CS 4700: Foundations of Artificial Intelligence

Prof. Bart Selman
selman@cs.cornell.edu

Machine Learning:
Neural Networks
R&N 18.7

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.**

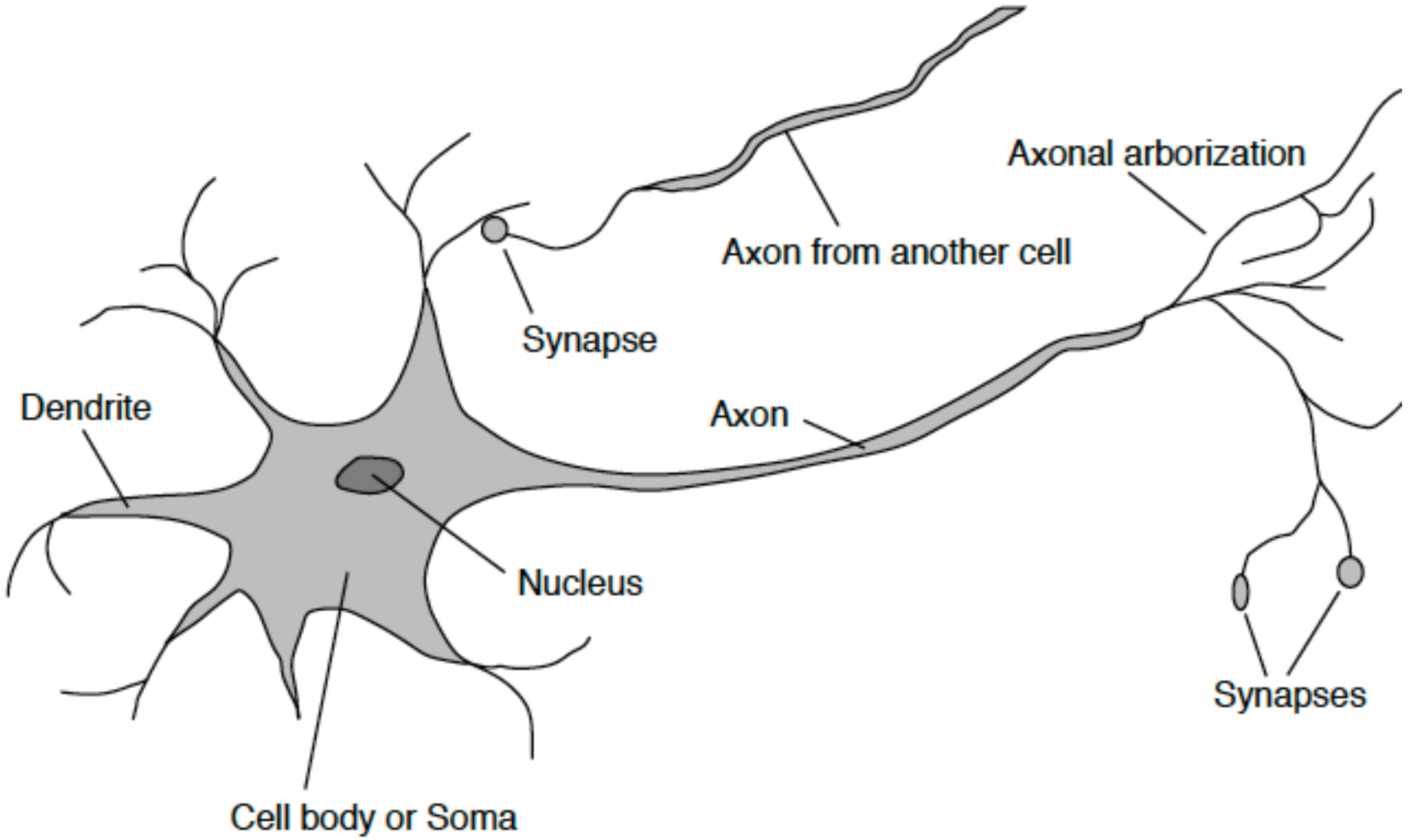
(Continuous; contrast decision trees.)

Much progress on both fronts.

Drawn interests from:

Neuro-science, Cognitive science, AI,

Physics, Statistics, and CS / EE.


Axonal arborization
Axon from another cell
Synapse
Dendrite
Axon
Nucleus
Synapses
Cell body or Soma

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 cells

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.**

	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).**
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!

We have: **units**, connected by **links**.

Each link has a **weight**.

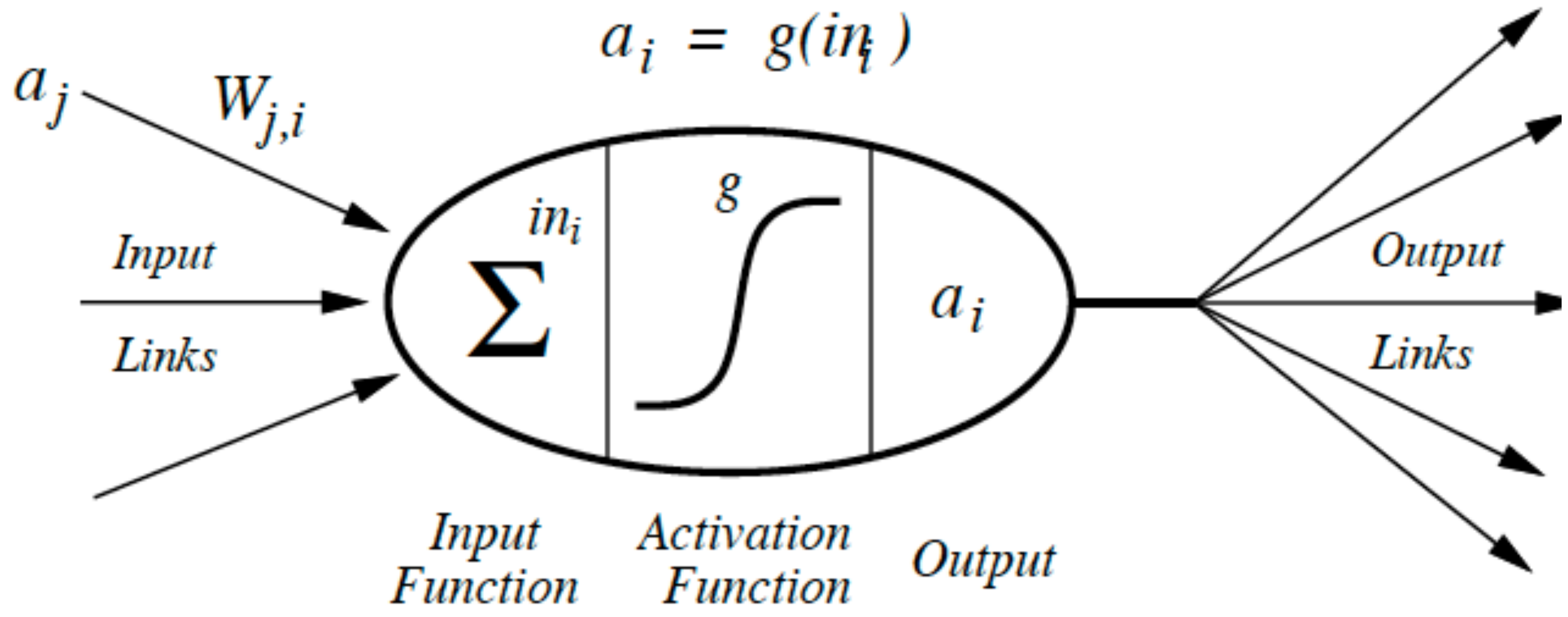
The primary means for long-term storage. (plasticity)

Later: our **learning** algorithms will modify these weights.

What about modifying the connectivity? ("rewiring" the brain ...).

Each unit has set of inputs links from other units set of output links to other units and an **activation level**.

