Neural networks

Chapter 20

Outline

◇ Brains

◇ Neural networks

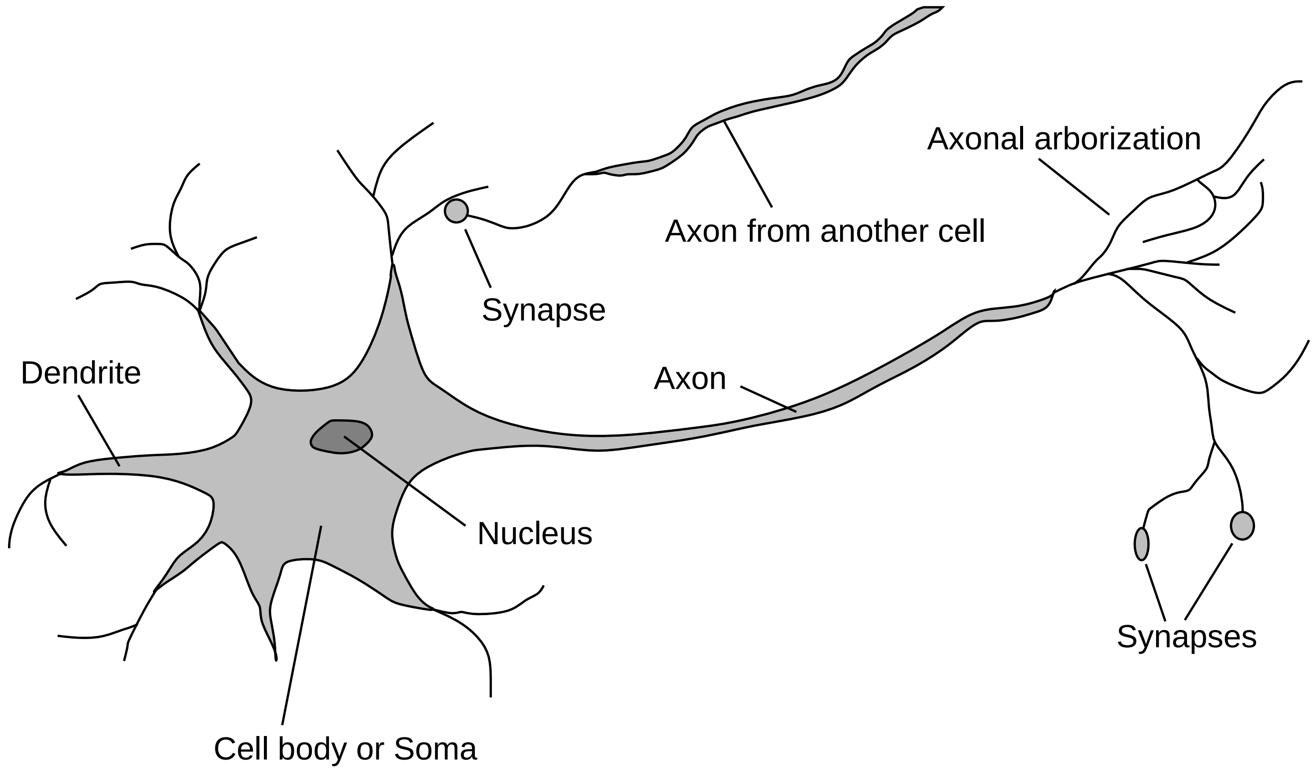
◇ Perceptrons

◇ Multilayer networks

◇ Applications of neural networks

Brains

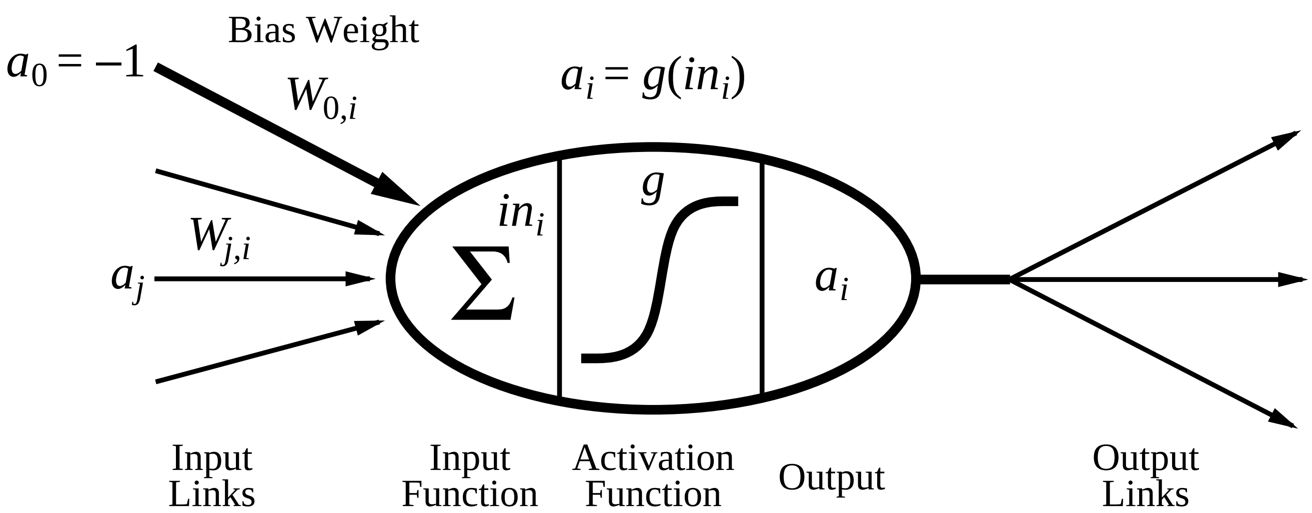
10^{11} neurons of > 20 types, 10^{14} synapses, 1ms–10ms cycle time
Signals are noisy "spike trains" of electrical potential



