CS 4700:
Foundations of Artificial Intelligence

Prof. Bart Selman
selman@cs.cornell.edu

Machine Learning:
Neural Networks
R&N 18.7

Intro & perceptron learning

Rich history, starting in the early forties.

(McCulloch and Pitts 1943)

(including at least on suspicious death . . .)

Two views:

- **Modeling the brain.**
- **“Just” representation of complex functions.**

(Continuous; contrast decision trees.)

Much progress on both fronts.

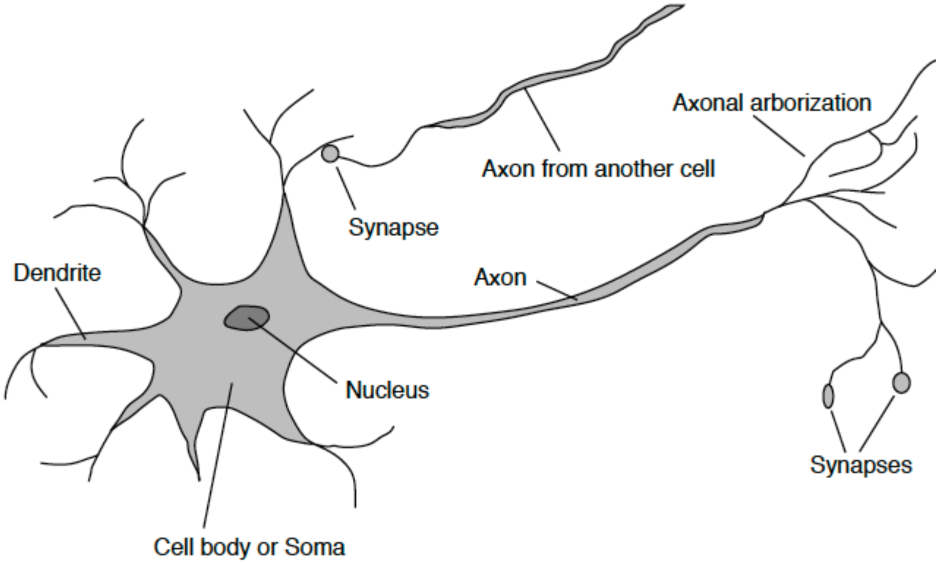
Drawn interests from:

Neuro-science, Cognitive science, AI,

Physics, Statistics, and CS / EE.

Neuron: How the brain works

neurons ~ 100 Billion



Neurons / nerve cells

cell body or **soma**
branches: **dendrited**
single long fiber: **axom**
(100 or more times the diameter of cell body)
axon connects via **synapsis** to dendrites of other c
signals propagated via complicated electrochemical reaction
each cell has a certain electrical potential
when above **threshold**, pulse is sent down axon

synapses can increase (**excitatory**) / decrease (**inhibitory**) potential (signal)
but most importantly: have **plasticity** — can **learn / remember!**
In fact, learning can happen to single cell!
Note: current model gives neuron with little stucture. Complexity arises out of connectivity. Not clear this is "final" model.

Idea: collection of simple cells leads to complex behavior: *thought, action, and consciousness* . . . .
Challenged by e.g. Penrose.
Contrast with current computer design.

Massively Parallel

Neurons: highly parallel computation.

10 to 100 steps — given simple timing constraints, one can deduce that certain visual and other cognitive computations are carried out in about 10 to 100 layers of neurons.

Interesting experiments about how visual features we can detect in parallel.

Appears to need **massive parallelism.**

neurons ~ 100 Billion

Why not build a model like a network of neurons?

	Computer	Human Brain
Computational units	1 CPU, 10^5 gates	10^{11} neurons
Storage units	10^9 bits RAM, 10^{10} bits disk	10^{11} neurons, 10^{14} synapses
Cycle time	10^{-8} sec	10^{-3} sec
Bandwidth	10^9 bits/sec	10^{14} bits/sec
Neuron updates/sec	10^5	10^{14}

Tempting enterprise:

Design computer modeled after the brain.

Good company: Von Neumann (1958)

The Computer and the Brain

But the **connection machine** was not successful

(Hillis 1989 / Thinking Machines)

64K processors.

What was the problem?

R&N:

The exact way in which the brain enables thought is one of the great mysteries of science.

Much recent progress ....

Still, there are skeptics. Especially in CS.

The Skeptic's Position

Related to "levels of abstractions" common in CS.
(less so in EE / Cogn. Sci.)

Consider: Try to figure out how a computer program
performing a heap sort works.

Q. How far would you get with a voltmeter? Wiring diagram?
Possibly the wrong level of abstraction!

Could be similar problem in understanding higher cognition
using FMRI scans!

Still, let's see what neural net research has achieved.

New York Times: "Scientists See Promise in Deep-Learning Programs," Saturday, Nov. 24, front page.

http://www.nytimes.com/2012/11/24/science/scientists-see-advances-in-deep-learning-a-part-of-artificial-intelligence.html?hpw

Multi-layer neural networks, a resurgence!

a) **Winner one of the most recent learning competitions**

b) **Automatic (unsupervised) learning of "cat" and "human face" from 10 million of Google images; 16,000 cores 3 days; multi-layer neural network (Stanford & Google). ImageNet http://image-net.org/**

c) **Speech recognition and real-time translation (Microsoft Research, China).**

Aside: see web site for great survey article "A Few Useful Things to Know About Machine Learning" by Domingos, CACM, 2012.

Start at min. 3:00. Deep Neural Nets in speech recognition.

Artificial Neural Networks

Mathematical abstraction!

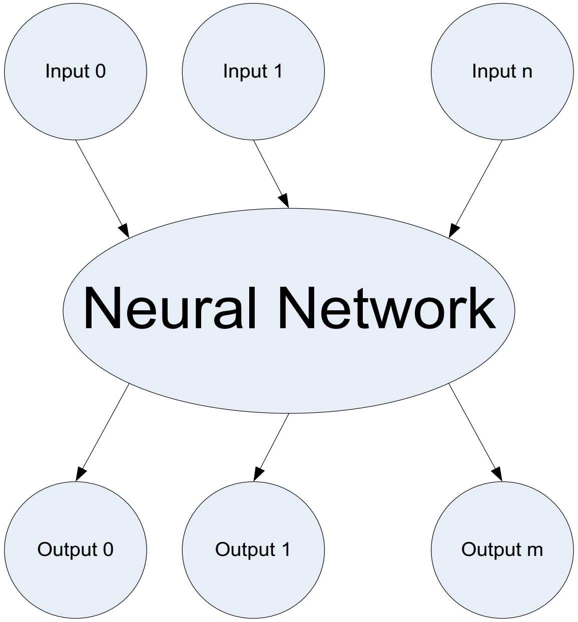
Basic Concepts

A Neural Network maps a set of inputs to a set of outputs

Number of inputs/outputs is variable

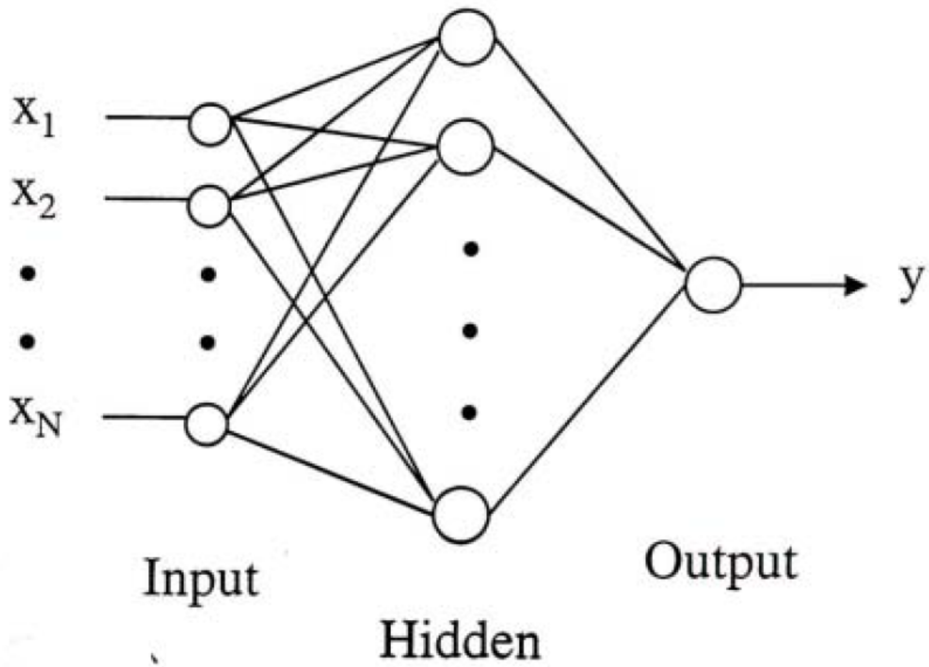
The Network itself is composed of an arbitrary number of nodes or units, connected by links, with an arbitrary topology.

A link from unit i to unit j serves to propagate the activation a_i to j, and it has a weight W_{ij}.



What can a neural networks do?

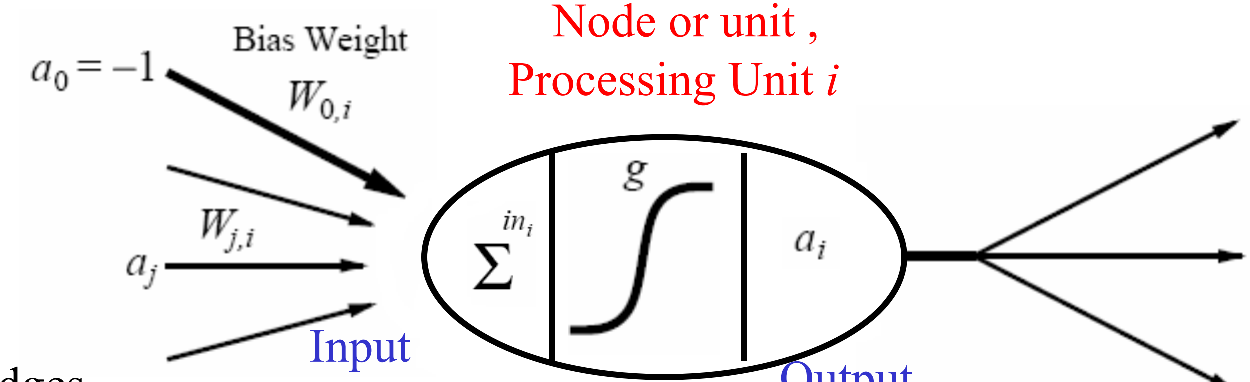
- Compute a known function / Approximate an unknown function
- Pattern Recognition / Signal Processing
- Learn to do any of the above



Different
types of nodes

An Artificial Neuron Node or Unit: A Mathematical Abstraction

Artificial Neuron, Node or unit , Processing Unit *i*



Input edges, each with *weights* (positive, negative, and change over time, learning)

Input function(in_i): weighted sum of its inputs, including fixed input a_0.

$$in_i = \sum_{j=0}^{n} W_{j,i} a_j$$

Activation function (g) applied to input function (typically non-linear).