A means to compute activation level at next time step.

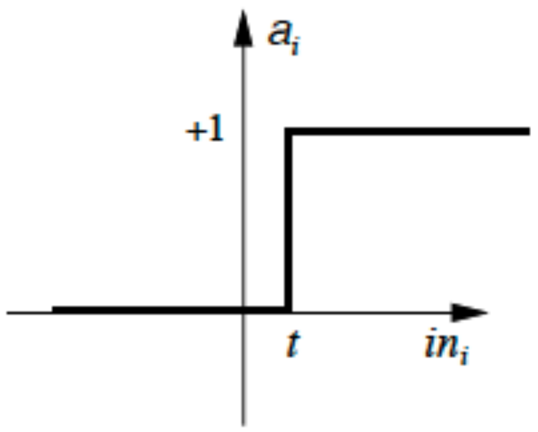

$a_i = g(in_i)$
a_j
$W_{j,i}$
Input
Links
in_i
Σ
g
a_i
Output
Links
Input Function
Activation Function
Output

$in_i = \sum_j W_{j,i} a_j$

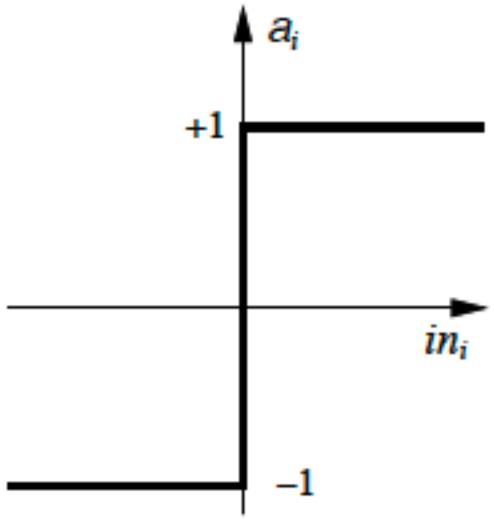
$a_i \leftarrow g(in_i) = g(\sum_j W_{j,i} a_j)$

g is the activation function:

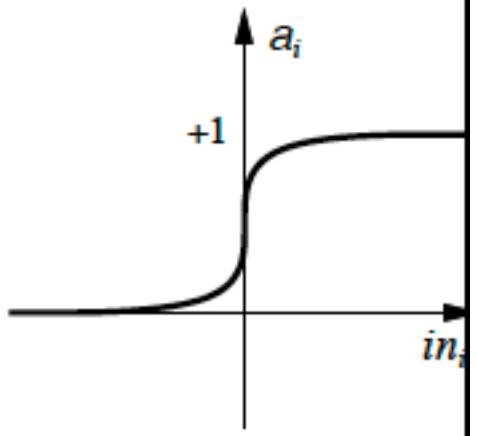
step, *sign*, and *sigmoid*.



(a) Step function



(b) Sign function



(c) Sigmoid function

a) $step_t(x) = 1$, if $x \geq t$; otherwise 0

b) $sign(x) = +1$, if $x \geq 0$; otherwise -1

c) $sigmoid(x) = \frac{1}{1+e^{-x}}$

What might be the advantage of c?

Aside: Can eliminate thresholds (it's a trick):

create extra input a_0 fixed at -1.

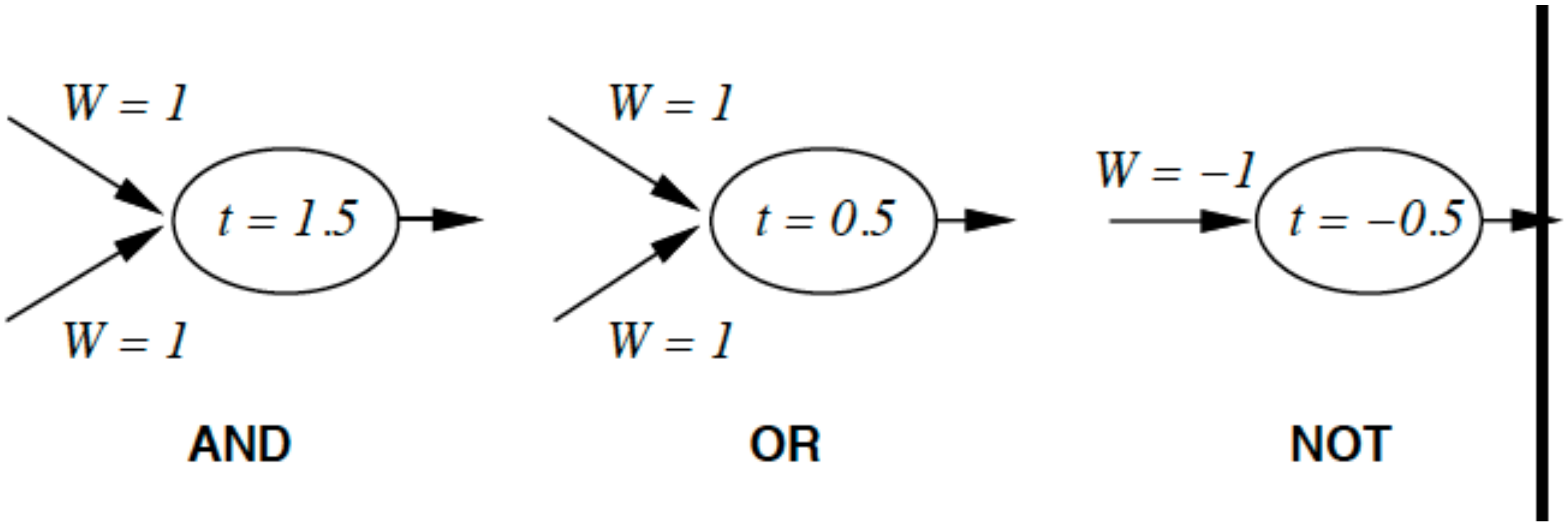
$$a_i = step_t(\sum_{j=1}^{n} \; W_{j,i}a_j) \;\; = \;\; step_0(\sum_{j=0}^{n} \; W_{j,i}a_j)$$

where $W_{0,i} = t$ and $a_0 = -1$.

Can simulate Boolean gates!

(original motivation McCulloch and Pitts (1943).

What does this mean?


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

Topics

Type of network structure.

Type of representations.

Type of learning algorithms (and applicability).

