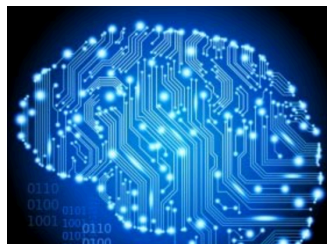




# Lecture 13: Learning 2

[http://cs.nju.edu.cn/yuy/course\\_ai16.ashx](http://cs.nju.edu.cn/yuy/course_ai16.ashx)



# Previously...

## Learning

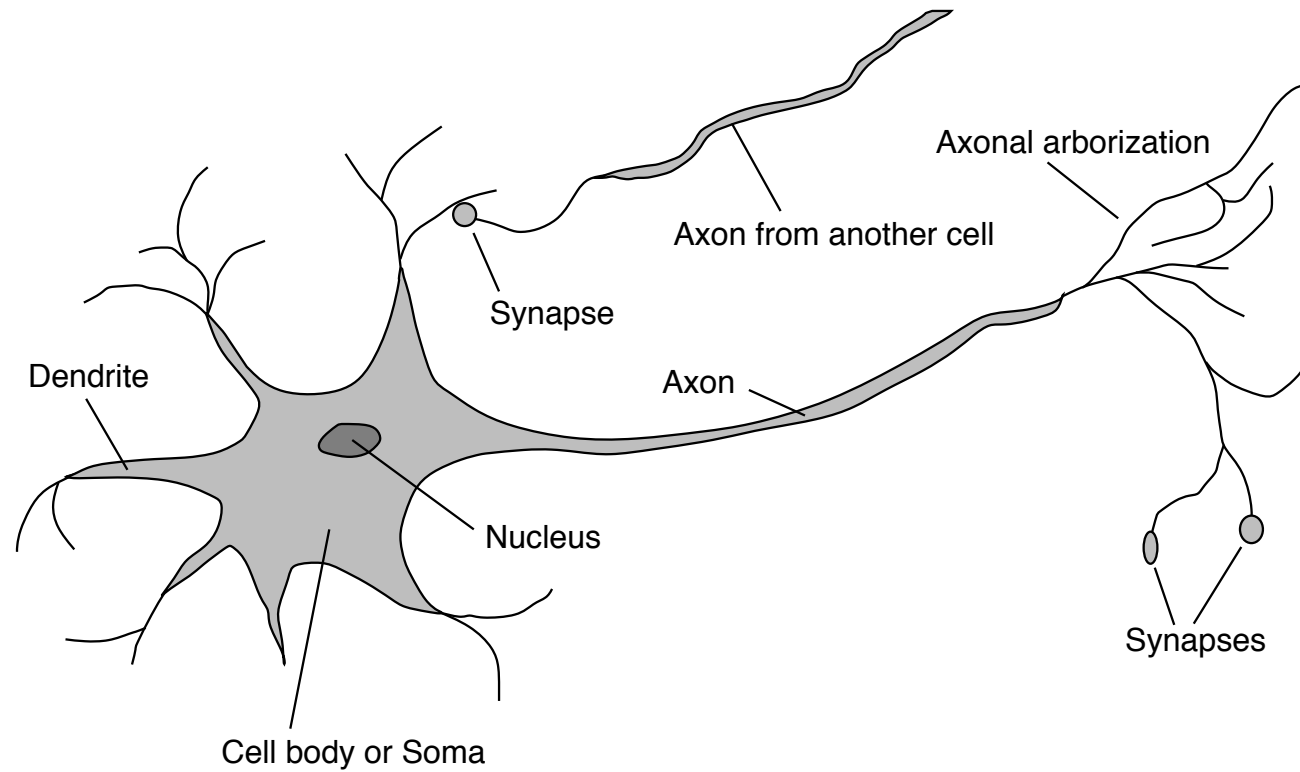
### Decision tree learning



# Brains



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

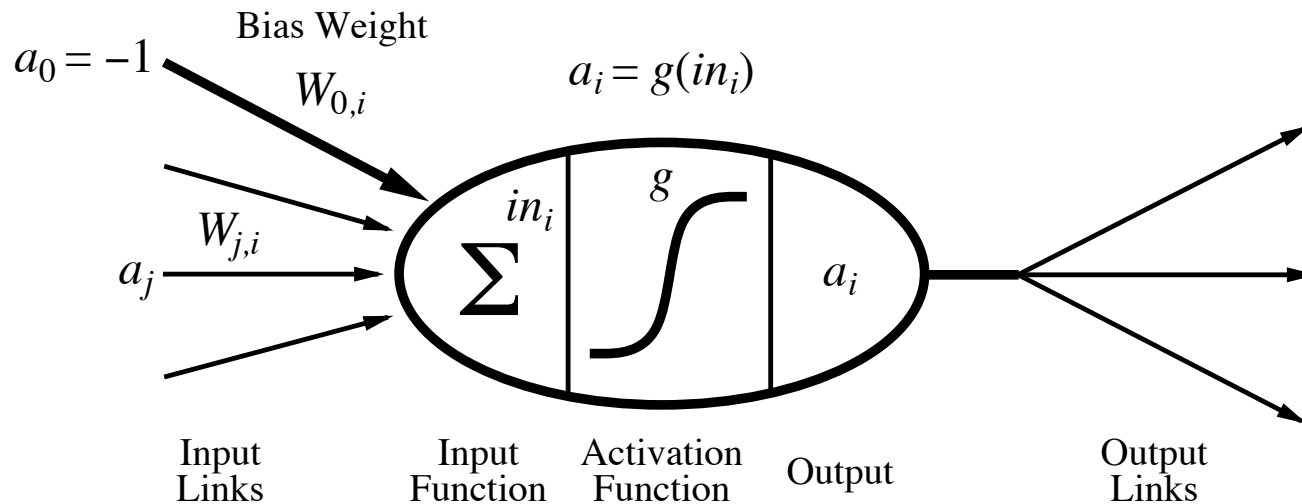


