Last update: December 4, 2008

Neural networks

CMSC 421: Chapter 20, Section 5

Outline

♢ Brains

♢ Neural networks

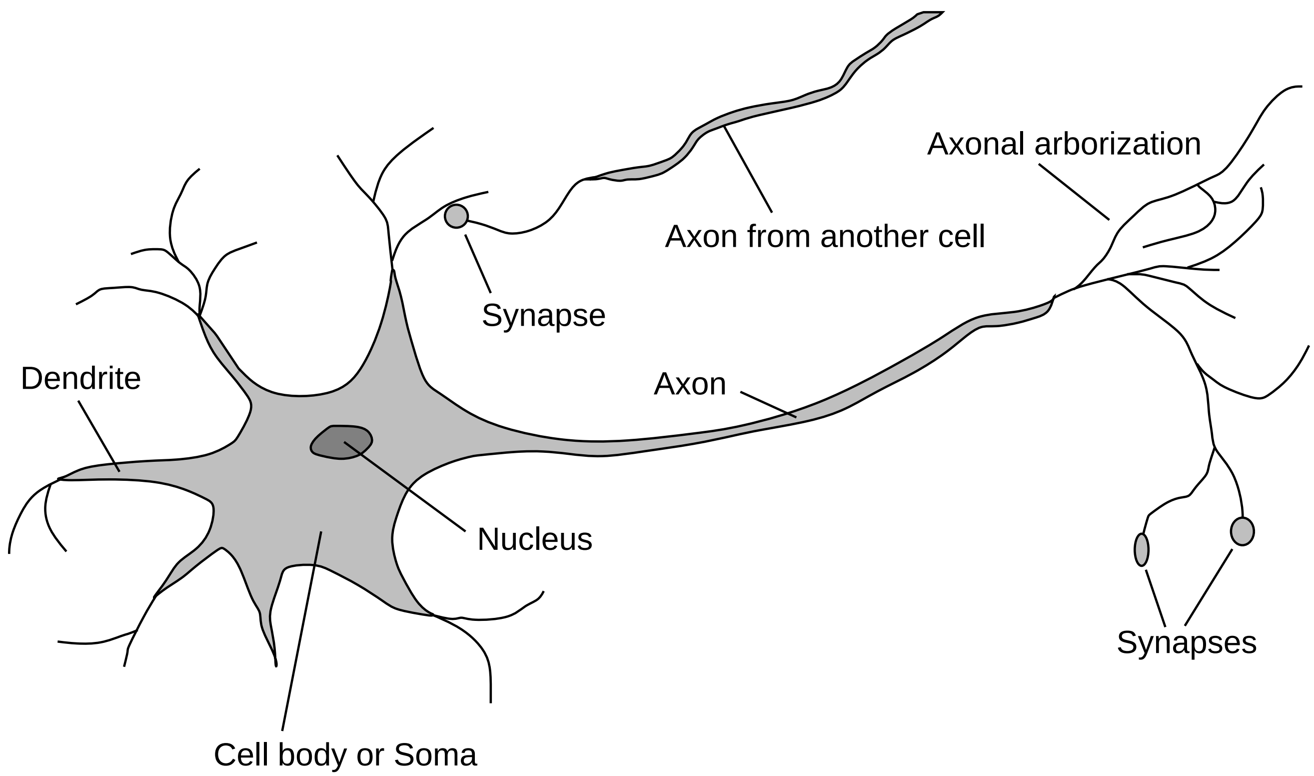
♢ Perceptrons

♢ Multilayer perceptrons

♢ Applications of neural networks

Brains

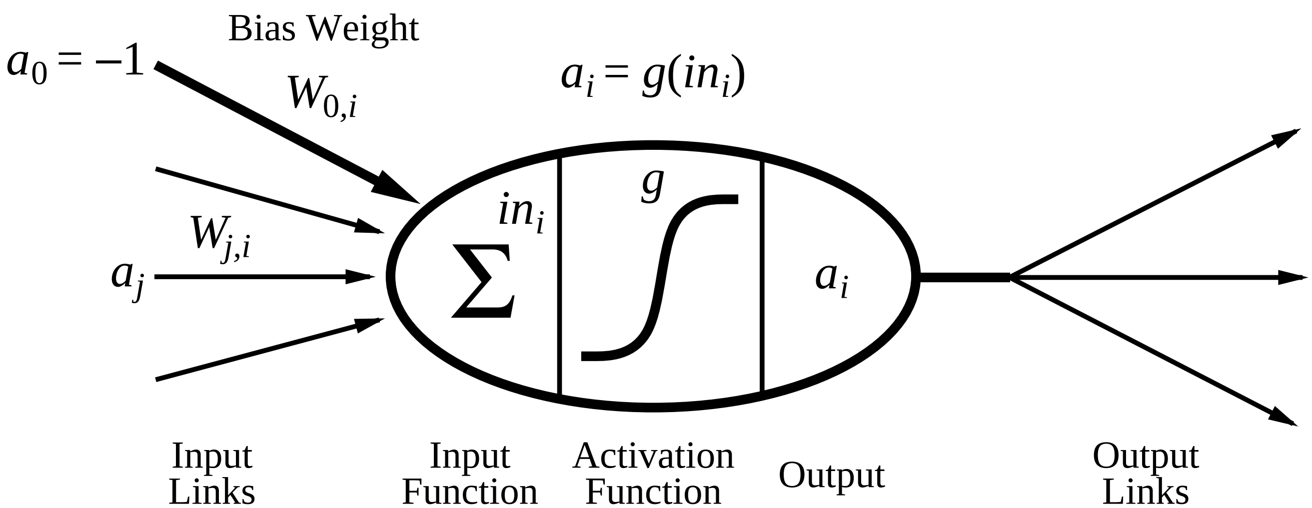
10^{11} neurons of >20 types, 10^{14} synapses, 1ms–10ms cycle time
Signals are noisy "spike trains" of electrical potential



