MC906
Introduction to Artificial Intelligence

Prof. Anderson Rocha
University of Campinas (Unicamp)
Campinas, Brazil

AIMA Chapter 20, Sec. 5
Machine Learning. Tom Mitchell, Chapter 4

Outline

◊ Brains

◊ Neural networks

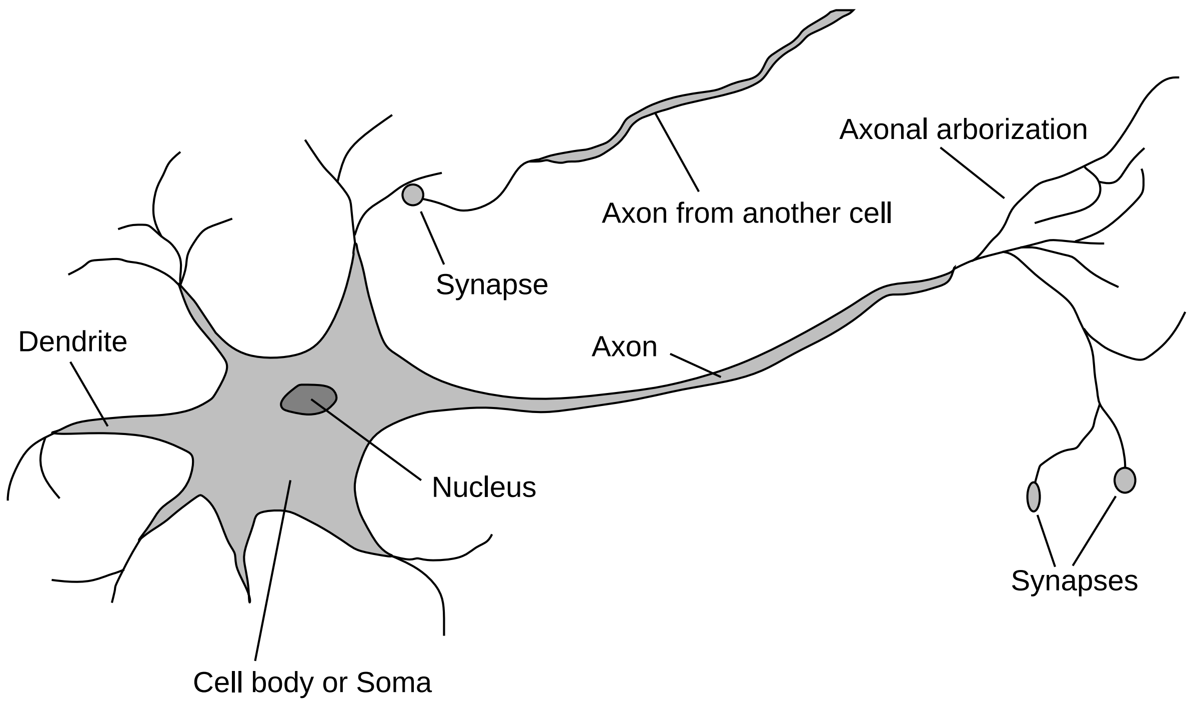
◊ Perceptrons

◊ Multilayer perceptrons

◊ Applications of neural networks

Brains

10^{11} neurons of > 20 types, 10^{14} synapses, 1ms–10ms cycle time
Signals are noisy "spike trains" of electrical potential



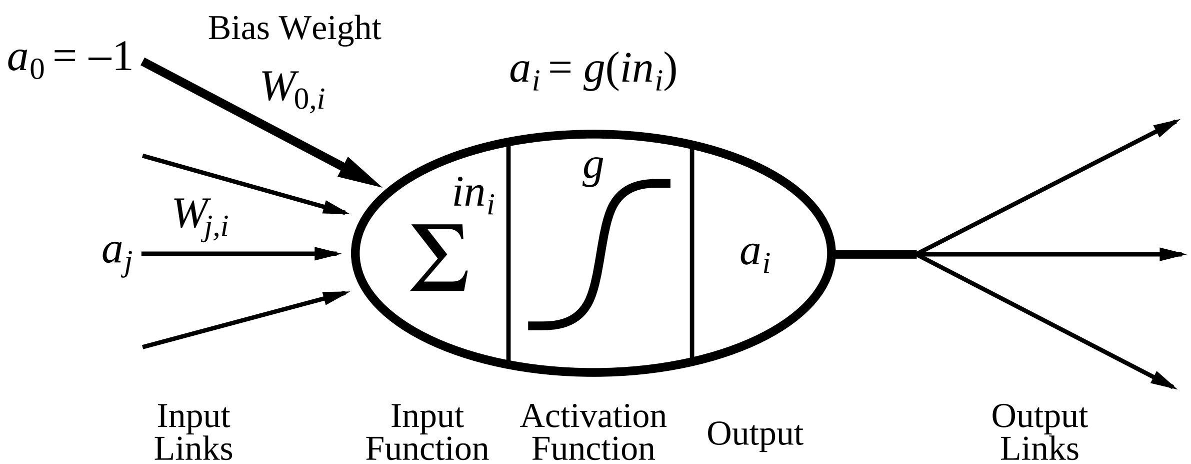
Time to recognize a face is about 0.1 s → parallelism!