# McCulloch-Pitts “unit”



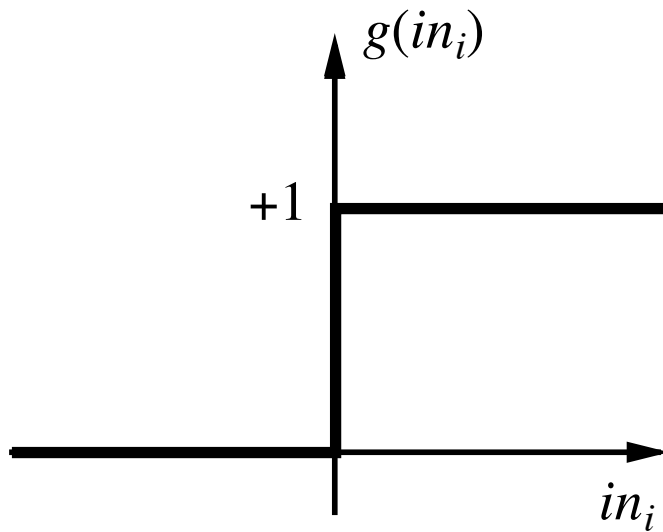
Output is a “squashed” linear function of the inputs:

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

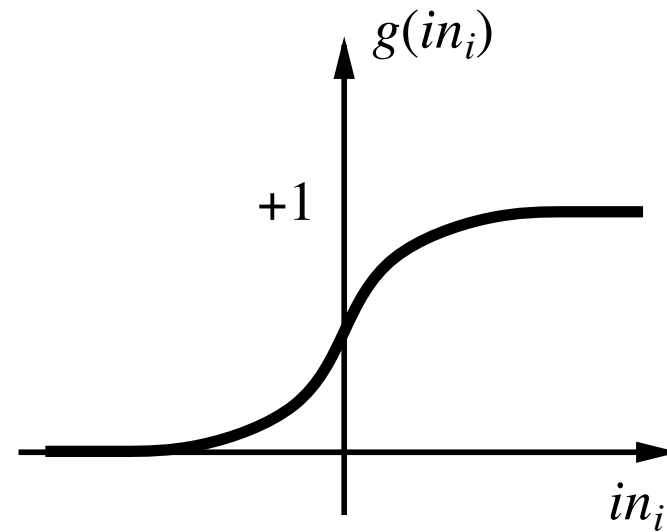


A gross oversimplification of real neurons, but its purpose is to develop understanding of what networks of simple units can do