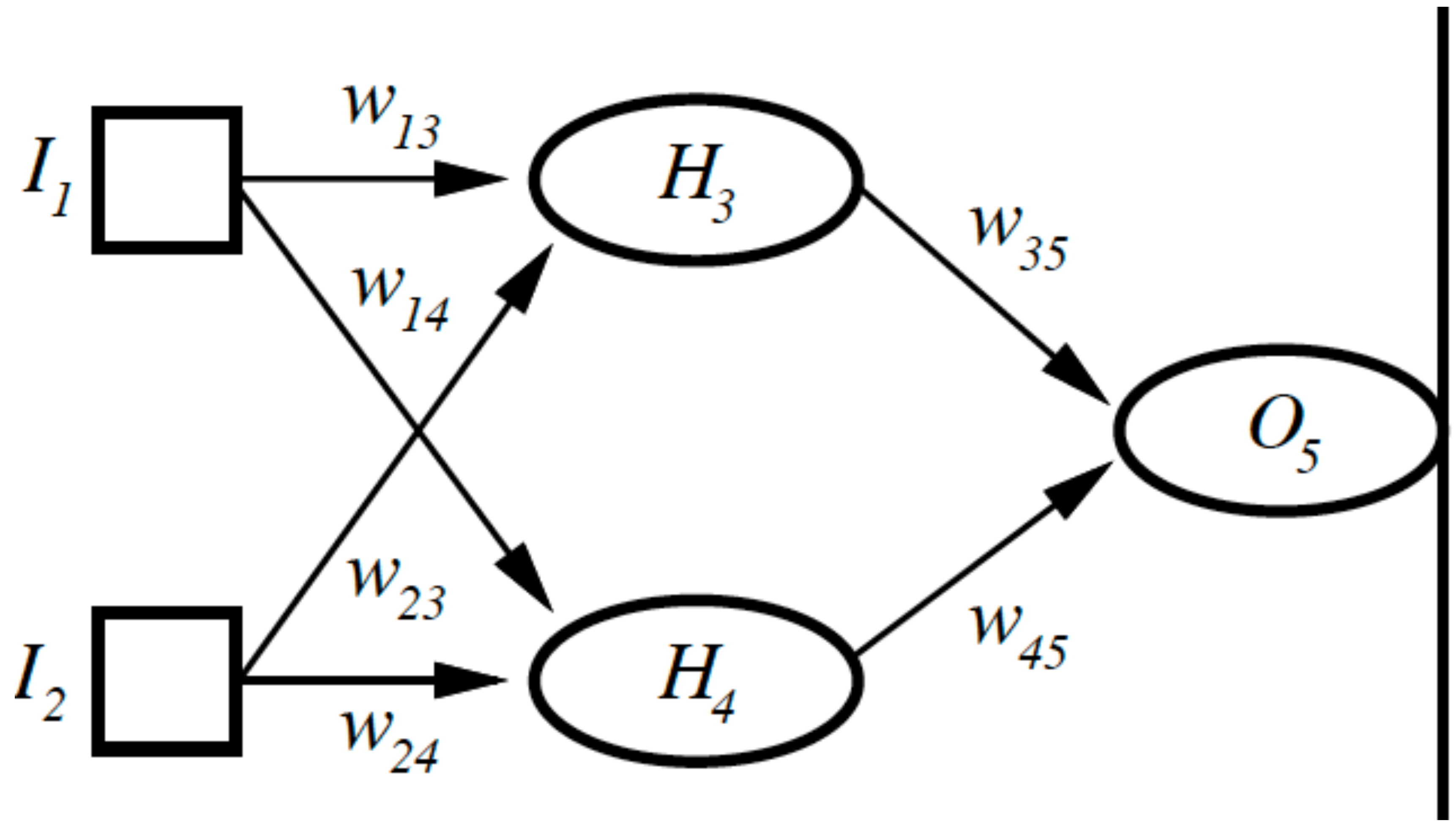
Network Structure

Main distinction: **feed-forward** vs. **recurrent**.

Feed-forward: no cycles. Activation flows one direction — from input layer via "hidden" layers to output layer.

Extreme (unlikely) example: input layer — retina cells / output layer — muscle control cells.

Next figure: two (three?) layers. Two input units / two hidden units one output unit


I_1
I_2
w_{13}
w_{14}
w_{23}
w_{24}
H_3
H_4
w_{35}
w_{45}
O_5

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

Activation passed from input to output. Does network have
internal state? Corresponds to simple reactive agents.

Much used!. *Good learning algorithms for classification / concepts.*

Brain cannot be just a feedforward network!
Need (need short-term memory)

Brain has many feed-back connections.
brain is **recurrent network.**
Cycles!

Hopfield Networks

Much harder to analyze. Can capture internal state. (activation keeps going around) More complex agents.

Two main types:

> **Hopfield networks.**
> **Boltzmann machines.**

Hopfield Networks

symmetric connections ($W_{i,j} = W_{j,i}$)

output 0/1 only.

train weights to obtain **associative memory**

eg. store patterns

It can be proven that an N unit Hopfiled net can store up to $0.138N$ patterns reliably.

Note: no explicit storage. All in the weights.

Boltzmann machines

symmetric connections ($W_{i,j} = W_{j,i}$)

  output 0/1 only but network in constant motion:

    compute **average output** value of each node.

  stochastic

has nice (but slow) learning algorithm. also closely

  connected to **probabilistic reasoning**

  belief networks.

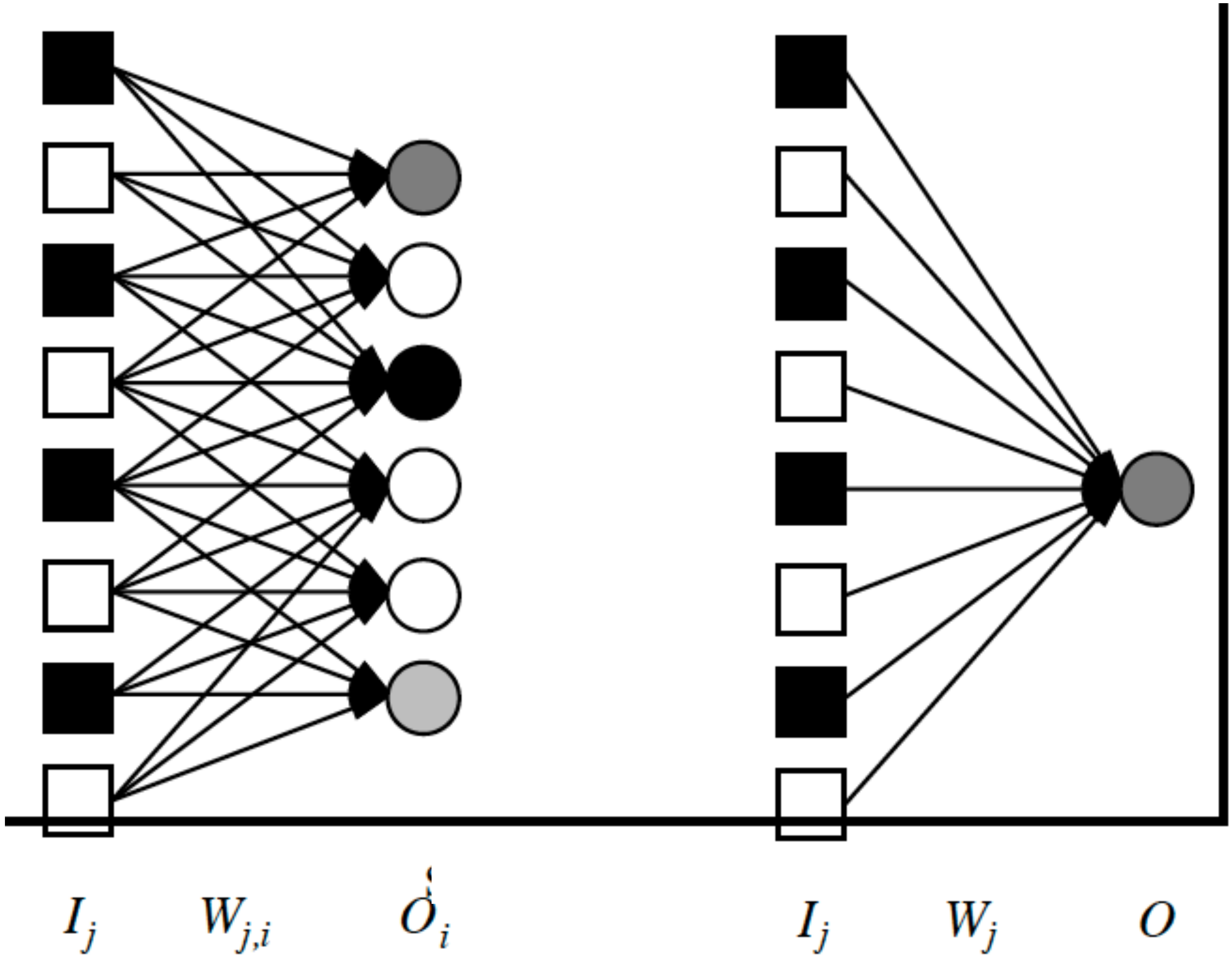
details beyond the scope of this course.