McCulloch–Pitts "unit"

*

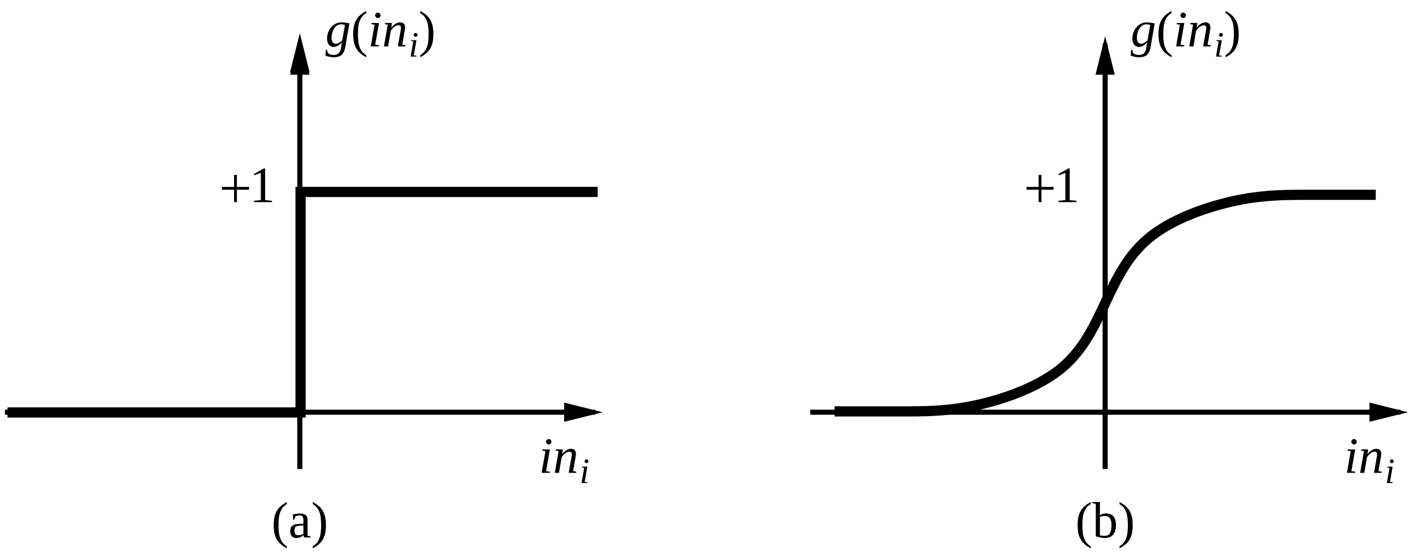
Output is a "squashed" linear function of the inputs:

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



A gross oversimplification of real neurons, but its purpose is
to develop understanding of what networks of simple units can do

Activation functions

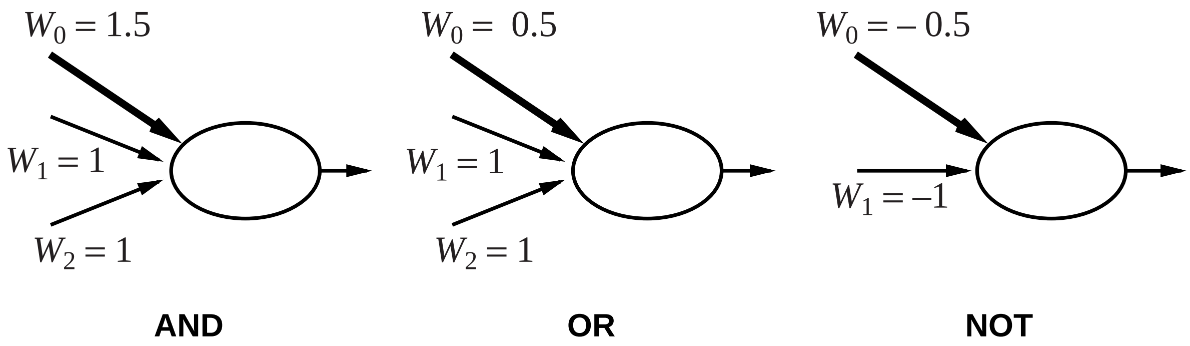


(a) is a step function or threshold function

(b) is a sigmoid function $1/(1 + e^{-x})$

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

Implementing logical functions



McCulloch and Pitts: every Boolean function can be implemented

Network structures

Feed-forward networks:

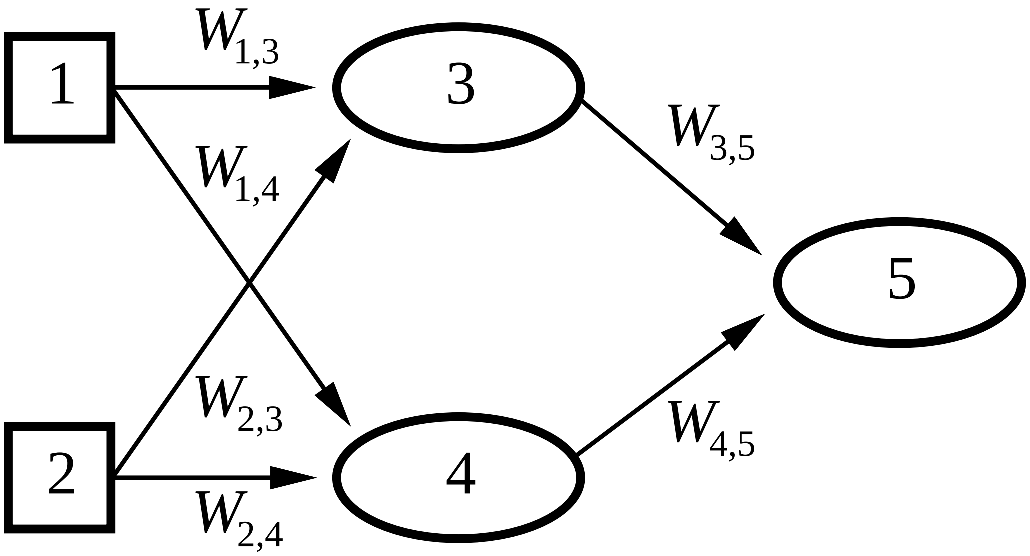
– single-layer perceptrons
– multi-layer perceptrons

Feed-forward networks implement functions, have no internal state

Recurrent networks:

– * recurrent neural nets have directed cycles with delays
$\Rightarrow$ have internal state (like flip-flops), can oscillate etc.

Feed-forward example

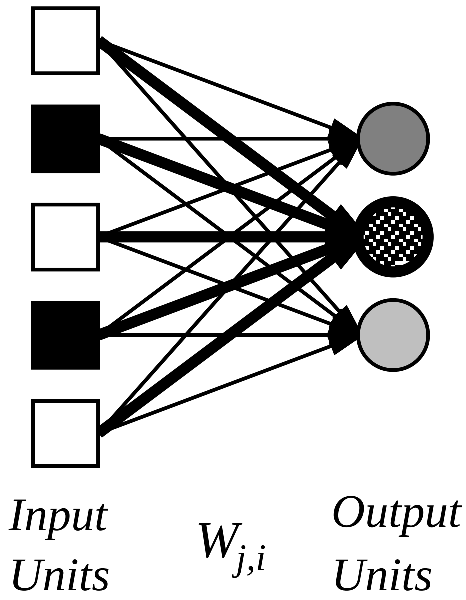


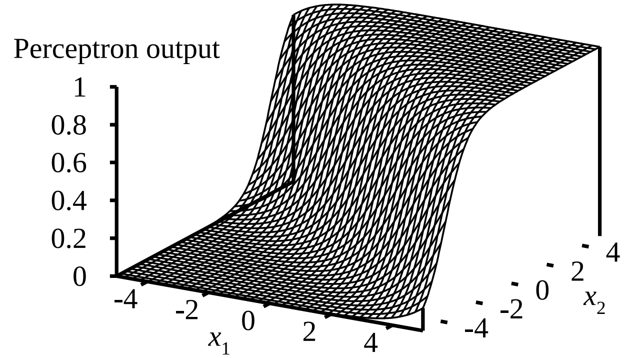
Feed-forward network = a parameterized family of nonlinear functions:

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

Adjusting weights changes the function: do learning this way!