McCulloch–Pitts "unit"

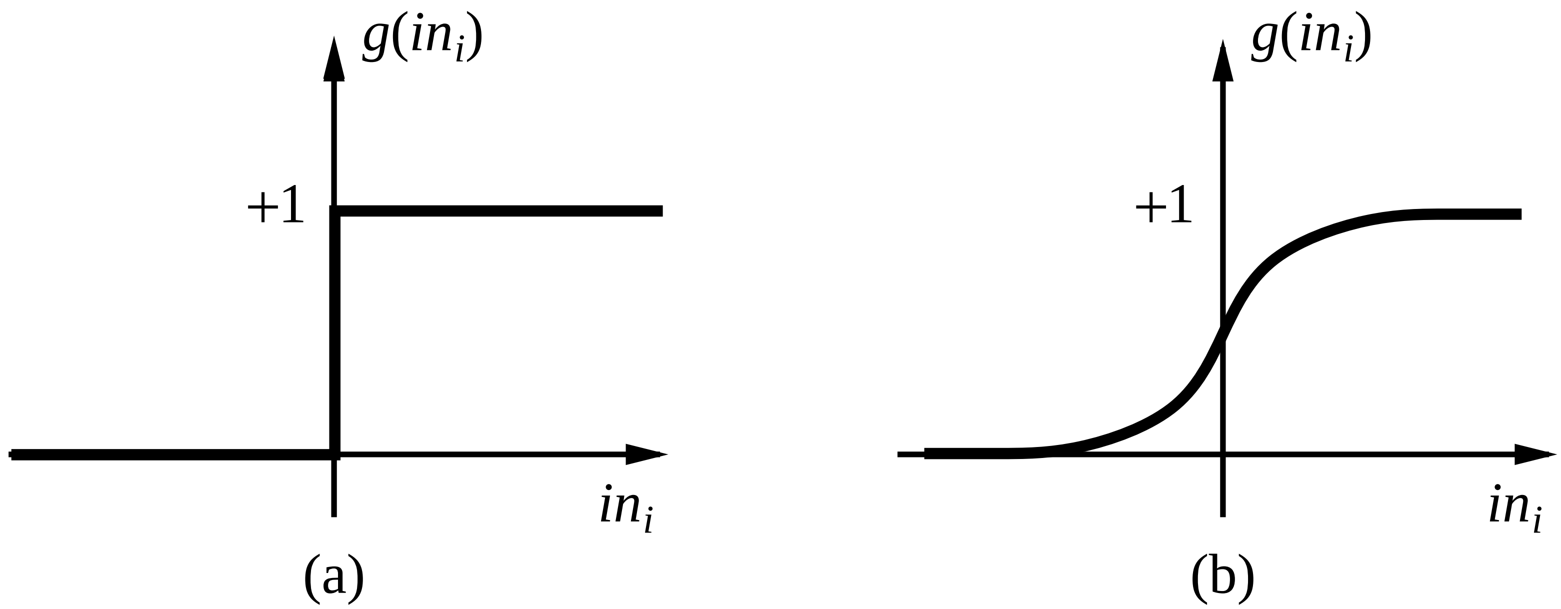
Output depends on a weighted sum of the inputs:

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



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

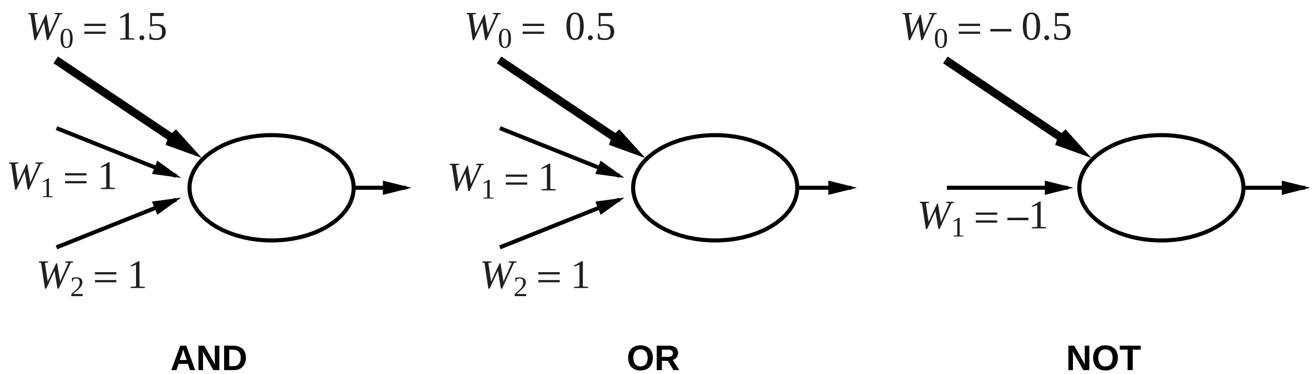
$g(in_i) = 1$ if $in_i > 0$; $g(in_i) = 0$ otherwise

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

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

Implementing logical functions

Use step function as activation function



McCulloch and Pitts: every Boolean function can be implemented as a collection of units