Output

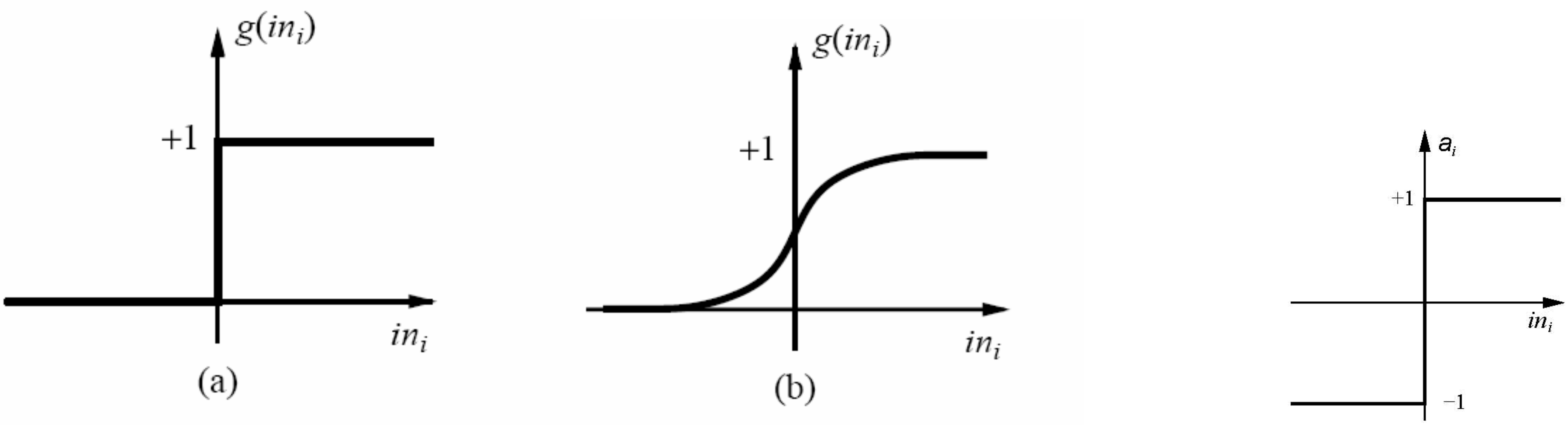
$$a_i = g(\sum_{j=0}^{n} W_{j,i} a_j)$$

Output edges, each with *weights* (positive, negative, and change over time, learning)

→ a processing element producing an output based on a function of its inputs

Note: the fixed input and bias weight are conventional; some authors instead, e.g., or a_0=1 and $-W_{0i}$

Activation Functions



(a) Threshold activation function → a step function or threshold function (outputs 1 when the input is positive; 0 otherwise).

(b) Sigmoid (or logistics function) activation function (key advantage: differentiable) $1/(1+e^{-x})$

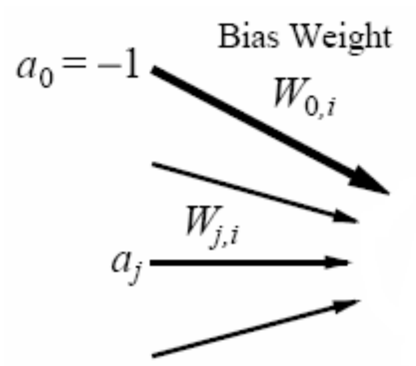
(c) Sign function, +1 if input is positive, otherwise -1.

These functions have a threshold (either hard or soft) at zero.

→ Changing the bias weight $W_{0,i}$ moves the threshold location.

Threshold Activation Function

$g(in_i)$



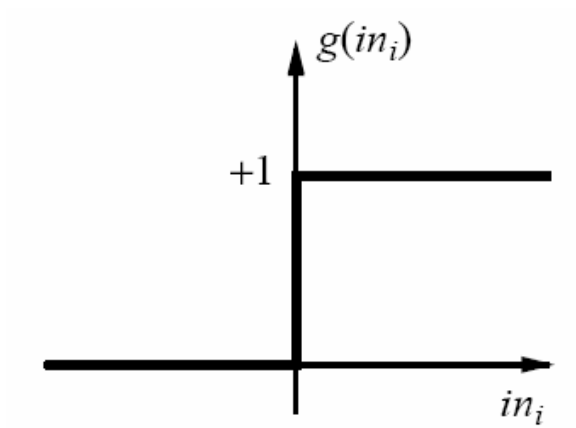
$$in_i = \sum_{j=0}^{n} W_{j,i} a_j > 0; \Leftrightarrow in_i = \sum_{j=1}^{n} w_{j,i} a_j + w_{0,i} a_0 > 0;$$

$$defining\ a_0 = -1\ we\ get \sum_{j=1}^{n} W_{j,i} a_j > w_{0,i},\ \theta_i = w_{0,i}$$

$$defining\ a_0 = 1\ we\ get \sum_{j=1}^{n} W_{j,i} a_j > -w_{0,i},\ \theta_i = -w_{0,i}$$

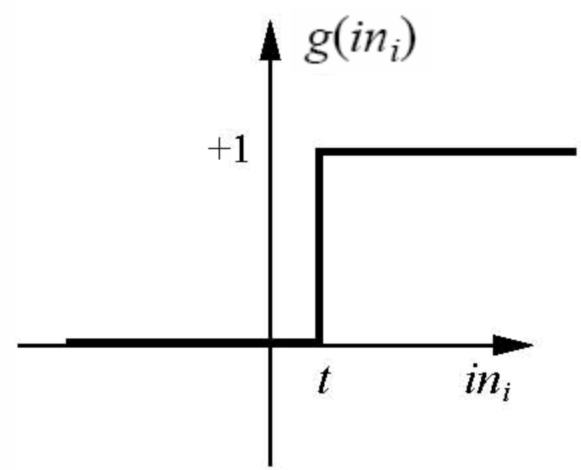
Input edges, each with *weights* (positive, negative, and change over time, learning)

θ_i threshold value associated with unit i



(a)

$\theta_i=0$



$\theta_i=t$

Implementing Boolean Functions

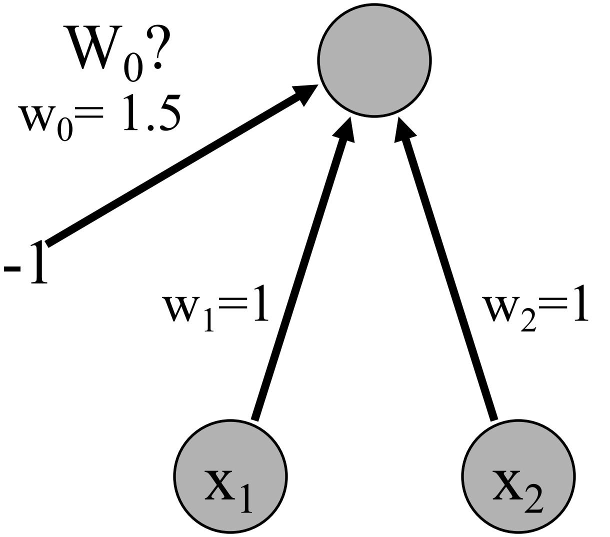
Units with a threshold activation function can act as logic gates; we can use these units to compute Boolean function of its inputs.

Activation of threshold units when:

$$\sum_{j=1}^{n} W_{j,i} a_j > W_{0,i}$$

Boolean AND

input x1	input x2	ouput
0	0	0
0	1	0
1	0	0
1	1	1



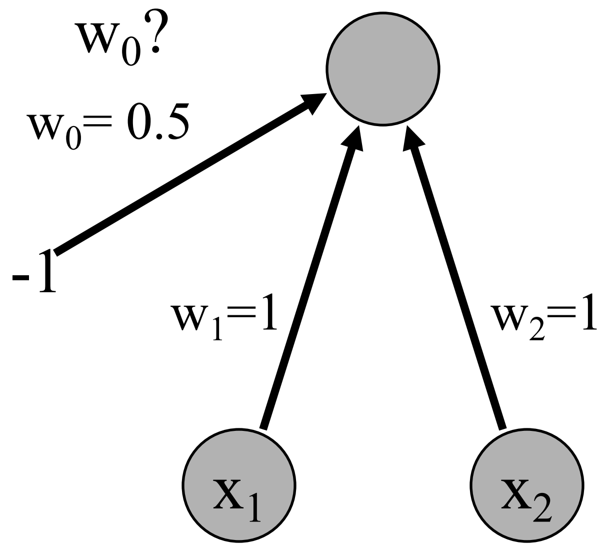
Activation of threshold units when:

$$\sum_{j=1}^{n} W_{j,i} a_j > W_{0,i}$$

What should W_0 be?

