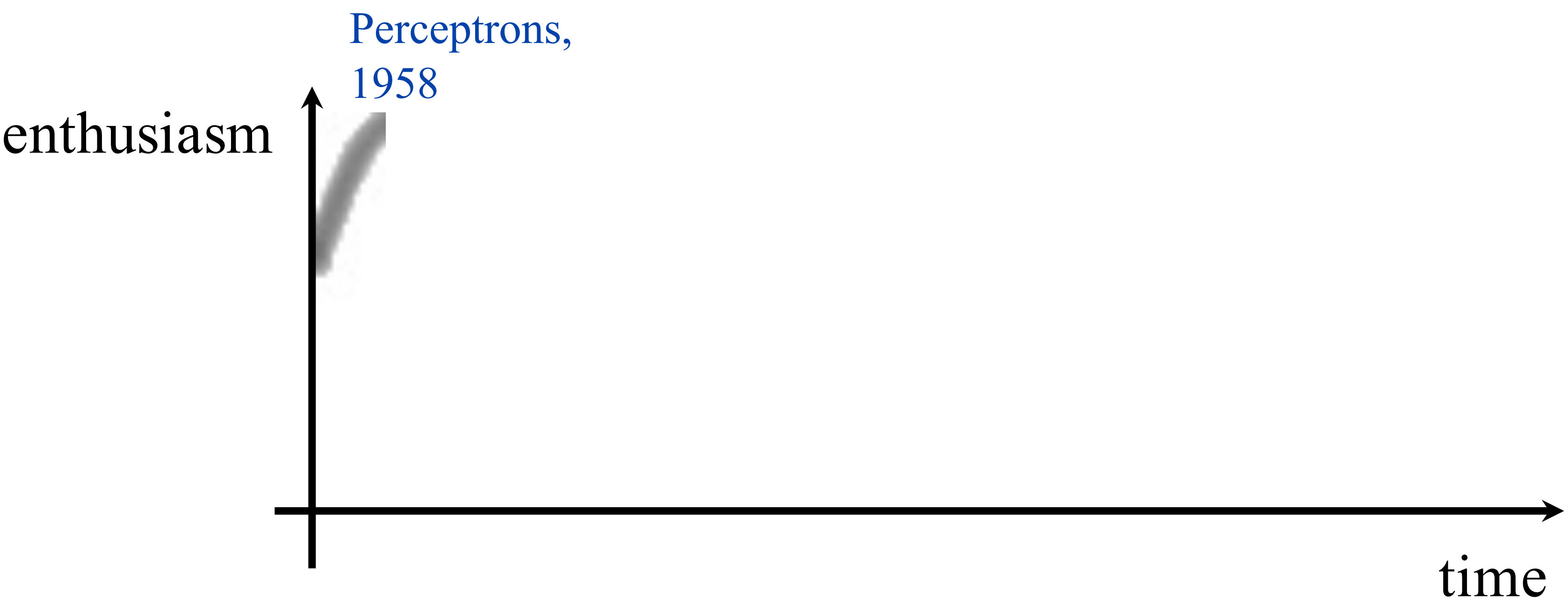
This is the last issue of Volume 65. Title page and index for the volume appear herein.

PUBLISHED BIMONTHLY BY THE
AMERICAN PSYCHOLOGICAL ASSOCIATION, INC.

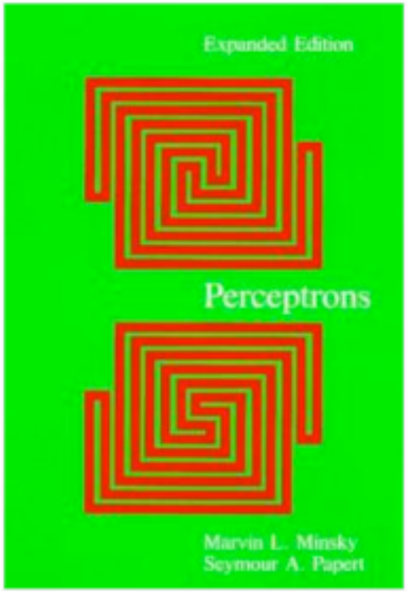
http://www.manhattanrarebooks-science.com/rosenblatt.htm

Perceptrons, 1958




Perceptrons,
1958
enthusiasm
time

Minsky and Papert, Perceptrons, 1972



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, expanded edition

An Introduction to Computational Geometry

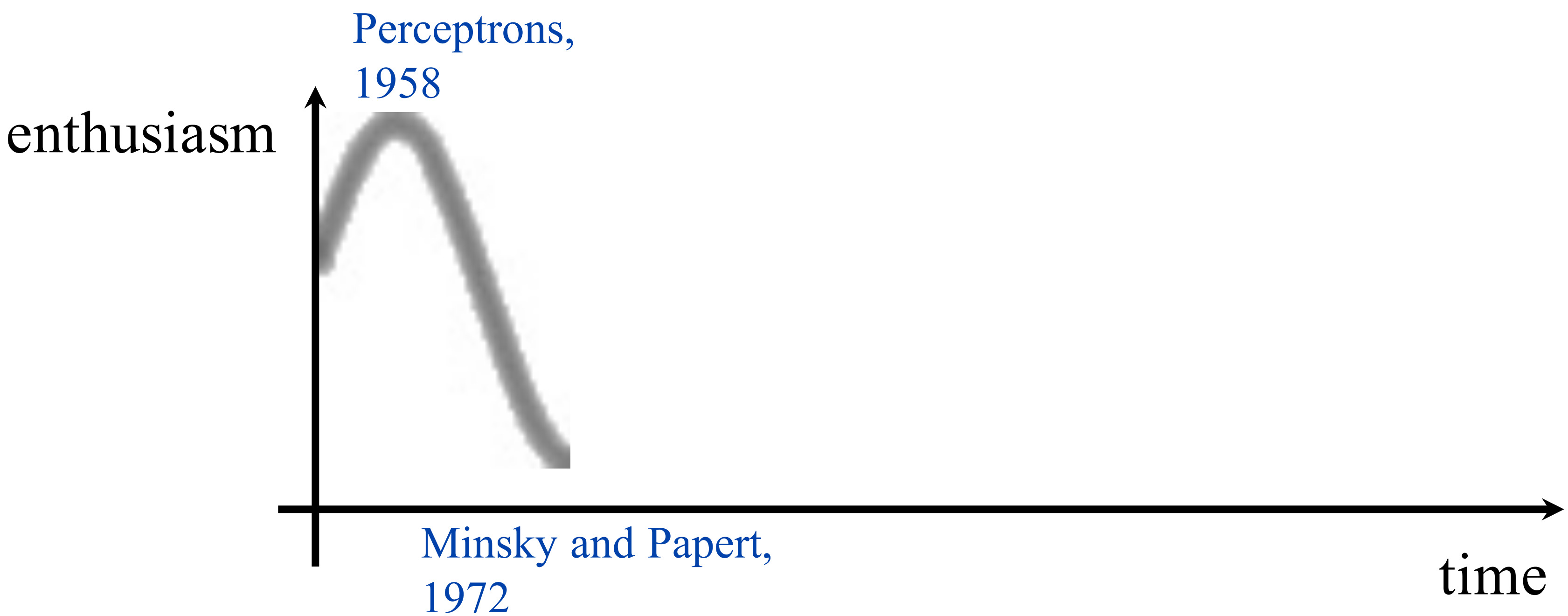
By Marvin Minsky and Seymour A. Papert

Overview

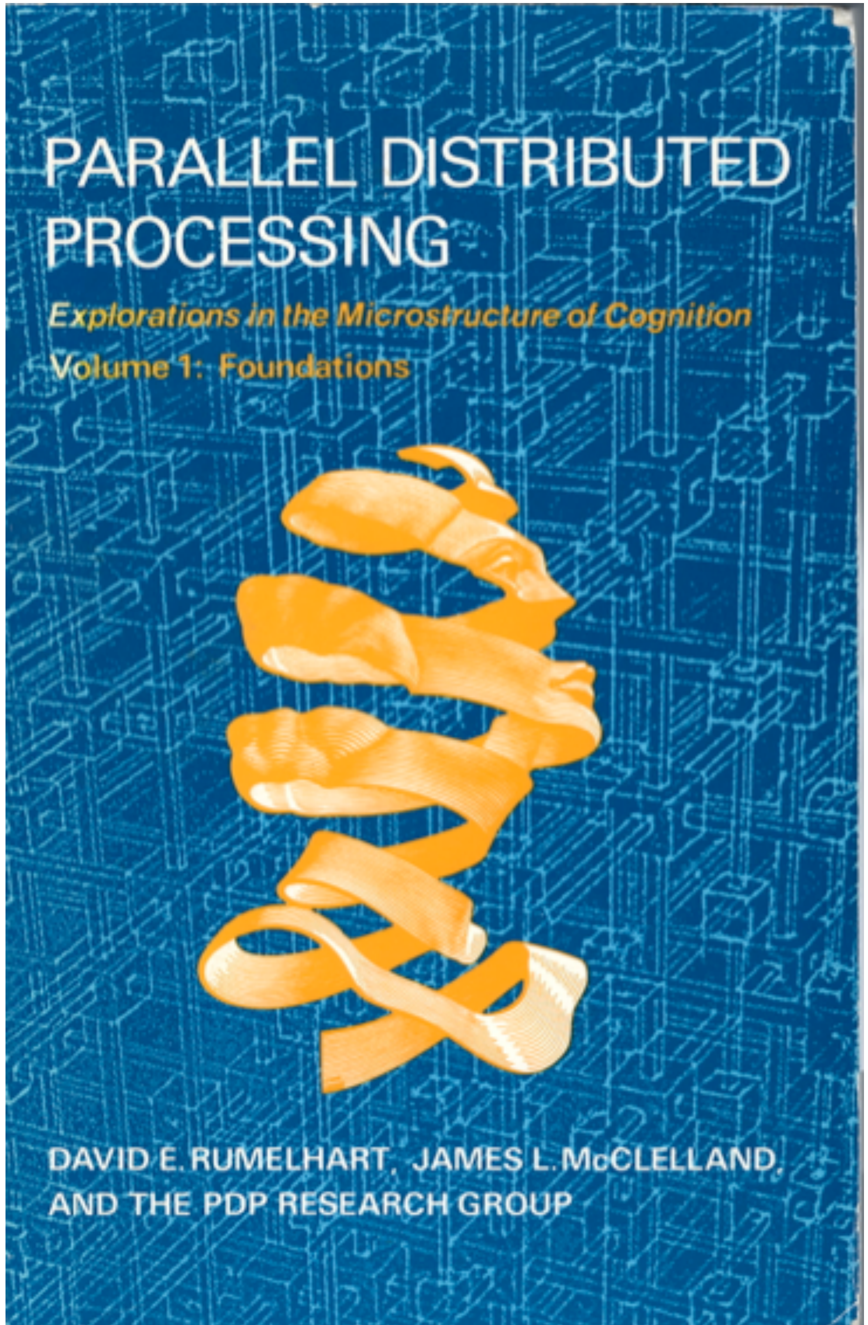
Perceptrons - the first systematic study of parallelism in computation - has remained a classical work on threshold automata networks for nearly two decades. It marked a historical turn in artificial intelligence, and it is required reading for anyone who wants to understand the connectionist counterrevolution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

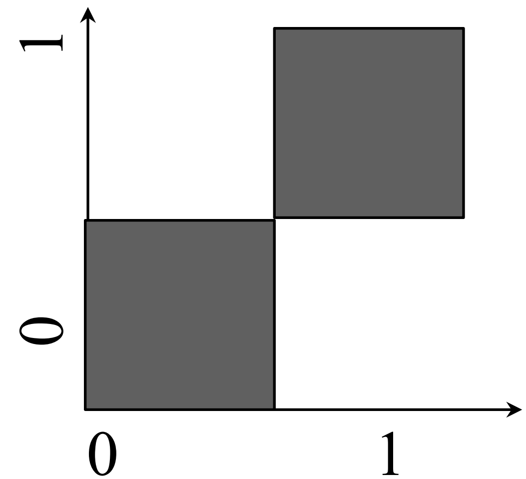

Perceptrons,
1958
enthusiasm
Minsky and Papert,
1972
time

Parallel Distributed Processing (PDP), 1986

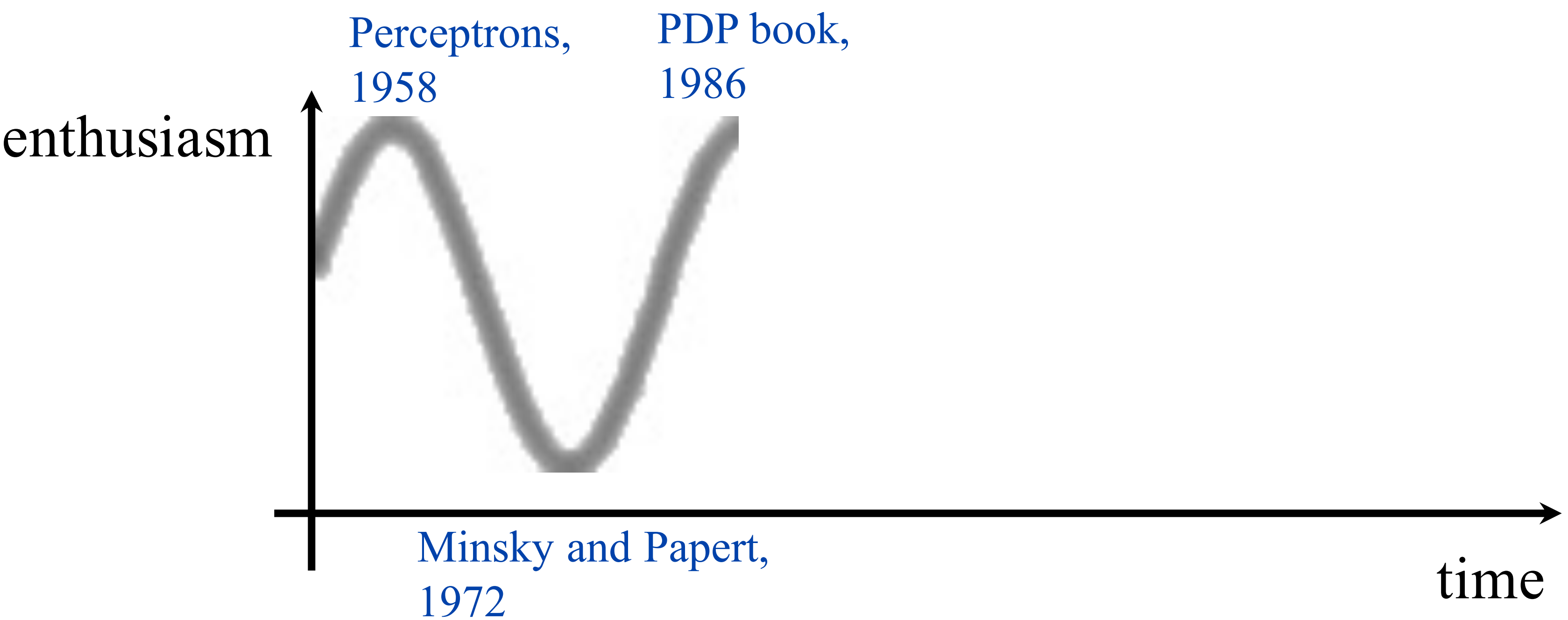


XOR problem

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.