Single-layer perceptrons





Output units all operate separately—no shared weights

Adjusting weights moves the location, orientation, and steepness of cliff

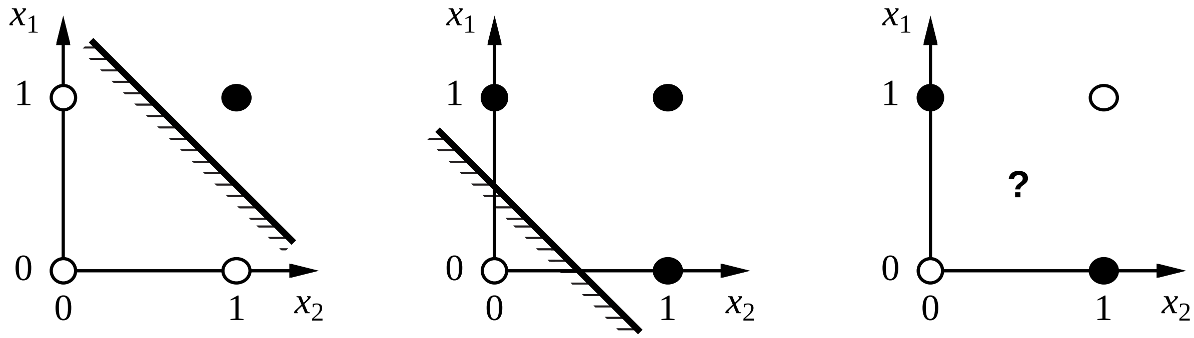
Expressiveness of perceptrons

Consider a perceptron with g = step function (Rosenblatt, 1957, 1960)

Can represent AND, OR, NOT, majority, etc., but not XOR

Represents a linear separator in input space:

$$\sum_j W_j x_j > 0 \quad \text{or} \quad \mathbf{W} \cdot \mathbf{x} > 0$$



(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_2

Minsky & Papert (1969) pricked the neural network balloon

Perceptron learning

Goal: Determine the weight vector $\mathbf{w}$ that produces the correct output for each training sample

The perceptron training rule: *

$$w_i = w_i + \Delta w_i \qquad where \;\; \Delta w_i = \eta(t - o)x_i$$

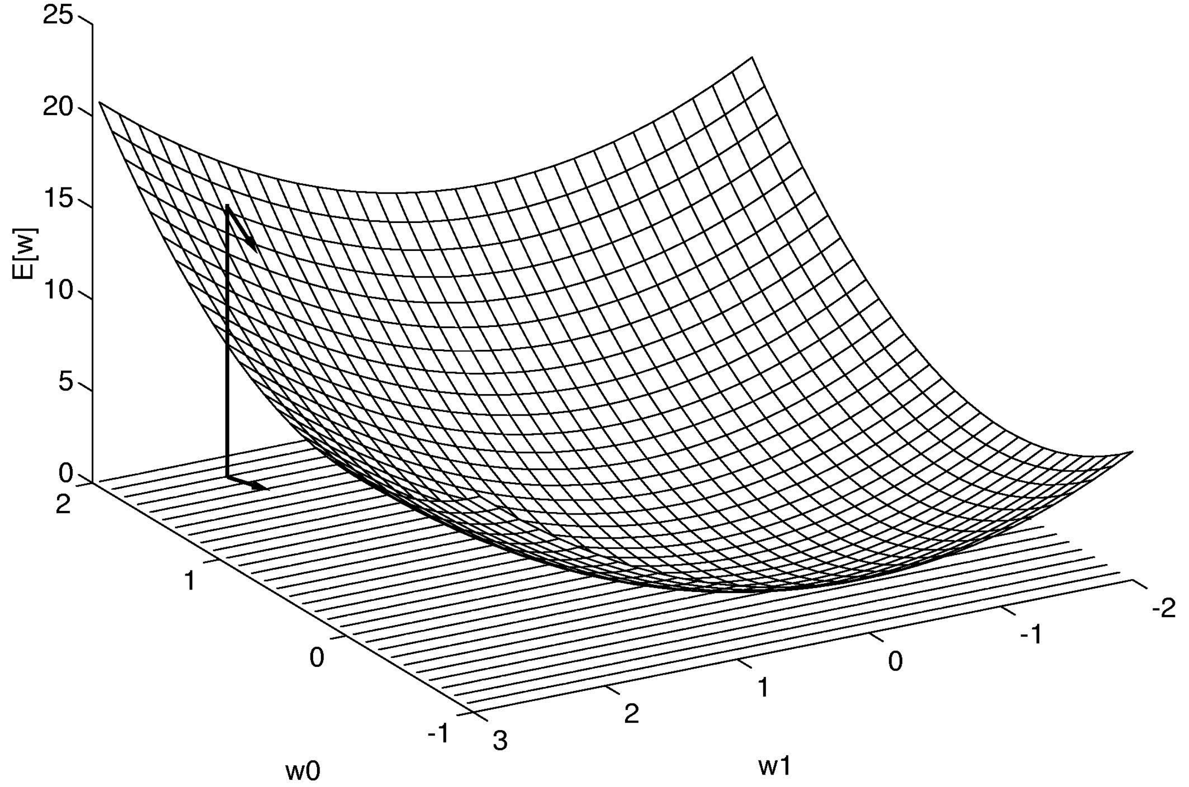
– Can be proven to converge within a finite number of iterations **provided the training examples are linearly separable**.