Boolean OR

input x1	input x2	ouput
0	0	0
0	1	1
1	0	1
1	1	1



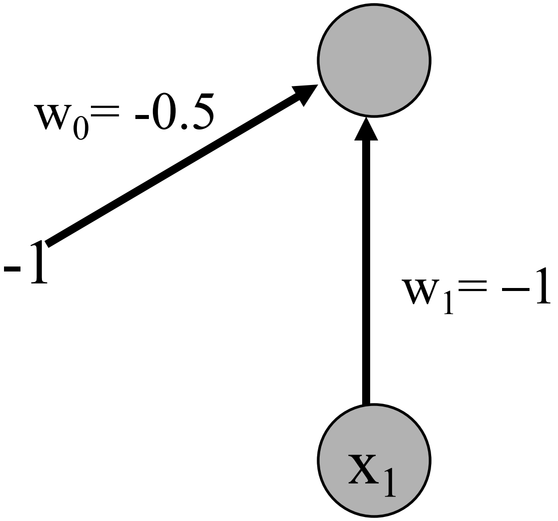
Activation of threshold units when:

$$\sum_{j=1}^{n} W_{j,i} a_j > W_{0,i}$$

What should W_0 be?

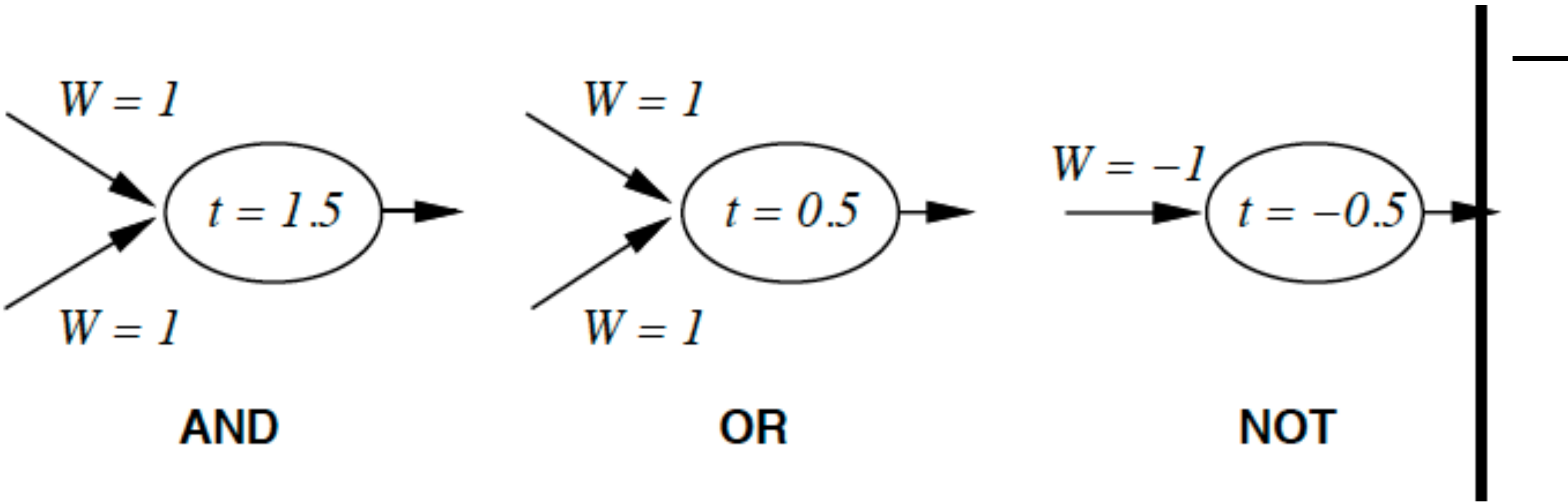
Inverter

input x1	output
0	1
1	0



Activation of threshold units when:

$$\sum_{j=1}^{n} W_{j,i} a_j > W_{0,i}$$


W = 1
t = 1.5
W = 1
AND
W = 1
t = 0.5
W = 1
OR
W = −1
t = −0.5
NOT

Network Structures

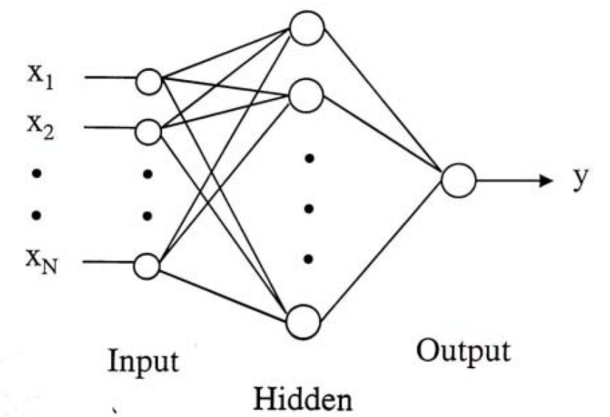
Acyclic or Feed-forward networks

Our focus

Activation flows from input layer to output layer

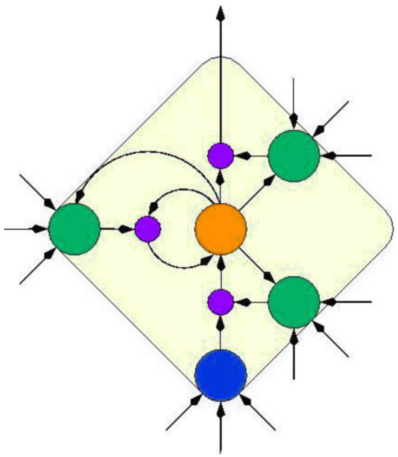
- single-layer perceptrons
- multi-layer perceptrons

Feed-forward networks implement functions, have no internal state (only weights).



Recurrent networks

- Feed the outputs back into own inputs
  →Network is a dynamical system
  (stable state, oscillations, chaotic behavior)
  →Response of the network depends on initial state
- Can support short-term memory
- More difficult to understand



Recurrent Networks

Can capture internal state (activation keeps going around);

→ more complex agents.

Brain cannot be a just a feed-forward network!

Brain has many feed-back connections and cycles

→ brain is a recurrent network!

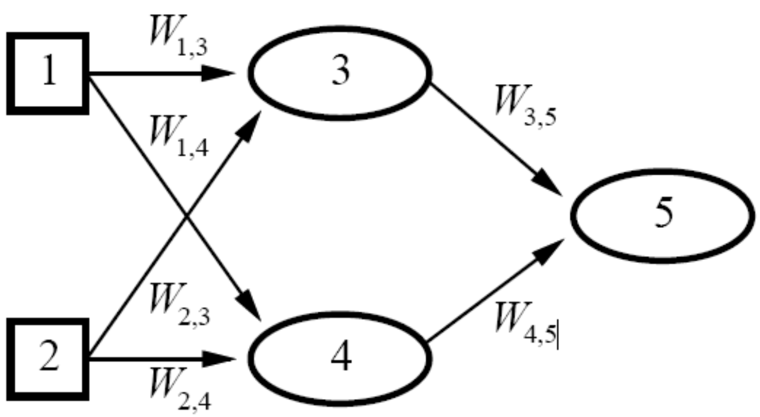
Two key examples:

Hopfield networks:

Boltzmann Machines .

Feed-forward Network: Represents a function of Its Input

Two input units **Two hidden units** **One Output**



(Bias unit omitted for simplicity)

Each unit receives input only from units in the immediately preceding layer.

Given an input vector $x = (x_1,x_2)$, the activations of the input units are set to values of the input vector, i.e., $(a_1,a_2)=(x_1,x_2)$, and the network computes:

Weights are the parameters of the function

$$
\begin{aligned}
a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\
&= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2))
\end{aligned}
$$

Feed-forward network computes a parameterized family of functions $\mathbf{h_W(x)}$

By adjusting the weights we get different functions:

that is how learning is done in neural networks!

Note: the input layer in general does not include computing units.

Perceptron

Cornell Aeronautical Laboratory

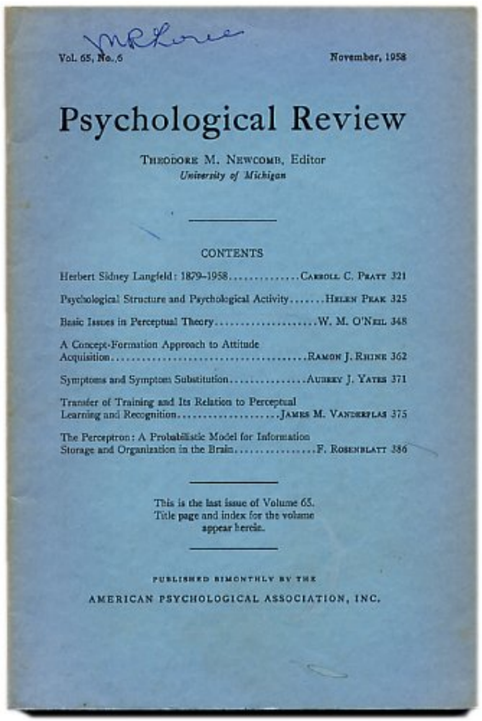
Rosenblatt &
Mark I Perceptron:
the first machine that could "learn" to recognize and identify optical patterns.

Perceptron

- Invented by Frank Rosenblatt in 1957 in an attempt to understand human memory, learning, and cognitive processes.
- The first neural network model by computation, with a remarkable learning algorithm:
  - If function can be represented by perceptron, the learning algorithm is guaranteed to quickly converge to the hidden function!
- Became the foundation of pattern recognition research

One of the earliest and most influential neural networks:
An important milestone in AI.

Perceptron


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.

ROSENBLATT, Frank.
(Cornell Aeronautical Laboratory at Cornell University)

The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain.

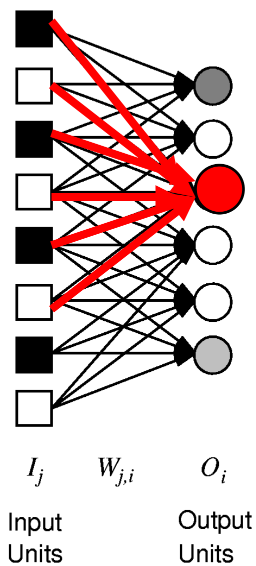
In, Psychological Review, Vol. 65, No. 6, pp. 386-408, November, 1958.