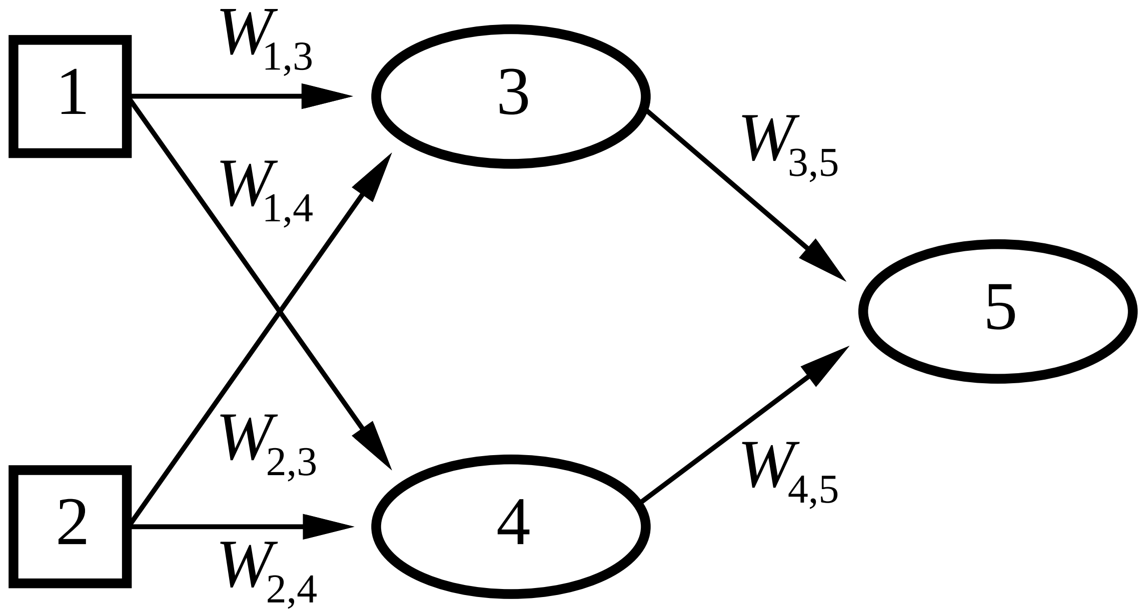
Network structures

Feed-forward networks:
- single-layer perceptrons
- multi-layer perceptrons

Feed-forward networks implement functions, have no internal state

Recurrent networks:
- Hopfield networks have symmetric weights ($W_{i,j} = W_{j,i}$)
  $g(x) = \mathsf{sign}(x)$, $a_i = \pm 1$; **holographic associative memory**
- Boltzmann machines use stochastic activation functions
- recurrent neural nets have directed cycles with delays
  $\Rightarrow$ have internal state (like flip-flops), can oscillate etc.

Feed-forward example

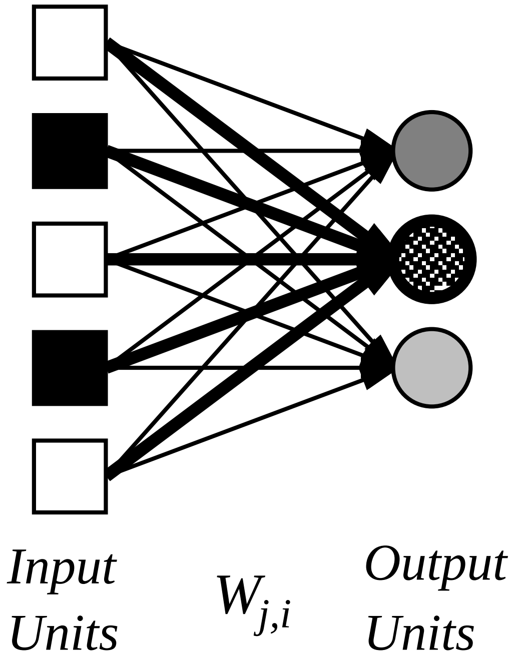


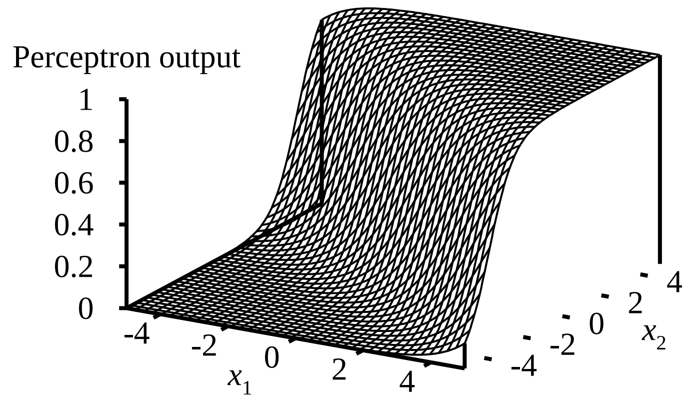
Feed-forward network = a parameterized family of nonlinear functions:

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

Adjusting weights changes the function: do learning this way!

Perceptrons

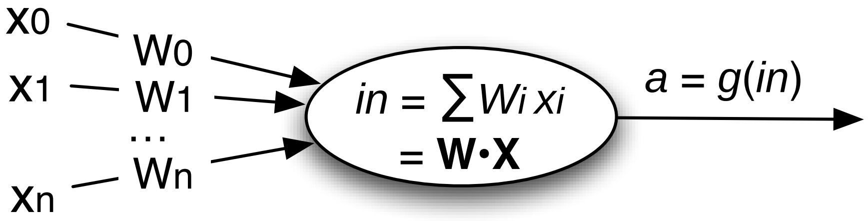




Output units all operate separately—no shared weights

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

Perceptron learning



Learn by adjusting weights to reduce error on training set

The squared error for an example with input $\mathbf{x}$ and true output y is

$$E = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y-a)^2$$

Perform optimization search by gradient descent:

$$\begin{aligned}\frac{\partial E}{\partial W_j} &= Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j}\left(y - g(\Sigma_{j=0}^{n} W_j x_j)\right) \\ &= -Err \times g'(in) \times x_j\end{aligned}$$

Simple weight update rule:

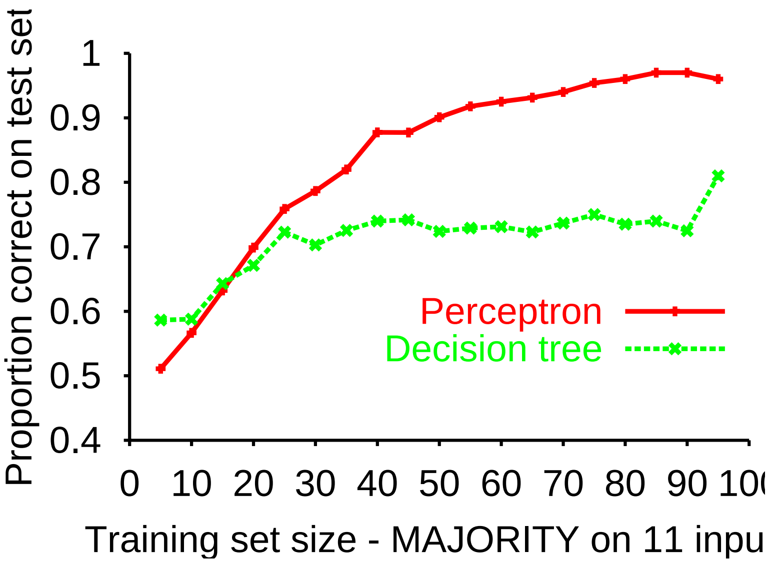
$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$$

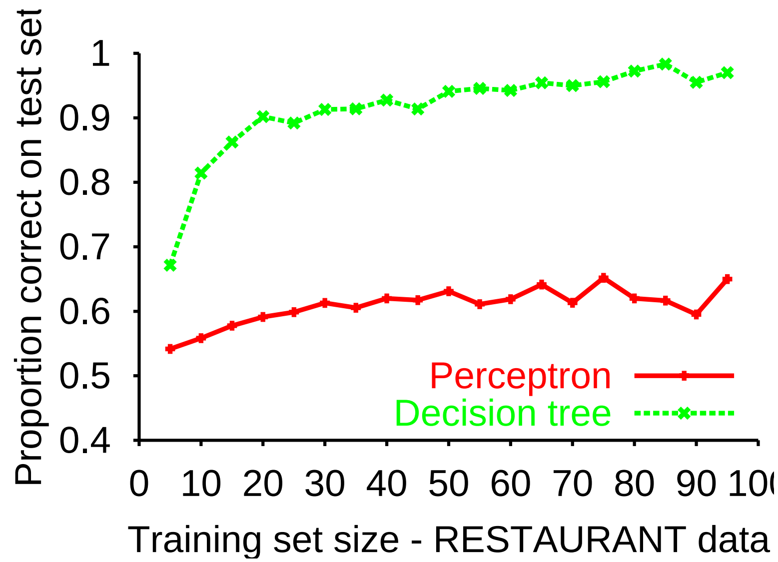
E.g., positive error $\Rightarrow$ increase network output

$\Rightarrow$ increase weights on pos. inputs, decrease on neg. inputs

Perceptron learning, continued

Perceptron learning rule converges to a consistent function
for any linearly separable data set





Perceptron learns majority function easily.

DTL doesn't: majority is representable, but only as a very large decision tree, DTL won't learn that without a very large data set.

DTL learns restaurant function easily, perceptron cannot represent it

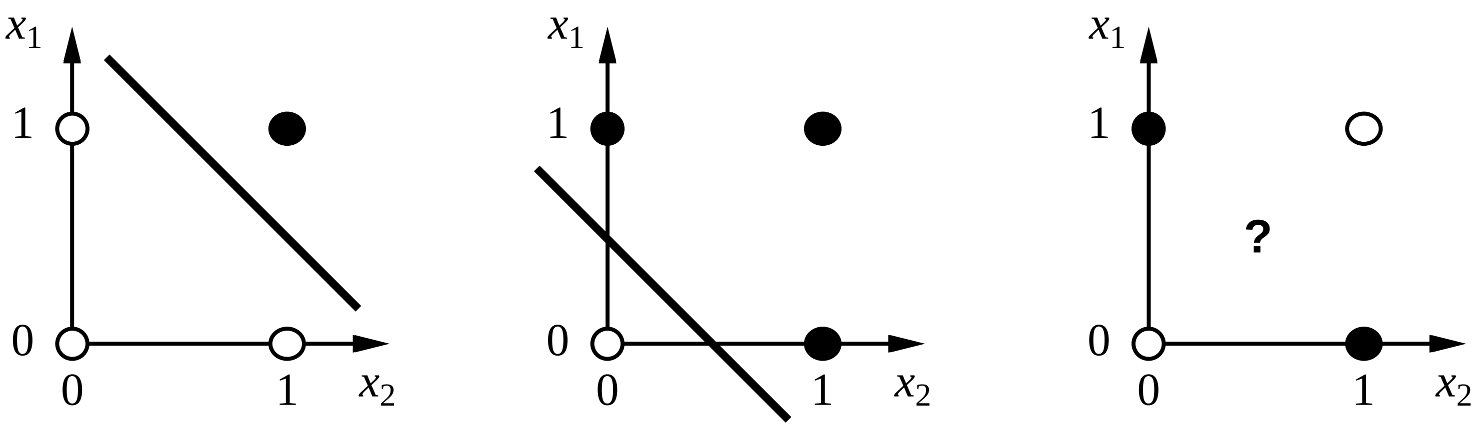
Expressiveness of perceptrons

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:

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

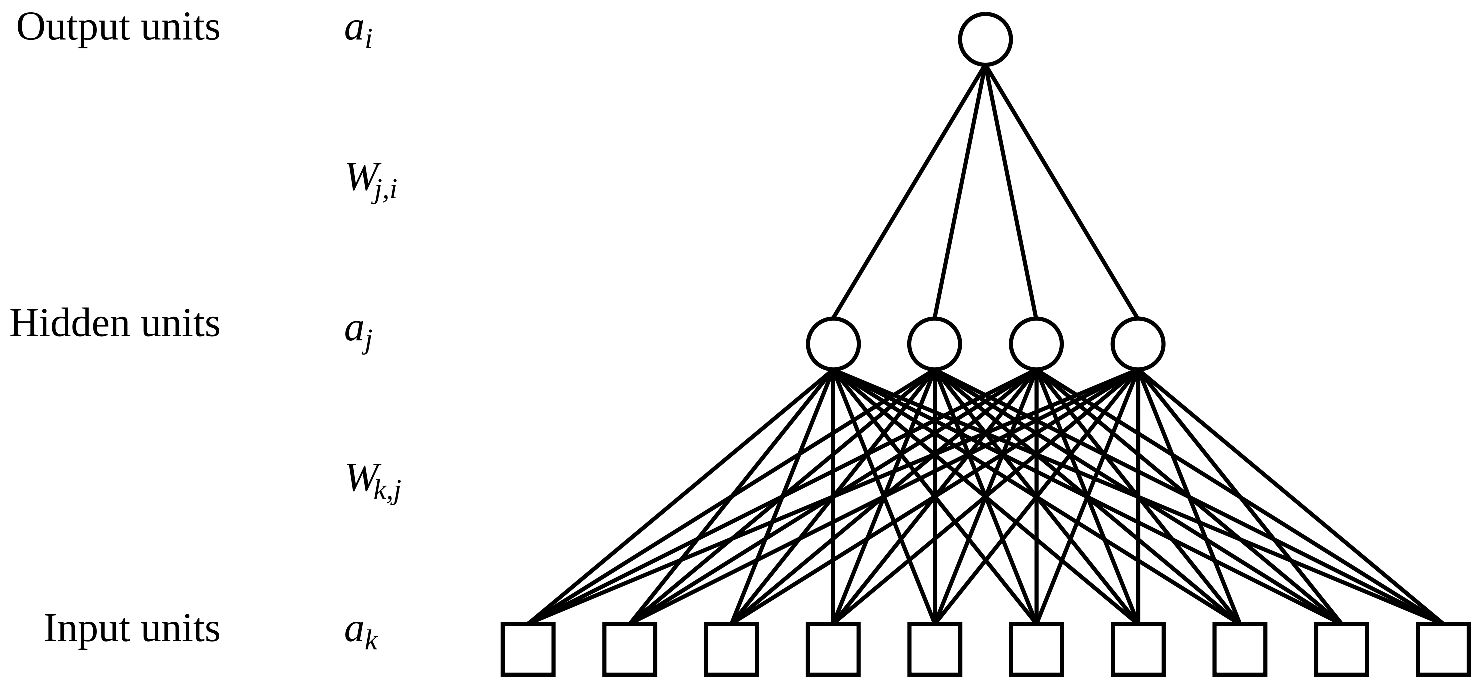


(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_2

Minsky & Papert (1969) pricked the neural network balloon

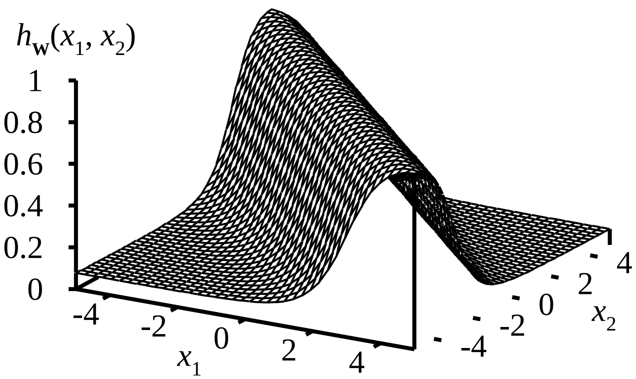
Multilayer feed-forward networks

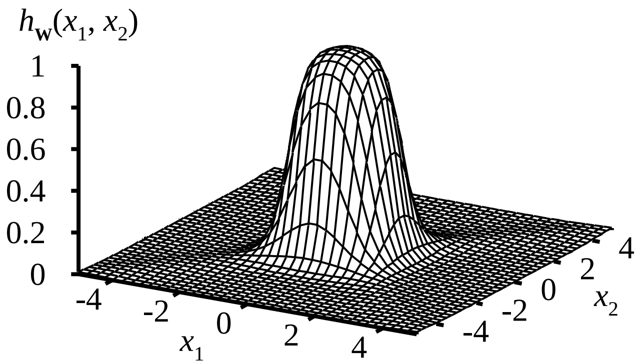
Layers are usually fully connected;
numbers of hidden units typically chosen by hand



Expressiveness of MLPs

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





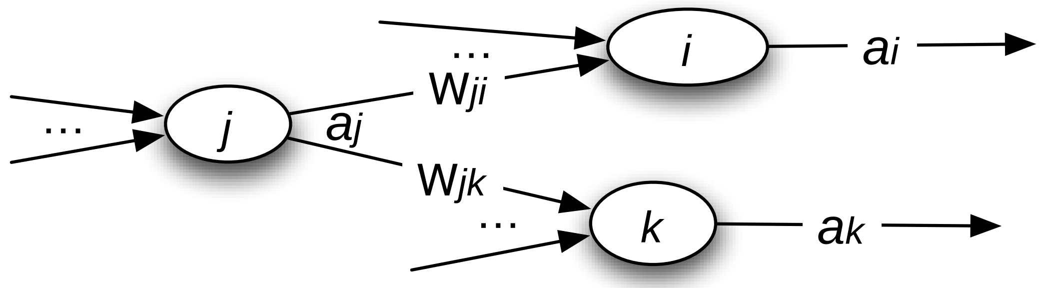
Combine two opposite-facing threshold functions to make a ridge

Combine two perpendicular ridges to make a bump

Add bumps of various sizes and locations to fit any surface

Proof requires exponentially many hidden units (like in the DTL proof)