Gradient descent rule:

$$w_i = w_i + \Delta w_i \qquad where \;\; \Delta w_i = \Delta w_i + \eta(t - o)x_i$$

– If the training samples are not linearly separable, this rule converges toward the best-fit approximation to the target.

Perceptron learning contd.



Hypothesis space of possible weight vectors and their associated errors.

Perceptron learning contd.

Learn by adjusting weights to best fit the training samples.

$$o(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} \qquad (linear\ unit)$$

Training error of the weight vector (D is the set of training samples)

$$E(\mathbf{w}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Perform optimization search by gradient descent:

$$\nabla E(\mathbf{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}\right]$$

The gradient specifies the direction that produces the steepest increase, therefore

$$\mathbf{w} = \mathbf{w} + \Delta\mathbf{w} \qquad where\ \ \Delta\mathbf{w} = -\eta \nabla E(\mathbf{w})$$

Perceptron learning contd.

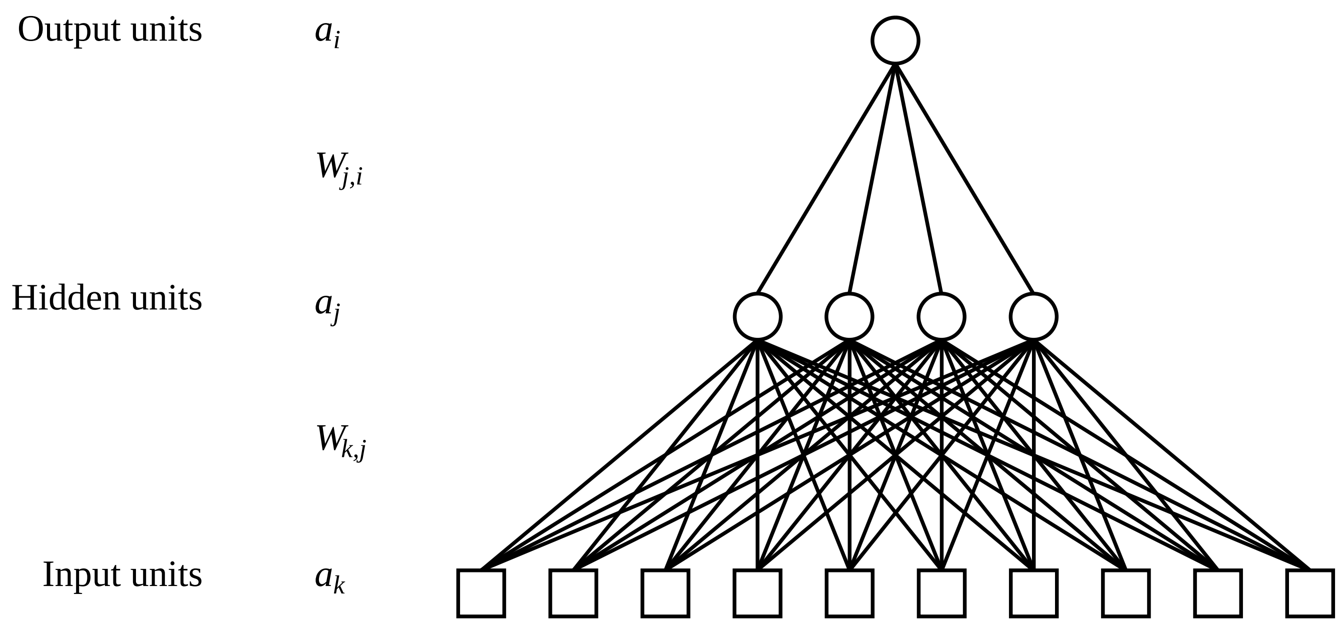
$$w_i = w_i + \Delta w_i \qquad where \;\; \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \mathbf{w} \cdot \mathbf{x}_d) \\
\frac{\partial E}{\partial w_i} &= \sum_{d \in D} (t_d - o_d)(-x_{id})
\end{aligned}$$

$$\Rightarrow \Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{id} \qquad\qquad *$$