Single Layer Feed-forward Neural Networks
Perceptrons

Single-layer neural network (perceptron network)

A network with all the inputs connected directly to the outputs

–Output units all operate separately: no shared weights

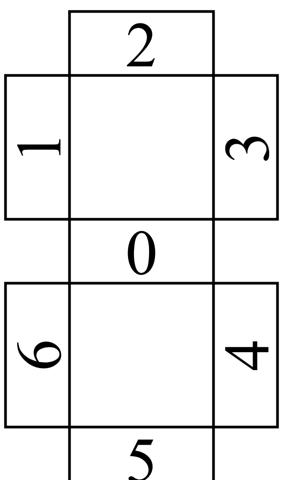


Perceptron Network

Since each output unit is independent of the others, we can limit our study to single output perceptrons.

Perceptron to Learn to Identify Digits
(From Pat. Winston, MIT)

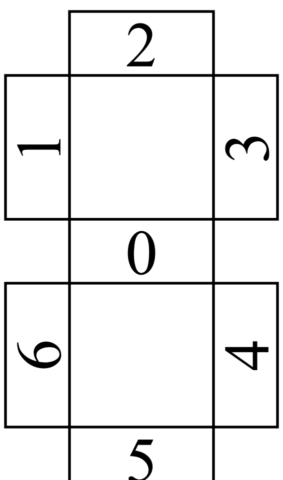
Seven line segments are enough to produce all 10 digits

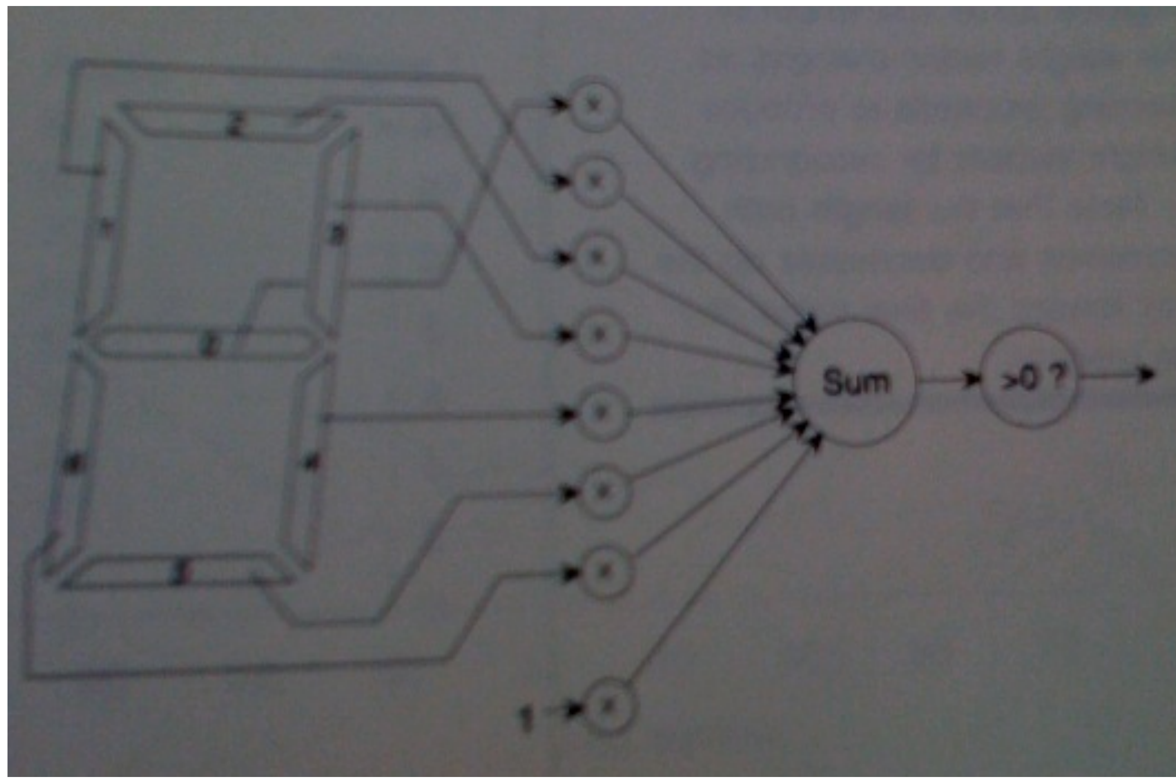


Digit	x_0	x_1	x_2	x_3	x_4	x_5	x_6
0	0	1	1	1	1	1	1
9	1	1	1	1	1	1	0
8	1	1	1	1	1	1	1
7	0	0	1	1	1	0	0
6	1	1	1	0	1	1	1
5	1	1	1	0	1	1	0
4	1	1	0	1	1	0	0
3	1	0	1	1	1	1	0
2	1	0	1	1	0	1	1
1	0	0	0	1	1	0	0

Perceptron to Learn to Identify Digits
(From Pat. Winston, MIT)

Seven line segments are enough to produce all 10 digits



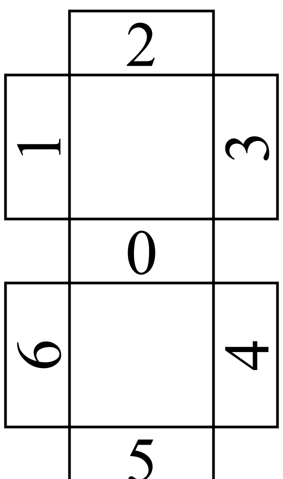


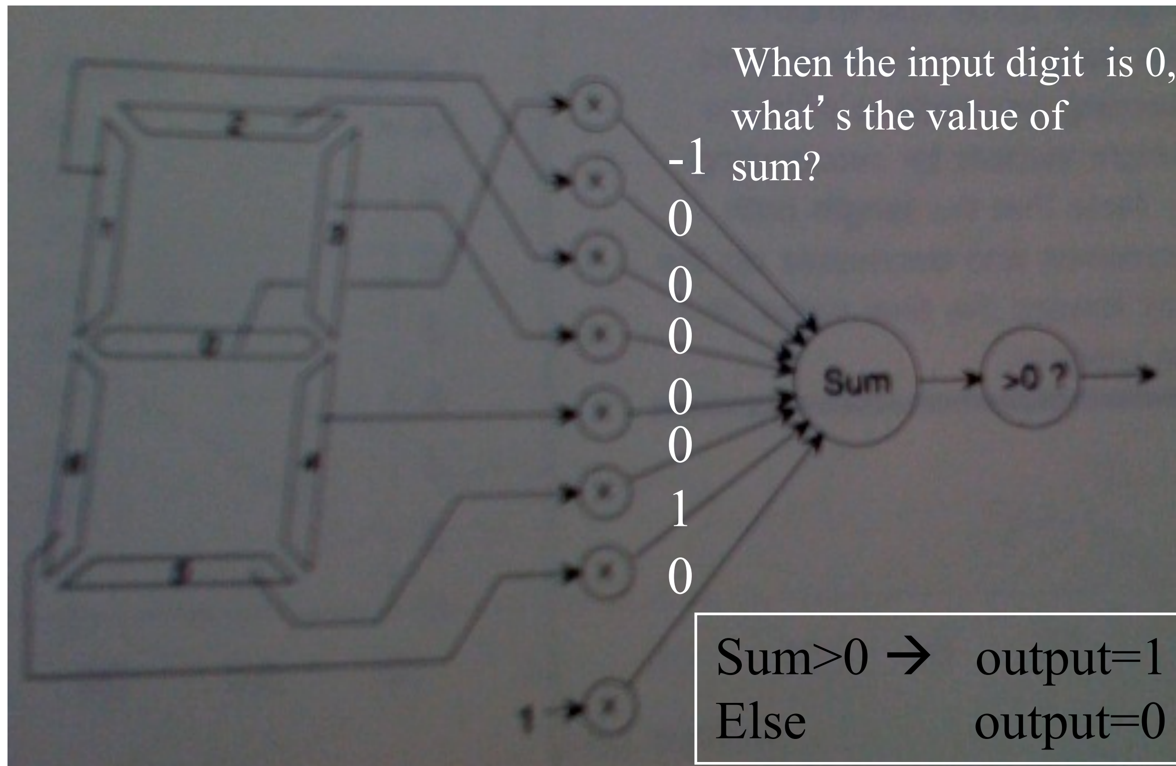
A vision system reports which of the seven segments in the display are on, therefore producing the inputs for the perceptron.

Perceptron to Learn to Identify Digit 0

Digit	x_0	x_1	x_2	x_3	x_4	x_5	x_6	X_7 (fixed input)
0	0	1	1	1	1	1	1	1

Seven line segments are enough to produce all 10 digits





A vision system reports which of the seven segments in the display are on, therefore producing the inputs for the perceptron.

Perceptrons

Remarkable learning algorithm: (Rosenblatt 1960)

if function can be represented by perceptron,

then learning algorithm is guaranteed to quickly converge

to the hidden function!

enormous popularity, early / mid 60's

But analysis by Minsky and Papert (1969)