# Activation functions



(a)



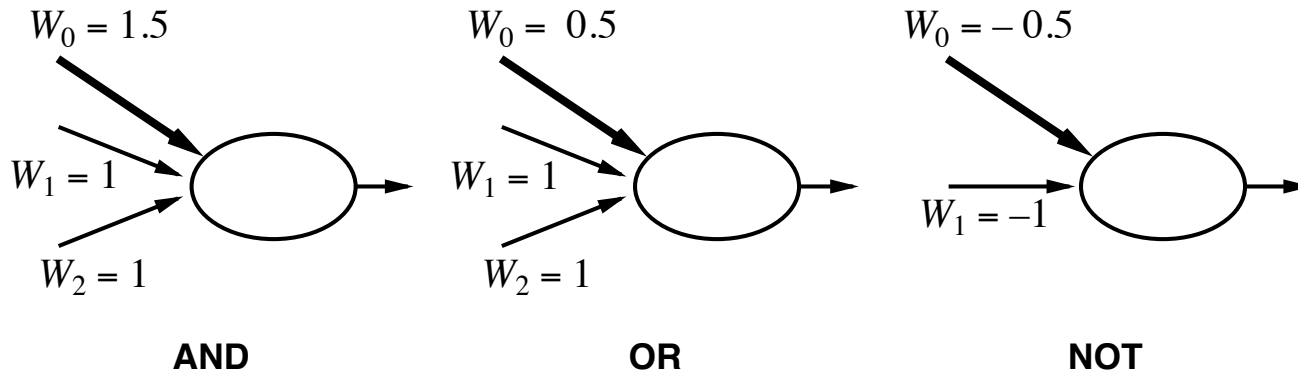
(b)

(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:

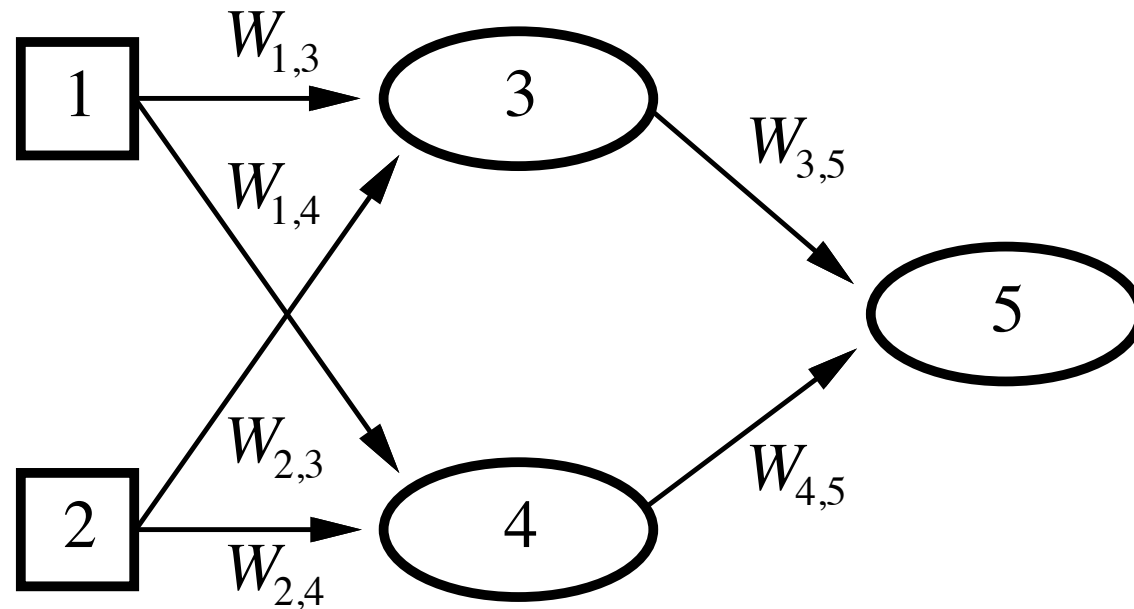
- single-layer perceptrons
- multi-layer perceptrons

