CSE 537 Fall 2015

LEARNING FROM EXAMPLES
AIMA CHAPTER 18.7

- Instructor: Sael Lee

Slides are mostly made from AIMA resources and Andrew W. Moore's tutorials: http://www.cs.cmu.edu/~awm/tutorials

ARTIFICIAL NEURAL NETWORKS

- Brains
- Neural networks
- Perceptrons
- Multilayer perceptrons
- Applications of neural networks

WHAT IS AN ARTIFICIAL NEURAL NETWORK?

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

It is a formalism for representing functions inspired from biological systems and composed of parallel computing units which each compute a simple function.

ANNs provide a general, practical method for learning real-valued, discrete-values, and vector-valued function from examples.

Algorithms such a Backpropagation use gradient descent to tune network parameters to best fit a training set of input-output pairs.

EXAMPLE APPLICATIONS & CHARACTERISTICS

ANN is robust to error in the training data and has been successfully applied to various real problems

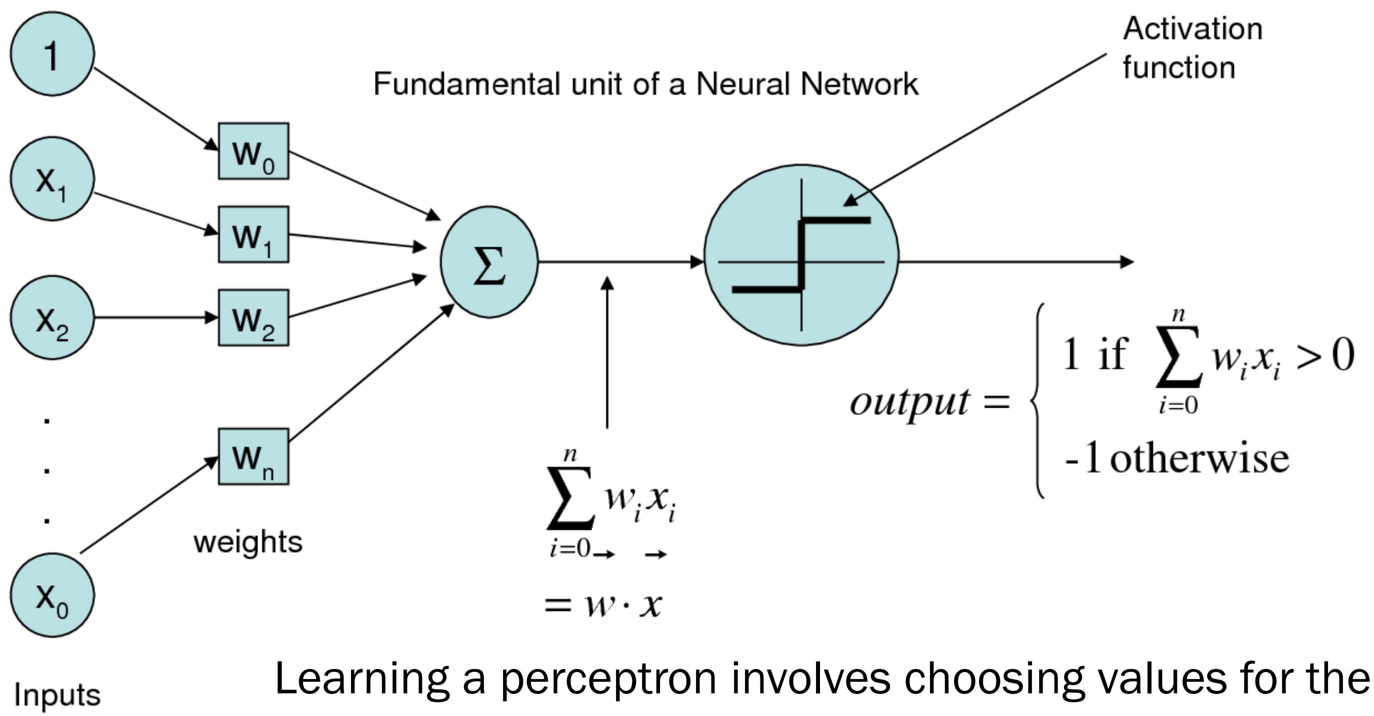
- Speech/voice recognition
- Face recognition
- Handwriting recognitions
- I can also be used where symbolic representations are used as cases for Decision tree learning

- Characteristics of ANN problems
  - Instances are represented by many attribute-value pair (supervised)
  - The target function output may be discrete, real, or vector.
  - Training data may contain error
  - Long training times are acceptable
  - Fast evaluation of the learning target function may be required
  - The ability of humans to understand the learned target function is not important.