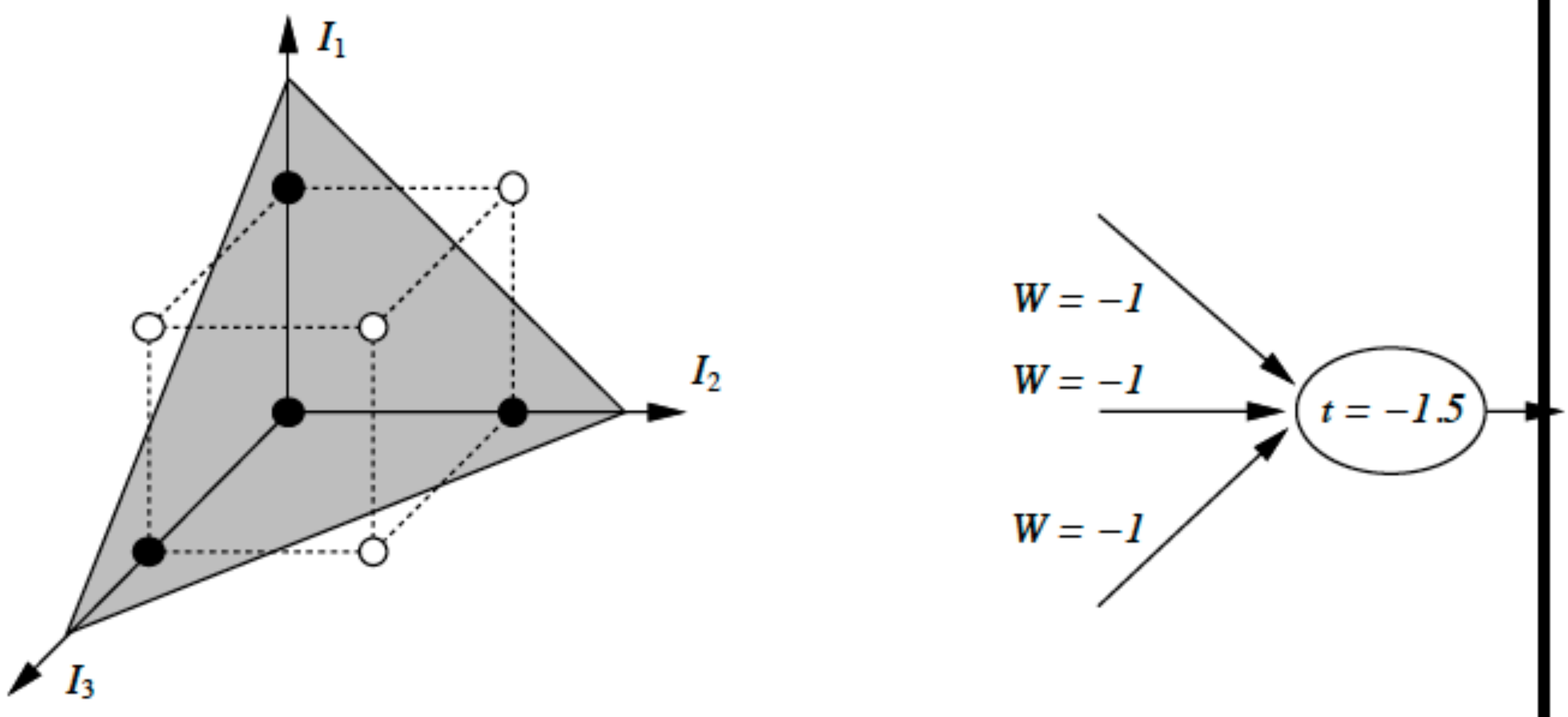
showed certain simple functions cannot be represented

(Boolean XOR)

Killed the field! (and possibly Rosenblatt (rumored)).

But Minsky used a simplified model. Single layer.

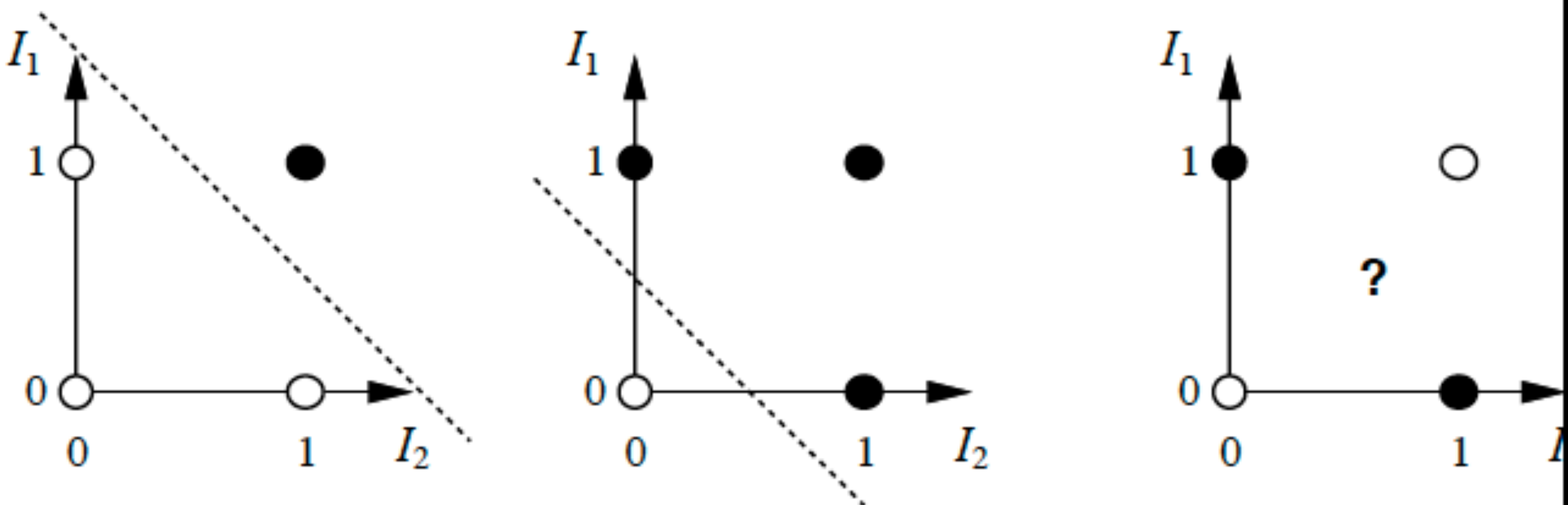
Linearly separable functions only



(a) Separating plane

(b) Weights and threshold

Assume: 0/1 signals. Open circles: "off" or "0". Closed "on" or "1".



(a) I_1 and I_2 (b) I_1 or I_2 (c) I_1 xor I_2

Mid eighties: comeback — multilayed networks

(Turing machine compatible)

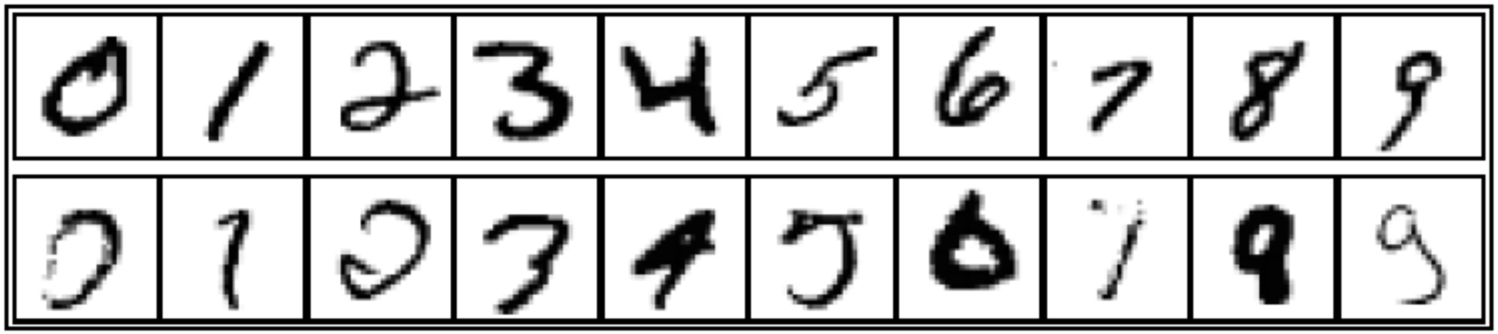
learning procedures: **backpropagation**

Possibly one of the most popular / widely used learning methods today.

John Denker: *"neural nets are the second best thing for learning anything!"* **Update: or perhaps the best!**

backprop and perceptron learning

Handwritten digit recognition



3-nearest-neighbor = 2.4% error
400–300–10 unit MLP = 1.6% error
LeNet: 768–192–30–10 unit MLP = 0.9% error

Current best (kernel machines, vision algorithms) ≈ 0.6% error (more specialized)

Representations

How are concepts represented in the brain / neural net?

local representations / grandmother cell

distributed representations

Pros / Cons?

distributed appeared to have won **but**

UCLA researchers showed (1997)

single cell can learn a concept! (concept: facial expressions / a cell responding to "angry face"!)

Note: can discover hidden features ("regularities") unsupervised with multi-layer networks.

- Neural Net Learning

Perceptron Learning: Intuition

Weight Update

→ Input I_j (j=1,2,…,n)

→ Single output O: target output, T.

Consider some initial weights

Define example error: $Err = T - O$

Now just move weights in right direction!

If the error is positive, then we need to increase O.

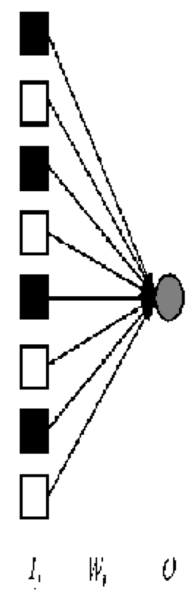
Err >0 → need to increase O;

Err <0 → need to decrease O;

Each input unit j, contributes $W_j I_j$ to total input:

if I_j is positive, increasing W_j tends to increase O;

if I_j is negative, decreasing W_j tends to increase O;



So, use:

$$W_j \leftarrow W_j + \alpha \times I_j \times Err$$

Perceptron Learning Rule (Rosenblatt 1960)

α is the learning rate (for now assume 1).

Perceptron Learning: Simple Example

Let' s consider an example (adapted from Patrick Wintson book, MIT)

Framework and notation:

0/1 signals

Input vector: $\vec{X} = < x_0, x_1, x_2 \cdots, x_n >$

Weight vector: $\vec{W} = < w_0, w_1, w_2 \cdots, w_n >$

$x_0 = 1$ and $\theta_0 = -w_0$, simulate the threshold.

O is output (0 or 1) (single output).

Learning rate = 1.

Threshold function: $S = \sum_{k=0}^{k=n} w_k x_k \quad S > 0 \; then \; O = 1 \quad else \quad O = 0$

$Err = T - O$

$W_j \leftarrow W_j + \alpha \times I_j \times Err$

Perceptron Learning: Simple Example

Set of examples, each example is a pair $(\vec{x}_i, y_i)$ i.e., an input vector and a label y (0 or 1).

This procedure provably converges (polynomial number of steps) if the function is represented by a perceptron (i.e., linearly separable)

Learning procedure, called the "*error correcting method*"

- Start with all zero weight vector.
- Cycle (repeatedly) through examples and for each example do:
  - If perceptron is 0 while it should be 1, add the input vector to the weight vector
  - If perceptron is 1 while it should be 0, subtract the input vector to the weight vector
  - Otherwise do nothing.