Perceptrons,
1958
PDP book,
1986
enthusiasm
Minsky and Papert,
1972
time

LeCun conv nets, 1998

PROC. OF THE IEEE, NOVEMBER 1998 7

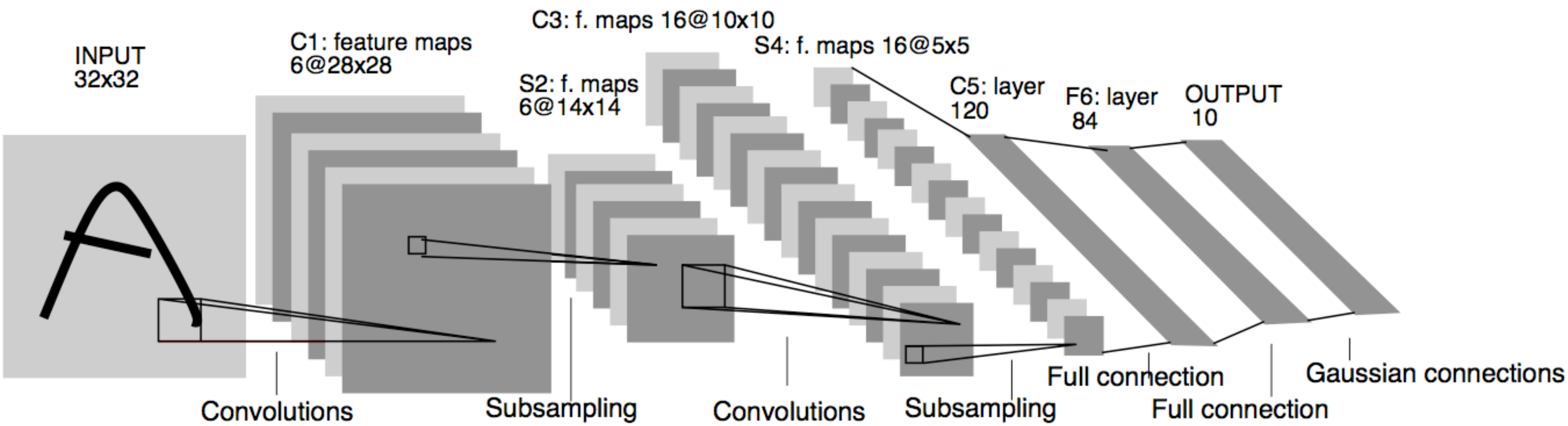


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Demos:

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

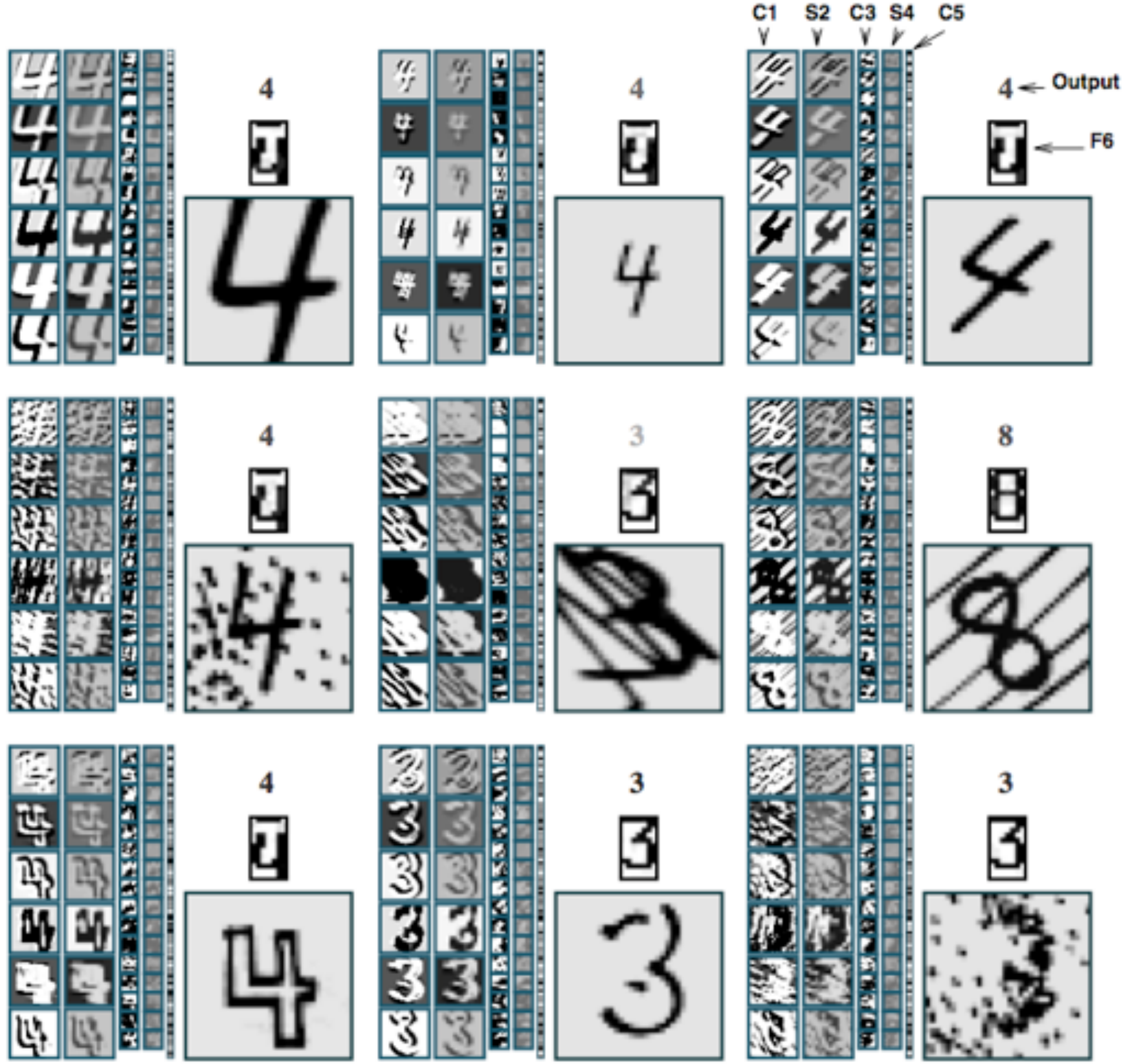
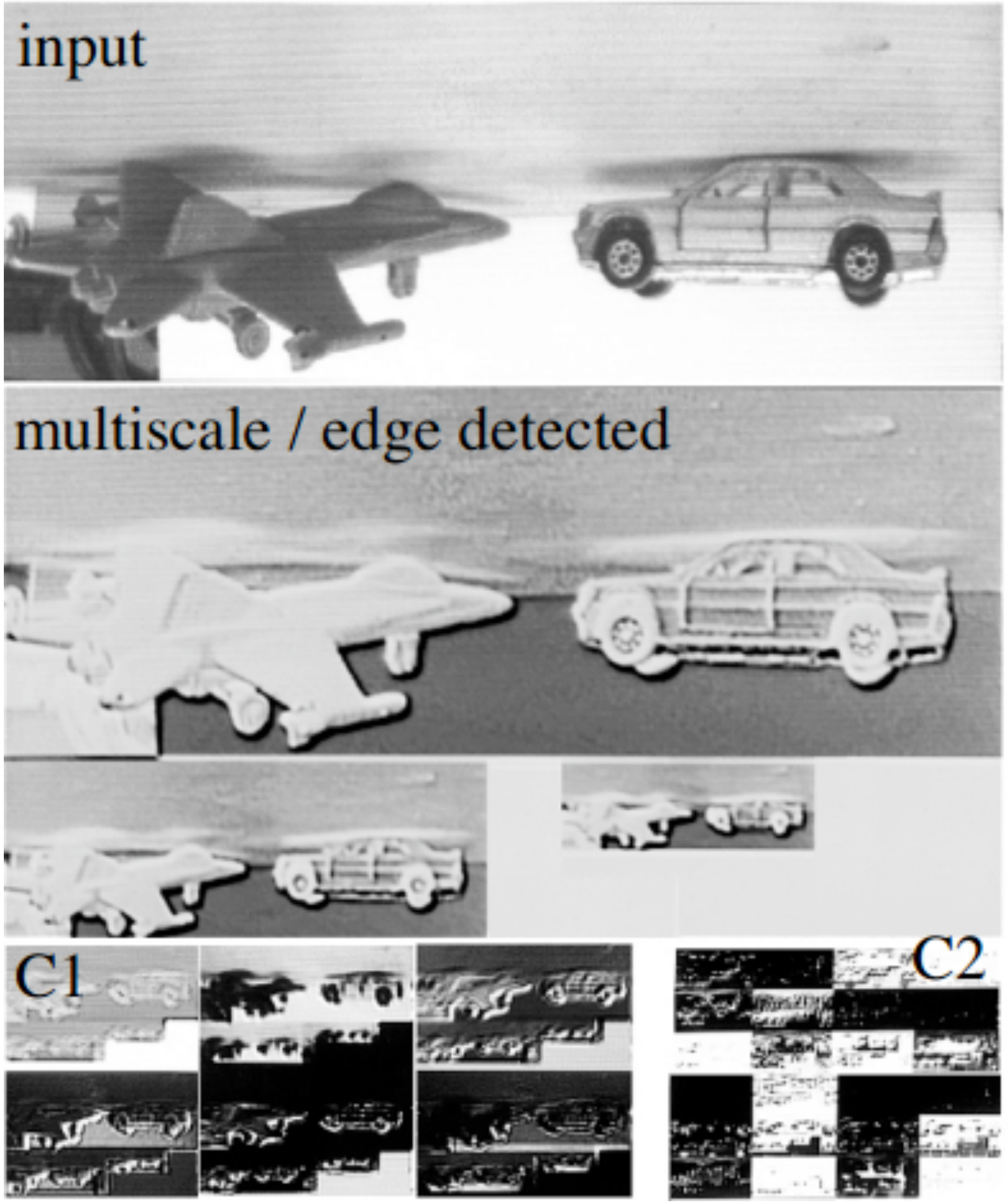


Fig. 13. Examples of unusual, distorted, and noisy characters correctly recognized by LeNet-5. The grey-level of the output label represents the penalty (lighter for higher penalties).



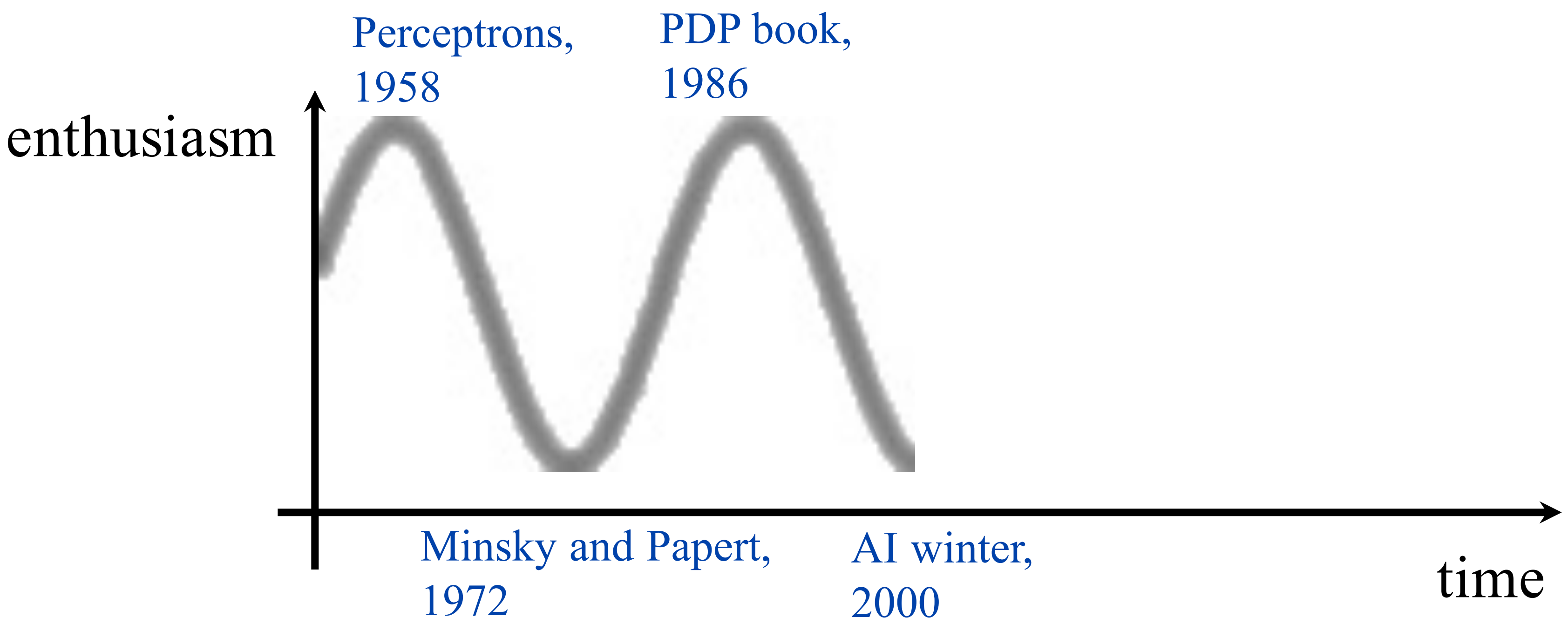
http://pub.clement.farabet.net/ecvw09.pdf

Neural networks to recognize handwritten digits? yes

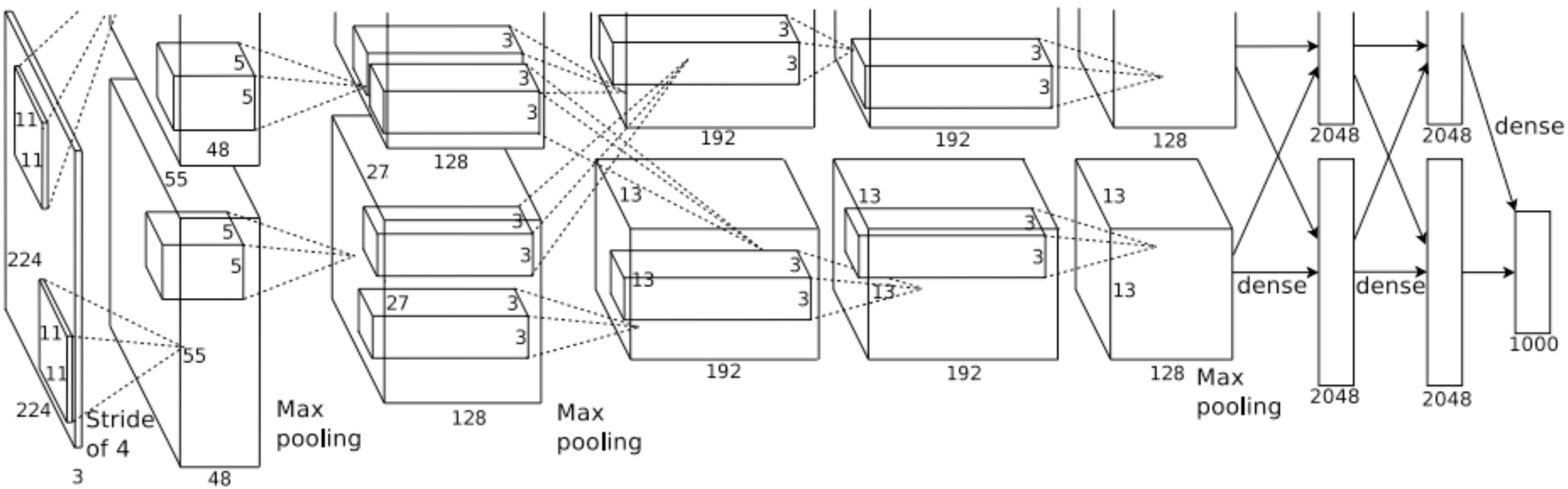
Neural networks for tougher problems? not really

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:
  - <u>title words predictive of paper acceptance</u>: "Belief Propagation" and "Gaussian".
  - <u>title words predictive of paper rejection</u>: "Neural" and "Network".


Perceptrons,
1958
PDP book,
1986
enthusiasm
Minsky and Papert,
1972
AI winter,
2000
time

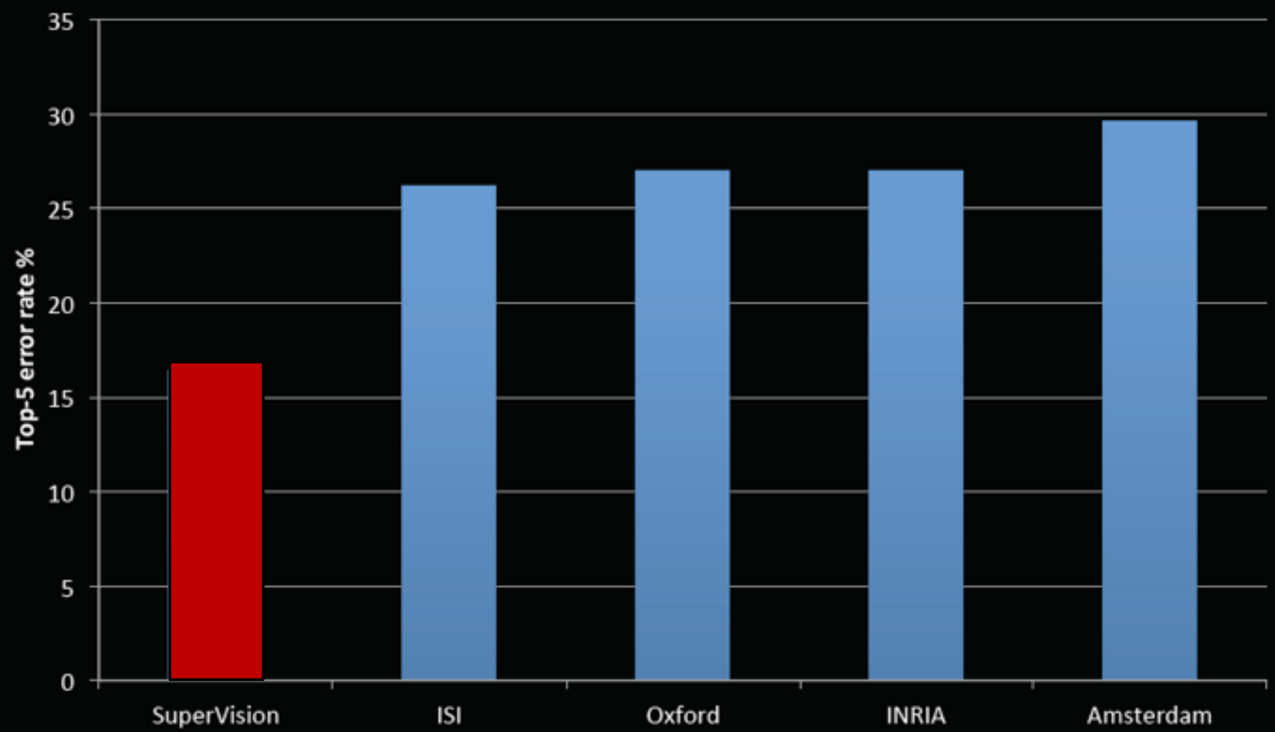
Krizhevsky, Sutskever, and Hinton, NIPS 2012