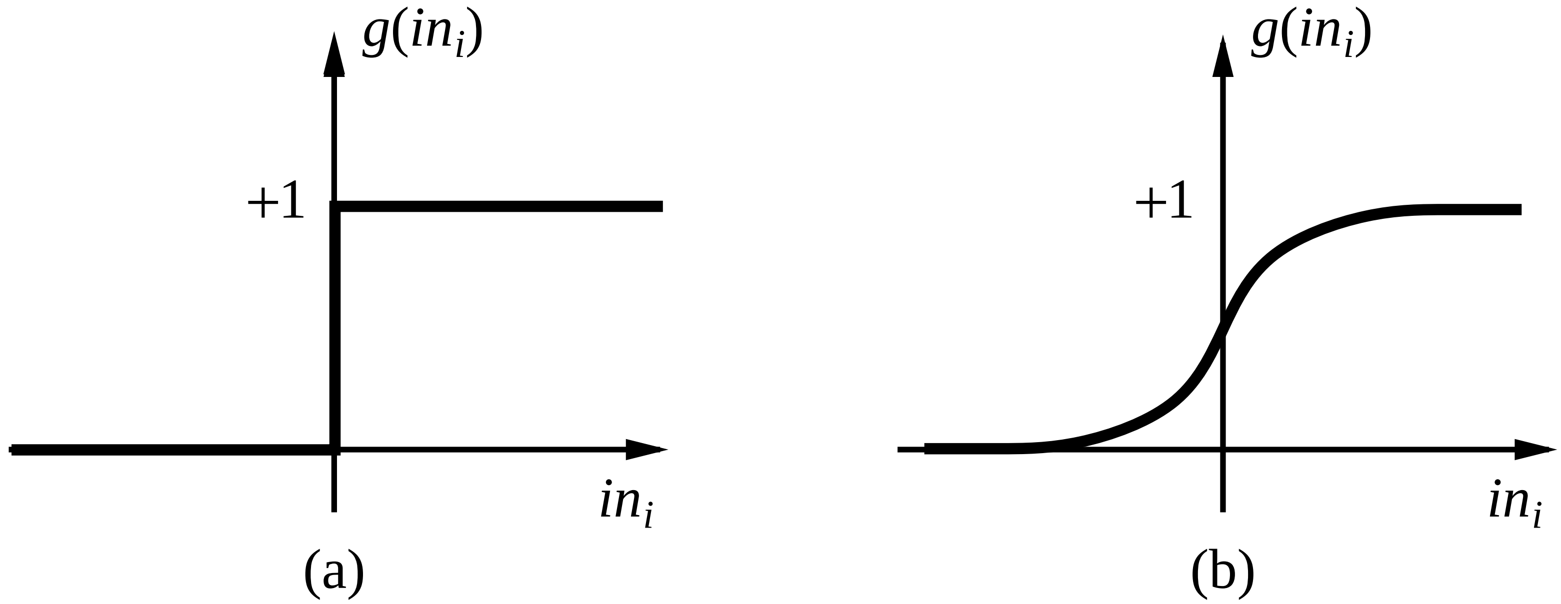
McCulloch–Pitts "unit"

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

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



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 (with large enough network)

AND?

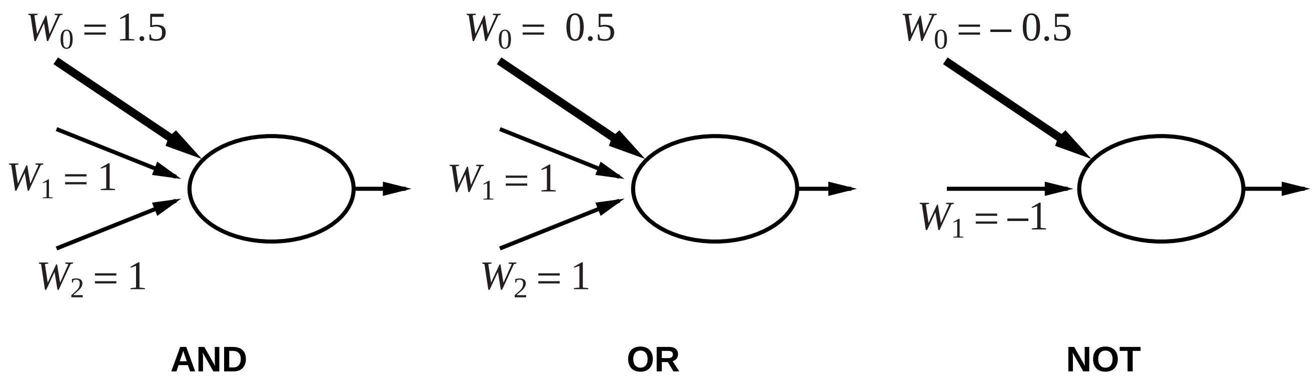
OR?

NOT?

MAJORITY?