← Intuitively correct, (e.g., if output is 0 but it should be 1, the weights are increased) !

Perceptron Learning: Simple Example

Consider learning the logical OR function.

Our examples are:

Sample	x0	x1	x2	label
1	1	0	0	0
2	1	0	1	1
3	1	1	0	1
4	1	1	1	1

Activation Function $$S = \sum_{k=0}^{k=n} w_k x_k \quad S > 0 \; then \; O = 1 \quad else \quad O = 0$$

$$S = \sum_{k=0}^{k=n} w_k x_k \quad S > 0 \textit{ then } O = 1 \quad \textit{else} \quad O = 0$$

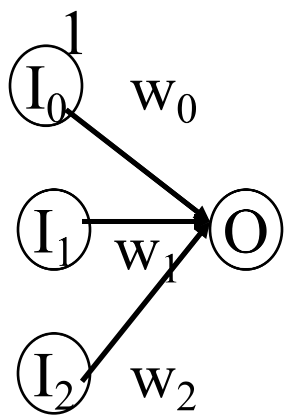
Error correcting method

If perceptron is 0 while it should be 1,
add the input vector to the weight vector
If perceptron is 1 while it should be 0,
subtract the input vector to the weight vector
Otherwise do nothing.

Perceptron Learning: Simple Example

We'll use a single perceptron with three inputs.

We'll start with all weights 0 W= <0,0,0>



Example 1 I= < 1 0 0> label=0 W= <0,0,0>

Perceptron (1×0+ 0×0+ 0×0 =0, S=0) output → 0

→it classifies it as 0, so correct, do nothing

Example 2 I=<1 0 1> label=1 W= <0,0,0>

Perceptron (1×0+ 0×0+ 1×0 = 0) output →0

→it classifies it as 0, while it should be 1, so we add input to weights

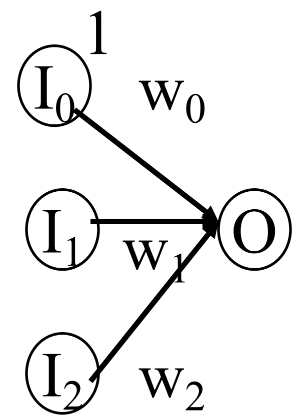
W = <0,0,0> + <1,0,1>= <1,0,1>

Example 3 I=<1 1 0> label=1 W= <1,0,1>

Perceptron ($1\times0+ 1\times0+ 0\times0 > 0$) output = 1

→it classifies it as 1, correct, do nothing

W = <1,0,1>



Example 4 I=<1 1 1> label=1 W= <1,0,1>

Perceptron ($1\times0+ 1\times0+ 1\times0 > 0$) output = 1

→it classifies it as 1, correct, do nothing

W = <1,0,1>

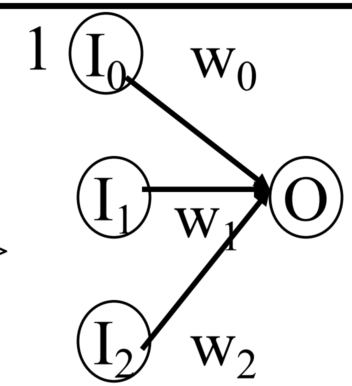
Error correcting method

If perceptron is 0 while it should be 1,
add the input vector to the weight vector

If perceptron is 1 while it should be 0,
subtract the input vector from the weight vector

Otherwise do nothing.

Perceptron Learning: Simple Example



Epoch 2, through the examples, W = <1,0,1> .

Example 1 I = <1,0,0> label=0 W = <1,0,1>

Perceptron (1×1+ 0×0+ 0×1 >0) output → 1

→it classifies it as 1, while it should be 0, so subtract input from weights

W = <1,0,1> - <1,0,0> = <0, 0, 1>

Example 2 I=<1 0 1> label=1 W= <0,0,1>

Perceptron (1×0+ 0×0+ 1×1 > 0) output →1

→it classifies it as 1, so correct, do nothing

Example 3 I=<1 1 0> label=1 W= <0,0,1>

Perceptron (1×0+ 1×0+ 0×1 > 0) output = 0

→it classifies it as 0, while it should be 1, so add input to weights

W = <0,0,1> + W = <1,1,0> = <1, 1, 1>

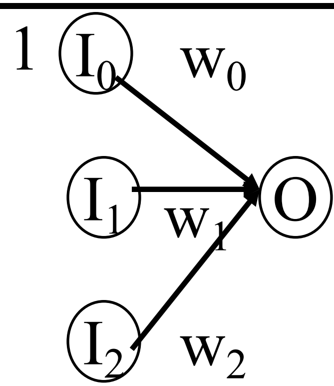
Example 4 I=<1 1 1> label=1 W= <1,1,1>

Perceptron (1×1+ 1×1+ 1×1 > 0) output = 1

→it classifies it as 1, correct, do nothing

W = <1,1,1>

Perceptron Learning: Simple Example



Epoch 3, through the examples, W = <1,1,1> .

Example 1 I=<1,0,0> label=0 W = <1,1,1>

Perceptron ($1 \times 1 + 0 \times 1 + 0 \times 1 > 0$) output → 1

→it classifies it as 1, while it should be 0, so subtract input from weights

W = <1,1,1> - W = <1,0,0> = <0, 1, 1>

Example 2 I=<1 0 1> label=1 W= <0, 1, 1>

Perceptron ($1 \times 0 + 0 \times 1 + 1 \times 1 > 0$) output →1

→it classifies it as 1, so correct, do nothing

Example 3 $I = <1\ 1\ 0>$ label=1 $W = <0, 1, 1>$

Perceptron $(1 \times 0 + 1 \times 1 + 0 \times 1 > 0)$ output = 1

→it classifies it as 1, correct, do nothing

Example 4 $I = <1\ 1\ 1>$ label=1 $W = <0, 1, 1>$

Perceptron $(1 \times 0 + 1 \times 1 + 1 \times 1 > 0)$ output = 1

→it classifies it as 1, correct, do nothing

$W = <1,1,1>$

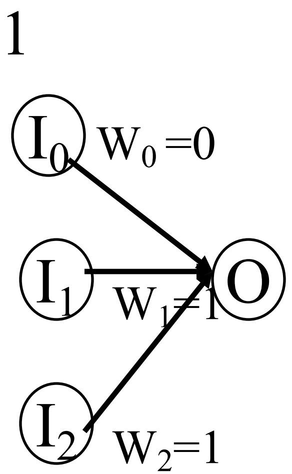
Perceptron Learning: Simple Example

Epoch 4, through the examples, W= <0, 1, 1>.

Example 1 I= <1,0,0> label=0 W = <0,1,1>

Perceptron ($1 \times 0 + 0 \times 1 + 0 \times 1 = 0$) output → 0

→it classifies it as 0, so correct, do nothing



OR

So the final weight vector W= <0, 1, 1> classifies all examples correctly, and the perceptron has learned the function!

Aside: in more realistic cases the bias (W0) will not be 0.
(This was just a toy example!)
Also, in general, many more inputs (100 to 1000)

Epoch	x0	x1	x2	Desired Target	w0	w1	w2	Output	Error	New w0	New w1	New w2
1 example 1	1	0	0	0	0	0	0	0	0	0	0	0

Epoch	x0	x1	x2	Desired Target	w0	w1	w2	Output	Error	New w0	New w1	New w2
1 example 1	1	0	0	0	0	0	0	0	0	0	0	0
example 2	1	0	1	1	0	0	0	0	1	1	0	1

Epoch	x0	x1	x2	Desired Target	w0	w1	w2	Output	Error	New w0	New w1	New w2
1 example 1	1	0	0	0	0	0	0	0	0	0	0	0
example 2	1	0	1	1	0	0	0	0	1	1	0	1
example 3	1	1	0	1	1	0	1	1	0	1	0	1

Epoch	x0	x1	x2	Desired Target	w0	w1	w2	Output	Error	New w0	New w1	New w2
1 example 1	1	0	0	0	0	0	0	0	0	0	0	0
example 2	1	0	1	1	0	0	0	0	1	1	0	1
example 3	1	1	0	1	1	0	1	1	0	1	0	1
example 4	1	1	1	1	1	0	1	1	0	1	0	1

Epoch	x0	x1	x2	Desired Target	w0	w1	w2	Output	Error	New w0	New w1	New w2
1 example 1	1	0	0	0	0	0	0	0	0	0	0	0
example 2	1	0	1	1	0	0	0	0	1	1	0	1
example 3	1	1	0	1	1	0	1	1	0	1	0	1
example 4	1	1	1	1	1	0	1	1	0	1	0	1
2 example 1	1	0	0	0	1	0	1	1	-1	0	0	1

Epoch	x0	x1	x2	Desired Target	w0	w1	w2	Output	Error	New w0	New w1	New w2
1 example 1	1	0	0	0	0	0	0	0	0	0	0	0
example 2	1	0	1	1	0	0	0	0	1	1	0	1
example 3	1	1	0	1	1	0	1	1	0	1	0	1
example 4	1	1	1	1	1	0	1	1	0	1	0	1
2 example 1	1	0	0	0	1	0	1	1	-1	0	0	1
example 2	1	0	1	1	0	0	1	1	0	0	0	1

Epoch	x0	x1	x2	Desired Target	w0	w1	w2	Output	Error	New w0	New w1	New w2
1 example 1	1	0	0	0	0	0	0	0	0	0	0	0
example 2	1	0	1	1	0	0	0	0	1	1	0	1
example 3	1	1	0	1	1	0	1	1	0	1	0	1
example 4	1	1	1	1	1	0	1	1	0	1	0	1
2 example 1	1	0	0	0	1	0	1	1	-1	0	0	1
example 2	1	0	1	1	0	0	1	1	0	0	0	1
example 3	1	1	0	1	0	0	1	0	1	1	1	1