Slide from Rob Fergus, NYU

ImageNet Classification 2012

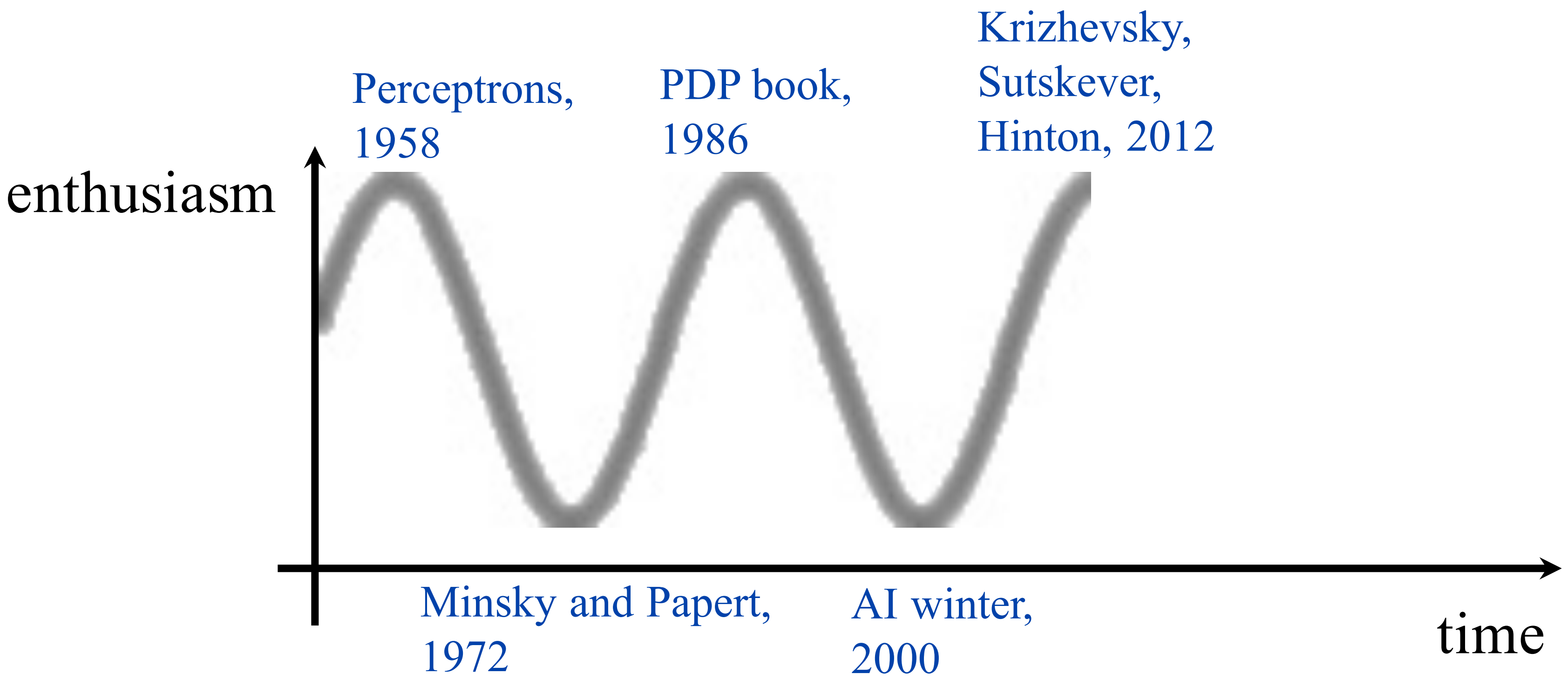
- Krizhevsky et al. -- 16.4% error (top-5)
- Next best (non-convnet) – 26.2% error

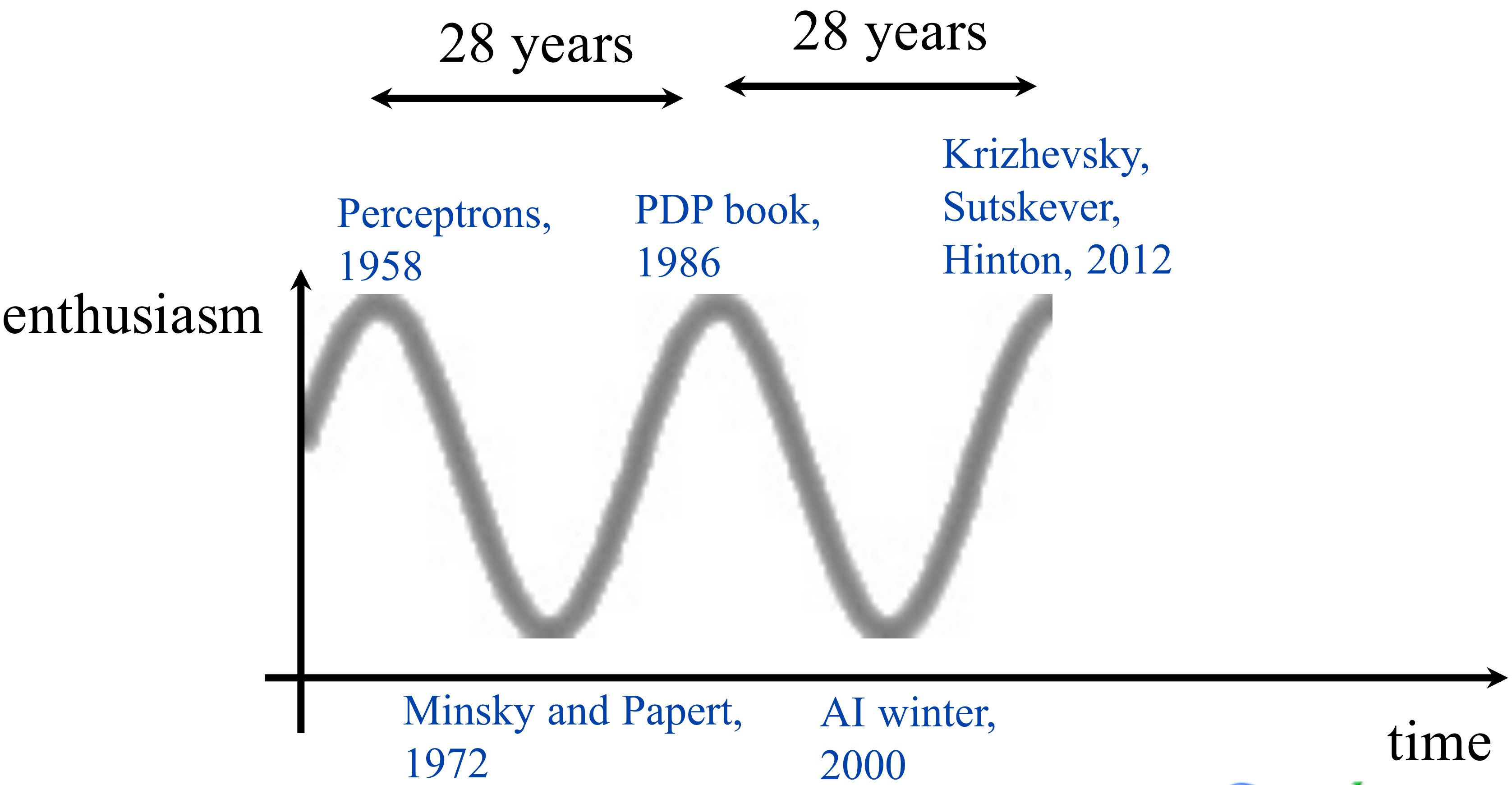


Krizhevsky, Sutskever, and Hinton, NIPS 2012

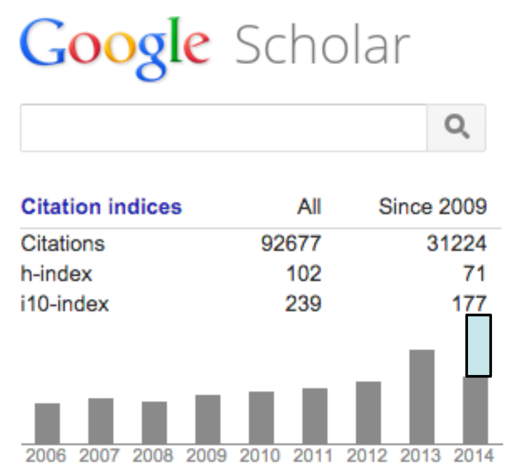
Test Nearby images, according to NN features

Krizhevsky, Sutskever, and Hinton, NIPS 2012


enthusiasm
Perceptrons,
1958
PDP book,
1986
Krizhevsky,
Sutskever,
Hinton, 2012
Minsky and Papert,
1972
AI winter,
2000
time

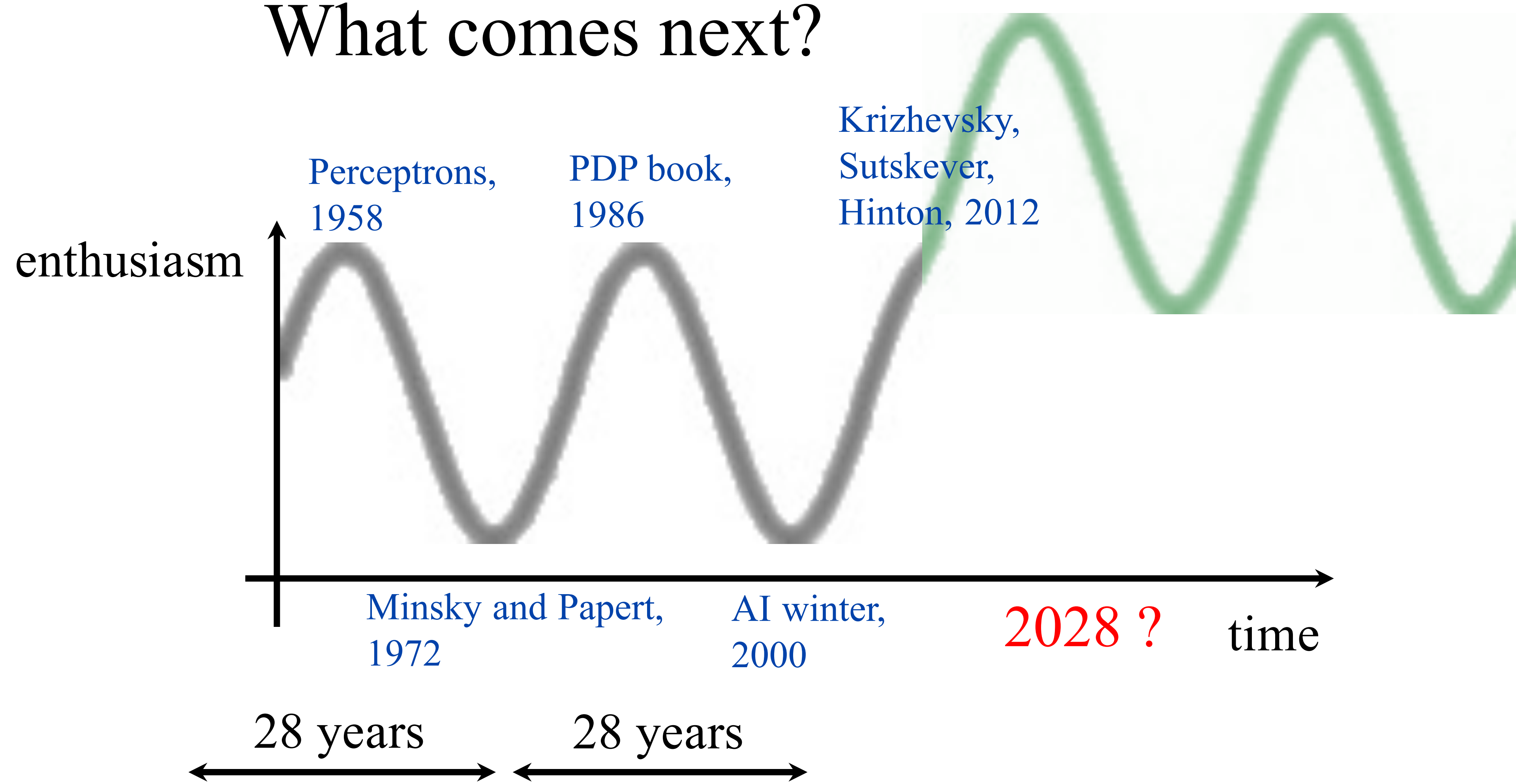


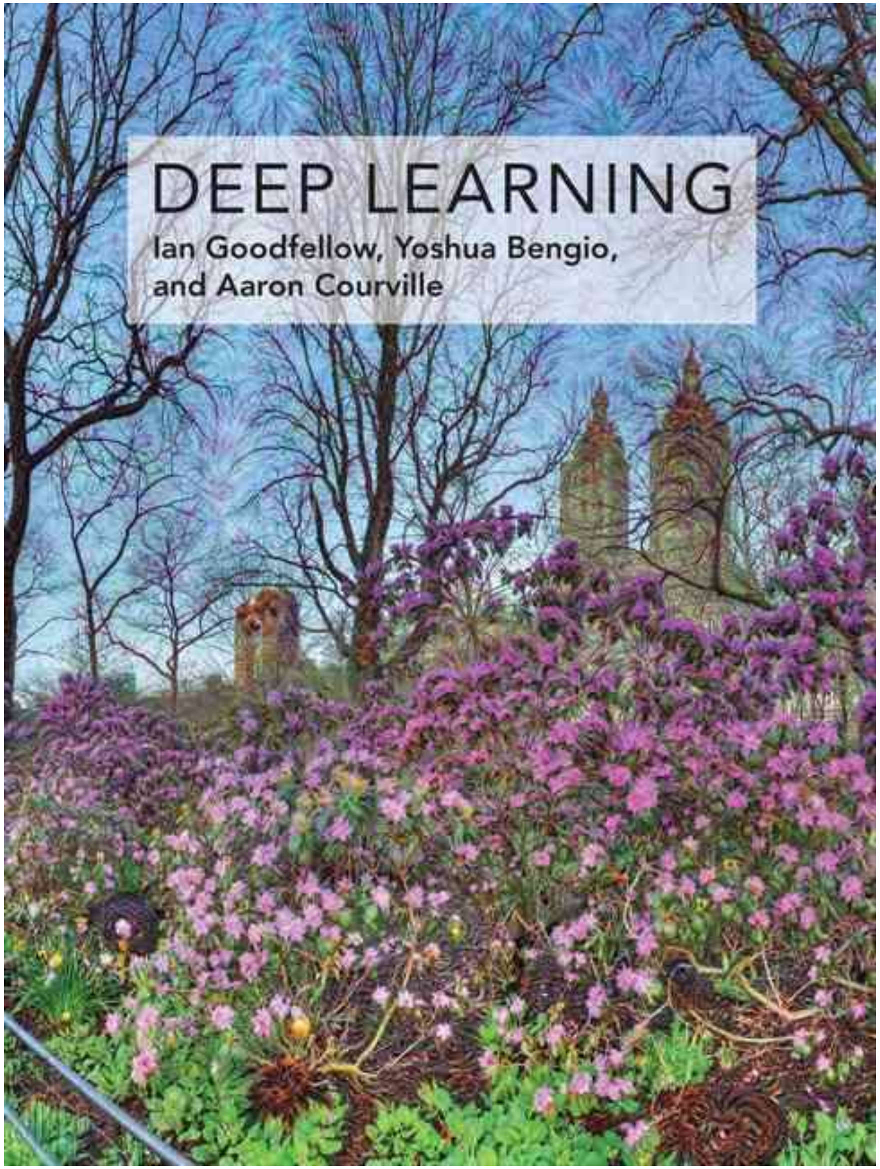
Geoff Hinton's citations



What comes next?

What comes next?





November 2016

http://www.deeplearningbook.org/

By Ian Goodfellow, Yoshua Bengio and Aaron Courville

Tutorials for Deep Learning Frameworks

TensorFlow Sessions:

- 5-6pm, Tue 10/3, 3-270, by Hunter
- 2-3pm, Thu 10/5, 3-270, by Jimmy

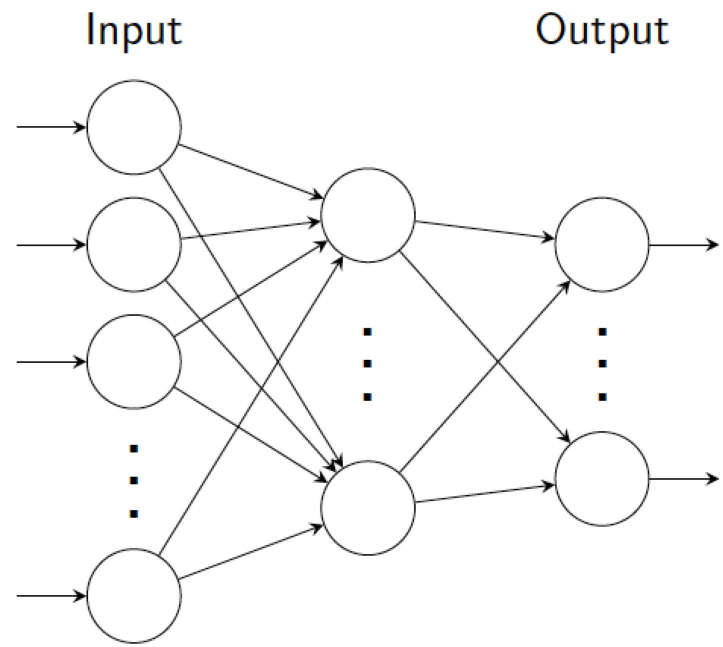
PyTorch Sessions:

- 4-5pm, Thu 10/5, 4-370, by Xiuming
- 6-7pm, Thu 10/5, 4-270, by Daniel

Information available on the class website

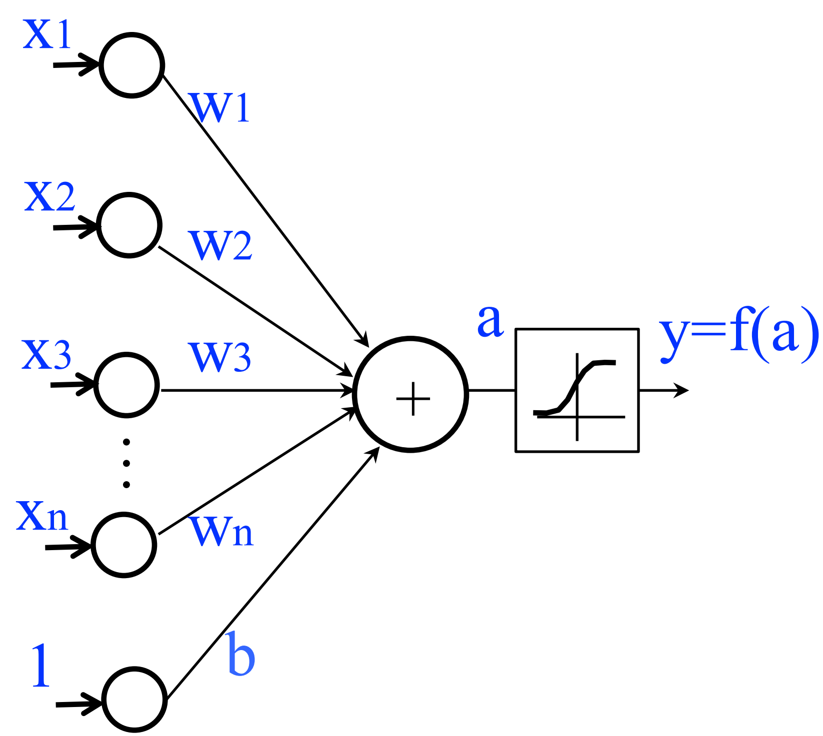
Neural networks

• 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 (unit)

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



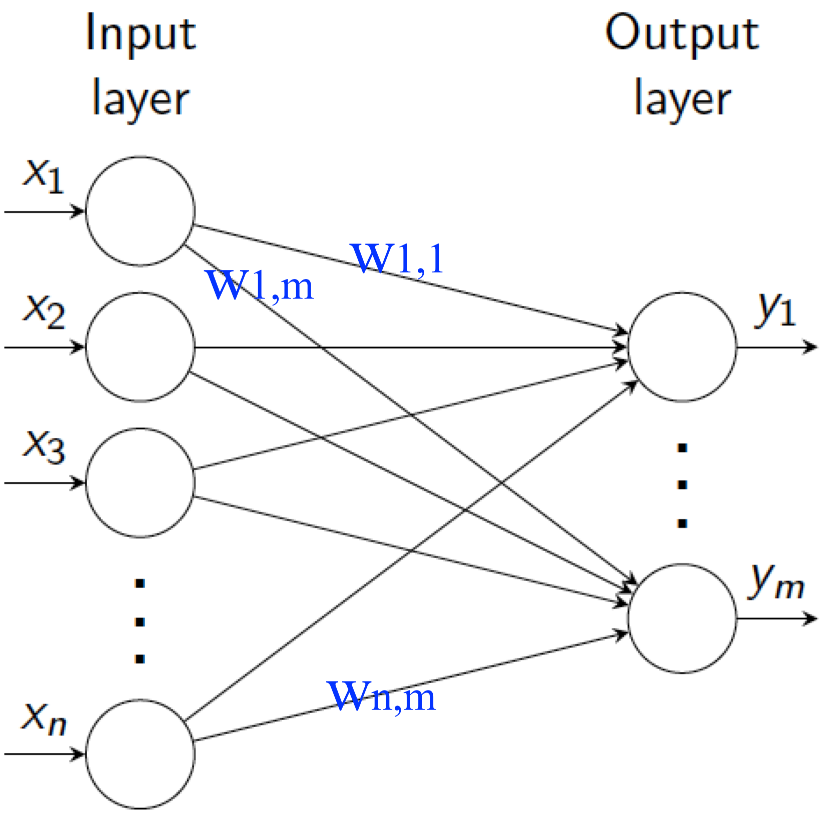
f(.) is a point-wise non-linear function. E.g.:

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

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: $a = Wx + b$



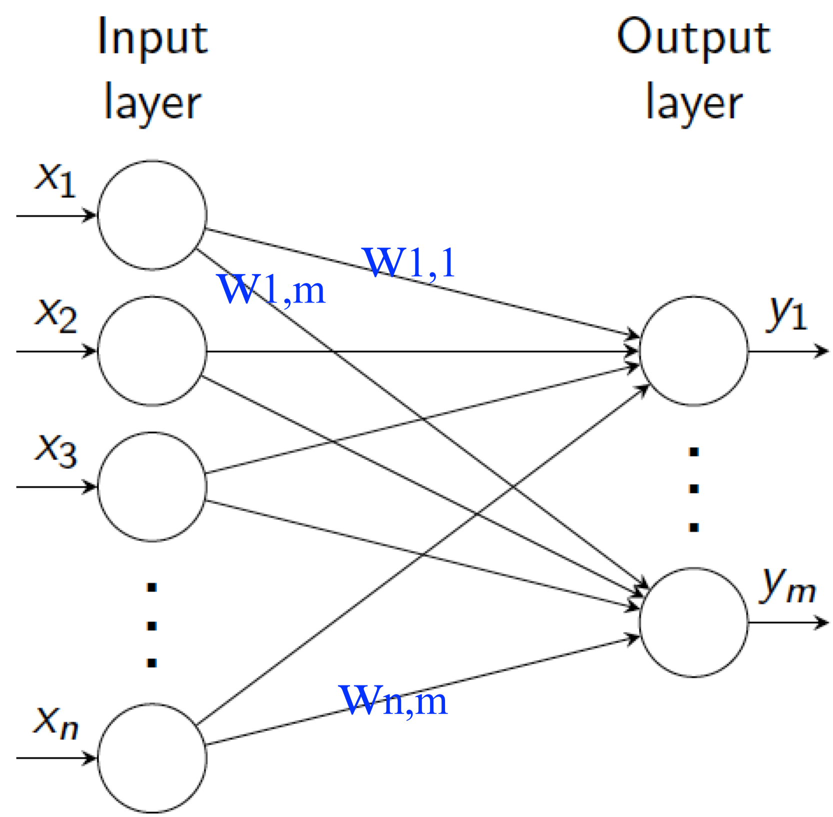
Single layer network

- 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

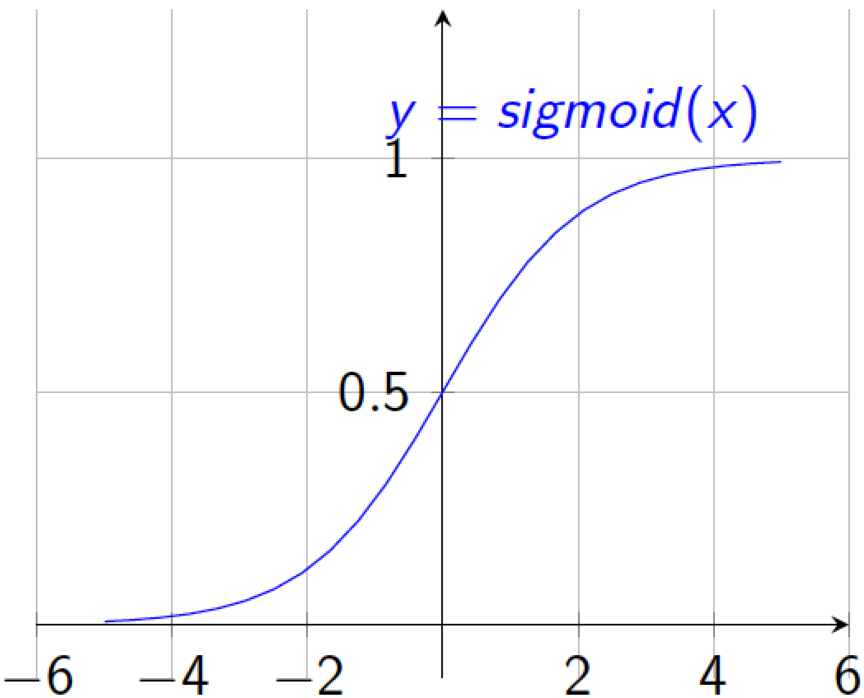
$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} w_{1,1} & & & \\ & & & \\ w_{3,1} & & & w_{4,3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

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



Non-linearities: sigmoid

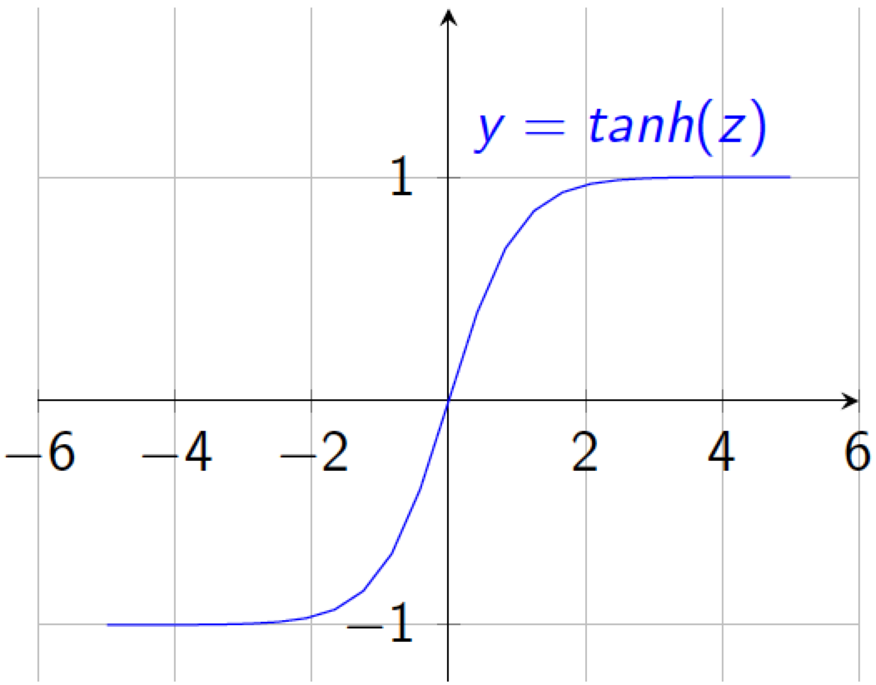
$$f(a) = sigmoid(a) = \frac{1}{1+e^{-a}}$$



- Interpretation as firing 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

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

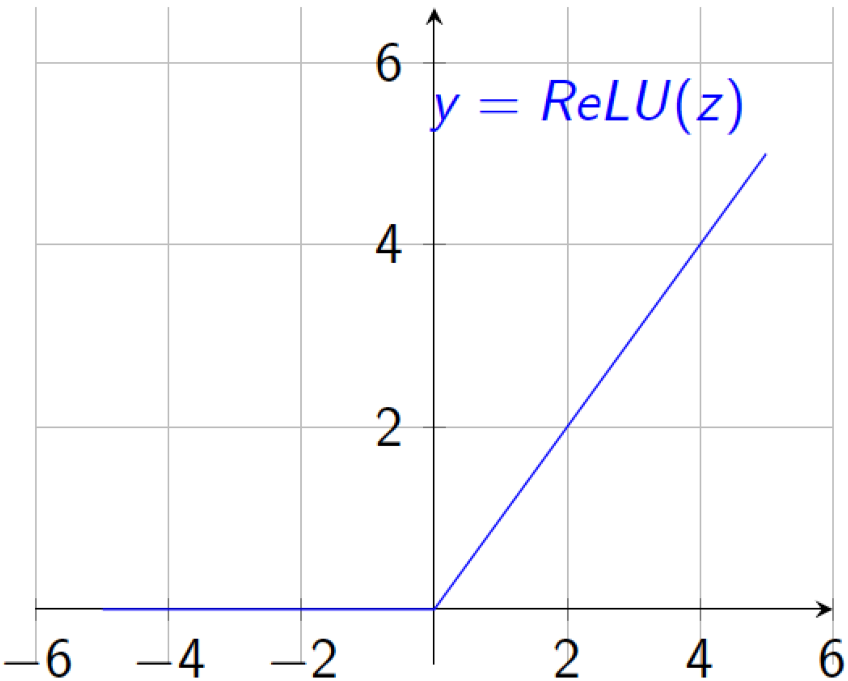


- Bounded between [-1,+1]
- Saturation for large +ve,-ve inputs
- Gradients go to zero
- Outputs centered at 0
- Preferable to sigmoid

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

Non-linearities: rectified linear (ReLU)

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



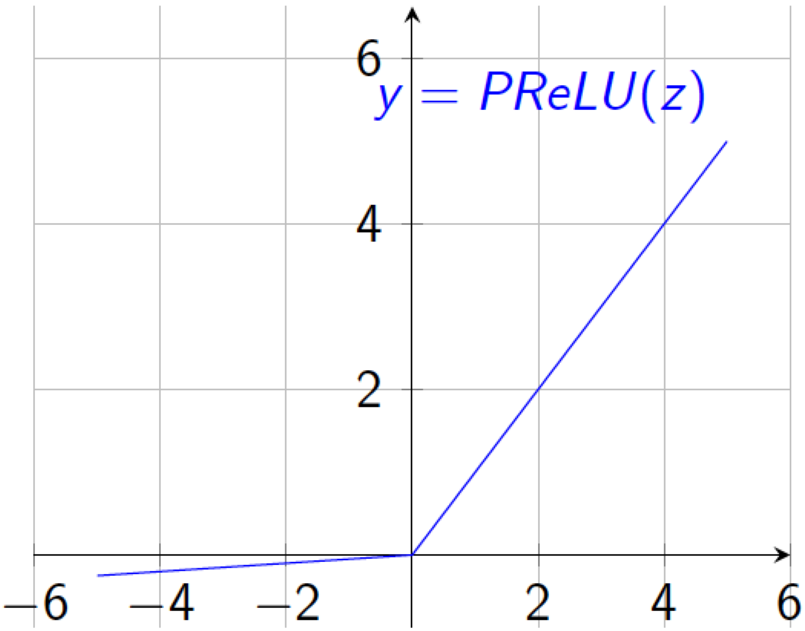
- Unbounded output (on positive side)
- Efficient to implement:

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

- 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

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



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

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

- 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

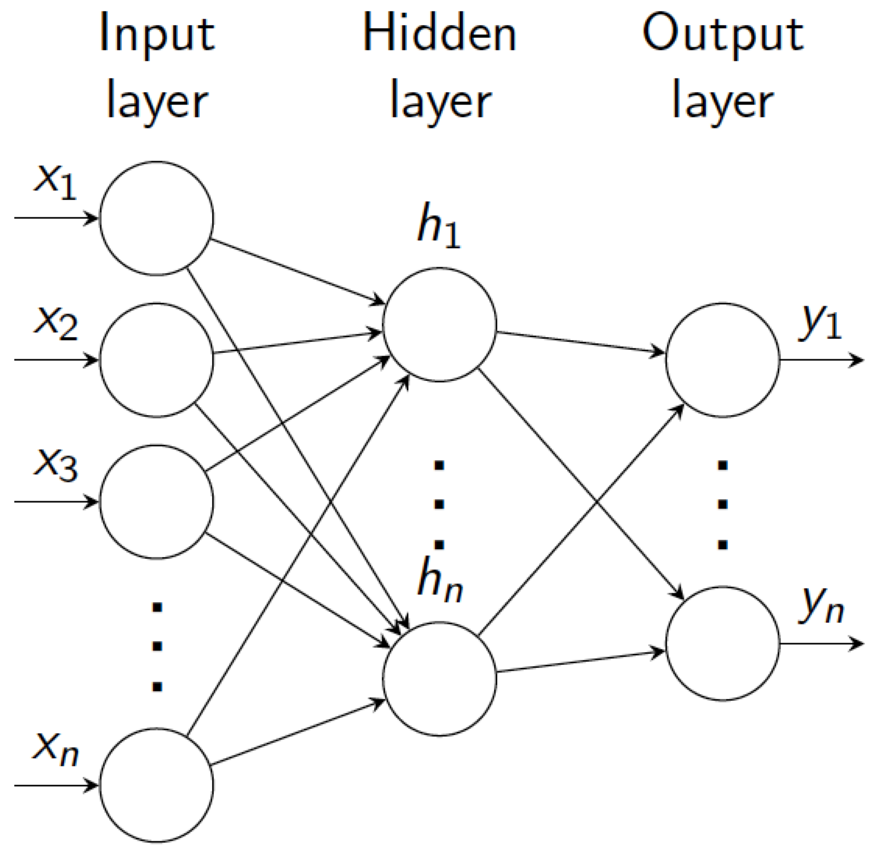
- Neural networks are composed of multiple layers of neurons.

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

- Layers between input and output are called hidden.

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

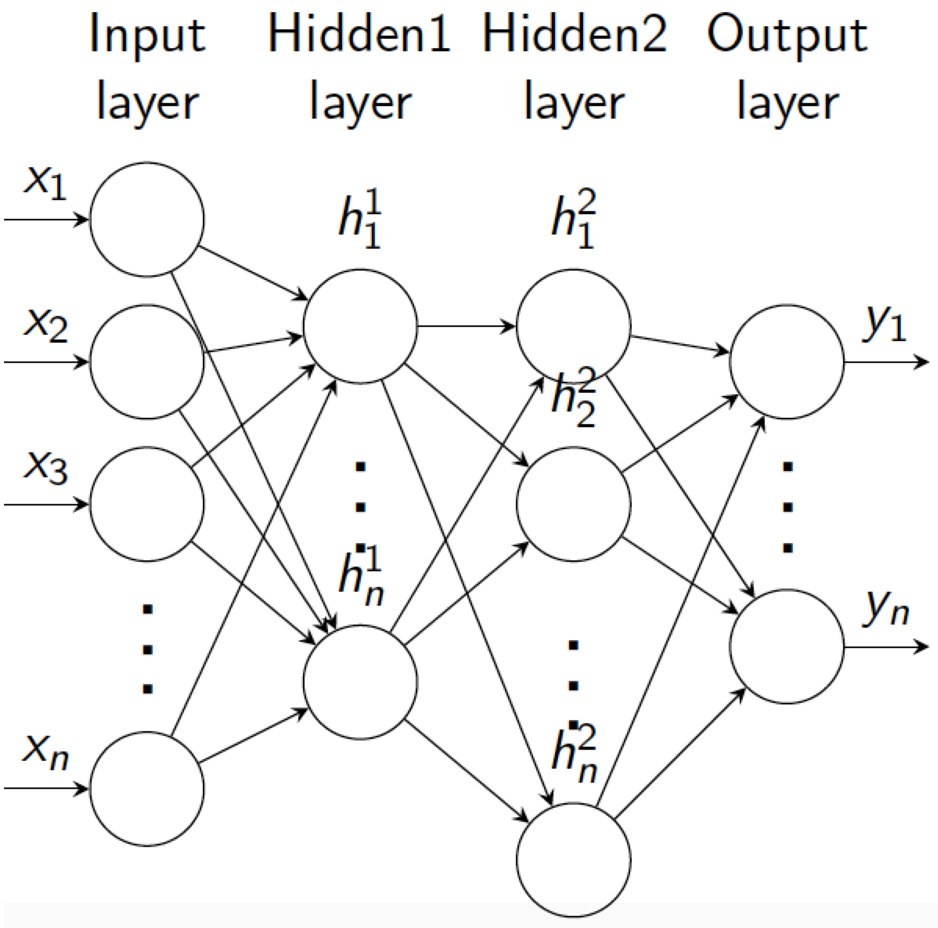
- Neurons typically called units.