Back-propagation learning



Output layer: same as for single-layer perceptron,

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

where $\Delta_i = Err_i \times g'(in_i)$

Hidden layer: **back-propagate** the error from the output layer:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i \ .$$

Update rule for weights in hidden layer:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j \ .$$

(Most neuroscientists deny that back-propagation occurs in the brain)

Back-propagation derivation

Like perceptron learning, back-propagation is optimization search by gradient descent

The squared error on a single example is defined as

$$E = \frac{1}{2}\sum_i (y_i - a_i)^2 \ ,$$

where the sum is over the nodes in the output layer.

$$\begin{aligned}
\frac{\partial E}{\partial W_{j,i}} &= -(y_i - a_i)\frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i)\frac{\partial g(in_i)}{\partial W_{j,i}} \\
&= -(y_i - a_i)g'(in_i)\frac{\partial in_i}{\partial W_{j,i}} = -(y_i - a_i)g'(in_i)\frac{\partial}{\partial W_{j,i}}\left(\sum_j W_{j,i}a_j\right) \\
&= -(y_i - a_i)g'(in_i)a_j = -a_j\Delta_i
\end{aligned}$$

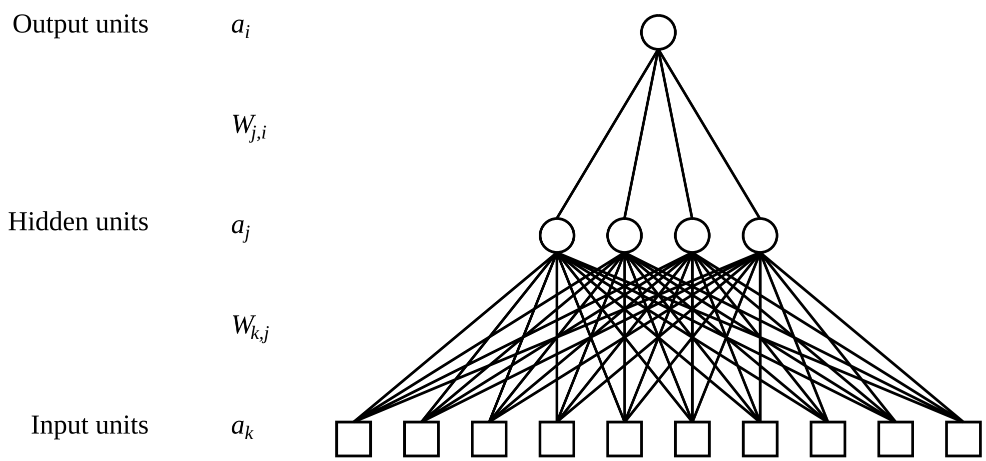
Back-propagation derivation, continued

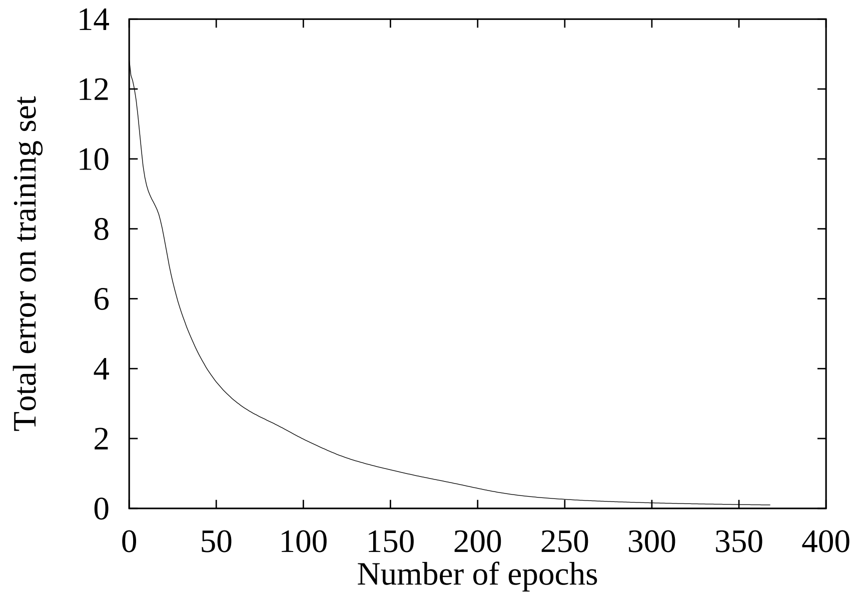
$$
\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i)\frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i)\frac{\partial g(in_i)}{\partial W_{k,j}} \\
&= -\sum_i (y_i - a_i) g'(in_i)\frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}}\left(\sum_j W_{j,i} a_j\right) \\
&= -\sum_i \Delta_i W_{j,i}\frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i}\frac{\partial g(in_j)}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial in_j}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial}{\partial W_{k,j}}\left(\sum_k W_{k,j} a_k\right) \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j
\end{aligned}
$$