Back to Feed-forward

input / output / hidden units.

perceptrons: no hidden units

multilayered


I_j
$W_{j,i}$
O_i
I_j
W_j
O

Perceptron

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

Cornell Aeronautical Laboratory

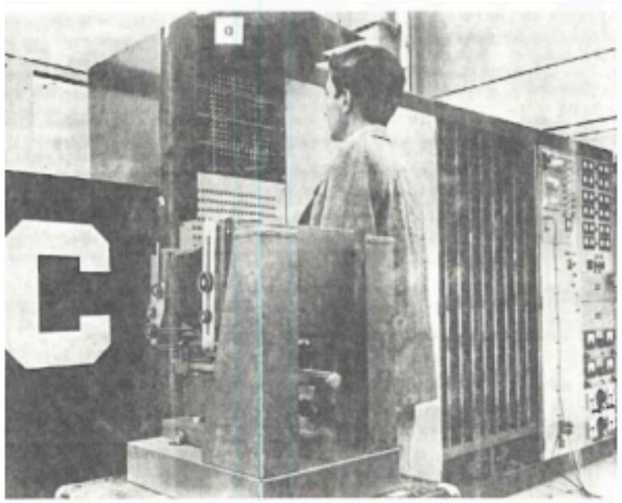
Rosenblatt &
Mark I Perceptron:
the first machine that could "learn" to recognize and identify optical patterns.

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

Machines

Mark I: 400 S-units,

512 A-units, 8 R-units



Tobermory

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

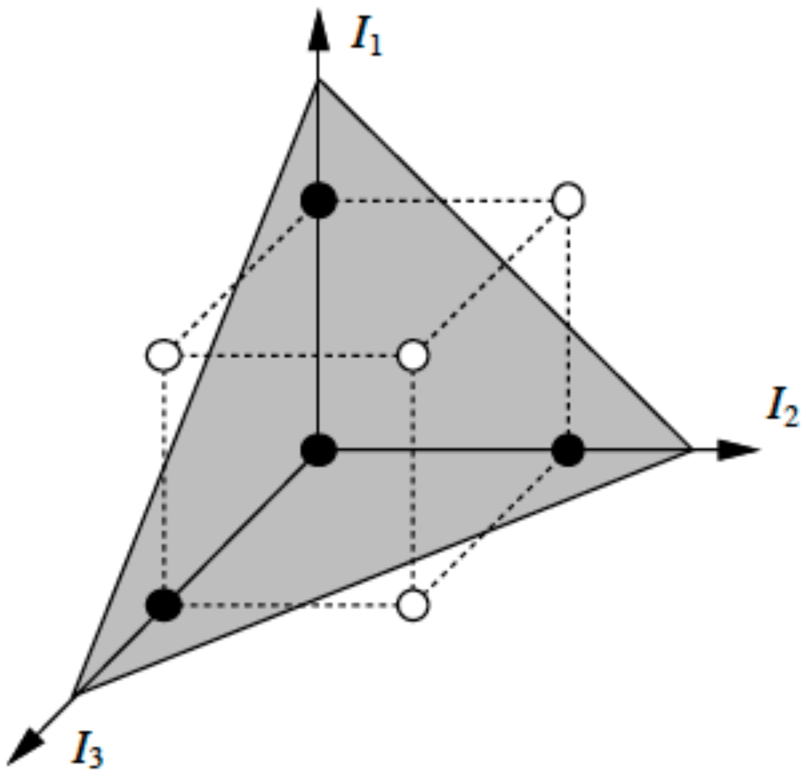
But analysis by Minsky and Papert (1969)

    showed certain simple functions cannot be represented
        (Boolean XOR)

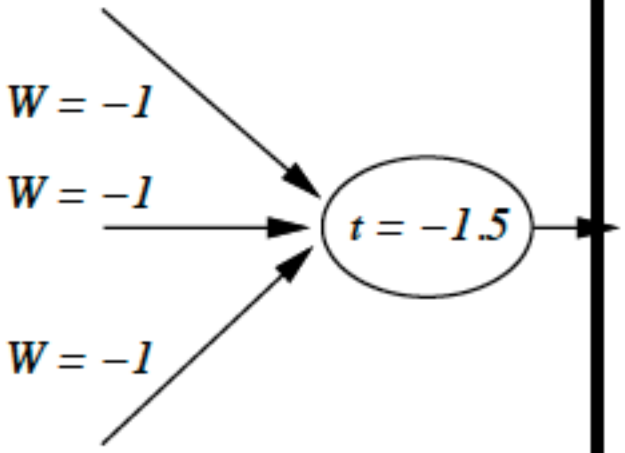
Killed the field! (and possibly Rosenblatt (rumored)).

But Minsky appears to have used a simplified model. Single layer.

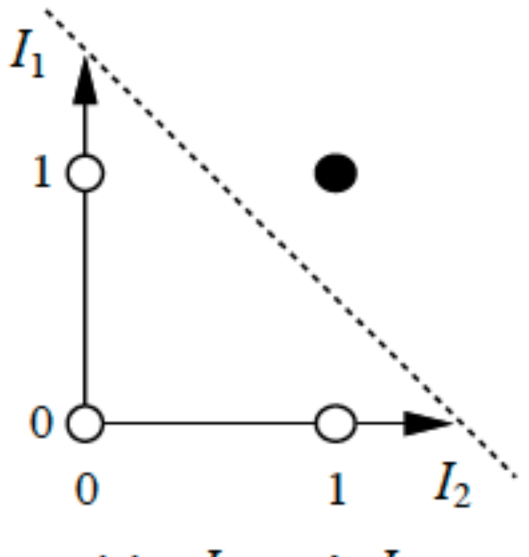
Linearly separable functions only

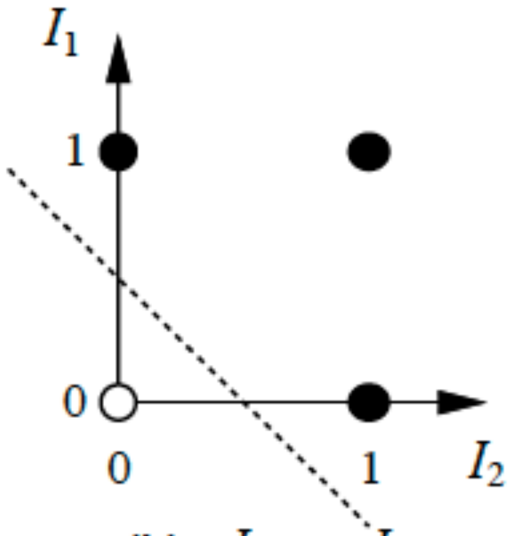


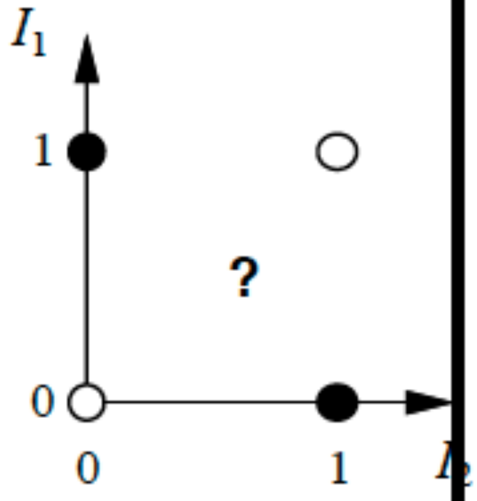
(a) Separating plane



(b) Weights and threshold


(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

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

Perceptrons

Recap previous slides

Representational limit:

- linearly separable functions only.
  - intuition? xor example.
  - xor with hidden layer.
  - connectedness example (Minsky/Papert)

Perceptron Learning

A perceptron can learn any linearly separable function, given enough enough training examples.

What does this say about linearly separable functions?

Key idea: **adjust weights till all examples correct.**

update weights repeatedly (epochs) for each example.

weight update

Single output O; target output for example T.