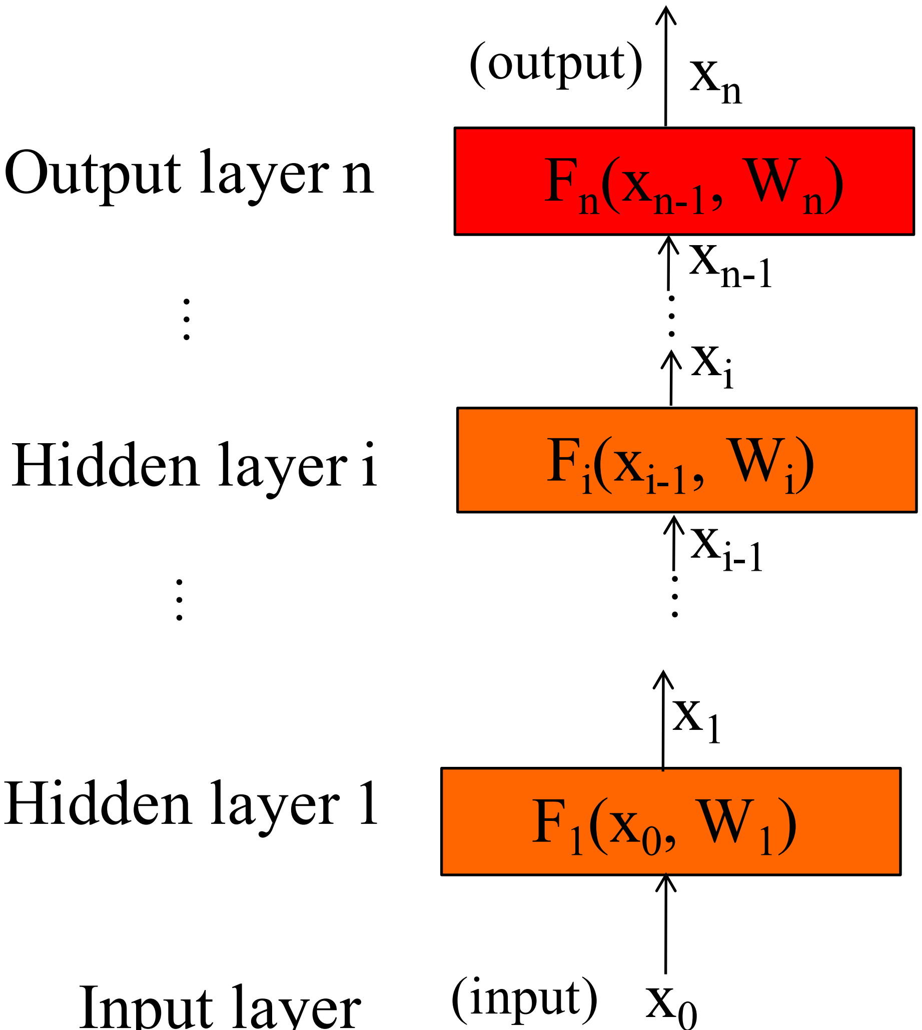
Example: 3 layer MLP

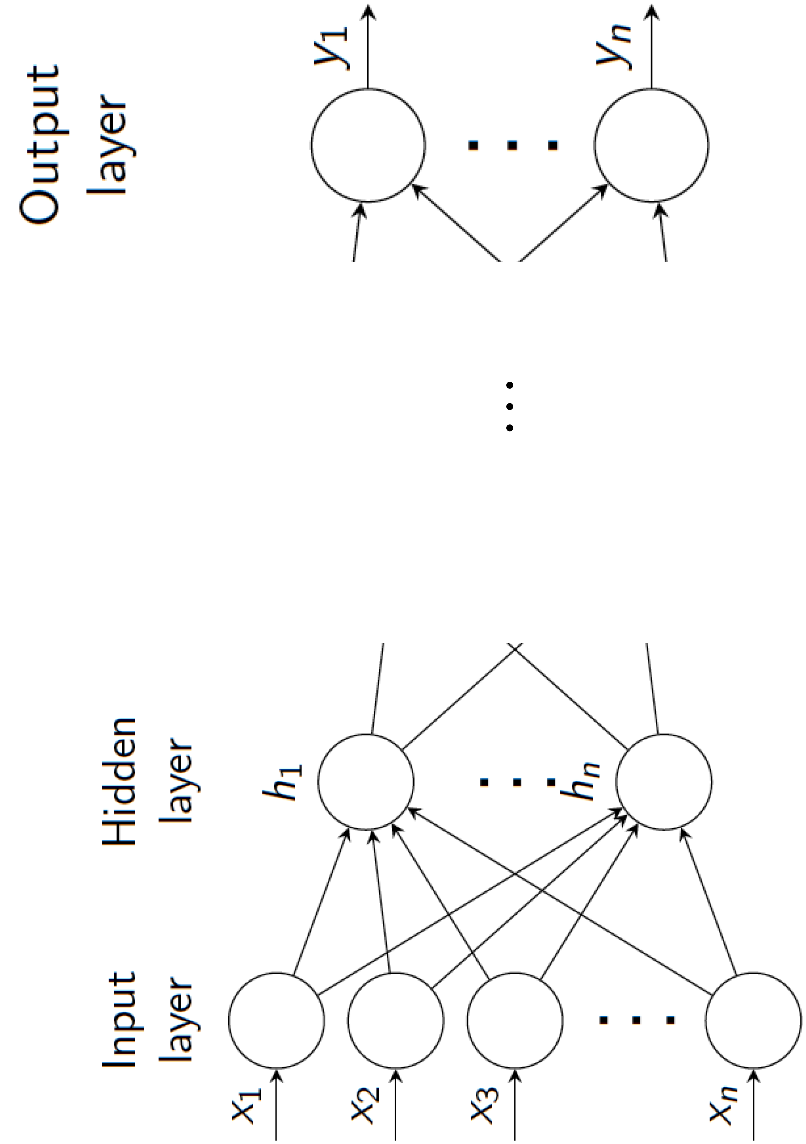
- By convention, number of layers is hidden + output (i.e. does not include input).
- So 3-layer model has 2 hidden layers.
- Parameters:

  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.
- Regarding number of units/layer:
  – Parameters grows with $(\text{units/layer})^2$.
  – With large units/layer, can easily overfit.

Representational power of two-layer network

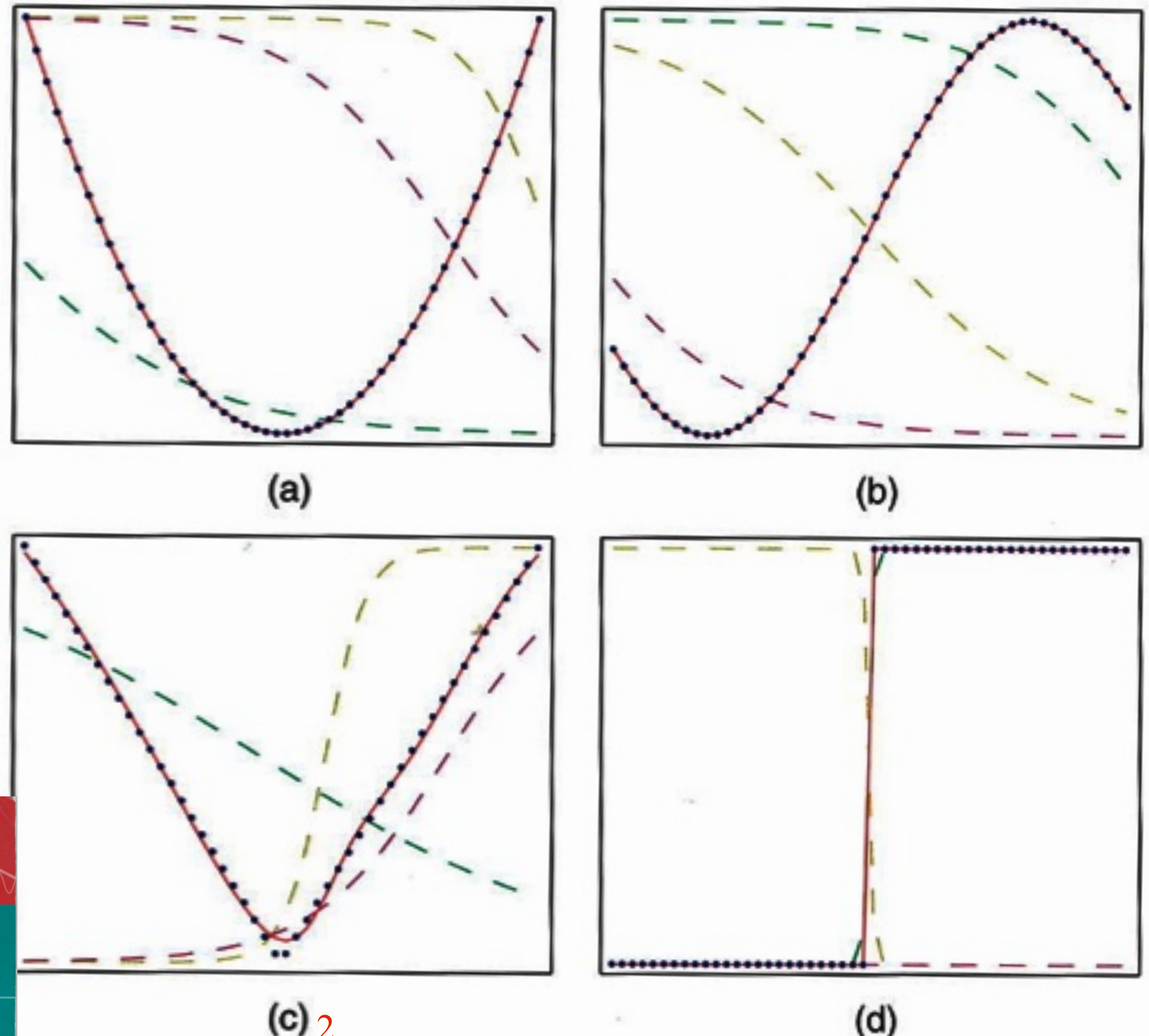
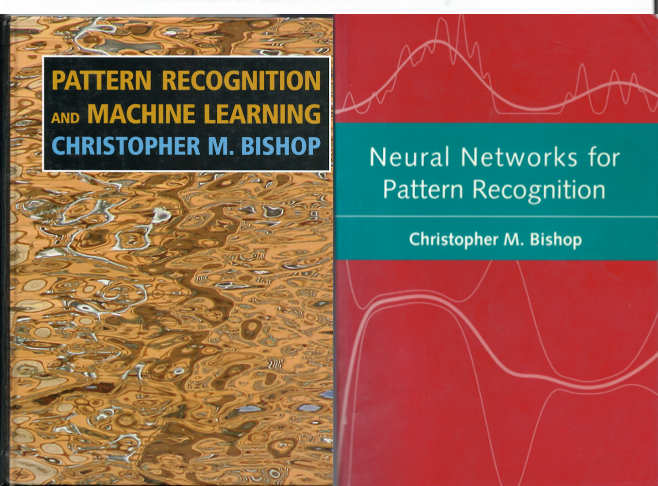
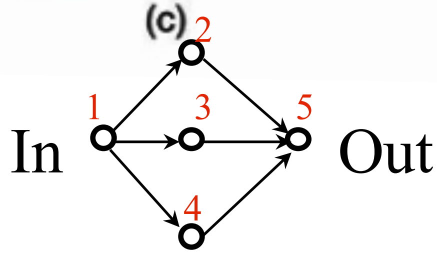


Figure 5.3 Illustration of the capability of a multilayer perceptron to approximate four different functions comprising (a) $f(x) = x^2$, (b) $f(x) = \sin(x)$, (c), $f(x) = |x|$, and (d) $f(x) = H(x)$ where $H(x)$ is the Heaviside step function. In each case, $N = 50$ data points, shown as blue dots, 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.





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

Representational power

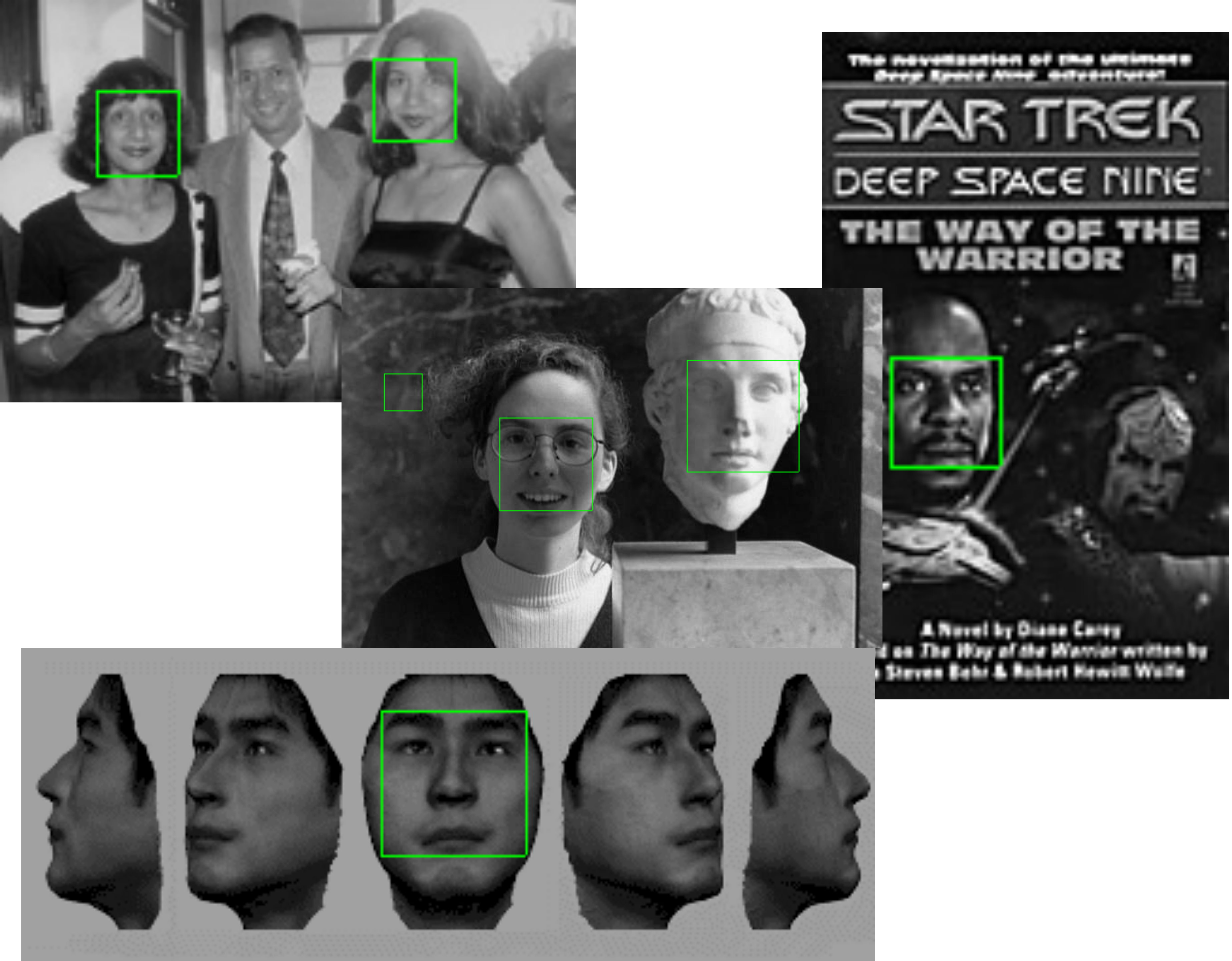
- 1 layer? Linear decision surface.
- 2+ layers? In theory, can represent any function. Assuming non-trivial non-linearity.
  - Bengio 2009,
    http://www.iro.umontreal.ca/~bengioy/papers/ftml.pdf
  - Bengio, Courville, Goodfellow book
    http://www.deeplearningbook.org/contents/mlp.html
  - Simple 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)