Feed-forward networks implement functions, have no internal state

## Recurrent networks:

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

# Feed-forward example



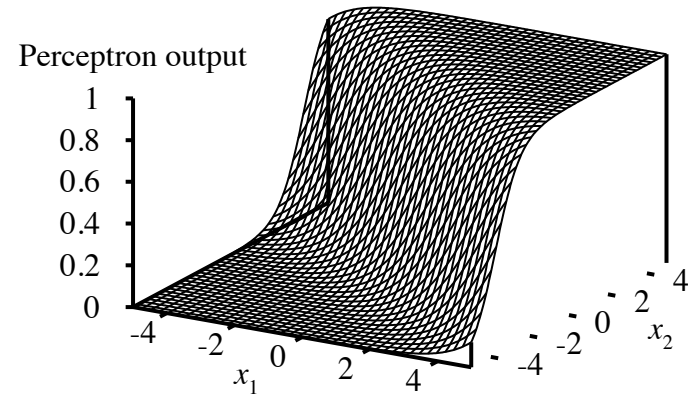
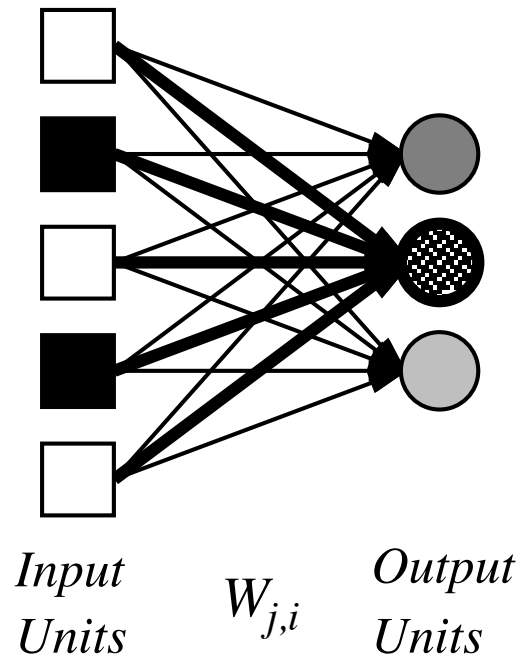
Feed-forward network = a parameterized family of nonlinear functions:

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

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

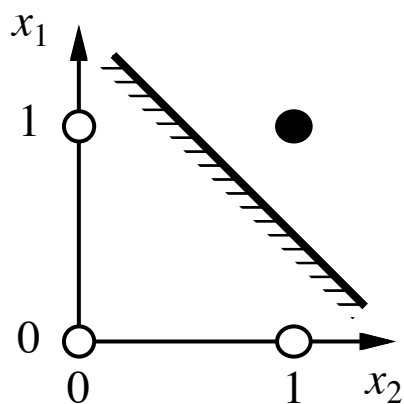


Consider a perceptron with  $g = \text{step function}$  (Rosenblatt, 1957, 1960)

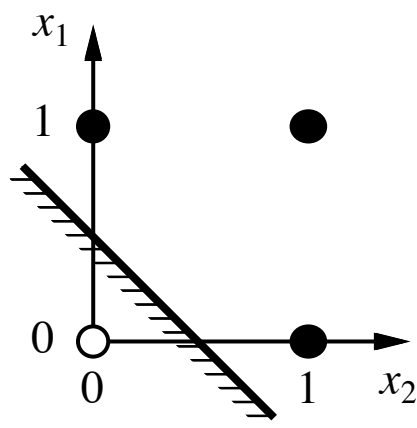
Can represent AND, OR, NOT, majority, etc., but not XOR

Represents a **linear separator** in input space:

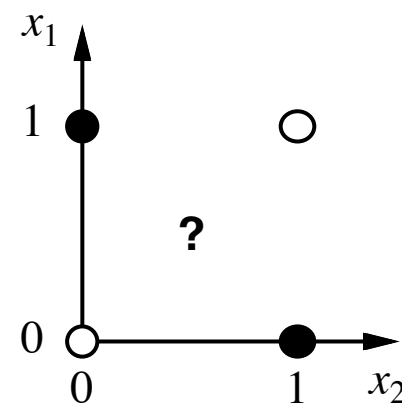
$$\sum_j W_j x_j > 0 \quad \text{or} \quad \mathbf{W} \cdot \mathbf{x} > 0$$