Back-propagation learning, continued

At each epoch, sum gradient updates for all examples and apply

Training curve for 100 restaurant examples: finds exact fit

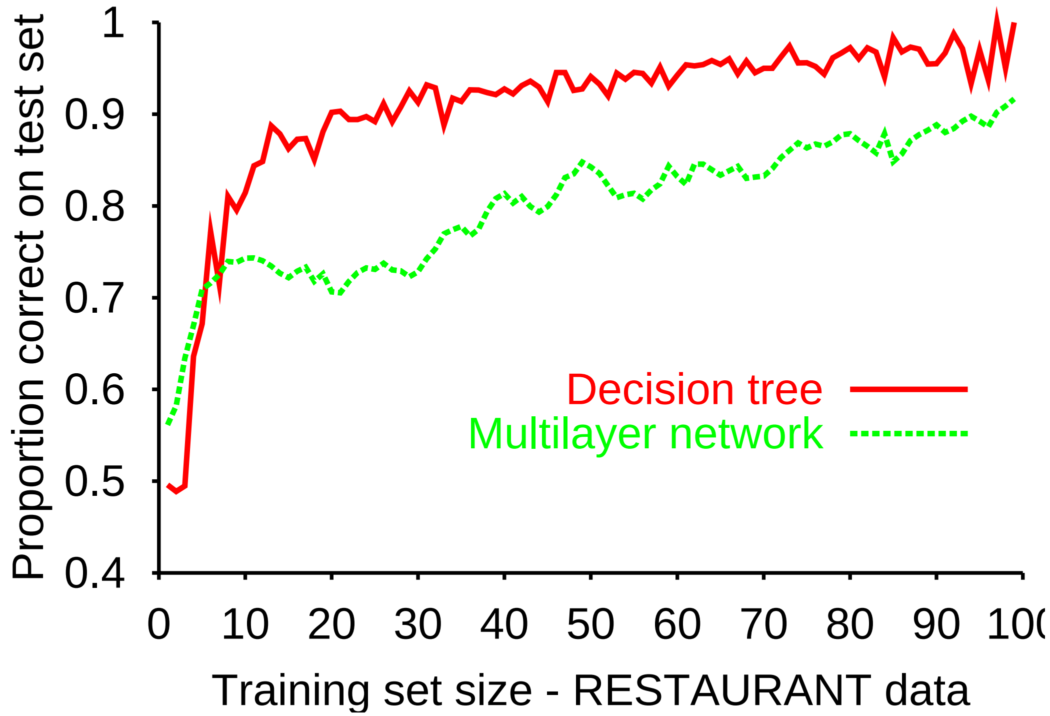




Typical problems: slow convergence, local minima

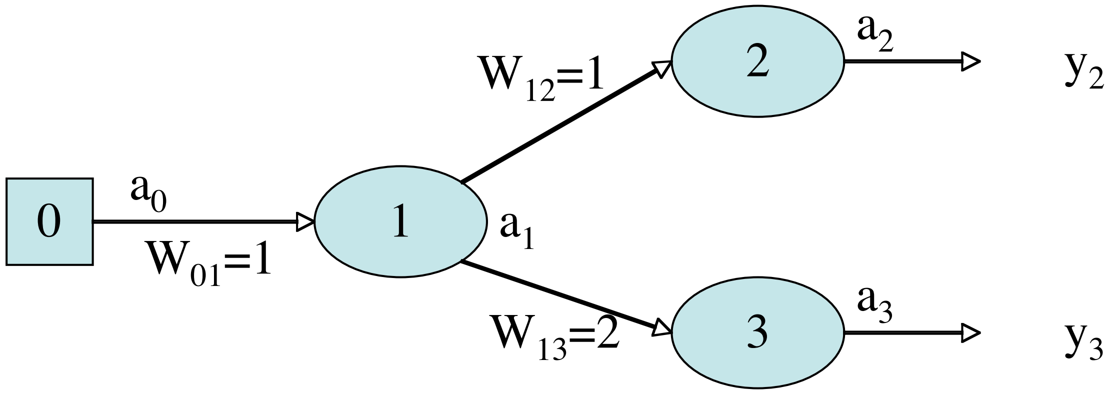
Back-propagation learning, continued

Learning curve for multi-layer perceptron with 4 hidden units:



Multi-layer perceptrons are quite good for complex pattern recognition tasks, but resulting hypotheses cannot be understood easily

Computational Example



a_i = output of unit i; W_{ij} = weight of link from unit i to unit j

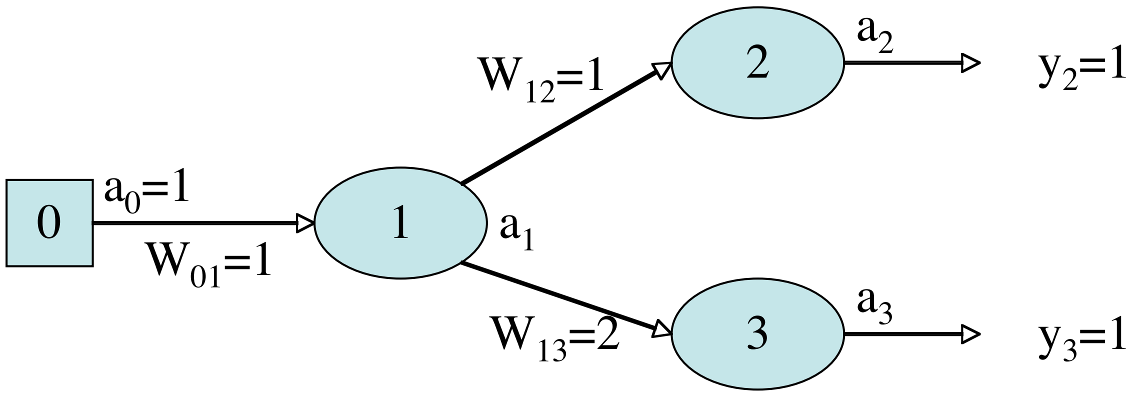
For output units, $\Delta_j = g'(in_j)(y_j - a_j)$. For hidden units, $\Delta_j = g'(in_j)\Sigma_i w_{ji}\Delta_i$.

Suppose $g(in_i) = in_i$ for every unit i. Then $g'(in_i) = 1$.
(Note: one wouldn't use such a g in practice!)

Problem 1. For what values of Δ_2 and Δ_3 will $\Delta_1 = 0$?

$0 = \Delta_1 = g'(in_1)(W_{12}\Delta_2 + W_{13}\Delta_3) = W_{12}\Delta_2 + W_{13}\Delta_3 = \Delta_2 + 2\Delta_3$
i.e., $\Delta_2 = -2\Delta_3$.