(

Object detection

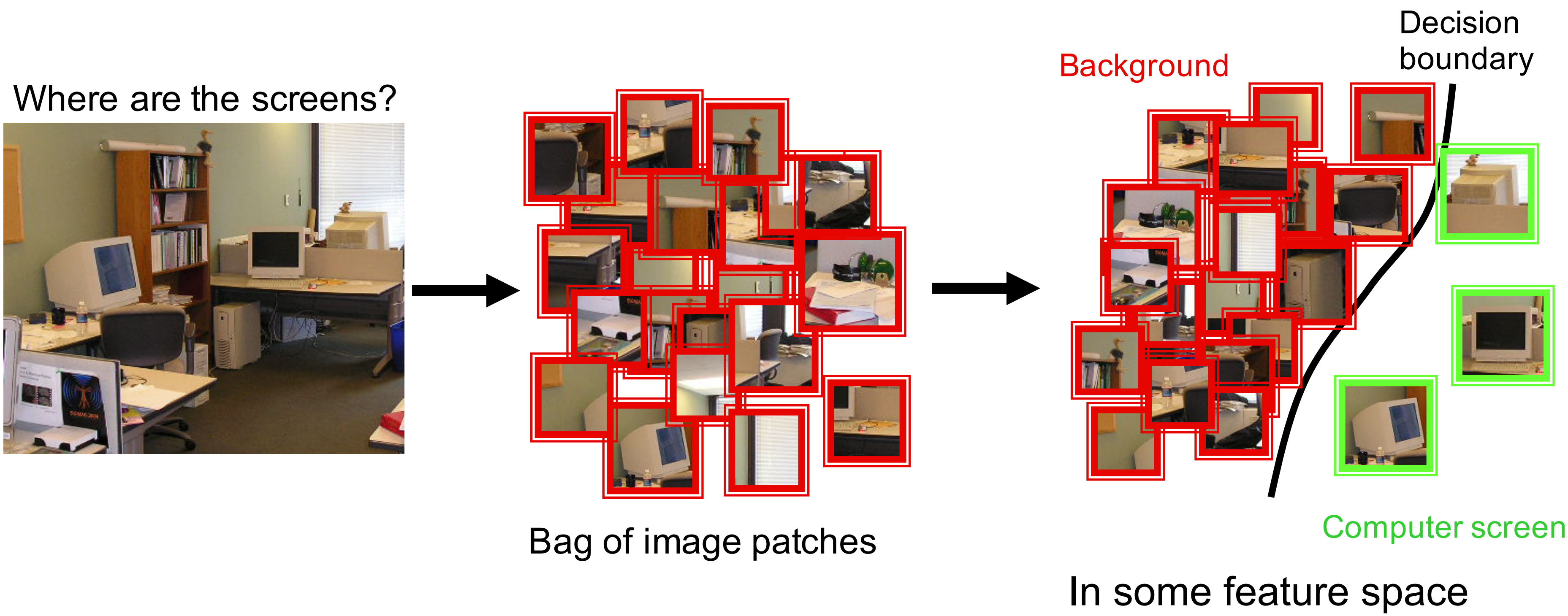

STAR TREK
DEEP SPACE NINE
THE WAY OF THE WARRIOR
A Novel by Diane Carey

Discriminative methods

Object detection and recognition is formulated as a classification problem.

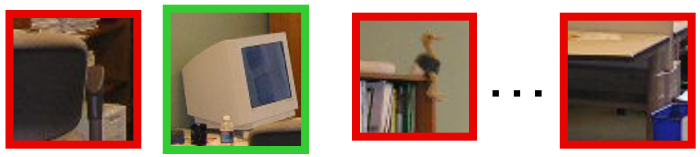
The image is partitioned into a set of overlapping windows

… and a decision is taken at each window about if it contains a target object or not.



Formulation

- Formulation: binary classification



Features	x =	x_1	x_2	x_3	$\cdots$ x_N	x_{N+1}	x_{N+2}	$\cdots$	x_{N+M}
Labels	y =	-1	+1	-1	-1	?	?		?

Training data: each image patch is labeled as containing the object or background

Test data

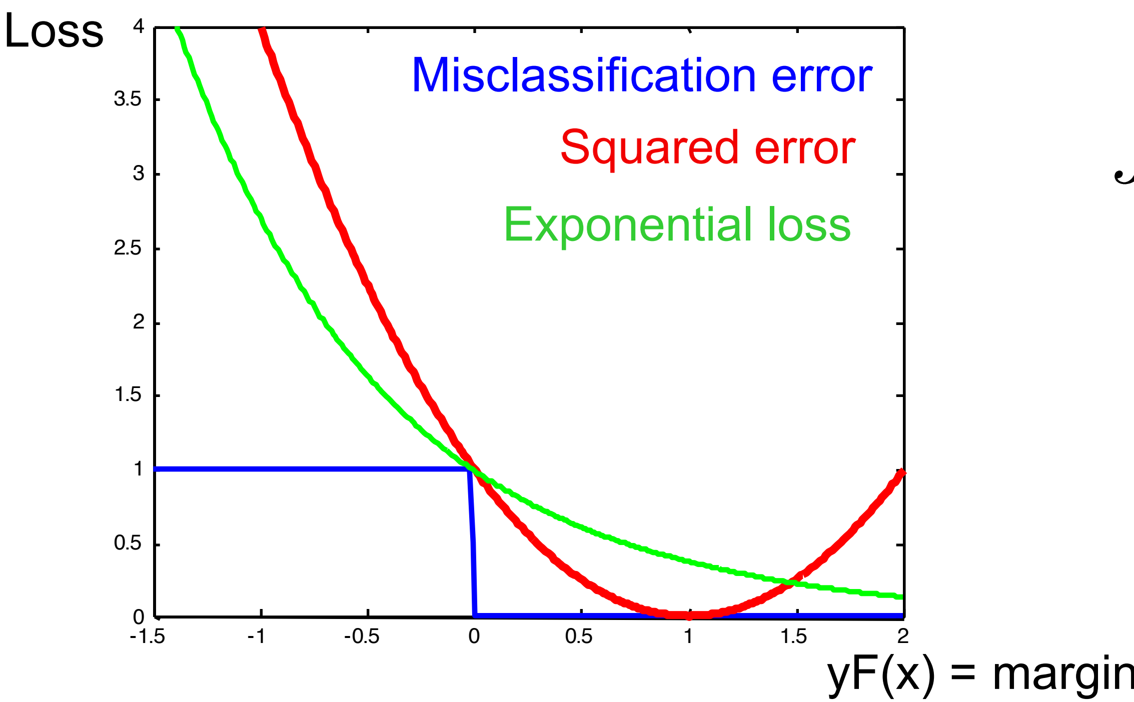
- Classification function

$$\hat{y} = F(x)$$ Where $F(x)$ belongs to some family of functions

- Minimize misclassification error

(Not that simple: we need some guarantees that there will be generalization)

Cost functions



Squared error

$$J = \sum_{t=1}^{N} [y_t - F(x_t)]^2$$

Exponential loss

$$J = \sum_{t=1}^{N} e^{-y_t F(x_t)}$$

MiniPlaces Challenge

- Goal: identify the scene category depicted in a photograph.
- Data
  - 100,000 images for training, 10,000 for validation and 10,000 for testing
  - 100 scene categories
- Task: produce up to 5 categories in descending order of confidence

MiniPlaces Challenge

Steps:

- Students have to sign up with a team name and the team members to receive a team code and instructions for submitting to the leaderboard. This can be done here.

- After sign-up, upload the prediction results here. Each team is allowed to upload a submission at most every 4 hours.

- The leaderboard is here.

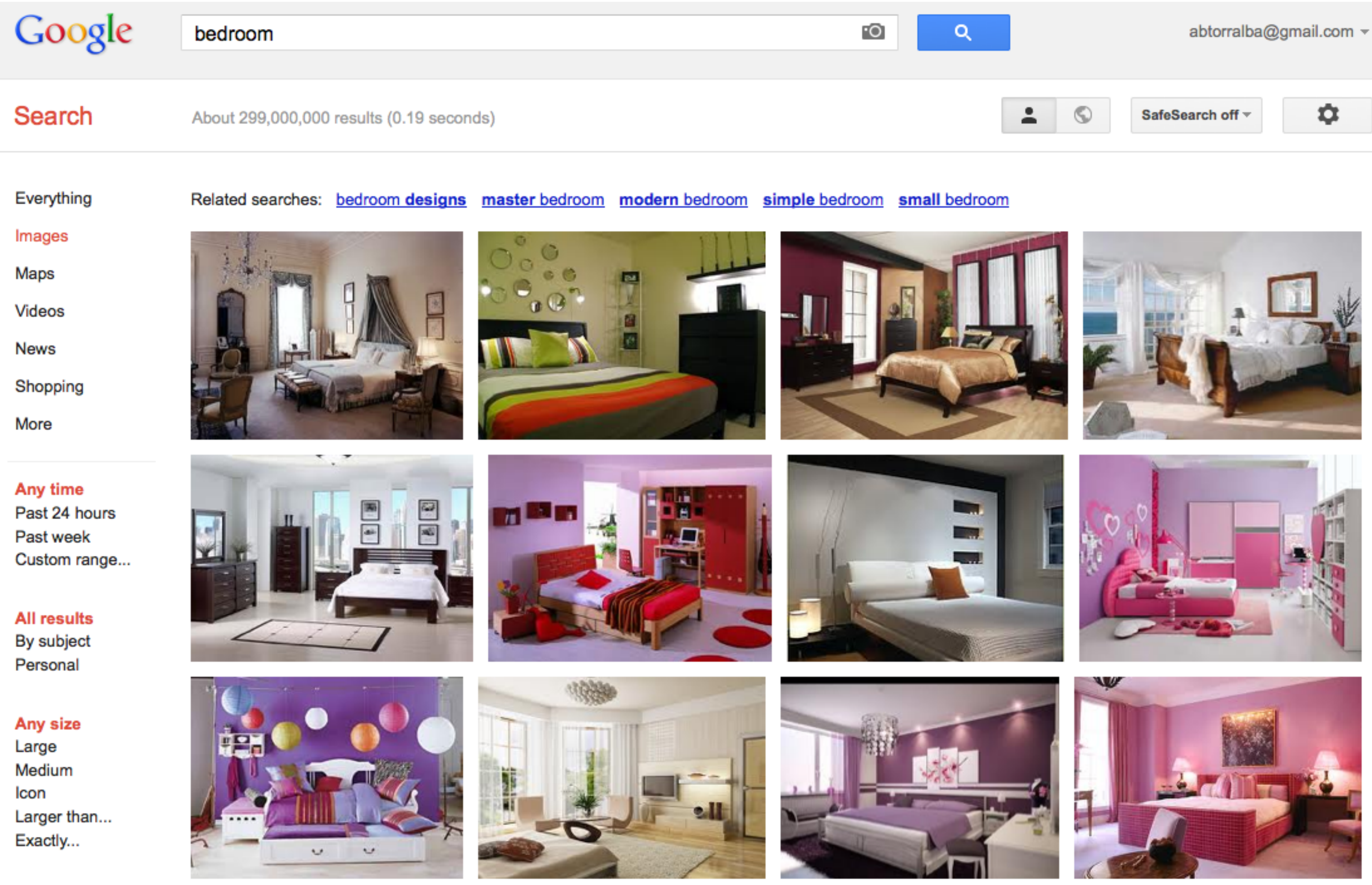
MiniPlaces Challenge

Suggestions:

- Computation: Amazon's EC2
  - Students can receive free $100 credit through the AWS Educate program.

- Model: It is very helpful to go through the examples of training convolutional neural nets in Caffe, TensorFlow, or PyTorch.

- Algorithm: Use data augmentation, deeper layers, or object annotation etc, to boost the classification accuracy.


Google
bedroom
abtorralba@gmail.com
Search
About 299,000,000 results (0.19 seconds)
SafeSearch off
Everything
Images
Maps
Videos
News
Shopping
More
Any time
Past 24 hours
Past week
Custom range...
All results
By subject
Personal
Any size
Large
Medium
Icon
Larger than...
Exactly...
Related searches: bedroom designs master bedroom modern bedroom simple bedroom small bedroom

Google

student bedroom

abtorr

Search

About 66,700,000 results (0.15 seconds)

SafeSearch of

Everything

Images

Maps

Videos

News

Shopping

More

Any time

Past 24 hours

Past week

Custom range...

All results

By subject

Personal

Any size

Large

Medium

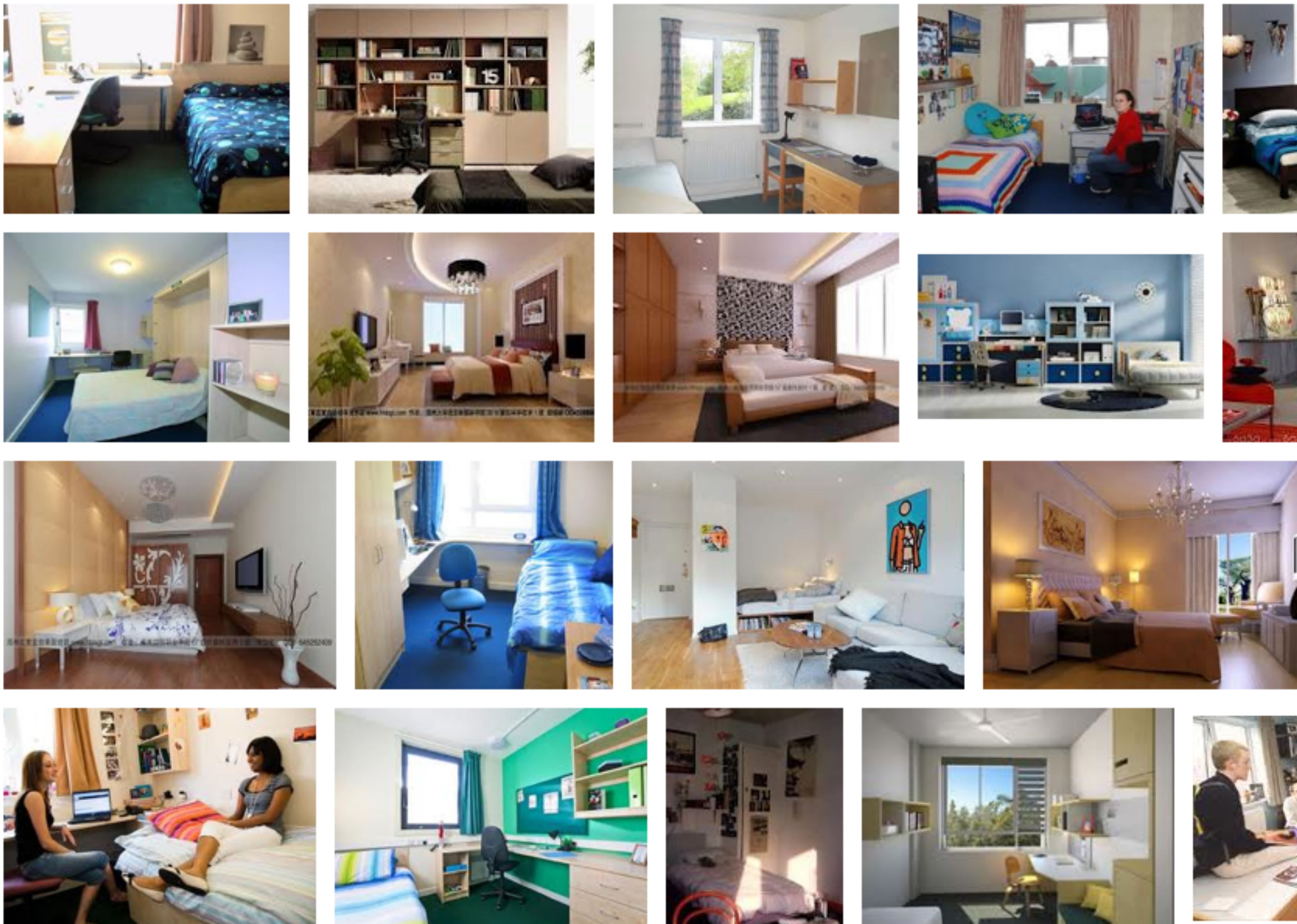
Icon

Larger than...

Exactly...