PRIMITIVE UNITS THAT MAKE UP ANN

Artificial Neural Networks

The Perceptron



Learning a perceptron involves choosing values for the weights wi

- Perceptrons
- Linear units
- Sigmoid units

ANN STRUCTURE: DECIDING THE MATHEMATICAL MODEL

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

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

Each unit is a linear regression

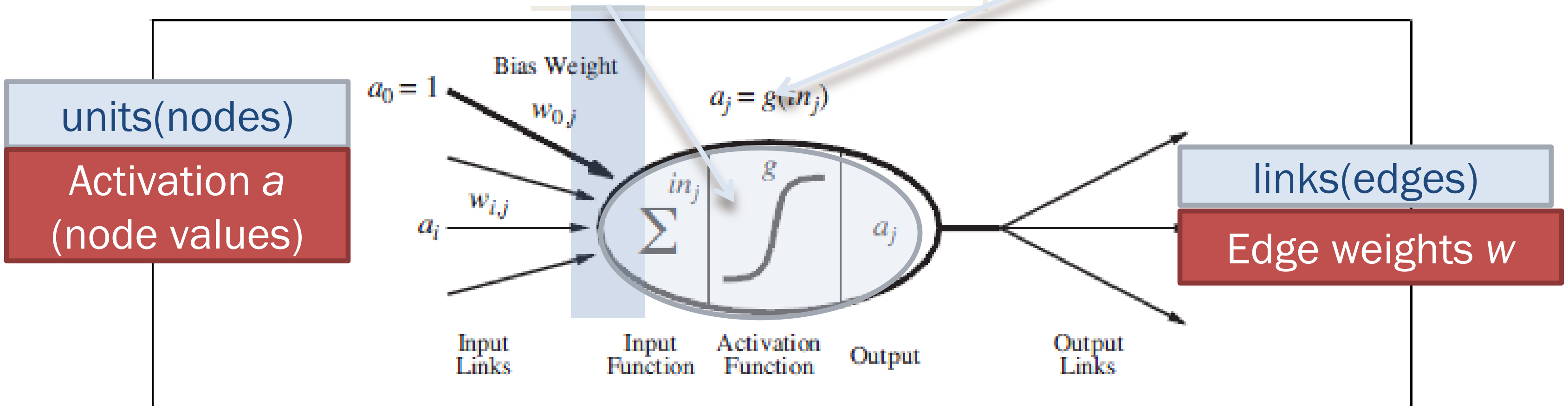


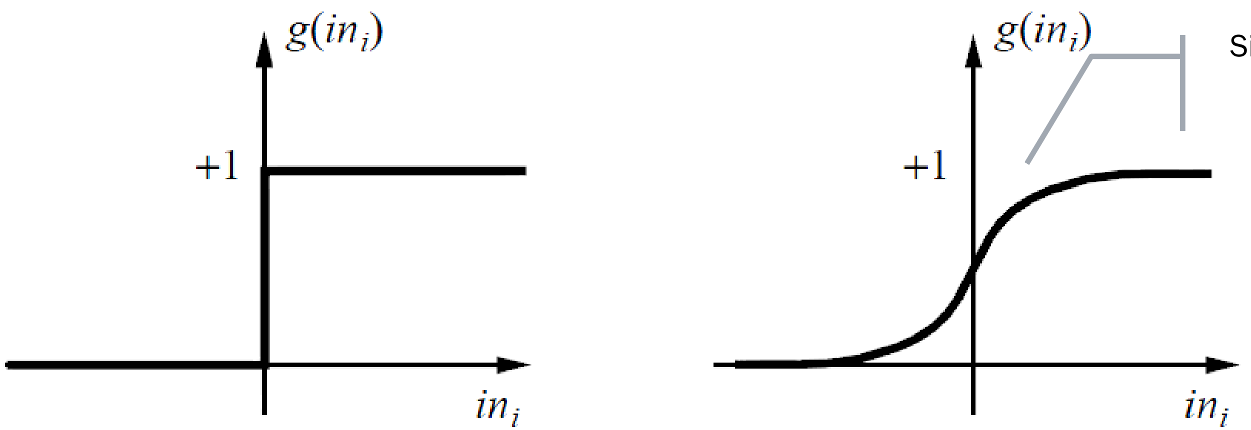
Figure 18.19 **FILES: figures/neuron-unit.eps (Wed Nov 4 11:23:13 2009).** A simple mathematical model for a neuron. The unit's output activation is $a_j = g(\sum_{i=0}^{n} w_{i,j} a_i)$, where a_i is the output activation of unit i and $w_{i,j}$ is the weight on the link from unit i to this unit.