Implementing logical functions

McCulloch and Pitts: every Boolean function can be implemented (with large enough network)



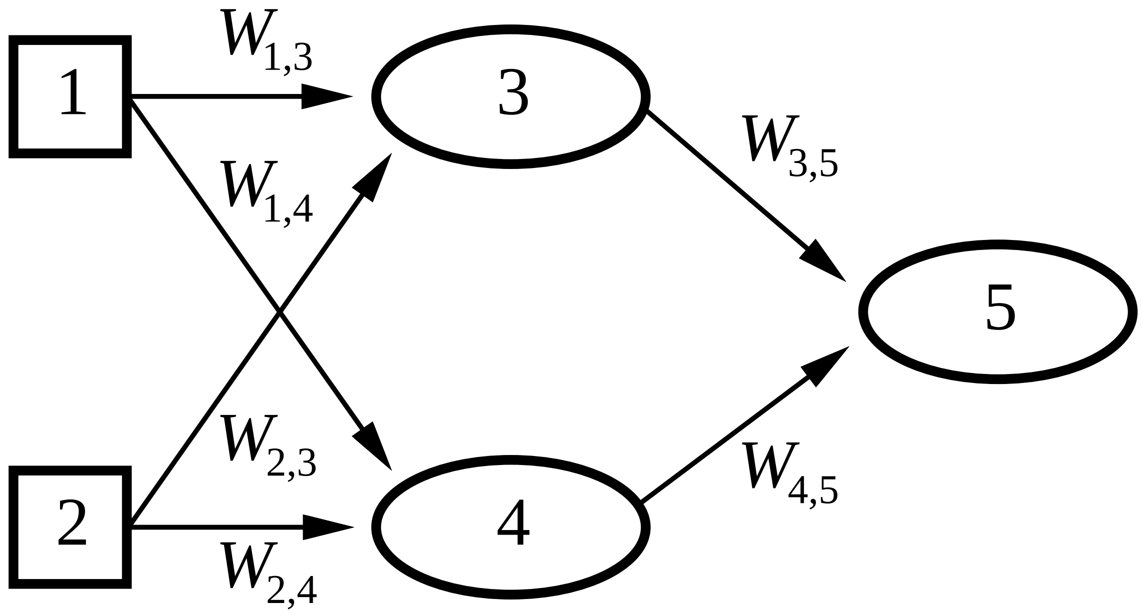
Network structures

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

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

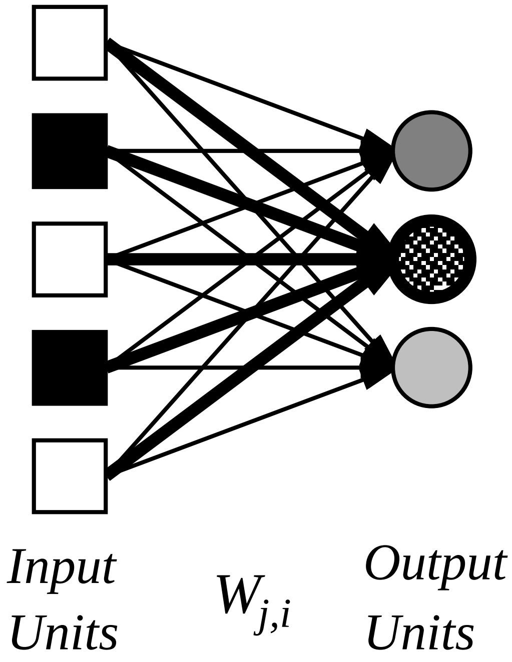
Feed-forward example

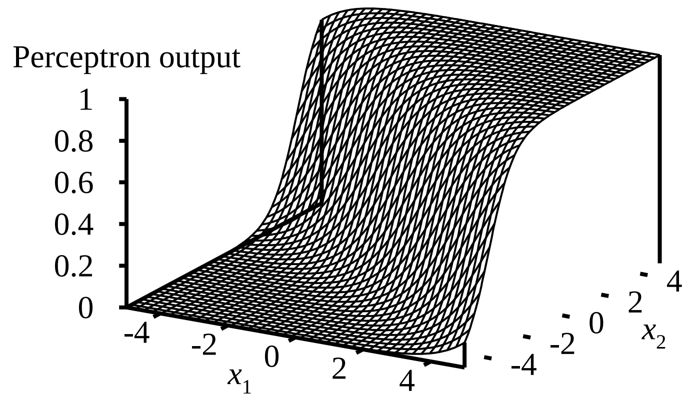


Feed-forward network = a parameterized family of nonlinear functions:

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

Perceptrons





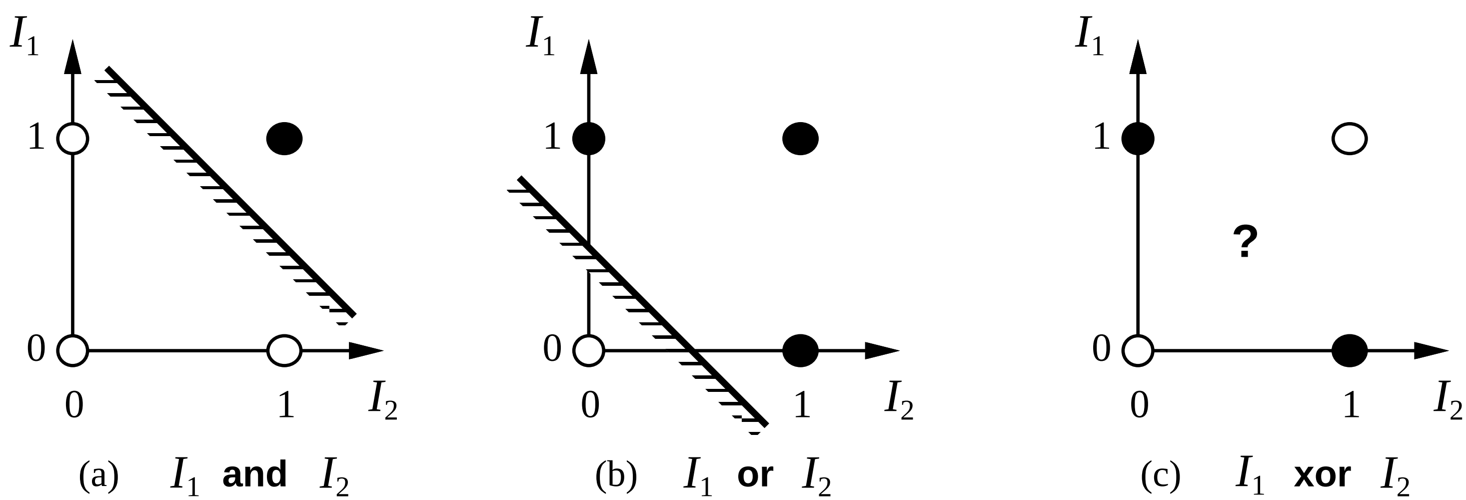
Expressiveness of perceptrons

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

Can represent AND, OR, NOT, majority, etc.

Represents a linear separator in input space:

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



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

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

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

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

Perform optimization search by gradient descent:

$$\frac{\partial E}{\partial W_j} = ?$$

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

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

Perform optimization search by gradient descent:

$$\frac{\partial E}{\partial W_j} = Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j}\left(y - g(\sum_{j=0}^{n} W_j x_j)\right)$$

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

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

Perform optimization search by gradient descent:

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

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

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

Perform optimization search by gradient descent:

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

Simple weight update rule:

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

E.g., +ve error $\Rightarrow$ increase network output
$\Rightarrow$ increase weights on +ve inputs, decrease on -ve inputs

Perceptron learning

```
W = random initial values
for iter = 1 to T
  for i = 1 to N (all examples)
    x⃗ = input for example i
    y = output for example i
    W_old = W
    Err = y − g(W_old · x⃗)
    for j = 1 to M (all weights)
      W_j = W_j + α · Err · g′(W_old · x⃗) · x_j
```

W = random initial values

for iter = 1 to T

  for i = 1 to N (all examples)

    $\vec{x} =$ input for example i

    $y =$ output for example i

    $W_{old} = W$

    $Err = y - g(W_{old} \cdot \vec{x})$

    for j = 1 to M (all weights)

      $W_j = W_j + \alpha \cdot Err \cdot g'(W_{old} \cdot \vec{x}) \cdot x_j$

Perceptron learning contd.

Derivative of sigmoid $g(x)$ can be written in simple form:

$$g(x) = \frac{1}{1+e^{-x}}$$
$$g'(x) = ?$$

Perceptron learning contd.

Derivative of sigmoid $g(x)$ can be written in simple form:

$$\begin{aligned} g(x) &= \frac{1}{1+e^{-x}} \\ g'(x) &= \frac{e^{-x}}{(1+e^{-x})^2} = e^{-x}g(x)^2 \end{aligned}$$

Also,

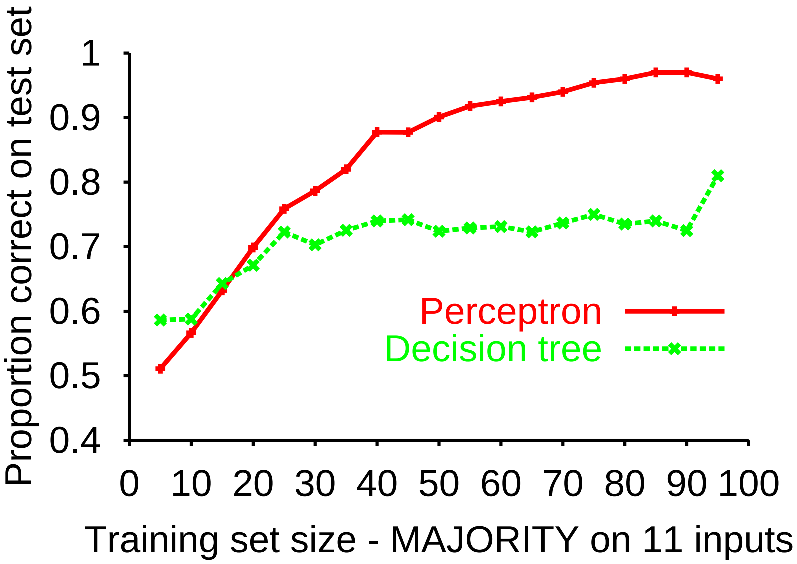
$$g(x) = \frac{1}{1+e^{-x}} \Rightarrow g(x) + e^{-x}g(x) = 1 \Rightarrow e^{-x} = \frac{1-g(x)}{g(x)}$$