(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



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

$$E = \frac{1}{2} \text{Err}^2 \equiv \frac{1}{2} (y - h_{\mathbf{W}}(\mathbf{x}))^2 ,$$

Perform optimization search by gradient descent:

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

Simple weight update rule:

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

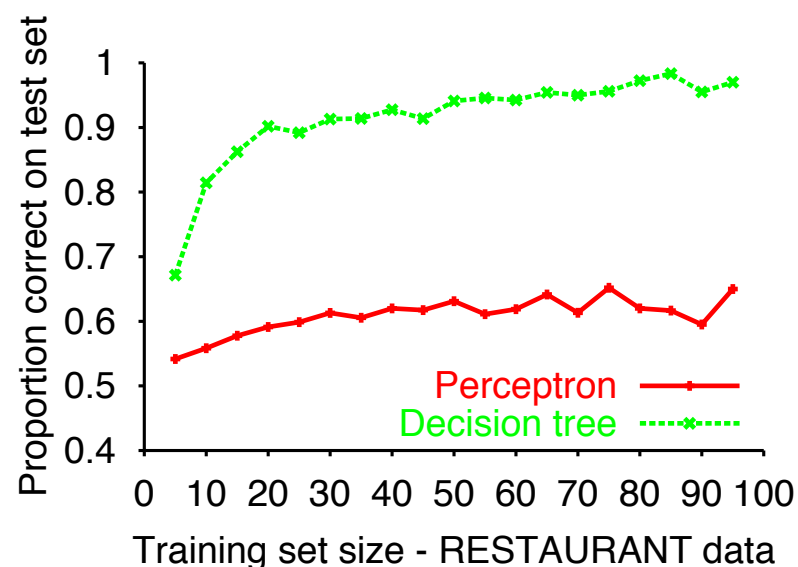
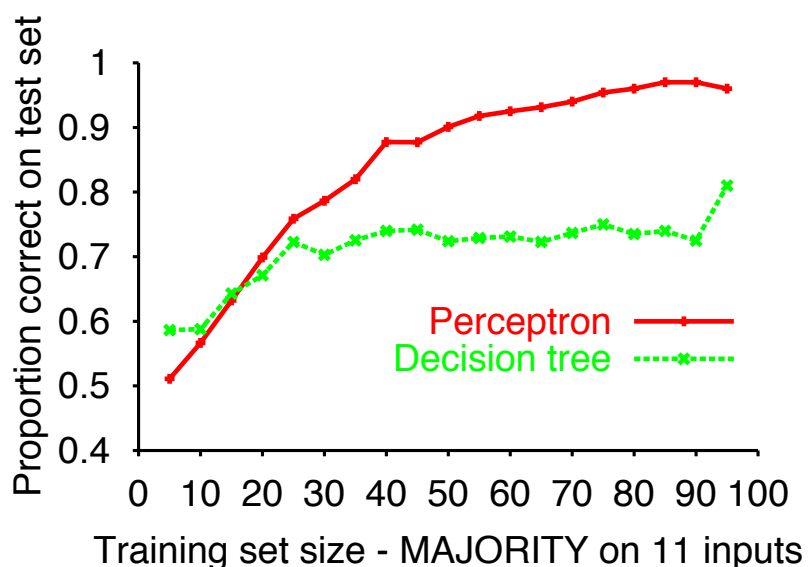
E.g., +ve error  $\Rightarrow$  increase network output