Any color

Full color

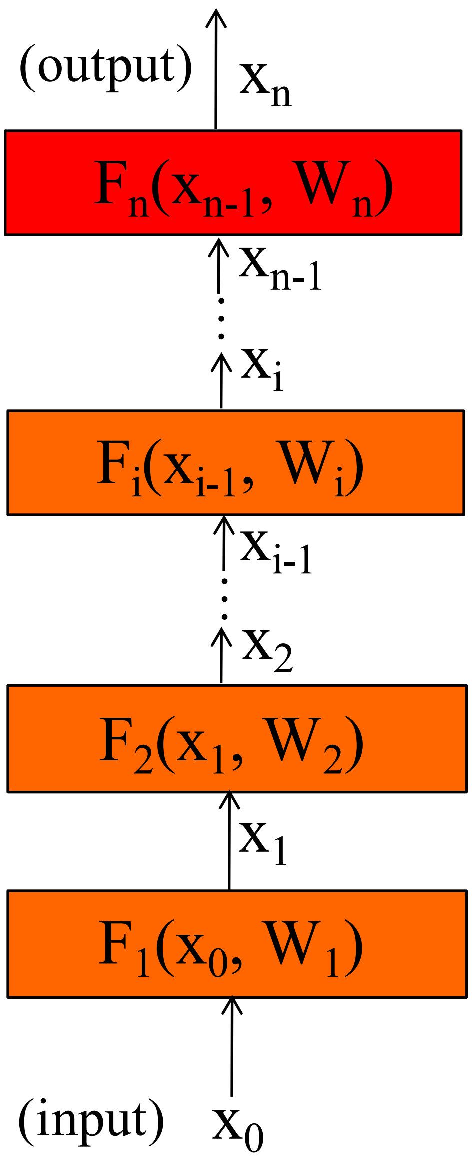



www.bigstock.com · 7067629

)

Cost function

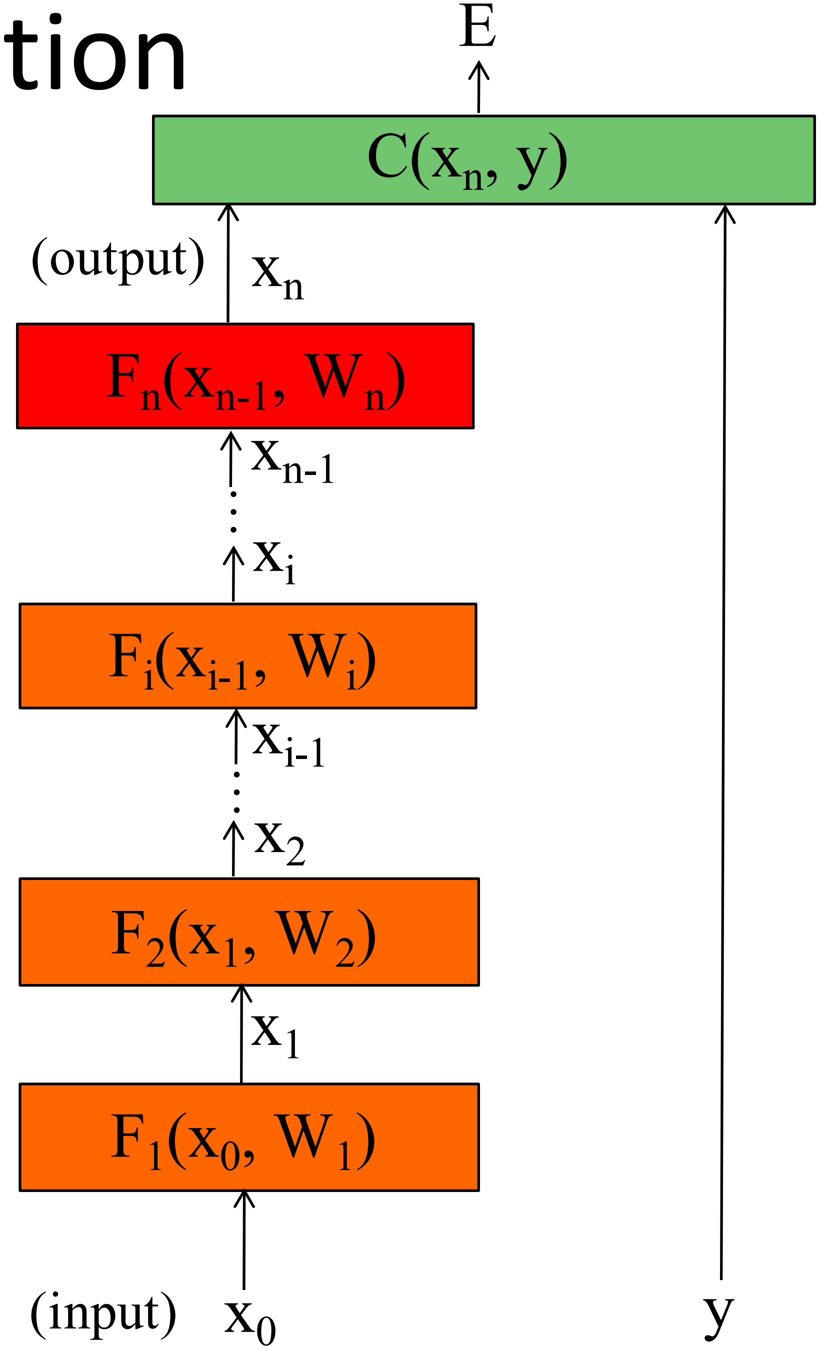
- Consider model with N layers. Layer i has vector of weights Wi.
- **Forward pass**: takes input x and passes it through each layer F_i:
  $$x_i = F_i\ (x_{i-1}, W_i)$$
- Output of layer i is x_i.
- Network output (top layer) is x_n.



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.

- By dessign, 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

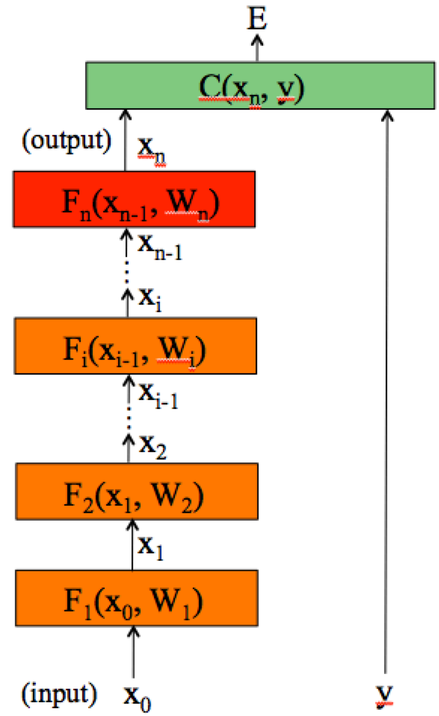
$$\nabla \theta = \frac{\partial E}{\partial \theta}$$

Where $q = \{w_1, w_2, \cdots, w_n\}$

Computing gradients

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

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



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 [m×1], then (*Jacobian formulation*):

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

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

(m rows and n columns)

Matrix calculus

- If y is a scalar and X is a matrix of size [n×m], then

$$\partial y/\partial X = \begin{bmatrix} \partial y/\partial x_{11} & \partial y/\partial x_{21} & \cdots & \partial y/\partial x_{n1} \\ \vdots & \vdots & & \vdots \\ \partial y/\partial x_{1m} & \partial y/\partial x_{12} & \cdots & \partial y/\partial x_{nm} \end{bmatrix}$$

The output is a matrix of size [m×n]

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:

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

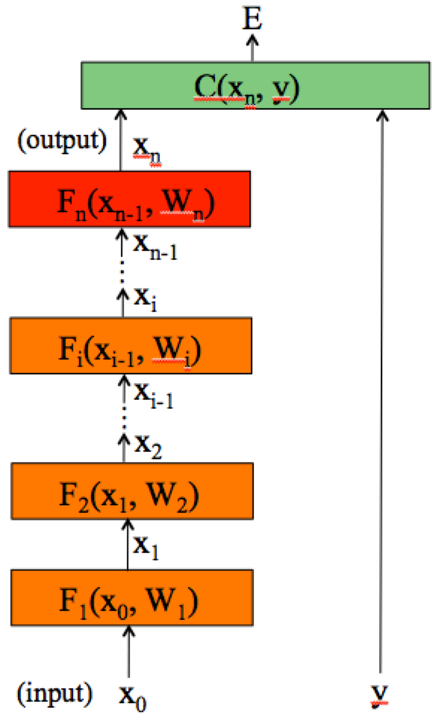
(exercise: check that all the matrix dimensions work fine)

)

Computing gradients

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

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



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

Computing gradients

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

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

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 θ 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 w_n: is the product between how much the cost changes when we change the output of the last layer and 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:

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

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!

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

This is easy.

Backpropagation

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

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

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

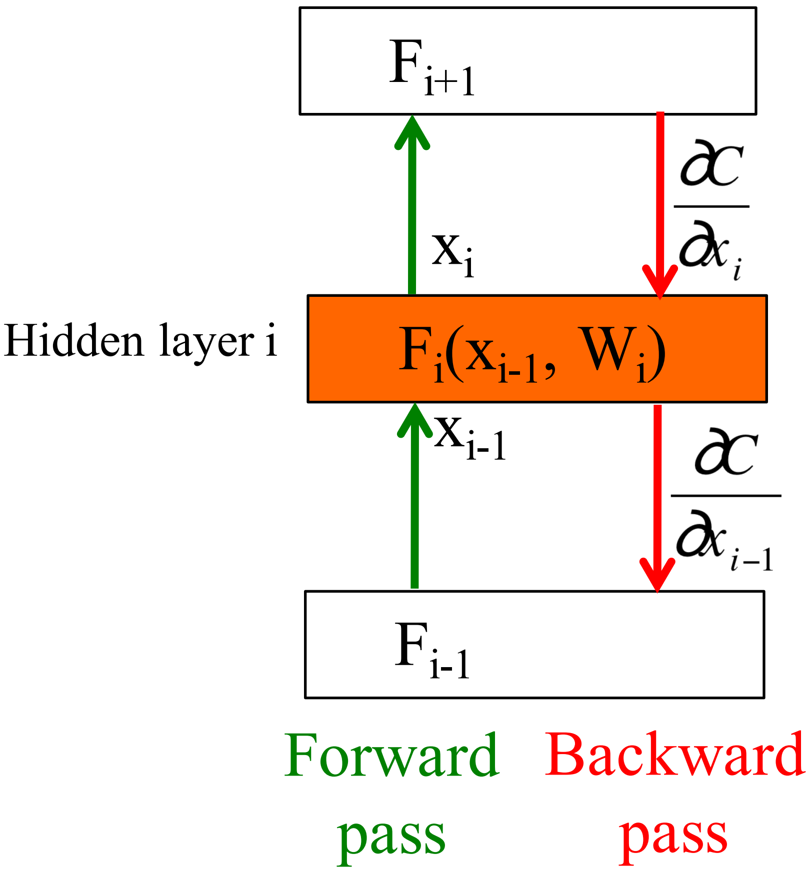
$$\frac{\partial C}{\partial x_{i-1}} = \frac{\partial C}{\partial x_i}\cdot\frac{\partial x_i}{\partial x_{i-1}}$$

Gradient layer i-1

Gradient layer i

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

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:

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

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

(sum over all training examples to get E)

Backpropagation: summary

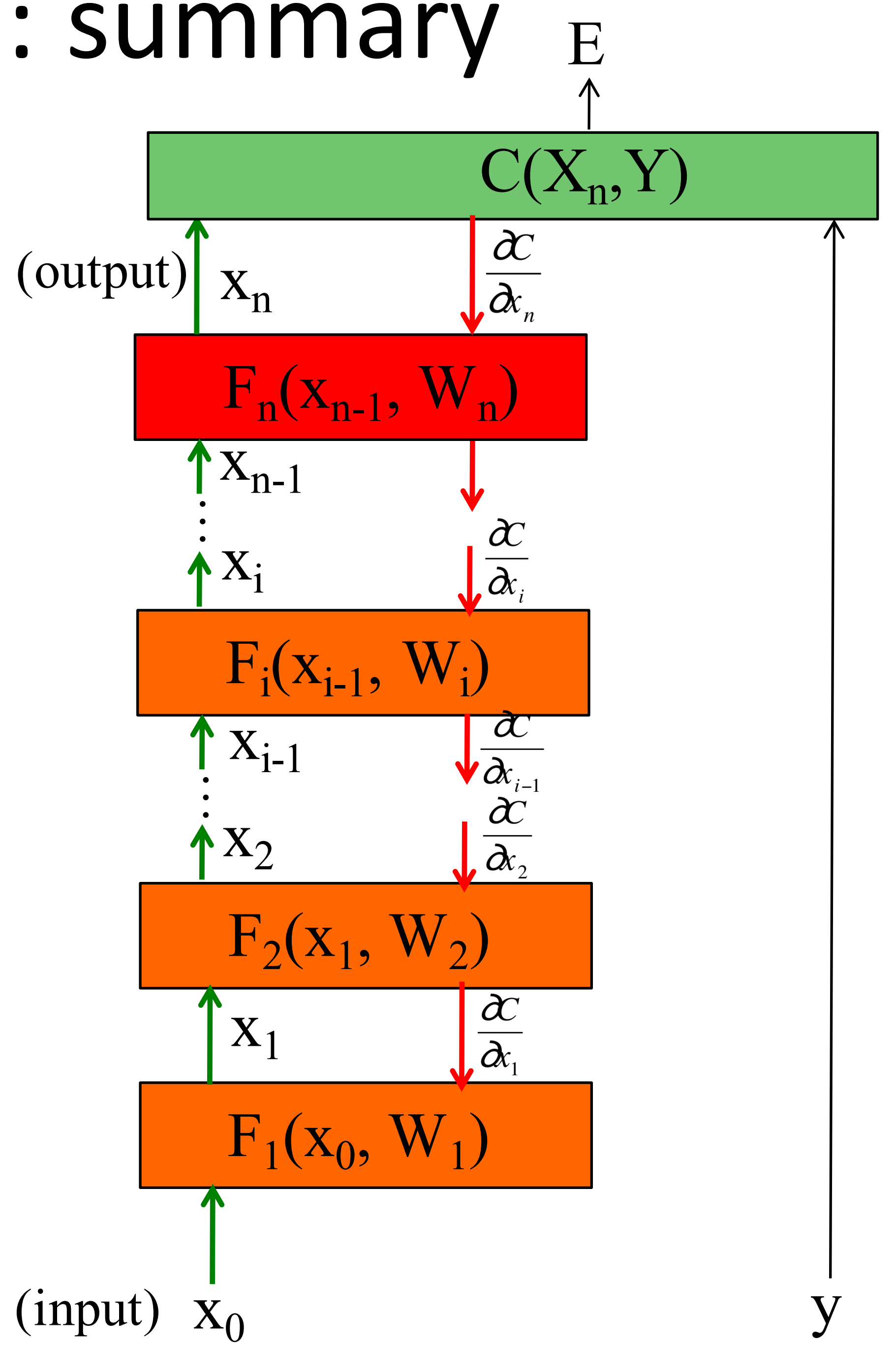
- Forward pass: for each training example. Compute the outputs for all layers

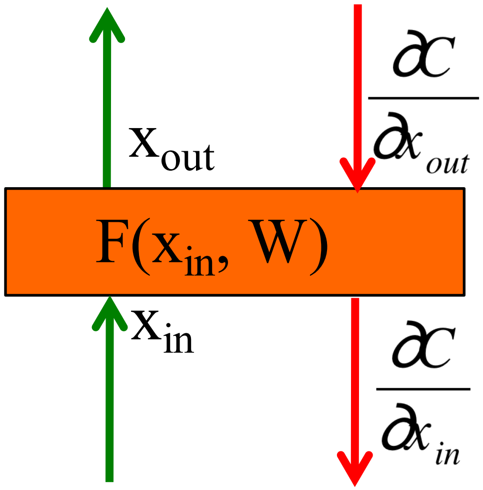
$$x_i = F_i(x_{i-1}, w_i)$$

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

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

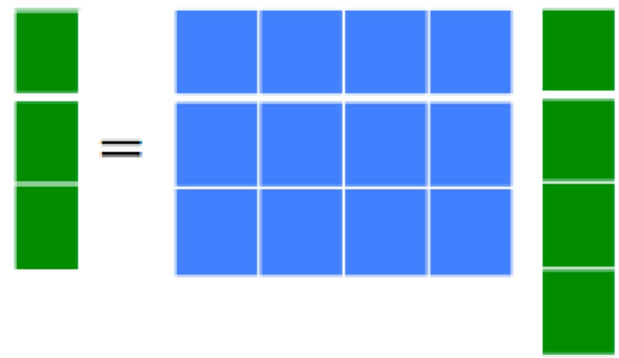
- Compute gradients and update weights.





Linear Module

- Forward propagation: $x_{out} = F(x_{in}, W) = Wx_{in}$



With W being a matrix of size $|x_{out}| \times |x_{in}|$

- Backprop to input:

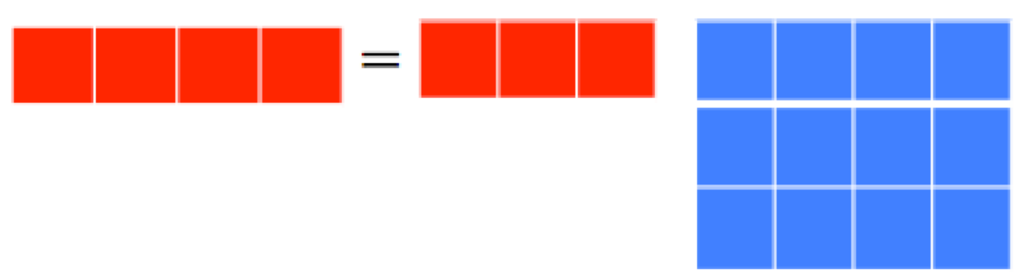
$$\frac{\partial C}{\partial x_{in}} = \frac{\partial C}{\partial x_{out}} \cdot \frac{\partial F(x_{in}, W)}{\partial x_{in}} = \frac{\partial C}{\partial x_{out}} \cdot \frac{\partial x_{out}}{\partial x_{in}}$$

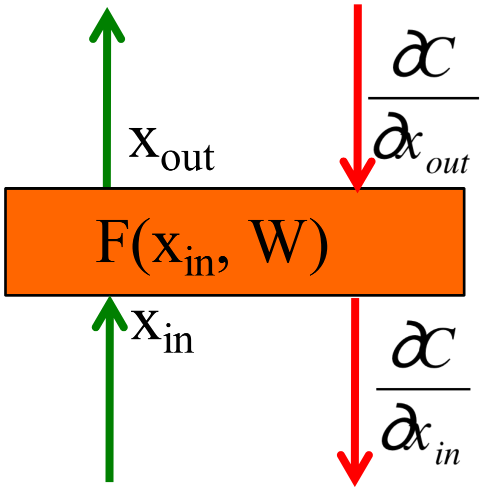
If we look at the j component of output x_{out}, with respect to the i component of the input, x_{in}:

$$\frac{\partial x_{out_i}}{\partial x_{in_j}} = W_{ij} \longrightarrow \frac{\partial F(x_{in}, W)}{\partial x_{in}} = W$$

Therefore:

$$\boxed{\frac{\partial C}{\partial x_{in}} = \frac{\partial C}{\partial x_{out}} \cdot W}$$





Linear Module

- Forward propagation: $x_{out} = F(x_{in}, W) = W x_{in}$
- Backprop to weights:

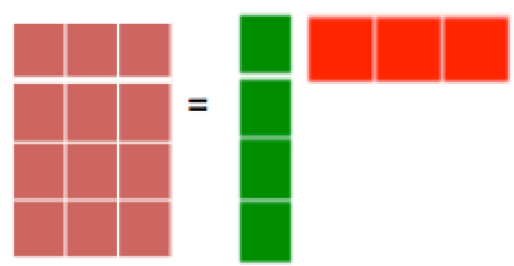
$$\frac{\partial C}{\partial W} = \frac{\partial C}{\partial x_{out}} \cdot \frac{\partial F(x_{in}, W)}{\partial W} = \frac{\partial C}{\partial x_{out}} \cdot \frac{\partial x_{out}}{\partial W}$$

If we look at how the parameter W_{ij} changes the cost, only the i component of the output will change, therefore:

$$\frac{\partial C}{\partial W_{ij}} = \frac{\partial C}{\partial x_{out_i}} \cdot \frac{\partial x_{out_i}}{\partial W_{ij}} = \frac{\partial C}{\partial x_{out_i}} \cdot x_{in_j}$$

$$\frac{\partial x_{out_i}}{\partial W_{ij}} = x_{in_j}$$

$$\boxed{\frac{\partial C}{\partial W} = x_{in} \cdot \frac{\partial C}{\partial x_{out}}}$$

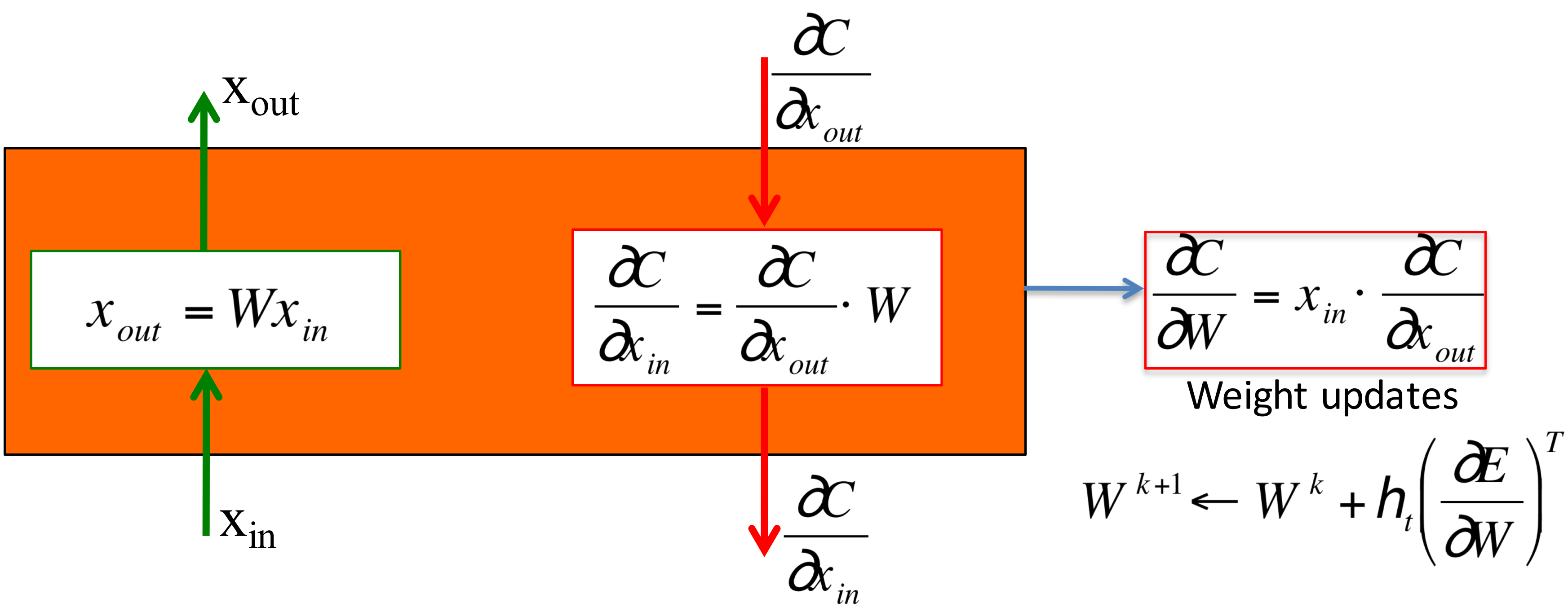


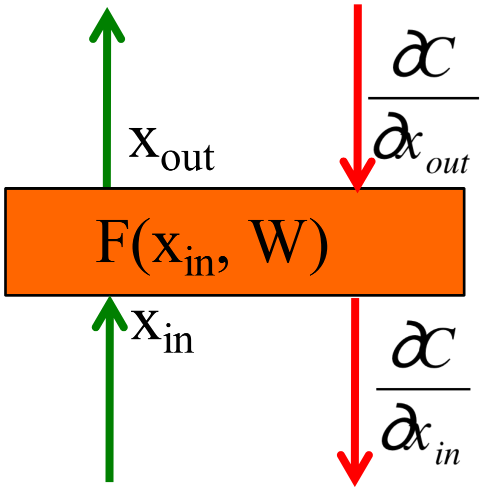
And now we can update the weights (by summing over all the training examples):

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

(sum over all training examples to get E)

Linear Module





Pointwise function

- Forward propagation:

$$x_{out_i} = h(x_{in_i} + b_i)$$

h = an arbitrary function, b_i is a bias term.

- Backprop to input: $$\frac{\partial C}{\partial x_{in_i}} = \frac{\partial C}{\partial x_{out_i}} \cdot \frac{\partial x_{out_i}}{\partial x_{in_i}} = \frac{\partial C}{\partial x_{out_i}} \cdot h'(x_{in_i} + b_i)$$

- Backprop to bias: $$\frac{\partial C}{\partial b_i} = \frac{\partial C}{\partial x_{out_i}} \cdot \frac{\partial x_{out_i}}{\partial b_i} = \frac{\partial C}{\partial x_{out_i}} \cdot h'(x_{in_i} + b_i)$$

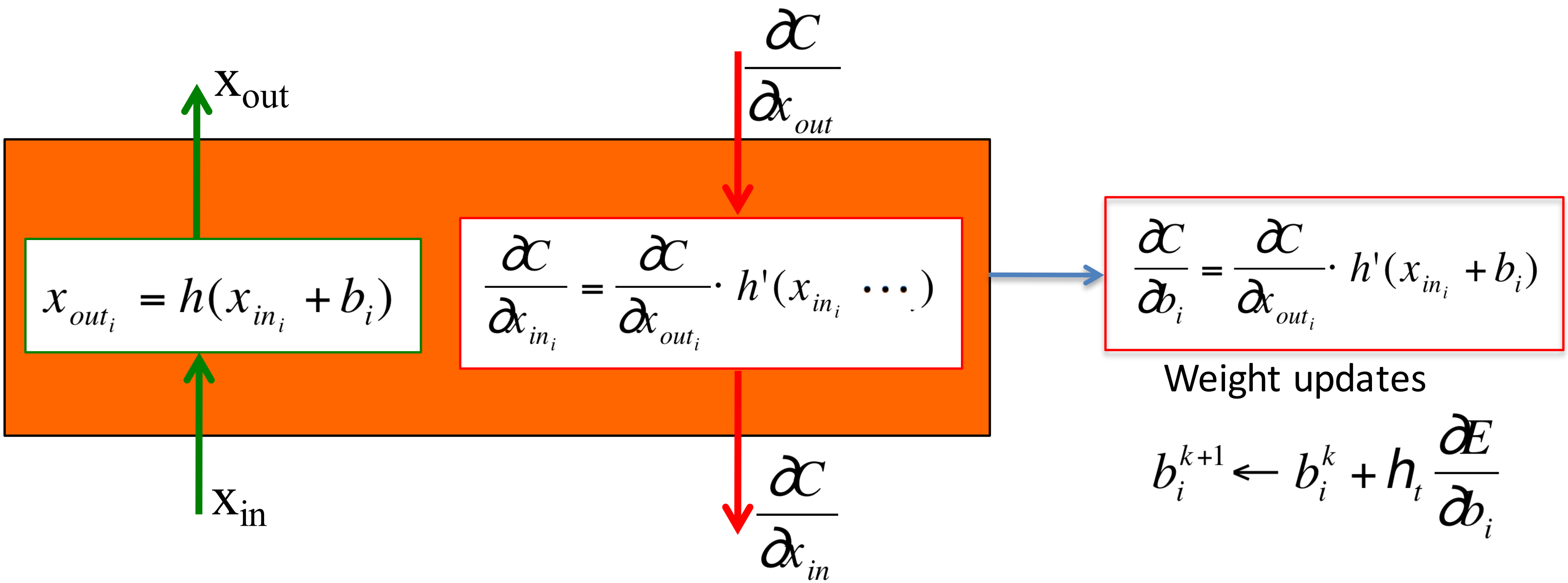
We use this last expression to update the bias.

Some useful derivatives:

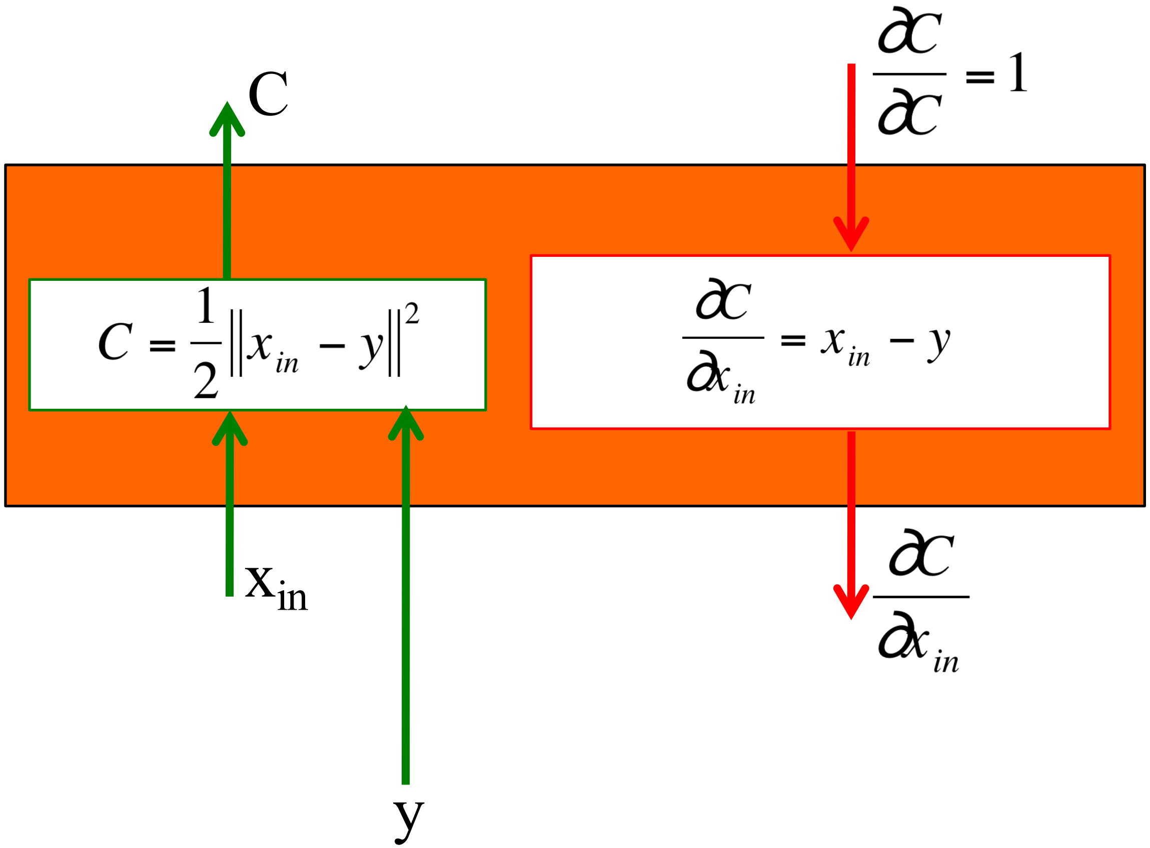
For hyperbolic tangent: $\tanh'(x) = 1 - \tanh^2(x)$

For ReLU: $h(x) = \max(0,x)$ $h'(x) = 1\ [x>0]$

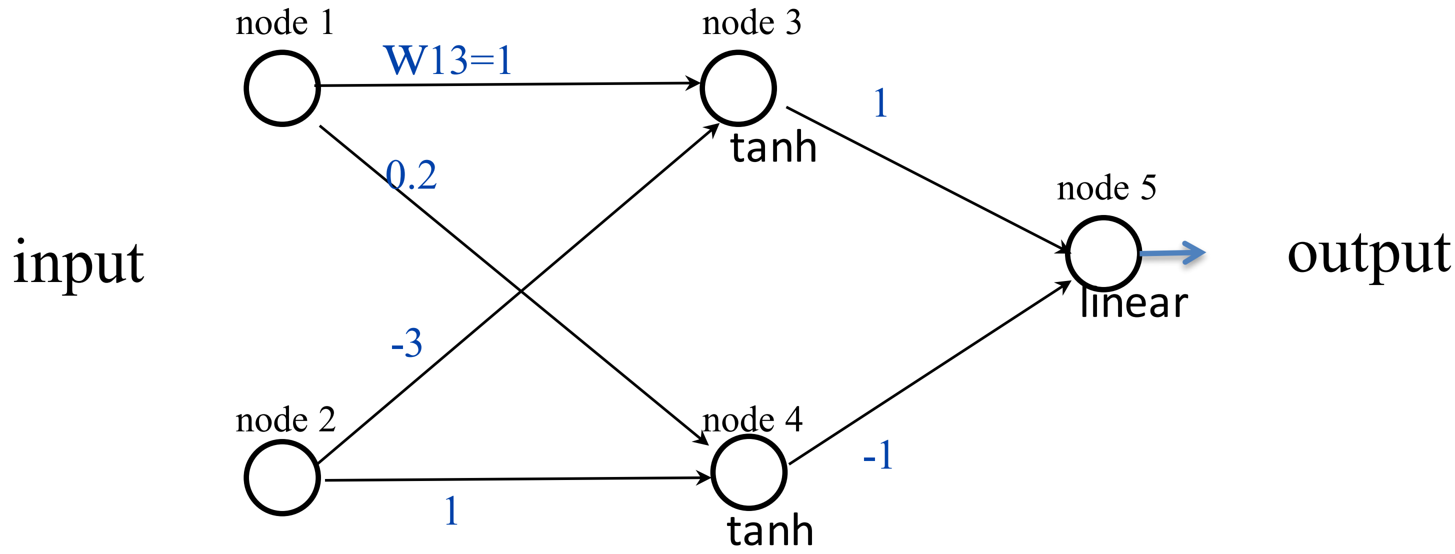
Pointwise function



Euclidean cost module



Back propagation example



Learning rate = -0.2 (because we used positive increments)

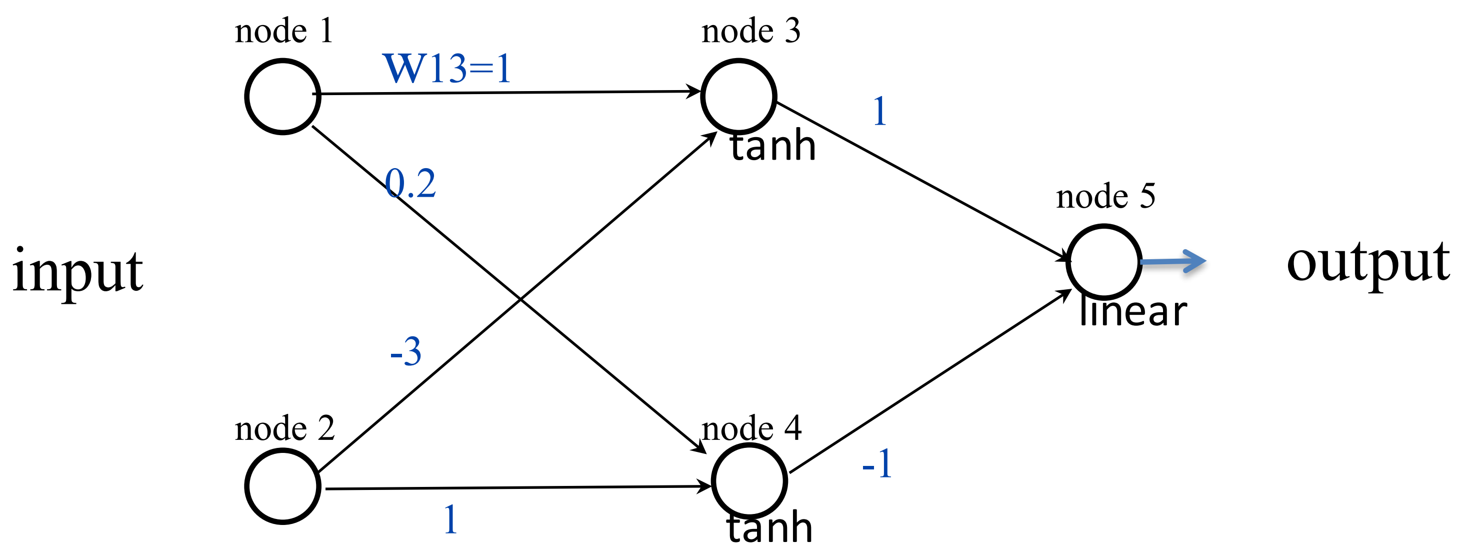
Euclidean loss

Training data:

	input		desired output
	node 1	node 2	node 5
	1.0	0.1	0.5

Exercise: run one iteration of back propagation

Back propagation example



After one iteration (rounding to two digits):