Epoch	x0	x1	x2	Desired Target	w0	w1	w2	Output	Error	New w0	New w1	New w2
1 example 1	1	0	0	0	0	0	0	0	0	0	0	0
example 2	1	0	1	1	0	0	0	0	1	1	0	1
example 3	1	1	0	1	1	0	1	1	0	1	0	1
example 4	1	1	1	1	1	0	1	1	0	1	0	1
2 example 1	1	0	0	0	1	0	1	1	-1	0	0	1
example 2	1	0	1	1	0	0	1	1	0	0	0	1
example 3	1	1	0	1	0	0	1	0	1	1	1	1
example 4	1	1	1	1	1	1	1	1	0	1	1	1

Epoch	x0	x1	x2	Desired Target	w0	w1	w2	Output	Error	New w0	New w1	New w2
1 example 1	1	0	0	0	0	0	0	0	0	0	0	0
example 2	1	0	1	1	0	0	0	0	1	1	0	1
example 3	1	1	0	1	1	0	1	1	0	1	0	1
example 4	1	1	1	1	1	0	1	1	0	1	0	1
2 example 1	1	0	0	0	1	0	1	1	-1	0	0	1
example 2	1	0	1	1	0	0	1	1	0	0	0	1
example 3	1	1	0	1	0	0	1	0	1	1	1	1
example 4	1	1	1	1	1	1	1	1	0	1	1	1
3 example 1	1	0	0	0	1	1	1	1	-1	0	1	1

Epoch	x0	x1	x2	Desired Target	w0	w1	w2	Output	Error	New w0	New w1	New w2
1 example 1	1	0	0	0	0	0	0	0	0	0	0	0
example 2	1	0	1	1	0	0	0	0	1	1	0	1
example 3	1	1	0	1	1	0	1	1	0	1	0	1
example 4	1	1	1	1	1	0	1	1	0	1	0	1
2 example 1	1	0	0	0	1	0	1	1	-1	0	0	1
example 2	1	0	1	1	0	0	1	1	0	0	0	1
example 3	1	1	0	1	0	0	1	0	1	1	1	1
example 4	1	1	1	1	1	1	1	1	0	1	1	1
3 example 1	1	0	0	0	1	1	1	1	-1	0	1	1
example 2	1	0	1	1	0	1	1	1	0	0	1	1

Epoch	x0	x1	x2	Desired Target	w0	w1	w2	Output	Error	New w0	New w1	New w2
1 example 1	1	0	0	0	0	0	0	0	0	0	0	0
example 2	1	0	1	1	0	0	0	0	1	1	0	1
example 3	1	1	0	1	1	0	1	1	0	1	0	1
example 4	1	1	1	1	1	0	1	1	0	1	0	1
2 example 1	1	0	0	0	1	0	1	1	-1	0	0	1
example 2	1	0	1	1	0	0	1	1	0	0	0	1
example 3	1	1	0	1	0	0	1	0	1	1	1	1
example 4	1	1	1	1	1	1	1	1	0	1	1	1
3 example 1	1	0	0	0	1	1	1	1	-1	0	1	1
example 2	1	0	1	1	0	1	1	1	0	0	1	1
example 3	1	1	0	1	0	1	1	1	0	0	1	1

Epoch	x0	x1	x2	Desired Target	w0	w1	w2	Output	Error	New w0	New w1	New w2
1 example 1	1	0	0	0	0	0	0	0	0	0	0	0
example 2	1	0	1	1	0	0	0	0	1	1	0	1
example 3	1	1	0	1	1	0	1	1	0	1	0	1
example 4	1	1	1	1	1	0	1	1	0	1	0	1
2 example 1	1	0	0	0	1	0	1	1	-1	0	0	1
example 2	1	0	1	1	0	0	1	1	0	0	0	1
example 3	1	1	0	1	0	0	1	0	1	1	1	1
example 4	1	1	1	1	1	1	1	1	0	1	1	1
3 example 1	1	0	0	0	1	1	1	1	-1	0	1	1
example 2	1	0	1	1	0	1	1	1	0	0	1	1
example 3	1	1	0	1	0	1	1	1	0	0	1	1
example 4	1	1	1	1	0	1	1	1	0	0	1	1

Epoch	x0	x1	x2	Desired Target	w0	w1	w2	Output	Error	New w0	New w1	New w2
1 example 1	1	0	0	0	0	0	0	0	0	0	0	0
example 2	1	0	1	1	0	0	0	0	1	1	0	1
example 3	1	1	0	1	1	0	1	1	0	1	0	1
example 4	1	1	1	1	1	0	1	1	0	1	0	1
2 example 1	1	0	0	0	1	0	1	1	-1	0	0	1
example 2	1	0	1	1	0	0	1	1	0	0	0	1
example 3	1	1	0	1	0	0	1	0	1	1	1	1
example 4	1	1	1	1	1	1	1	1	0	1	1	1
3 example 1	1	0	0	0	1	1	1	1	-1	0	1	1
example 2	1	0	1	1	0	1	1	1	0	0	1	1
example 3	1	1	0	1	0	1	1	1	0	0	1	1
example 4	1	1	1	1	0	1	1	1	0	0	1	1
4 example 1	1	0	0	0	0	1	1	0	0	0	1	1

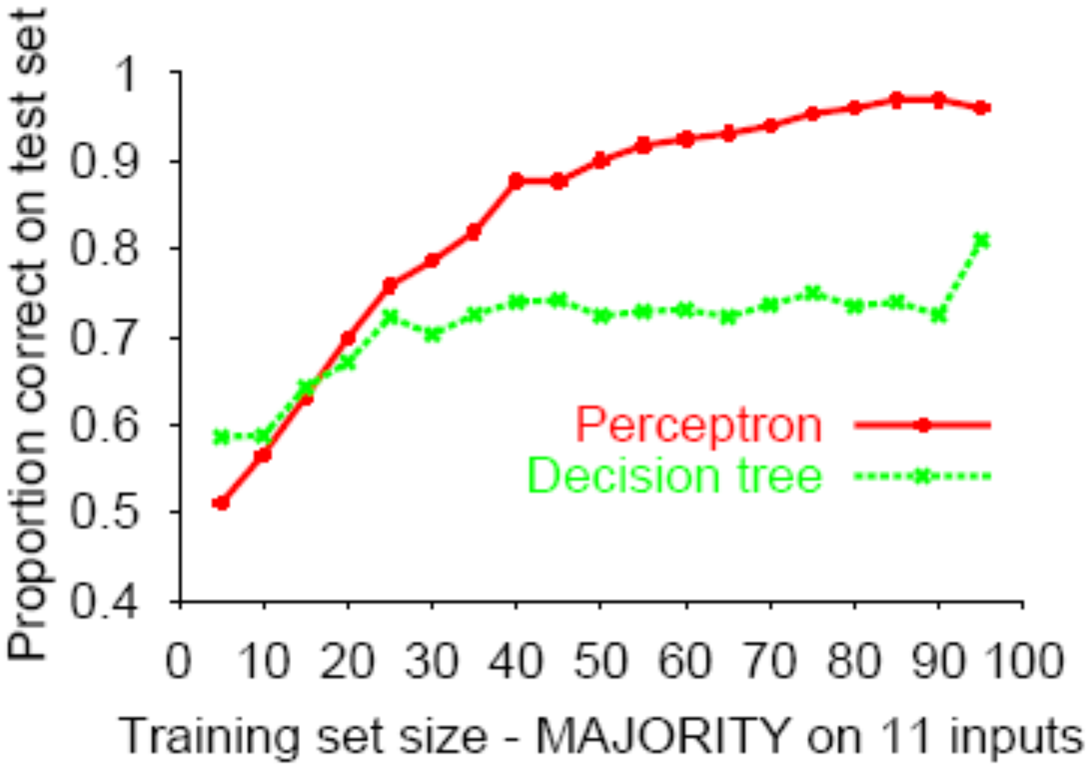
Convergence of Perceptron Learning Algorithm