$\Rightarrow$  increase weights on +ve inputs, decrease on -ve inputs

# Perceptron learning contd.



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



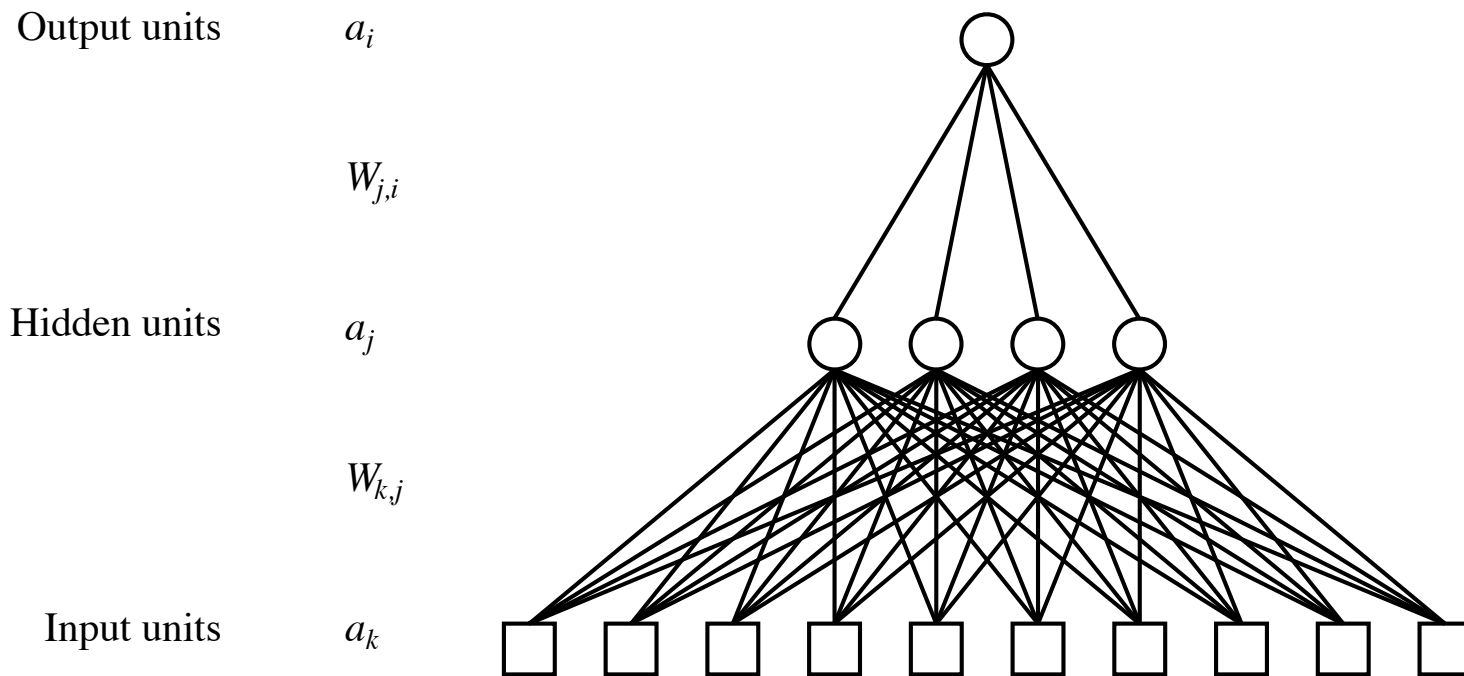
Perceptron learns majority function easily, DTL is hopeless