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

ACTIVATION FUNCTIONS *G*

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

Activation function enables the model to be nonlinear



(a) (b)

Sigmoid function allows the model to be **differentiable**

Hard threshold: perceptron

Logistic function: Sigmoid perceptron

(a) is a step function or threshold function

(b) is a sigmoid function $1/(1 + \exp(-W^T A))$

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

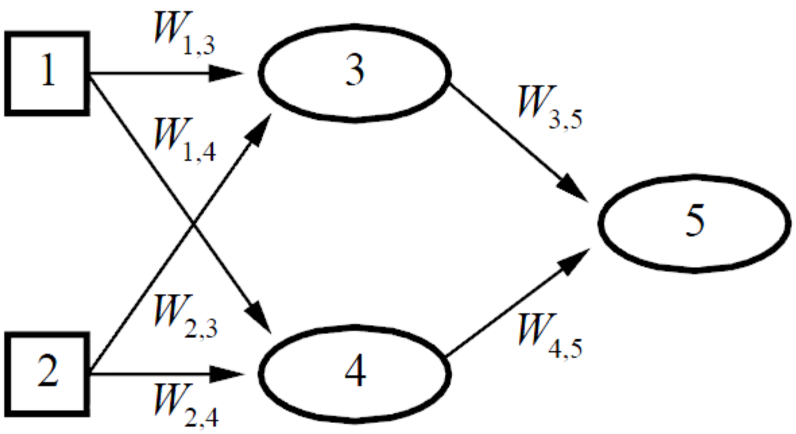
Two types of ANN structure:

- **Feed-forward networks:** connections only in one direction (directed acyclic graph)
  - Feed-forward network implement functions, have no internal state
  - Examples:
    - single-layer perceptrons (output is 0 or 1)
    - multi-layer perceptrons
- Recurrent networks:
  - Interesting models of the brain but more difficult to understand.
  - Have directed cycles with delays ⇒ have internal state (like flip-flops), can oscillate etc.

TASKS TO BE SOLVED BY ANN

- Controlling the movements of a robot based on self-perception and other information (e.g., visual information);
- Deciding the category of potential food items (e.g., edible or non-edible) in an artificial world;
- Recognizing a visual object (e.g., a familiar face);
- Predicting where a moving object goes, when a robot wants to catch it.

FEED-FORWARD EXAMPLE



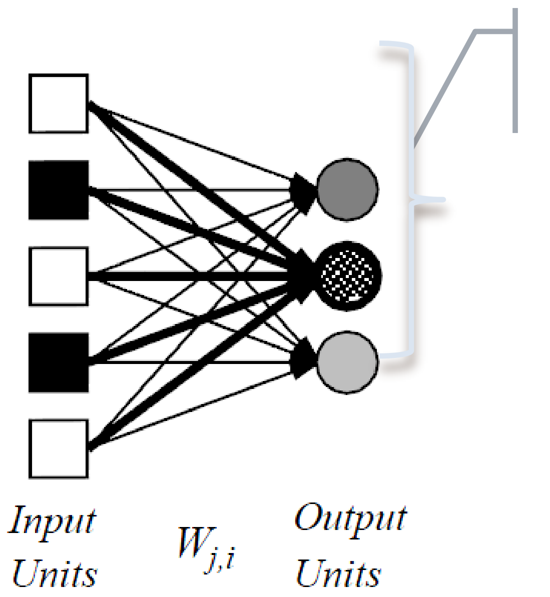
Feed-forward network = a parameterized family of nonlinear functions:

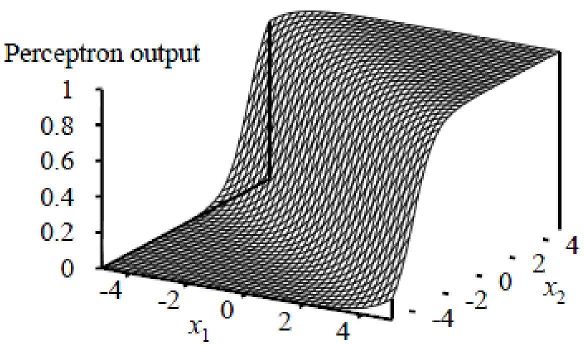
$$\begin{aligned} a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\ &= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2)) \end{aligned}$$

Adjusting weights changes the function: do learning this way!