Perceptron converges to a consistent function, if…

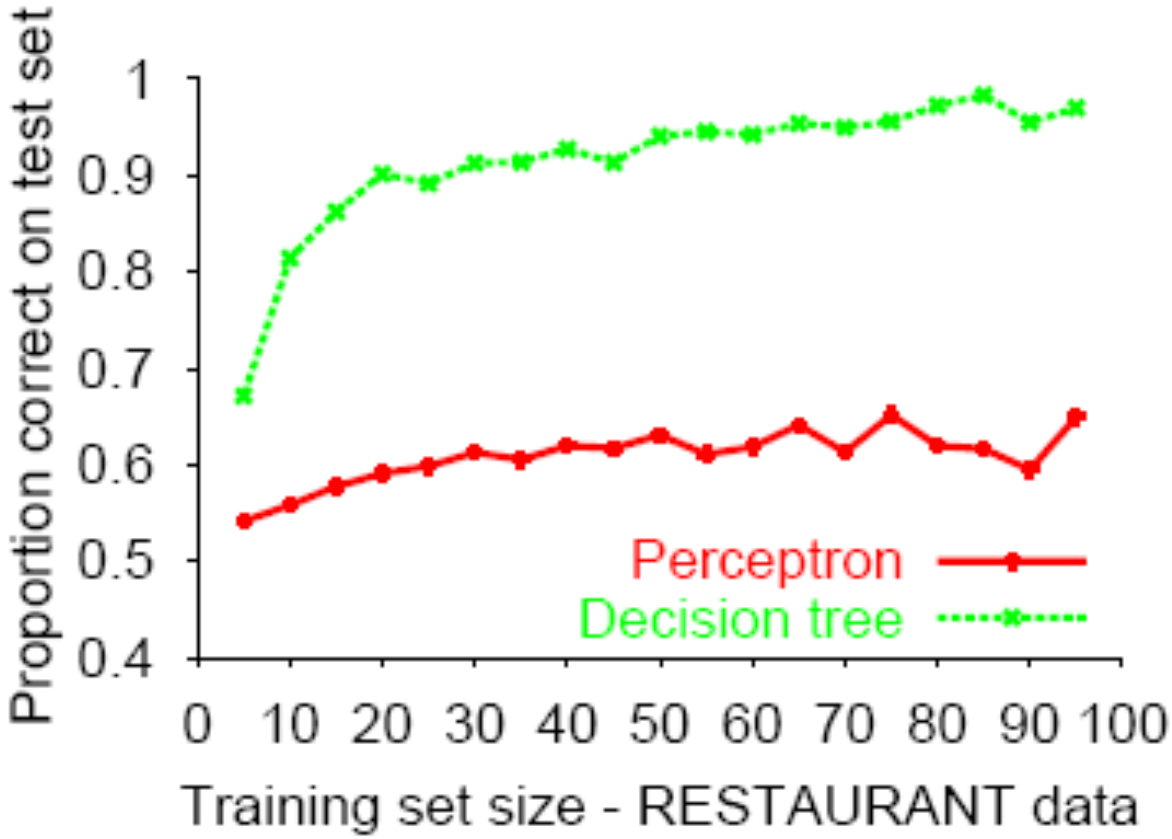
… training data linearly separable

… step size α sufficiently small

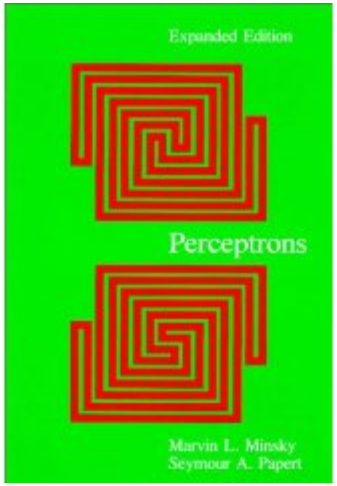
… no "hidden" units



Perceptron learns majority function easily, DTL is hopeless



DTL learns restaurant function easily, perceptron cannot represent it



Good news: Adding hidden layer allows more target functions to be represented.

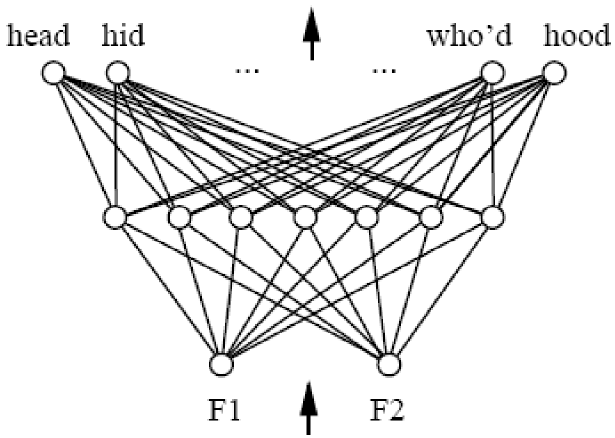
Minsky & Papert (1969)

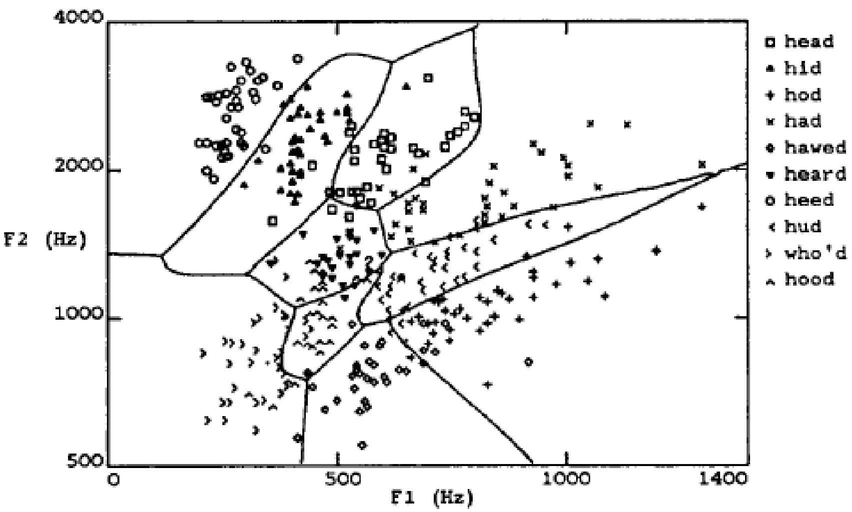
Multi-layer Perceptrons (MLPs)

Single-layer perceptrons can only represent linear decision surfaces.

Multi-layer perceptrons can represent non-linear decision surfaces.

Hidden Layer





Output units

The choice of how to represent the output then determines the form of the cross-entropy function

1. Linear output $z = W^T h + b$. Often used as mean of Gaussian distribution.

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I}).$$

2. Sigmoid function for Bernoulli distribution. Output $P(y=1 \mid x)$

Output units

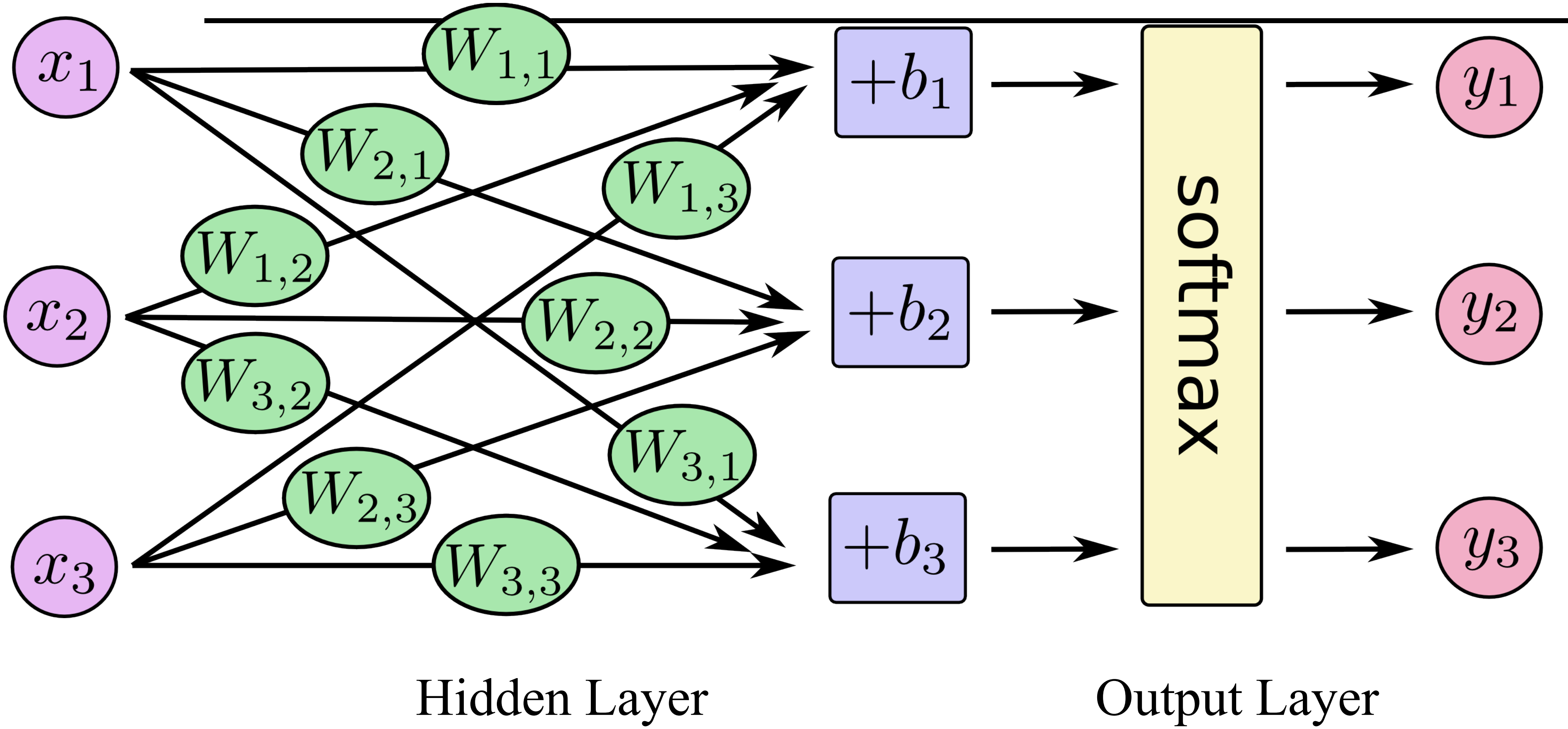
3. Softmax for Multinoulli output distributions. Predict a vector, each element being P(y = i| x)

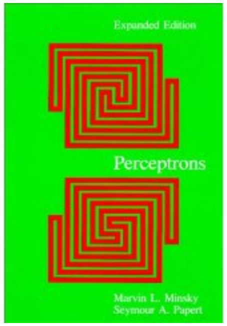
A linear layer

$$\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b},$$

Softmax function

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}.$$


x_1
x_2
x_3
$W_{1,1}$
$W_{2,1}$
$W_{1,3}$
$W_{1,2}$
$W_{2,2}$
$W_{3,2}$
$W_{3,1}$
$W_{2,3}$
$W_{3,3}$
$+b_1$
$+b_2$
$+b_3$
softmax
y_1
y_2
y_3
Hidden Layer
Output Layer



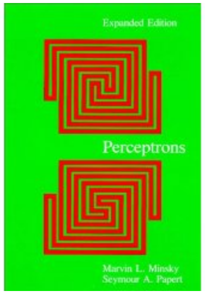
Bad news: No algorithm for learning in multi-layered networks, and no convergence theorem was known in 1969!

Minsky & Papert (1969) *"[The perceptron] has many features to attract attention: its linearity; its intriguing learning theorem; its clear paradigmatic simplicity as a kind of parallel computation. There is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgment that the extension is sterile."*

Minsky & Papert (1969) pricked the neural network balloon …they almost killed the field.

Rumors say these results may have killed Rosenblatt….

Winter of Neural Networks 69-86.



Two major problems they saw were

1. How can the learning algorithm apportion credit (or blame) to individual weights for incorrect classifications depending on a (sometimes) large number of weights?

2. How can such a network learn useful higher-order features?

Back Propagation - Next

Good news: Successful credit-apportionment learning algorithms developed soon afterwards (e.g., back-propagation). Still successful, in spite of lack of convergence theorem.