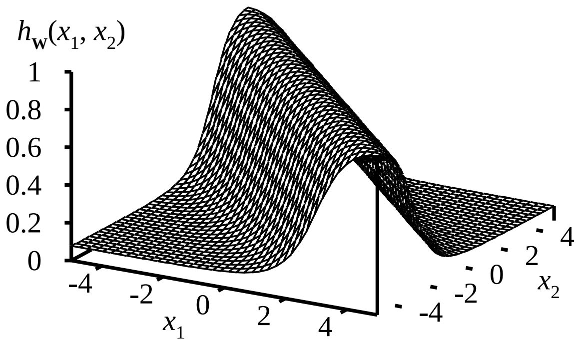
Multilayer perceptrons

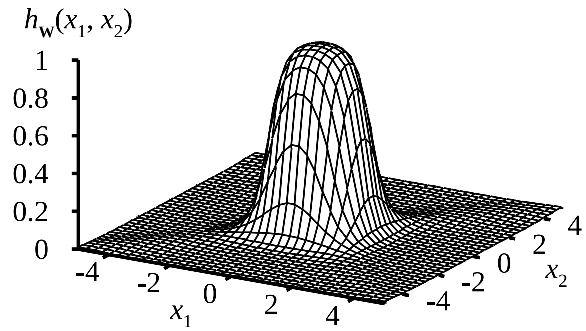
Layers are usually fully connected;
numbers of hidden units typically chosen by hand



Expressiveness of MLPs

All continuous functions w/ 2 layers, all functions w/ 3 layers



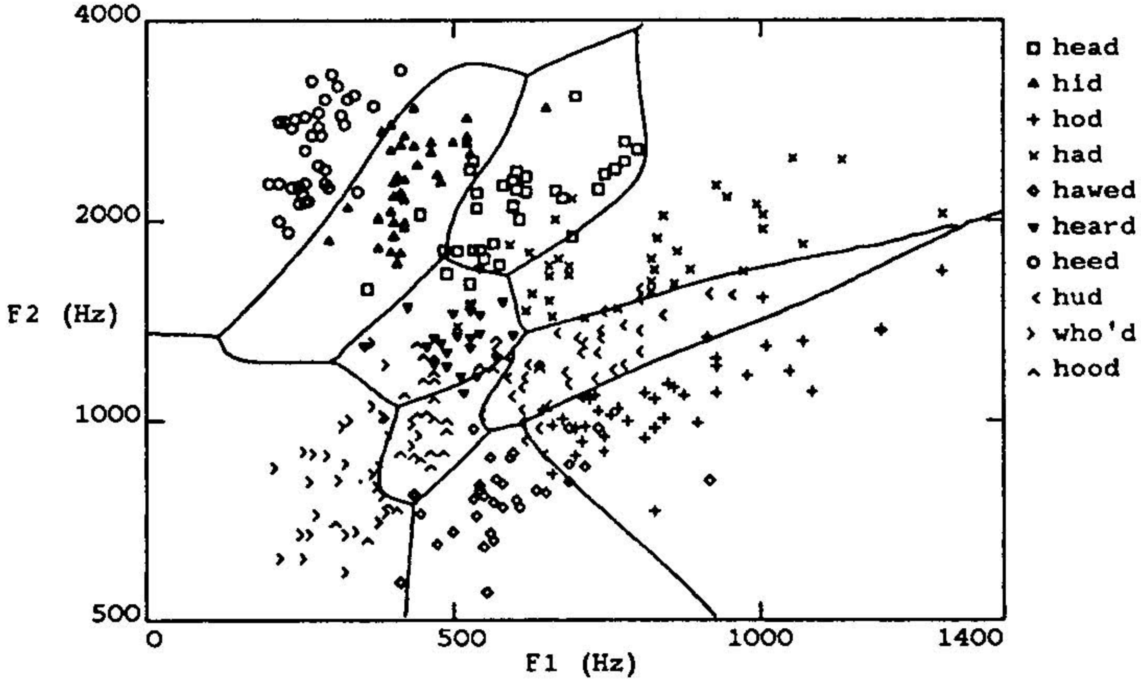


Combine two opposite-facing threshold functions to make a ridge

Combine two perpendicular ridges to make a bump

Add bumps of various sizes and locations to fit any surface

Expressiveness of MLPs contd.



Decision regions of a multilayer feedforward network trained to recognize 10 vowel sounds.

Back-propagation learning

Problem: Error surface has multiple local minima. Therefore, the gradient descent converges to a local minimum.

Back-propagation algorithm has two steps

- forward: input instance $\mathbf{x}$, compute output o_u for every unit u
- backward: propagate the errors backward through the network

Back-propagation learning contd.

For each $(\mathbf{x}, \mathbf{t})$ do

propagate input forward

1. input $\mathbf{x}$ and compute output o_u for every unit u

propagate errors backward

2. for each output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

3. for each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_h$$

4. update each network weight w_{ji}

$$w_{ij} \leftarrow w_{ji} + \Delta w_{ji} \qquad where\ \Delta w_{ji} = \eta \delta_j x_{ji}$$

Back-propagation learning contd.