DTL learns restaurant function easily, perceptron cannot represent it

# Multilayer perceptrons



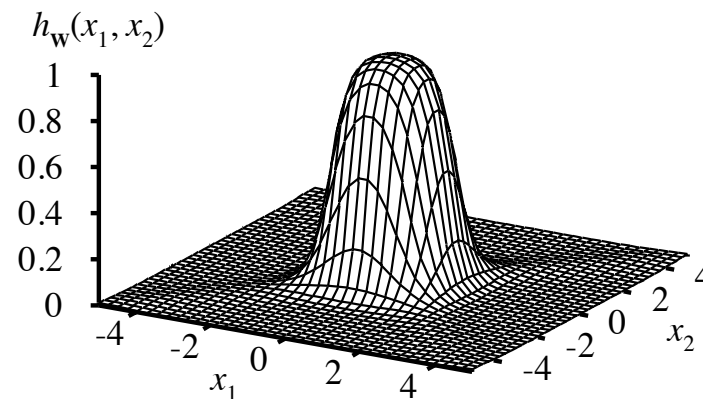
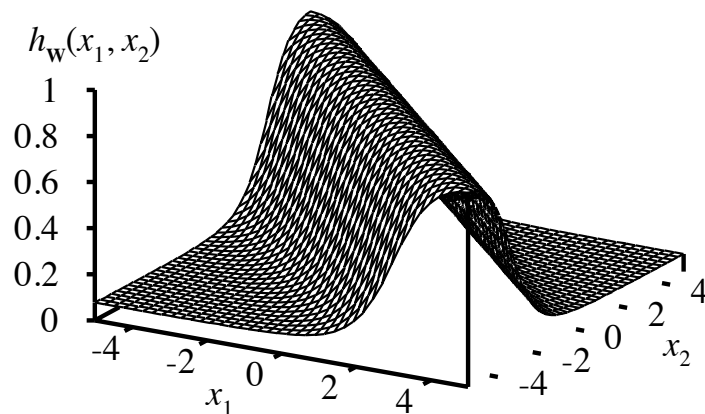
Layers are usually fully connected;  
numbers of **hidden units** typically chosen by hand



# 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

Proof requires exponentially many hidden units (cf 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



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



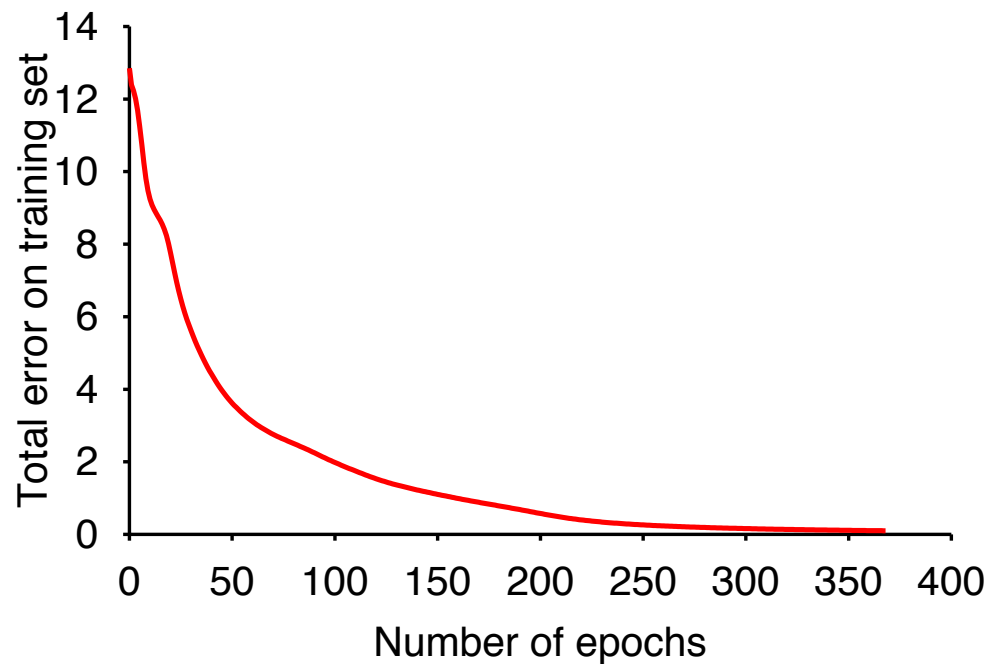
$$\begin{aligned}\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i) \frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i) \frac{\partial g(in_i)}{\partial W_{k,j}} \\ &= -\sum_i (y_i - a_i) g'(in_i) \frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}}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