Define error: $Err = T - O$

Now, just move weights in right direction!

If error is positive, then need to increase 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 **learning rate**.

Perceptrons

From Patrick Winston (MIT) book

Basic learning strategy for perceptron:

(simplified from R&N; e.g., learning rate = 1)

Framework and notation:

0/1 signals

input signal: $\vec{X} =< x_1, x_2, \ldots, x_n, x_{n+1} >$

weight vetor: $\vec{W} =< w_1, w_2, \ldots, w_n, w_{n+1} >$

$x_{n+1} = 1$ with w_{n+1} simulates threshold.

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

Threshold function of perceptron:

Let $S = \sum_{k=1}^{k=n+1} w_k \times x_k$

If $S > 0$ then $O = 1$,

else $O = 0$.

We want to train the perceptron with a
 set of examples.
Each example is given by an
 a pair $(\vec{X}, l)$,
 i.e., an input vector with a label l (0 or 1).

Learning procedure for perceptron:

Called "the error correcting method"

```
Start with all zero weight vector.
Cycle (repeatedly) through examples and for
each example $(\vec{X}, l)$ do:
    If perceptron gives wrong answer,
        if output perceptron is 0 while it should be 1,
            add the input vector to the weight vector.
        if output perceptron is 1 while it should be 0,
            subtract the input vector to the weight vector.
    otherwise do nothing.
```

Note that procedure is intuitively correct.
(e.g., if output is 0 but should be 1 the weights are increased.)

The remarkable thing is that the procedure can be proved to converge (in poly steps) if the function can be represented by perceptron (i.e. is linearly separable)

Let's do an example.

Consider learning the logical "or" function.

Our examples are:

sample	x_1	x_2	x_3	1
1	0	0	1	0
2	0	1	1	1
3	1	0	1	1
4	1	1	1	1

Note: x_3 input fixed at 1 to model threshold.
Side benefit: no total 0 input.

Why needed? Note: if all zero input is "incorrect."

We'll use a single perceptron with three inputs (x_1, x_2, x_3) and single output (l).

Note artificial input x_3 fixed at 1.

Let $S = \sum_{k=1}^{k=n+1} w_k \times x_k$

If $S > 0$ then $O = 1$,
else $O = 0$.

We start with all weights at 0: $< 0, 0, 0 >$

Let's consider the 1st example.

Perceptron with current weights classifies
sample 1 correctly as 0, so no change in weights.

Consider sample 2

Perceptron says 0 (note all weights still 0; $S = 0$)
should be 1, so we add input vector to weight vector

New weights: $< 0, 1, 1 >$ ($=< 0, 0, 0 > + < 0, 1, 1 >$)

Note: standard addition and substraction. Weights can
take on arbitrary integer values.

Consider sample 3

Classified correctly; do nothing. Consider sample 4

Classified correctly; do nothing. Consider sample 1

Perceptron says 1; should be 0.

Subtract input vector:

New weights: $< 0, 1, 0 >$ $(=< 0, 1, 1 > - < 0, 0, 1 >)$

The next slide shows the next couple of steps.

Please verify those for yourself.

```
sample 2, correctly classified
sample 3, perceptron 0; should be 1
          new weights: <1, 1, 1>
sample 4, correctly classified
sample 1, perceptron 1; should be 0
          new weights: <1, 1, 0>
sample 2, correctly classified
sample 3, correctly classified
sample 4, correctly classified
sample 1, correctly classified
```

So, the weight vector $< 1, 1, 0 >$ classifies
all examples correctly, and the perceptron
has learned the function.

Aside: in more realistic cases the
weight w_3 will not be 0.
(This was just a toy example!)
Also, in general many more inputs (100 to 1000).
& 1M+

Always works, if inputs are linearly separable!
Single layer of units.

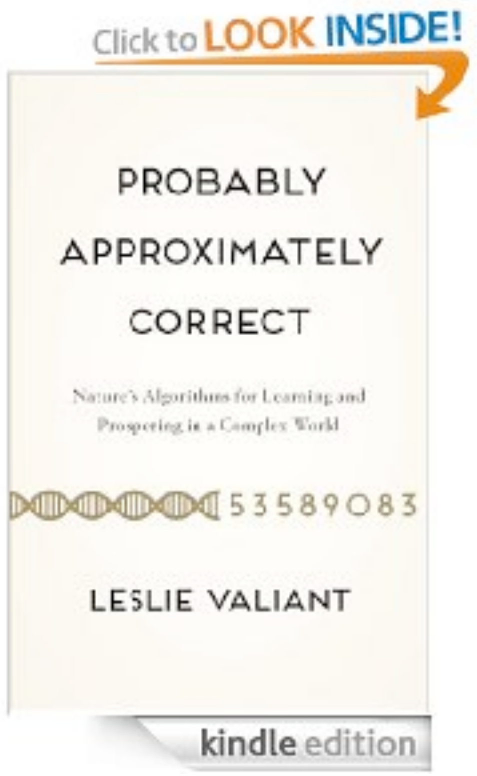
What about multi-layer NNs? No known learning method until early 80s…

Backpropagation

In order to learn multi-layer neural nets,
we need another learning algorithm.
(invented ca. 1984)
A multi-layer net has one or more hidden layers.

We will consider the **backpropagation** algorithm for training such networks.
See also R&N. Here we will present a more detailed example.

Side note: New book by Les Valiant on PAC! Also, an attempt to quantify evolution as a PAC learning process. Bottom-line: evolution based on random permutations is too slow! (Earth not old enough to explain our complexity.) But a mutation process guided by structure in our environment may be sufficiently fast.


Click to LOOK INSIDE!

PROBABLY APPROXIMATELY CORRECT

Nature's Algorithms for Learning and Prospering in a Complex World

53589083

LESLIE VALIANT

kindle edition

NYT review

Backpropagation

Based on Nilsson (Stanford)

Note that in the perceptron case, we looked at the output value, compared it to the desired value and changed the weights accordingly.

We want to do something similar in the multi-layer case but one difficulty is to determine the error in the outputs of the hidden units.
Luckily, we can approximate those errors by "backpropagating" the final output error.
To do so, we need an activition function that is differentiable.

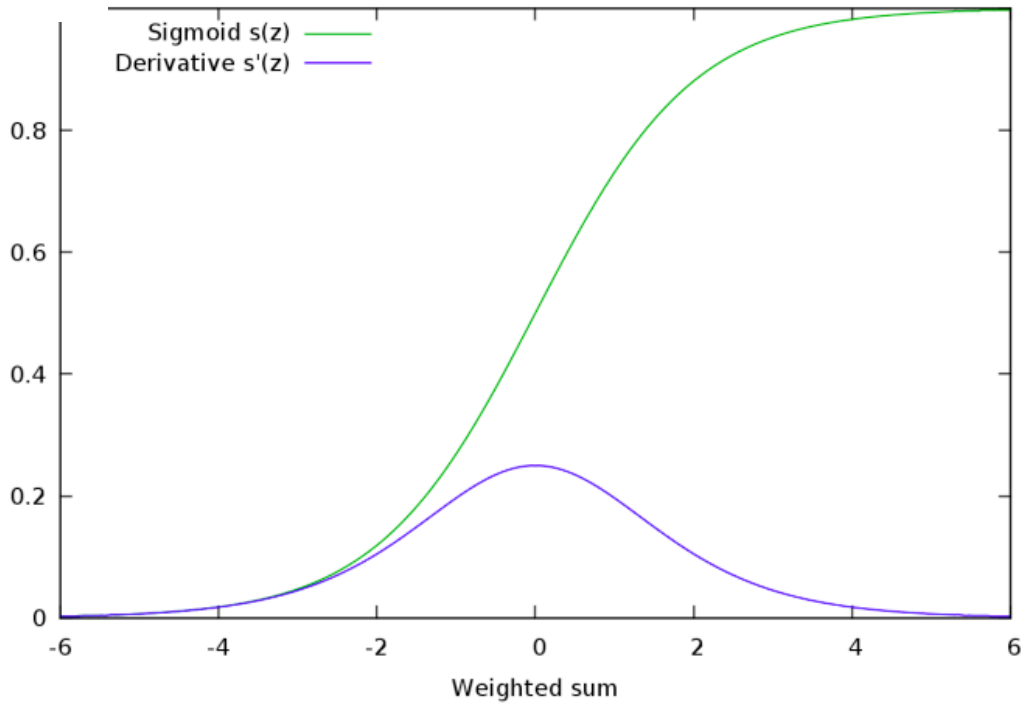
We use the **sigmoid** function to compute our output values.