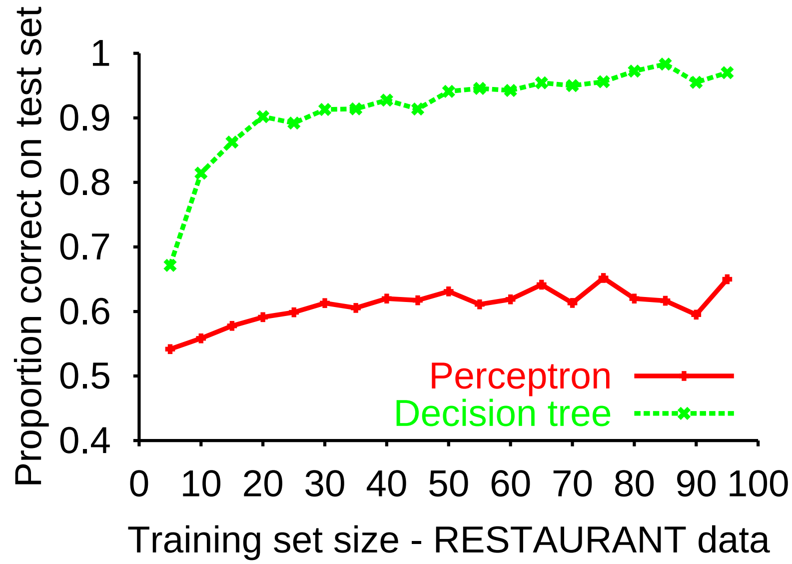
So

$$\begin{aligned} g'(x) &= \frac{1-g(x)}{g(x)}g(x)^2 \\ &= (1-g(x))g(x) \end{aligned}$$

Perceptron learning contd.

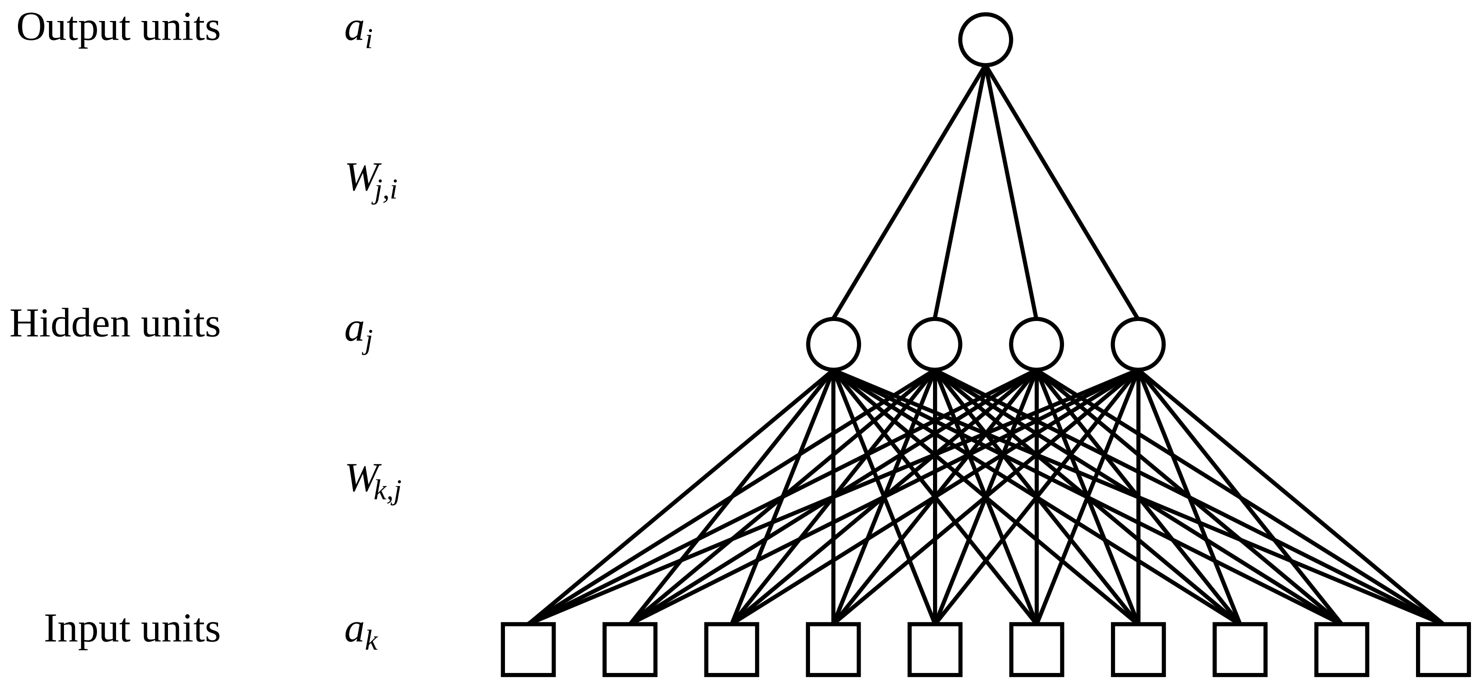
Perceptron learning rule converges to a consistent function
for any linearly separable data set