SINGLE LAYER FEED-FORWARD NEURAL NETWORKS: PERCEPTRON NETWORK

Every unit connects directly form the network's inputs to it's output





Output units all operate separately — no shared weights

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

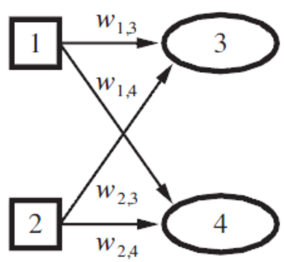
EXPRESSIVENESS OF SINGLE LAYER PERCEPTRONS

- 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:

EX> Two bit adder

Two separate component

1. Carry 2. sum

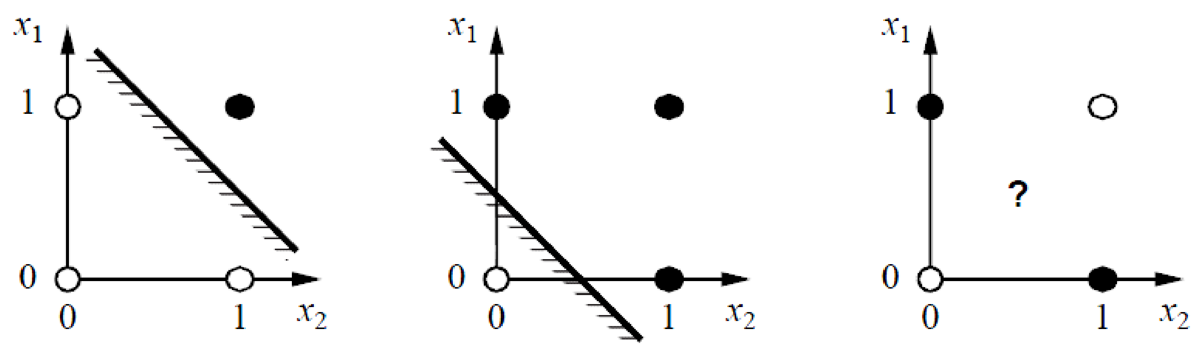


Carry: AND

Sum: XOR

X1	X2	Y3 (carry)	Y4 (sum)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

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



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

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

PERCEPTRON LEARNING

Learn by adjusting weights to reduce error on training set

The squared error for an example with input 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**
(just like logistic regression)

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

* Chain rule:

$$\frac{\partial g(f(x))}{\partial x} = \frac{g'(f(x))\partial f(x)}{\partial x}$$

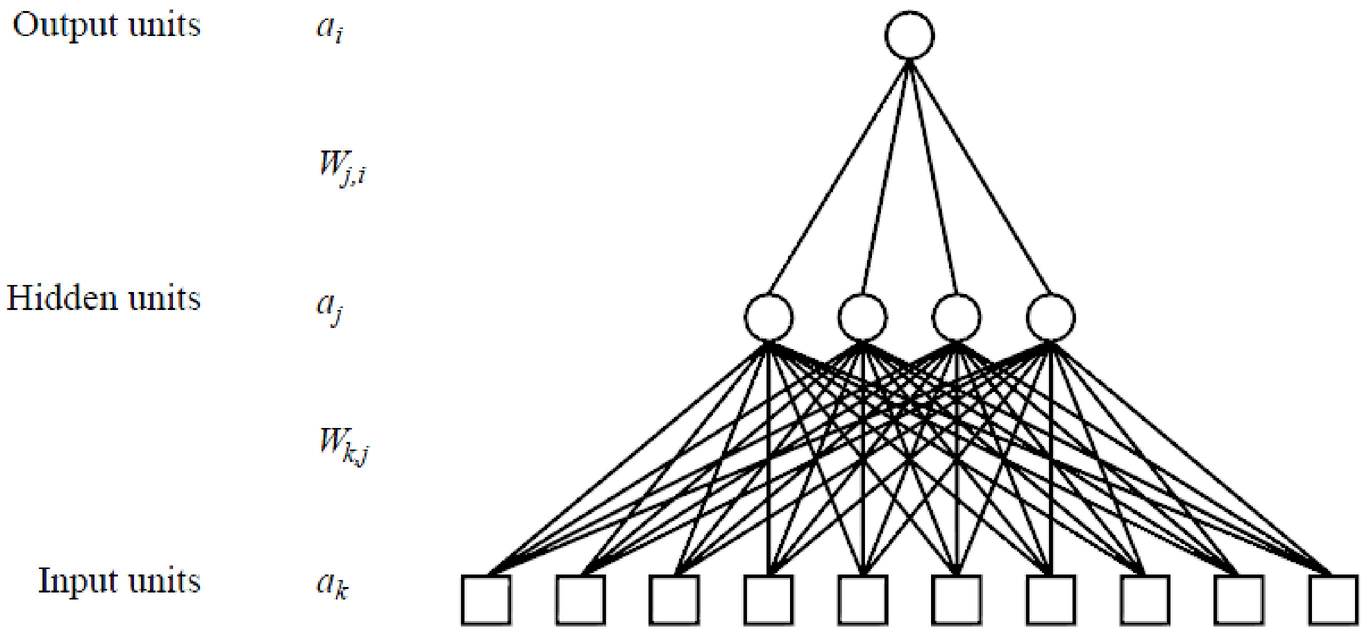
Simple weight update rule:

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

E.g., +ve error ⇒ increase network output
⇒ increase weights on +ve inputs, decrease on -ve inputs

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

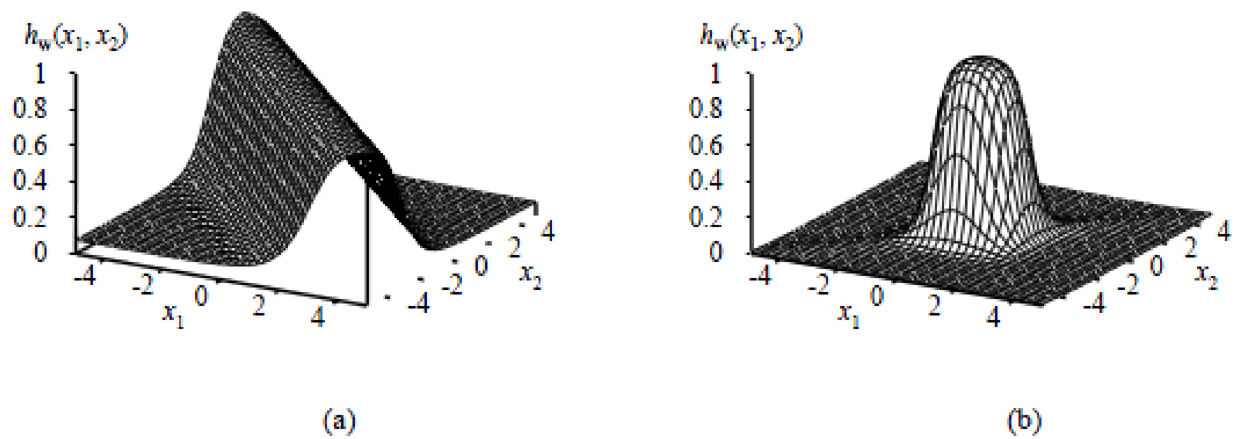


Figure 18.23 **FILES: .** (a) The result of combining two opposite-facing soft threshold functions to produce a ridge. (b) The result of combining two ridges to produce a bump.

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

LEARNING IN MULTILAYER NETWORKS

- Complications in the error estimation:
  - Interactions among the learning problems when the network has multiple output!
  - Need to think of network as implementing a vector hypothesis $\mathbf{h}_w$ rater than scalar function h_w.
  - In terms of loss function dependency is additive across the components of the error vector

    $\mathbf{y} - \mathbf{h}_w(\mathbf{x})$

$$\frac{\partial Loss(\boldsymbol{w})}{\partial \mathrm{x}} = \frac{\partial}{\partial w}|\mathbf{y} - \mathbf{h}_{\mathbf{w}}(\mathbf{x})|^{2} = \frac{\partial}{\partial w}\sum_{k}(y_k - a_k)^2 = \sum_{k}\frac{\partial}{\partial w}(y_k - a_k)^2$$

However, if the there are multi-layers, the intermediate error are not trivial.