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

The derivative of $f(x)$ is:

$f'(x) = f(x) \times (1 - f(x))$

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



Note: largest derivative at $x = 0$

That's where neuron is most sensitive to weight changes (effect of changes is well "controlled").

Output value of neuron is simply the weighted sum of its input "pushed" through the sigmoid.

$f(s) = \frac{1}{1+e^{-s}}$, where
$s = \sum_{k=1}^{k=n+1} w_k \times x_k$

Again, we assume the threshold is replaced by an extra input fixed at 1. Inputs: 0 or 1.

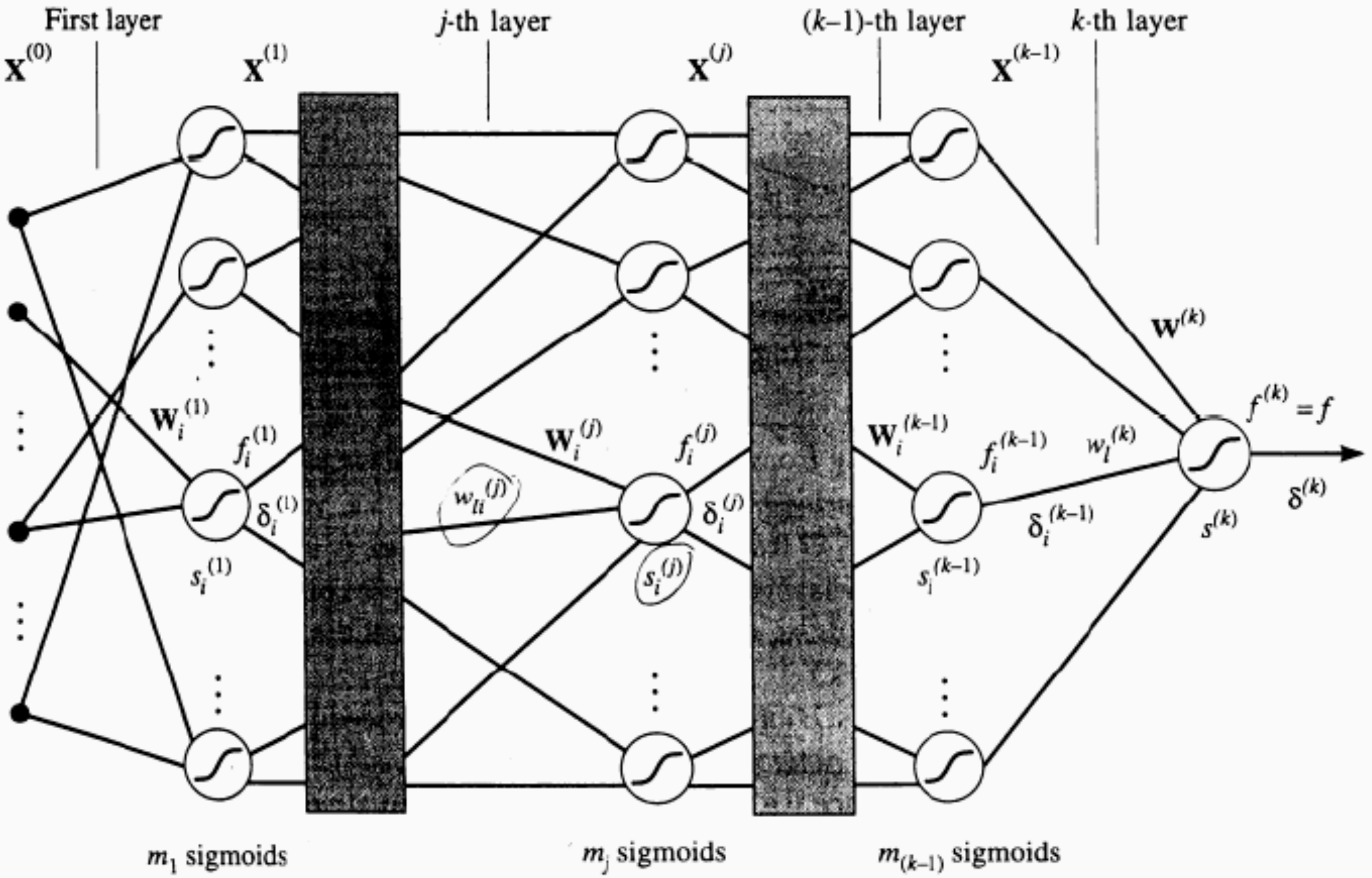


Figure 3.5

A k–layer Network of Sigmoid Units

Ok… details backpropagation NOT on exam.

Setup and notation:

k-layer network.

Input vector:

$\vec{X}^{(0)} = < x_1^{(0)}, x_2^{(0)}, \ldots, x_{m_0}^{(0)} >$

The first layer has m_1 units and its output is:

$\vec{X}^{(1)} = < f_1^{(1)}, f_2^{(1)}, \ldots, f_{m_1}^{(1)} >$

The weights for the ith unit in the first layer are given by:

$\vec{W}_i^{(1)} = < w_{1,i}^{(1)}, w_{2,i}^{(1)}, \ldots, w_{m_0,i}^{(1)} >$

These outputs $\vec{X^{(1)}}$ are fed into the second layer of m_2 units.

The number of units in the j-th layer is m_j.

In general, the weight vector of unit i in the j-th layer is $\vec{W_i^{(j)}}$ with components $w_{l,i}^j$ for $l = 1, m_{(j-1)} + 1$.

Note that the previous layer had $m_{(j-1)}$ units (and thus outputs) all connected to each unit in the j-th layer. We use one additional weight and fixed input to model the threshold. (Not given on previous slide.)

The weighted sum of the inputs to i-th sigmoid unit in the j-th layer is denoted by $s_i^{(j)}$

The output is $f_i^{(j)} = \frac{1}{1+e^{-s_i^{(j)}}}$

We have:

$s_i^{(j)} = \overrightarrow{X^{(j-1)}} \,.\, \overrightarrow{W^{(j)}}$

We used the vector dot product, i.e.,

$s_i^{(j)} = \sum_{l=1}^{l=m_{(j-1)}+1} x_l^{(j-1)} \times w_{l,j}^{(j)}$

$x_l^{(j-1)}$ is the output of unit l in the previous layer. It's connected with weight $w_{l,i}^{(j)}$ to the i-th unit in the j-th layer.

Note: again, concerning the "+1" here, we assume that an extra "1" is added to the input vector and an extra weight, to model the threshold.

Finally, the k-th layer is the output layer.
It has a single unit with input $s^{(k)}$ (weighted sum) and output value $f^{(k)} = f$.

The objective of the backprogation algorithm is
to minimize the output error on each example.

That is, we want to minimize:
$(L - f)^2$
Where, L the label (0 or 1) of the input example
under consideration and f is the output of the
network given that example.
Note: we will update the weights after each example
An alternative approach considers the combined
error over the total training set and update after
seeing all examples. In the limit the approaches are the same.

A key observation is that the error $(L - f)^2$ is
only a function of the weights.
Note: the number of units etc. is fixed.
Also, the inputs are fixed, since we are considering
a particular example.