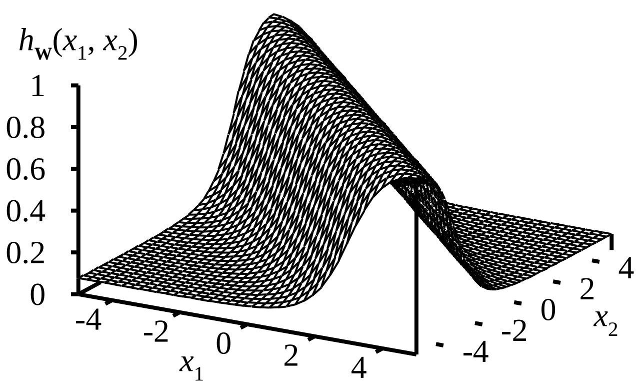
Multilayer networks

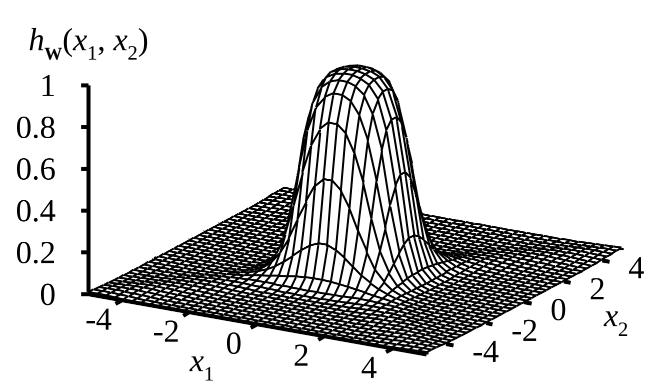
Layers are usually fully connected;
numbers of hidden units typically chosen by hand



Expressiveness of MLPs

All continuous functions w/ 1 hidden layer, all functions w/ 2 hidden layers





Training a MLP

In general have n output nodes,

$$E \equiv \frac{1}{2} \sum_i Err_i^2,$$

where $Err_i = (y_i - a_i)$ and Σ_i runs over all nodes in the output layer.

Need to calculate

$$\frac{\partial E}{\partial W_{ij}}$$

for any W_{ij}.

Training a MLP cont.

Can approximate derivatives by:

$$
\begin{aligned}
f'(x) &\approx \frac{f(x+h) - f(x)}{h} \\
\frac{\partial E}{\partial W_{ij}}(\mathbf{W}) &\approx \frac{E(\mathbf{W} + (0, \ldots, h, \ldots, 0)) - E(\mathbf{W})}{h}
\end{aligned}
$$

What would this entail for a network with n weights?

Training a MLP cont.

Can approximate derivatives by:

$$
\begin{aligned}
f'(x) &\approx \frac{f(x+h)-f(x)}{h} \\
\frac{\partial E}{\partial W_{ij}}(\mathbf{W}) &\approx \frac{E(\mathbf{W}+(0,\ldots,h,\ldots,0))-E(\mathbf{W})}{h}
\end{aligned}
$$

What would this entail for a network with n weights?

- one iteration would take $O(n^2)$ time

Complicated networks have tens of thousands of weights, $O(n^2)$ time is intractable.

Back-propagation is a recursive method of calculating all of these derivatives in $O(n)$ time.

Back-propagation learning

In general have n output nodes,

$$E \equiv \frac{1}{2}\sum_i Err_i^2,$$

where $Err_i = (y_i - a_i)$ and Σ_i runs over all nodes in the output layer.

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 layers: **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 layers:

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

Back-propagation derivation

For a node i in the output layer:

$$\frac{\partial E}{\partial W_{j,i}} = -(y_i - a_i)\frac{\partial a_i}{\partial W_{j,i}}$$

Back-propagation derivation

For a node i in the output layer:

$$\frac{\partial E}{\partial W_{j,i}} = -(y_i - a_i)\frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i)\frac{\partial g(in_i)}{\partial W_{j,i}}$$

Back-propagation derivation

For a node i in the output layer:

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

Back-propagation derivation

For a node i in the output layer:

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

Back-propagation derivation

For a node i in the output layer:

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

where $\Delta_i = (y_i - a_i)g'(in_i)$

Back-propagation derivation: hidden layer

For a node j in a hidden layer:

$$\frac{\partial E}{\partial W_{k,j}} = \text{ ?}$$

“Reminder”: Chain rule for partial derivatives

For $f(x, y)$, with f differentiable wrt x and y, and x and y differentiable wrt u and v:

$$\frac{\partial f}{\partial u} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial u} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial u}$$

and

$$\frac{\partial f}{\partial v} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial v} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial v}$$

Back-propagation derivation: hidden layer

For a node j in a hidden layer:

$$\frac{\partial E}{\partial W_{k,j}} = \frac{\partial}{\partial W_{k,j}} E(a_{j_1}, a_{j_2}, \ldots, a_{j_m})$$

where $\{j_i\}$ are the indices of the nodes in the same layer as node j.

Back-propagation derivation: hidden layer

For a node j in a hidden layer:

$$\frac{\partial E}{\partial W_{k,j}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial W_{k,j}} + \sum_i \frac{\partial E}{\partial a_i}\frac{\partial a_i}{\partial W_{k,j}}$$

where Σ_i runs over all other nodes i in the same layer as node j.