# Back-propagation learning



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

Training curve for 100 restaurant examples: finds exact fit

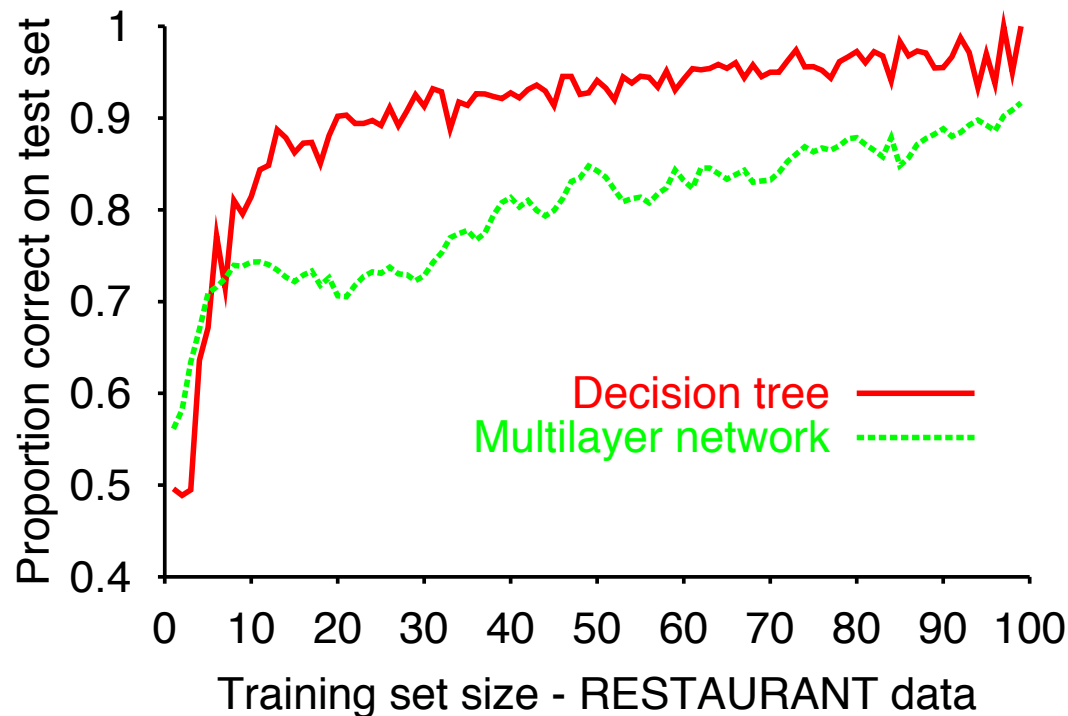


Typical problems: slow convergence, local minima

# Back-propagation learning

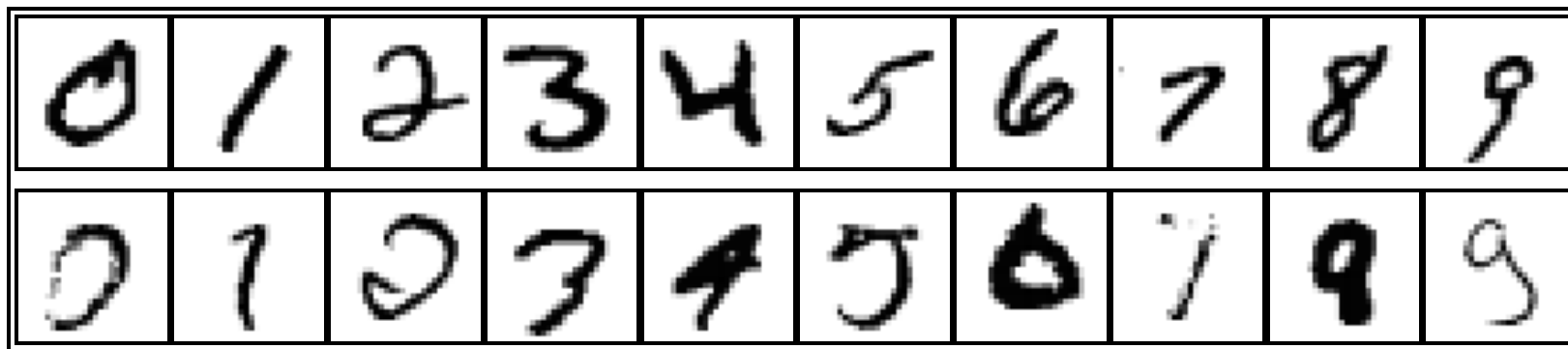


Learning curve for MLP with 4 hidden units:



MLPs are quite good for complex pattern recognition tasks, but resulting hypotheses cannot be understood easily

# 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)  $\approx$  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