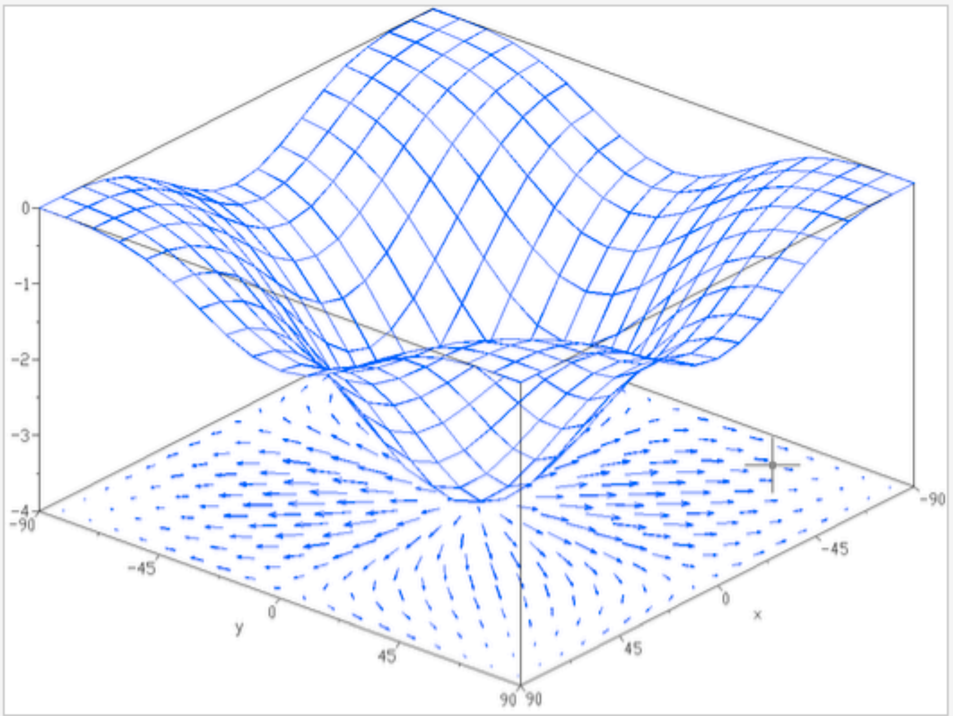
The idea is now to do a **gradient** descent in the
weight space to minimize the error.

The derivation (basically calculus) is somewhat
involved but not difficult.

It's "just" multivariate (or multivariable) calculus.

Optimization and gradient descent.

Consider x and y our two weights and f(x,y) the error signal.



The gradient of the function $f(x,y) = -(\cos^2 x + \cos^2 y)^2$ depicted as a projected vector field on the bottom plane.

Multi-layer network: composition of sigmoids; use chain rule.

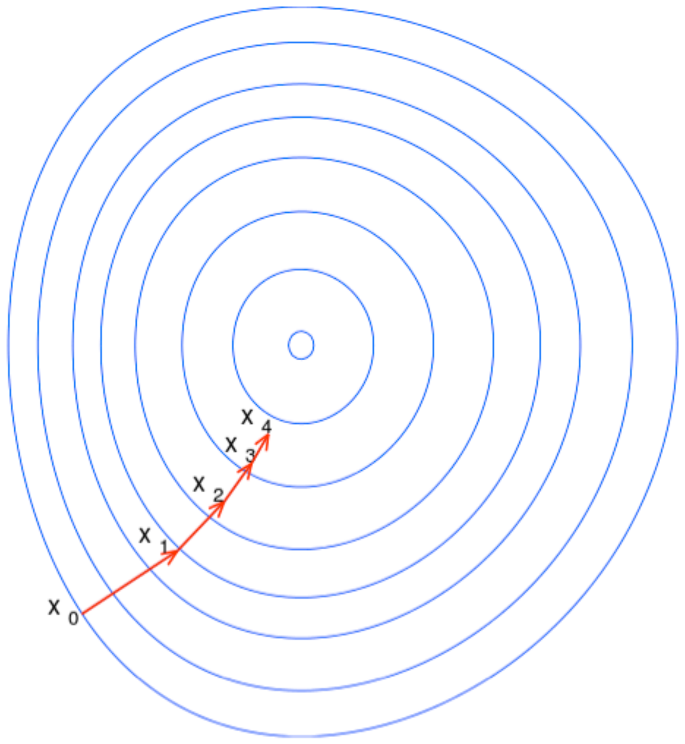
$$\nabla f = \frac{\partial f}{\partial x_1}\mathbf{e}_1 + \cdots + \frac{\partial f}{\partial x_n}\mathbf{e}_n$$

Reminder:
Gradient descent optimization. (Wikipedia)

Gradient descent is based on the observation that if the multivariable function $F(\mathbf{x})$ is defined and differentiable in a neighborhood of a point $\mathbf{a}$, then $F(\mathbf{x})$ decreases *fastest* if one goes from $\mathbf{a}$ in the direction of the negative gradient of F at $\mathbf{a}$, $-\nabla F(\mathbf{a})$. It follows that, if

$$\mathbf{b} = \mathbf{a} - \gamma \nabla F(\mathbf{a})$$

for γ small enough, then $F(\mathbf{a}) \geq F(\mathbf{b})$. With this observation in mind, one starts with a guess $\mathbf{x}_0$ for a local minimum of F, and considers the sequence $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ such that



$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n), \ n \geq 0.$$

We have

$$F(\mathbf{x}_0) \geq F(\mathbf{x}_1) \geq F(\mathbf{x}_2) \geq \cdots,$$

so hopefully the sequence $(\mathbf{x}_n)$ converges to the desired local minimum.

Here we just give the results of the calculation.

First, the weight adjustment for the single unit in the output layer is given by:

$$\vec{W}^{(k)} \leftarrow \vec{W}^{(k)} + \alpha \times (d - f) \times f \times (1 - f) \times \vec{X}^{(k-1)}$$

Note: f is the output of the unit; d is desired output.

This rule is analogous to the perceptron rule, except that we are now using the sigmoid.

The $(d - f)$ is the error signal.

α is the learning rate (a constant chosen by the user).

$f \times (1 - f)$ comes from the derivative of the sigmoid.

$\vec{X}^{(k-1)}$ is the input to the unit under consideration.

So, again we're adding (subtracting) the input vector, depending on $(d - f)$.

Please check for yourself that the correction is in the right direction!

We're basically going to do something similar for the weights in the hidden layer. The only problem is that we don't have a direct measure of the error in the outputs on the hidden units.

However, we can estimate those errors by considering the constribution (an estimate) of each unit to the final output value (layer k).

Starting with the final layer and moving backwards,

we compute for the i-th unit in the j-th layer:

$\delta_i^{(j)} = f_i^j(1 - f_i^j)\sum_{l=1}^{m_{j+1}} \delta_l^{(j+1)} w_{i,l}^{(j+1)}$

Note that $f_i^{(j)}$ is the output value of the unit.

So, the δ for a unit in the i-th layer is

computed by considering the delta in the $i+1$-th layer.

The base case is the output layer k:

$\delta^{(k)} = (d - f) \times f \times (1 - f)$

I.e., the real output error times the gradient.

This values is propagated backwards to get error measures

for the internal units (times the gradient again).

Finally, using these δ's, we can compute the weight updates for the hidden units:

$$\vec{W}_i^{(j)} \leftarrow \vec{W}_i^{(j)} + \alpha \times \delta_i^j \times \vec{X}^{(j-1)}$$

Verify that this rule is consistent with the update rule for the final node. (Check $j = k$; drop i, since only one unit in layer.)

So, first we "backpropagate" the error signal, and then we update the weights, layer by layer.

That's why it's called "backprop learning." Now changing speech recognition, image recognition etc.!

Although, the rules are somewhat intuitive, only by doing the full derivation, can one explain all the terms.