Back-propagation derivation: hidden layer

For a node j in a hidden layer:

$$\begin{aligned}\frac{\partial E}{\partial W_{k,j}} &= \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial W_{k,j}} + \sum_i \frac{\partial E}{\partial a_i}\frac{\partial a_i}{\partial W_{k,j}} \\ &= \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial W_{k,j}} \qquad \text{since } \frac{\partial a_i}{\partial W_{k,j}} = 0 \text{ for } i \neq j\end{aligned}$$

Back-propagation derivation: hidden layer

For a node j in a hidden layer:

$$\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial W_{k,j}} + \sum_i \frac{\partial E}{\partial a_i}\frac{\partial a_i}{\partial W_{k,j}} \\
&= \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial W_{k,j}} \qquad \text{since } \frac{\partial a_i}{\partial W_{k,j}} = 0 \text{ for } i \neq j \\
&= \frac{\partial E}{\partial a_j} \cdot g'(in_j) a_k
\end{aligned}$$

Back-propagation derivation: hidden layer

For a node j in a hidden layer:

$$\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial W_{k,j}} + \sum_i \frac{\partial E}{\partial a_i}\frac{\partial a_i}{\partial W_{k,j}} \\
&= \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial W_{k,j}} \qquad \text{since } \frac{\partial a_i}{\partial W_{k,j}} = 0 \text{ for } i \neq j \\
&= \frac{\partial E}{\partial a_j} \cdot g'(in_j) a_k
\end{aligned}$$

$$\frac{\partial E}{\partial a_j} = \text{?}$$

Back-propagation derivation: hidden layer

For a node j in a hidden layer:

$$\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial W_{k,j}} + \sum_i \frac{\partial E}{\partial a_i}\frac{\partial a_i}{\partial W_{k,j}} \\
&= \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial W_{k,j}} \qquad \text{since } \frac{\partial a_i}{\partial W_{k,j}} = 0 \text{ for } i \neq j \\
&= \frac{\partial E}{\partial a_j} \cdot g'(in_j)a_k
\end{aligned}$$

$$\frac{\partial E}{\partial a_j} = \frac{\partial}{\partial a_j}E(a_{k_1}, a_{k_2}, \ldots, a_{k_m})$$

where $\{k_i\}$ are the indices of the nodes in the layer after node j.

Back-propagation derivation: hidden layer

For a node j in a hidden layer:

$$\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial W_{k,j}} + \sum_i \frac{\partial E}{\partial a_i}\frac{\partial a_i}{\partial W_{k,j}} \\
&= \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial W_{k,j}} \qquad \text{since } \frac{\partial a_i}{\partial W_{k,j}} = 0 \text{ for } i \neq j \\
&= \frac{\partial E}{\partial a_j} \cdot g'(in_j)a_k
\end{aligned}$$

$$\frac{\partial E}{\partial a_j} = \sum_k \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial a_j}$$

where Σ_k runs over all nodes k that node j connects to.

Back-propagation derivation: hidden layer

For a node j in a hidden layer:

$$
\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial W_{k,j}} + \sum_i \frac{\partial E}{\partial a_i}\frac{\partial a_i}{\partial W_{k,j}} \\
&= \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial W_{k,j}} \qquad \text{since } \frac{\partial a_i}{\partial W_{k,j}} = 0 \text{ for } i \neq j \\
&= \frac{\partial E}{\partial a_j} \cdot g'(in_j) a_k
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial E}{\partial a_j} &= \sum_k \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial a_j} \\
&= \sum_k \frac{\partial E}{\partial a_k} g'(in_k) W_{j,k}
\end{aligned}
$$