Computational Example

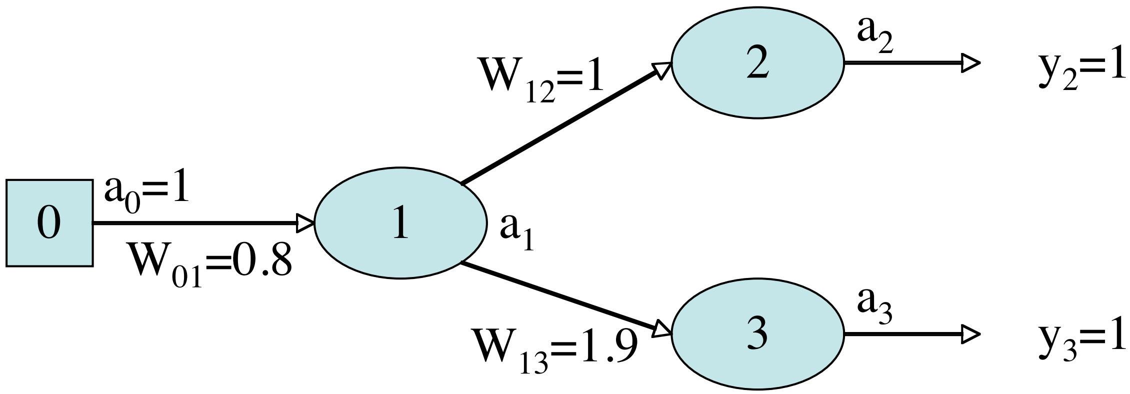


Problem 2. Let $\alpha = 0.1$.
Suppose there is just one training example: $a_0 = 1, y_2 = 1, y_3 = 1$.
First epoch:

$$a_1 = g(in_1) = W_{01}a_0 = 1 * 1 = 1 \qquad \Delta_2 = y_2 - a_2 = 1 - 1 = 0$$
$$a_2 = g(in_2) = W_{12}a_1 = 1 * 1 = 1 \qquad \Delta_3 = y_3 - a_3 = 1 - 2 = -1$$
$$a_3 = g(in_3) = W_{13}a_1 = 2 * 1 = 2 \qquad \Delta_1 = \Delta_2 + 2\Delta_3 = 0 + 2(-1) = -2$$

$$W_{12} \leftarrow W_{12} + \alpha a_1 \Delta_2 = 1 + 0.1 * 1 * 0 = 1$$
$$W_{13} \leftarrow W_{13} + \alpha a_1 \Delta_3 = 2 + 0.1 * 1 * (-1) = 1.9$$
$$W_{01} \leftarrow W_{01} + \alpha a_0 \Delta_1 = 1 + 0.1 * 1 * (-2) = 0.8$$

Computational Example



Second epoch:

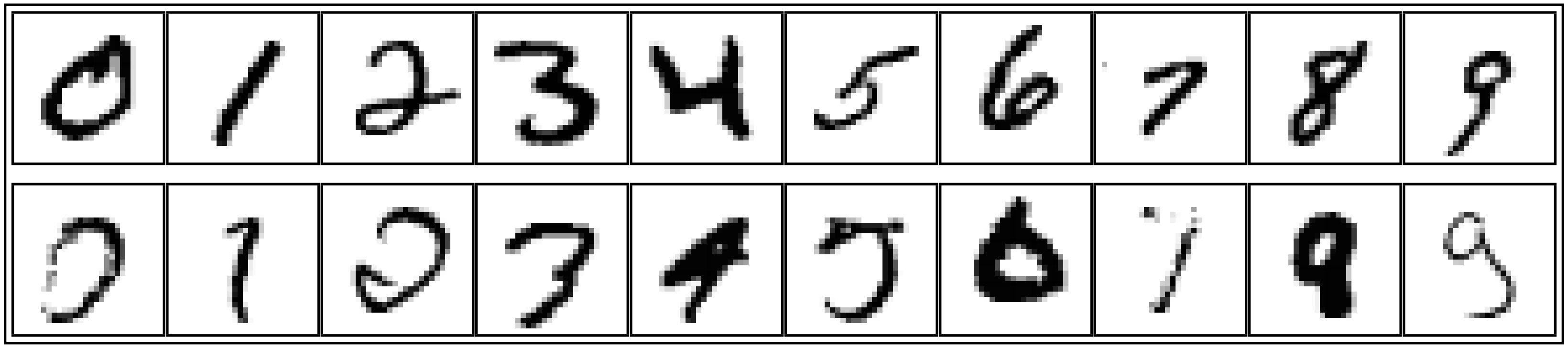
$$a_1 = g(in_1) = W_{01}a_0 = 0.8 * 1 = 0.8$$
$$a_2 = g(in_2) = W_{12}a_1 = 1 * 0.8 = 0.8$$
$$a_3 = g(in_3) = W_{13}a_1 = 1.9{*}0.8 = 1.52$$

$$\Delta_2 = y_2 - a_2 = 1 - 0.8 = 0.2$$
$$\Delta_3 = y_3 - a_3 = 1 - 1.52 = -0.52$$

$$\Delta_1 = W_{12}\Delta_2 + W_{13}\Delta_3 = 1 * 0.2 + 1.9 * (-0.52) = 0.2 - 0.988 = -0.788$$

$$W_{12} \leftarrow W_{12} + \alpha a_1 \Delta_2 = 1 + 0.1 * 0.8 * 0.2 = 1.016$$
$$W_{13} \leftarrow W_{13} + \alpha a_1 \Delta_3 = 1.9 + 0.1 * 0.8 * (-0.52) = 1.8584$$
$$W_{01} \leftarrow W_{01} + \alpha a_0 \Delta_1 = 0.8 + 0.1 * 1 * (-0.788) = 0.7212.$$

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