Let us consider an example of the procedure in action.

For home (but not on exam).

Trace example at home!

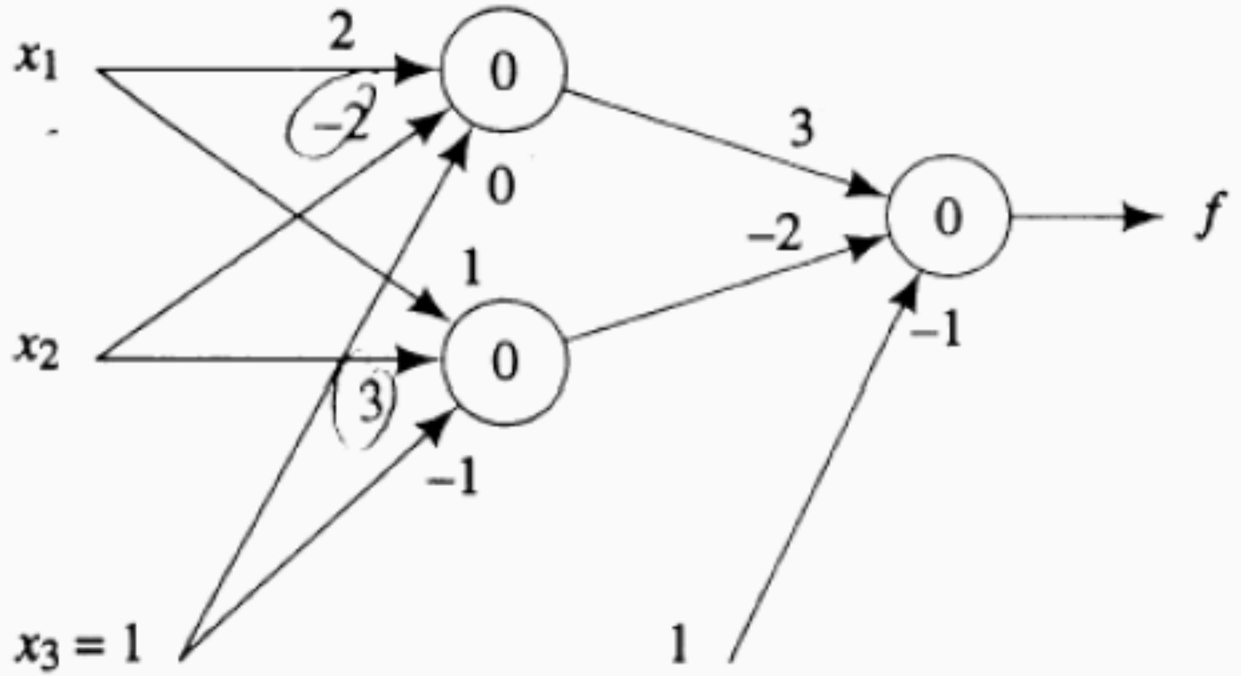


Figure 3.6

A Network to Be Trained by Backprop

We have two layers ($k = 2$).

Two units in first layer, with a total
of three inputs $(x_1^{(0)}, x_2^{(0)}, x_3^{(0)})$.
$x_3^{(0)}$ will be fixed to 1.

The two units in the first layer are connected
to a node in the final layer. This node
has three inputs: $(x_1^{(1)}, x_2^{(1)}, x_3^{(1)})$.

$x_1^{(1)}$ is the output of the 1st unit in the 1st layer.

$x_2^{(1)}$ is the output of the 2nd unit in the 2nd layer.

$x_3^{(1)}$ is fixed at 1.

Note the given weights in the figure.

We want to train the network to capture the following patterns:

ex 1.) $x_1^{(0)} = 1, x_2^{(0)} = 0, x_3^{(0)} = 1, d = 0$
ex 2.) $x_1^{(0)} = 0, x_2^{(0)} = 0, x_3^{(0)} = 1, d = 1$
ex 3.) $x_1^{(0)} = 0, x_2^{(0)} = 1, x_3^{(0)} = 1, d = 0$
ex 4.) $x_1^{(0)} = 1, x_2^{(0)} = 1, x_3^{(0)} = 1, d = 1$

Again, d is the desired output; $x_3^{(0)}$ is the fixed unit. Let's consider the update after the first pattern.

Ex 1.) gives input vector $< 1, 0, 1 >$, which leads via the sigmoid to the following output values:

$$f_1^{(1)} = \frac{1}{1+e^{-2}} = 0.881$$
$$f_2^{(1)} = \frac{1}{1+e^{-0}} = 0.5$$
$$f = \frac{1}{1+e^{-(3\times 0.881+(-2)\times 0.5-1)}} = 0.665$$

We now compute the values for the δ's.

First the base case:

$$\delta^{(2)} = (0 - 0.665) \times 0.665 \times (1 - 0.665) = -0.148$$

Backpropagating through the weights gives:

$$\delta_1^{(1)} = 0.881 \times (1 - 0.881) \times (-0.148 \times 3) = -0.047$$
$$\delta_2^{(1)} = 0.5 \times (1 - 0.5) \times (-0.148 \times -2) = 0.074$$

Double-check at least one of these!

After computing the δ's, we can now update the weights. (Use learning rate $\alpha = 1$.) We get for the new weights:

$$\vec{W}_1^{(1)} = < 1.953, -2.0, -0.047 >$$
$$\vec{W}_2^{(1)} = < 1.074, 3.0, -0.926 >$$
$$\vec{W}^{(2)} = < 2.870, -2.074, -1.148 >$$

Aside: the original weights were:

$$\vec{W_1}^{(1)} = < 2.0, -2.0, 0.0 >$$

$$\vec{W_2}^{(1)} = < 1.0, 3.0, -1.0 >$$

$$\vec{W}^{(2)} = < 3.0, -2.0, -1.0 >$$

Let's do an example calculation of the first weight vector.

$\vec{W_1}^{(1)} =< 1.953, -2.0, -0.047 >$

$$w_{1,1}^{(1)} = w_{1,1}^{(1)} + (1 \times \delta_1^1 \times x_1^{(0)})$$
$$= 2 + (1 \times (-0.047) \times 1) = 1.953$$

$$w_{1,2}^{(1)} = w_{1,2}^{(1)} + (1 \times \delta_1^1 \times x_2^{(0)})$$
$$= -2 + (1 \times (-0.047) \times 0) = -2.0$$

$$w_{1,3}^{(1)} = w_{1,3}^{(1)} + (1 \times \delta_1^1 \times x_3^{(0)})$$
$$= 0 + (1 \times (-0.047) \times 1) = -0.047$$

Let's do the calculation of the third weight vector.

$\vec{W}(2) = < 2.870, -2.074, -1.148 >$ $w_1^{(2)} = w_1^{(2)} + (1 \times \delta^2 \times x_1^{(1)})$

$$= 3 + (1 \times (-0.148) \times 0.881)$$
$$= 2.870$$

$$w_2^{(2)} = w_2^{(2)} + (1 \times \delta^2 \times x_2^{(1)})$$
$$= -2 + (1 \times (-0.148) \times 0.5)$$
$$= -2.074$$

$$w_3^{(2)} = w_3^{(2)} + (1 \times \delta^2 \times x_3^{(1)})$$
$$= -1 + (1 \times (-0.148) \times 1)$$
$$= -1.148$$

Summary ☺

A tour of AI:

I) AI

--- motivation

--- intelligent agents

II) Problem-Solving

--- search techniques

--- adversarial search

--- reinforcement learning

Summary, cont. ☺

III) Knowledge Representation and Reasoning

--- logical agents

--- Boolean satisfiability (SAT) solvers

--- first-order logic and inference

V) Learning

--- from examples

--- decision tree learning (info gain)

--- neural networks

The field has grown (and continues to grow) exponentially but you have now seen a good part!

Have a great winter break!!!!