Convergence and Local Minima:

– Guaranted to converge to a local minimum

– Solutions: add momentum, train multiple networks

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1)$$

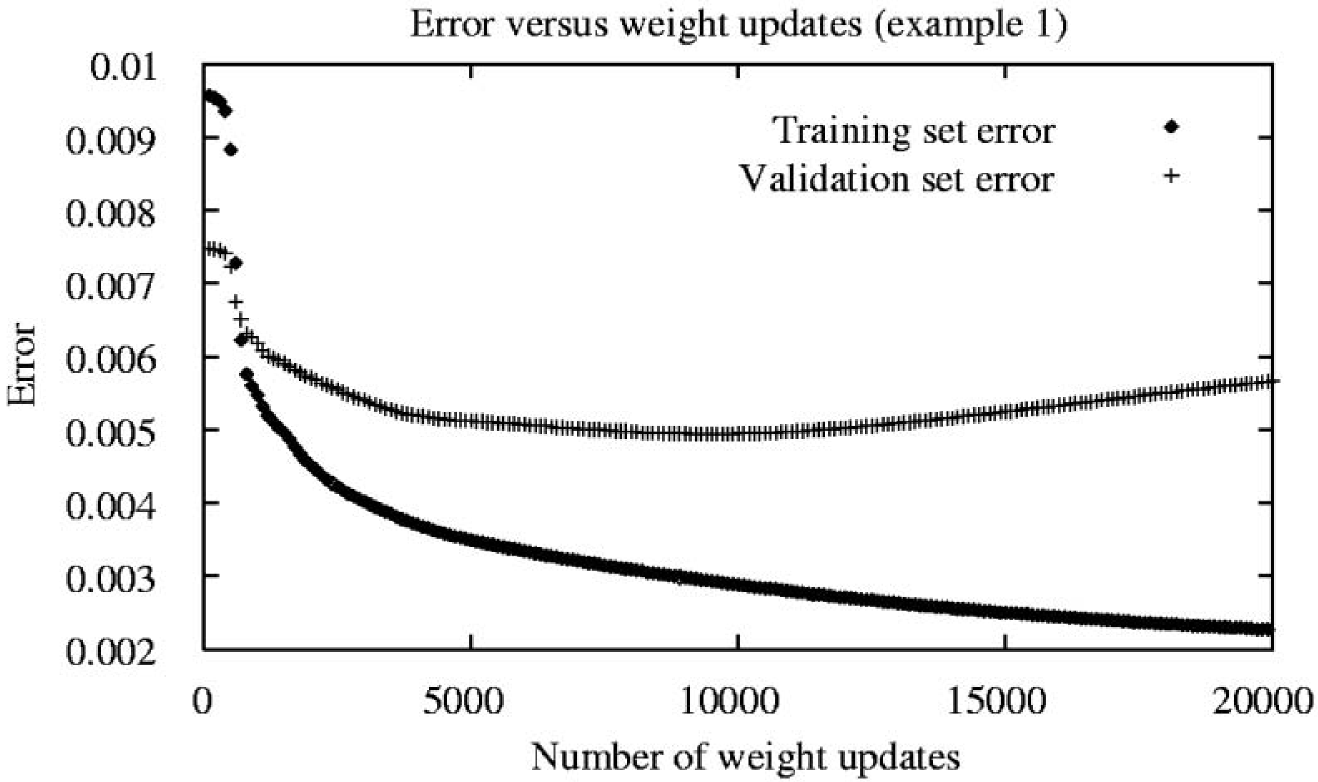
Representational power:

– Boolean functions, continuous functions, arbitrary functions

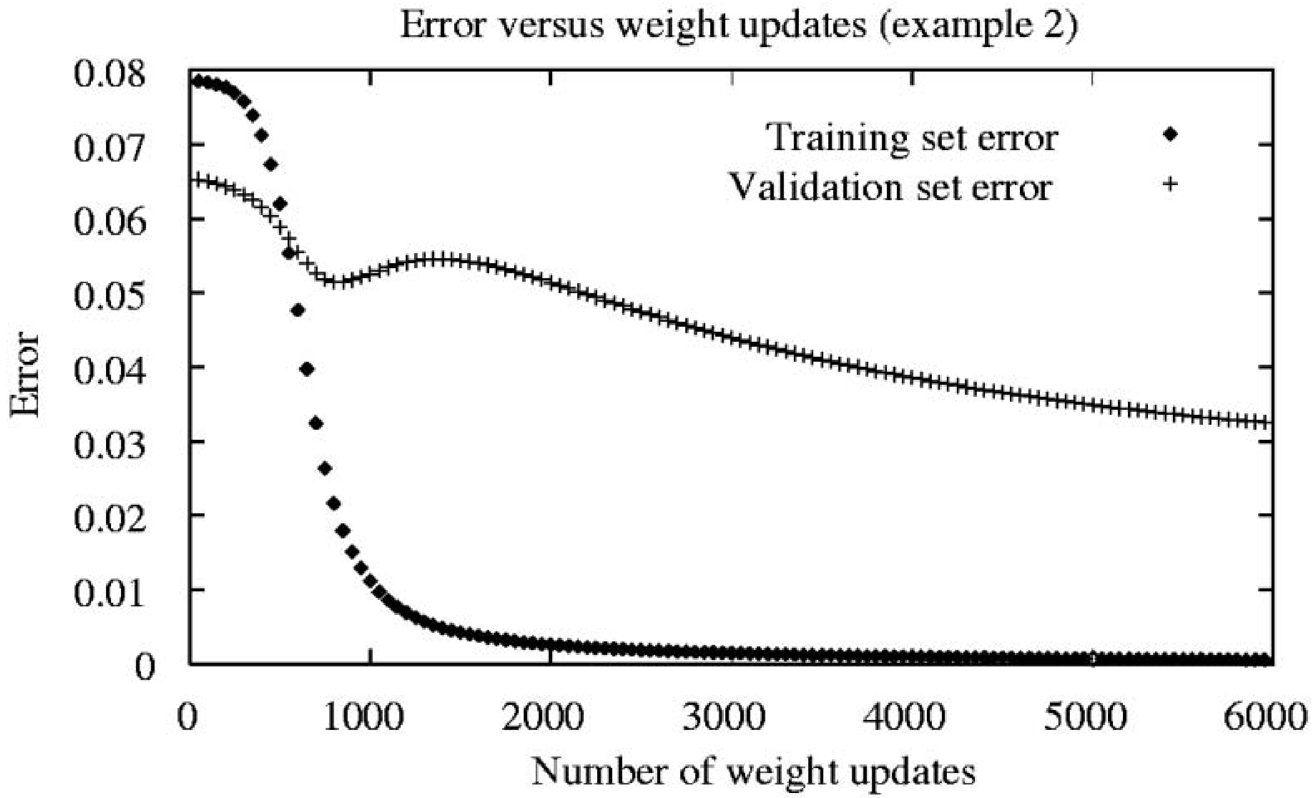
Overfitting:

– Given enough weight-tuning iterations, the back-propagation will create a complex decision surface that fits noise.

Back-propagation learning contd.



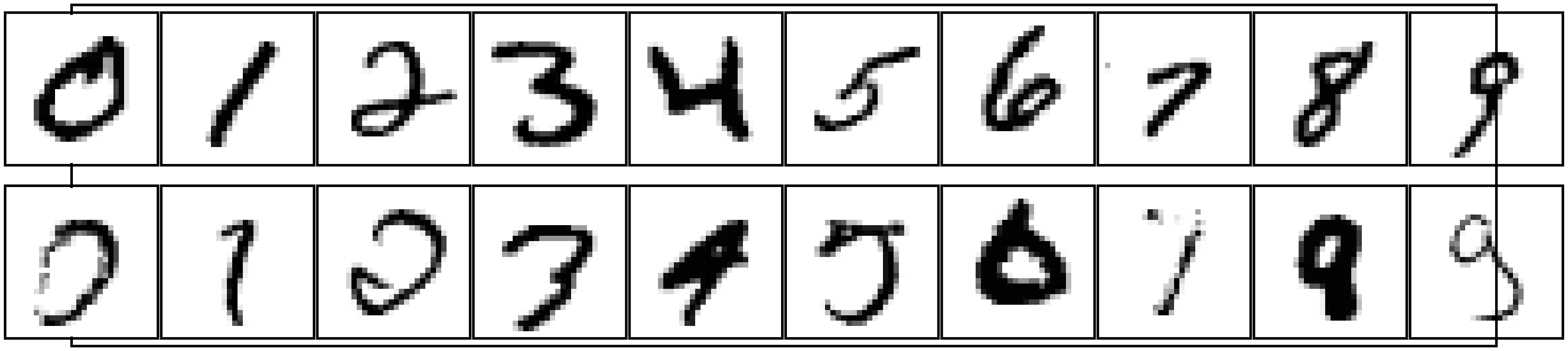
Back-propagation learning contd.



Appropriate problems for neural networks

- Instances are represented by many attribute-value pairs.
- The target function may be discrete-valued, real-value, or a vector of values.
- The training examples may contain errors.
- Long training times are acceptable.
- Fast evaluation of the learned target function may be required.
- The ability of humans to understand the learned target function is not important.

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

Summary

Most brains have lots of neurons; each neuron $\approx$ linear–threshold unit (?)

Perceptrons (one-layer networks) insufficiently expressive

Multi-layer networks are sufficiently expressive; can be trained by gradient descent, i.e., error back-propagation

Many applications: speech, driving, handwriting, fraud detection, etc.

Engineering, cognitive modelling, and neural system modelling subfields have largely diverged