Back-propagation derivation: hidden layer

If we define

$$\Delta_j \equiv g'(in_j) \sum_k W_{j,k} \Delta_k$$

then

$$\frac{\partial E}{\partial W_{k,j}} = -\Delta_j a_k$$

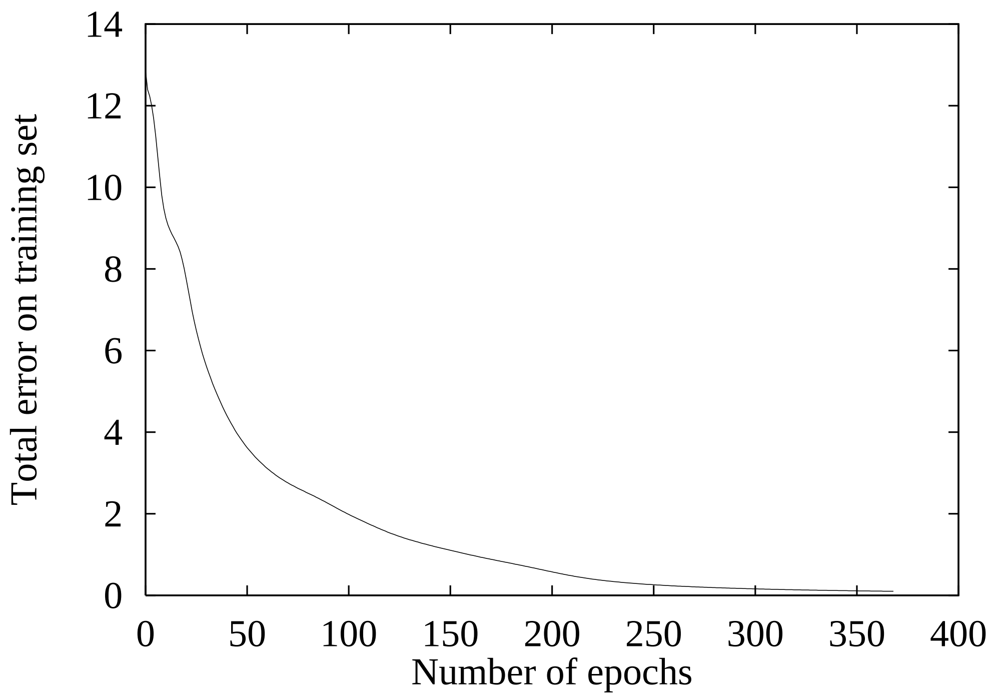
Back-propagation pseudocode

for iter = 1 to T
  for e = 1 to N (all examples)
    $\vec{x} =$ input for example e
    $\vec{y} =$ output for example e
    run $\vec{x}$ forward through network, computing all $\{a_i\}, \{in_i\}$
    for all nodes i (in reverse order)
      compute $\Delta_i = \begin{cases} (y_i - a_i) \times g'(in_i) & \texttt{if i is output node} \\ g'(in_i) \, \Sigma_k \, W_{i,k} \Delta_k & \texttt{o.w.} \end{cases}$
    for all weights $W_{j,i}$
      $W_{j,i} = W_{j,i} + \alpha \times a_j \times \Delta_i$

Back-propagation learning contd.

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

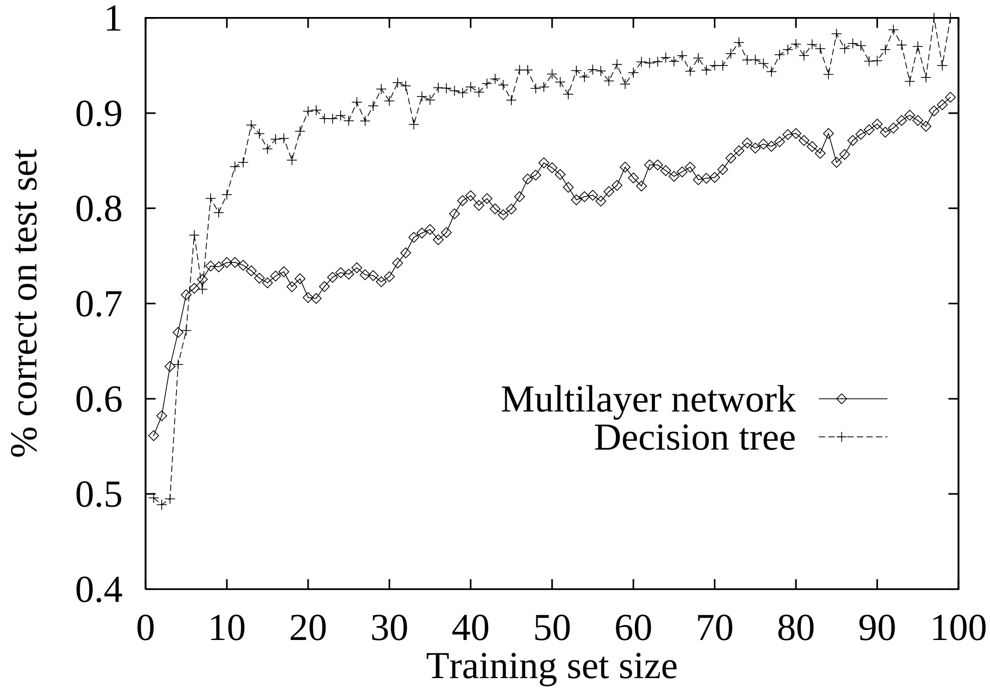
Restaurant data:



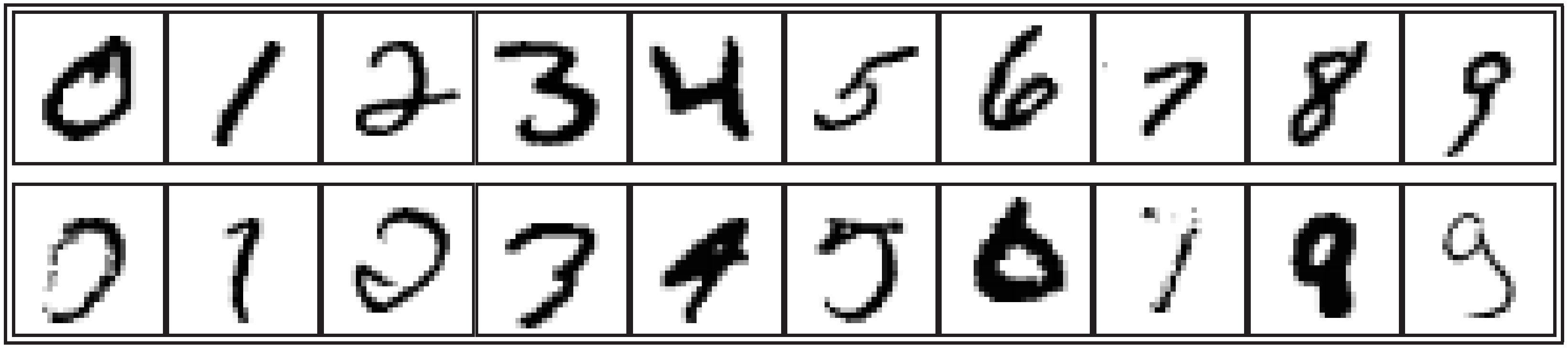
Usual problems with slow convergence, local minima

Back-propagation learning contd.

Restaurant data:



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

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, credit cards, etc.