BACK-PROPAGATION LEARNING

1. **Output layer: weight update rules** are same as for single-layer perceptron,

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

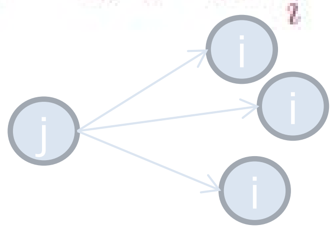
where $\Delta_i = Err_i \times g'(in_i)$

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

2. **Hidden layer: Error back-propagation rule**

back-propagate the error from the output layer:

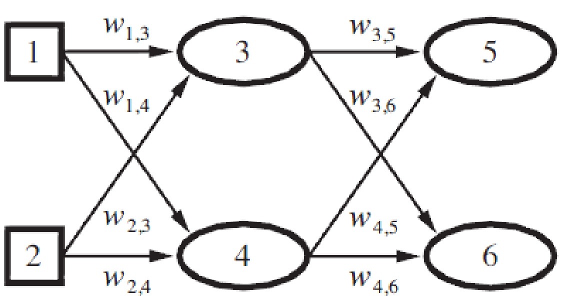
$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i$$



Hidden layer is responsible for Δ_i portion of error according to strength of the connection.

3. **Update rule** for weights in hidden layer:

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



THE BACK-PROPAGATION ALGORITHM FOR LEARNING IN MULTILAYER NETWORKS

```
function BACK-PROP-LEARNING(examples, network) returns a neural network
  inputs: examples, a set of examples, each with input vector x and output vector y
          network, a multilayer network with L layers, weights w_{i,j}, activation function g
  local variables: Δ, a vector of errors, indexed by network node

  repeat
     for each weight w_{i,j} in network do
        w_{i,j} ← a small random number
     for each example (x, y) in examples do
        /* Propagate the inputs forward to compute the outputs */
        for each node i in the input layer do
           a_i ← x_i
        for ℓ = 2 to L do
           for each node j in layer ℓ do
              in_j ← Σ_i w_{i,j} a_i
              a_j ← g(in_j)
        /* Propagate deltas backward from output layer to input layer */
        for each node j in the output layer do
           Δ[j] ← g'(in_j) × (y_j − a_j)
        for ℓ = L − 1 to 1 do
           for each node i in layer ℓ do
              Δ[i] ← g'(in_i) Σ_j w_{i,j} Δ[j]
        /* Update every weight in network using deltas */
        for each weight w_{i,j} in network do
           w_{i,j} ← w_{i,j} + α × a_i × Δ[j]
  until some stopping criterion is satisfied
  return network
```

Compute the Δ values for the output units using the observed error

Propagate the Δ values back to the previous layer.

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

Update the weights between the two layers.

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

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.

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

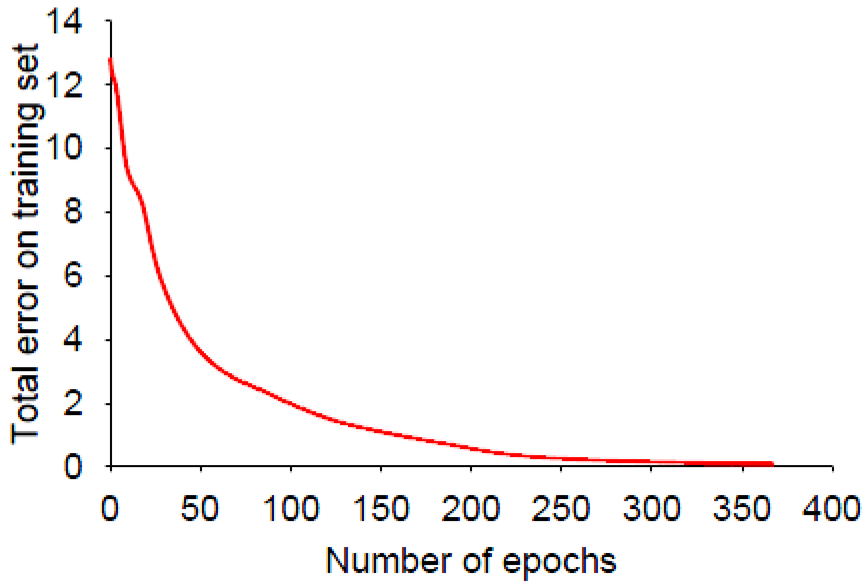
BACK-PROPAGATION DERIVATION CONT.

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

BACK-PROPAGATION LEARNING CONT.

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

LEARNING NEURAL NETWORK STRUCTURE

- Cross-validation
  - If we stay with fully connected networks, structural parameters to choose from are:
    - Number of hidden layers and their sizes.
- Optimal brain damage
  - Start with fully connected network and start removing links and units iteratively.
- Tiling
  - Starting from single unit and start adding units to take care of the examples that current units got wrong.

SUMMARY

- Most brains have lots of neurons; each neuron ≈ 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 modeling, and neural system modeling subfields have largely diverged