


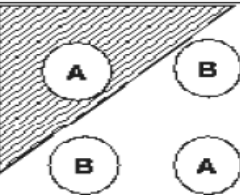
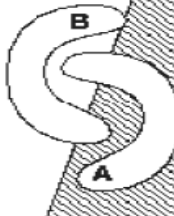
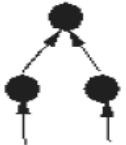
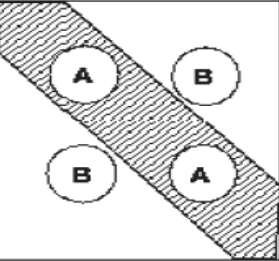

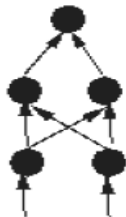
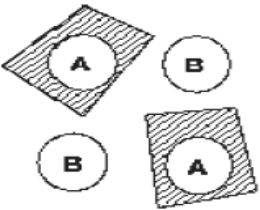
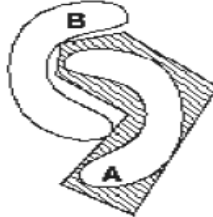
# **Lecture 9: Neural Networks**

**Machine Learning**

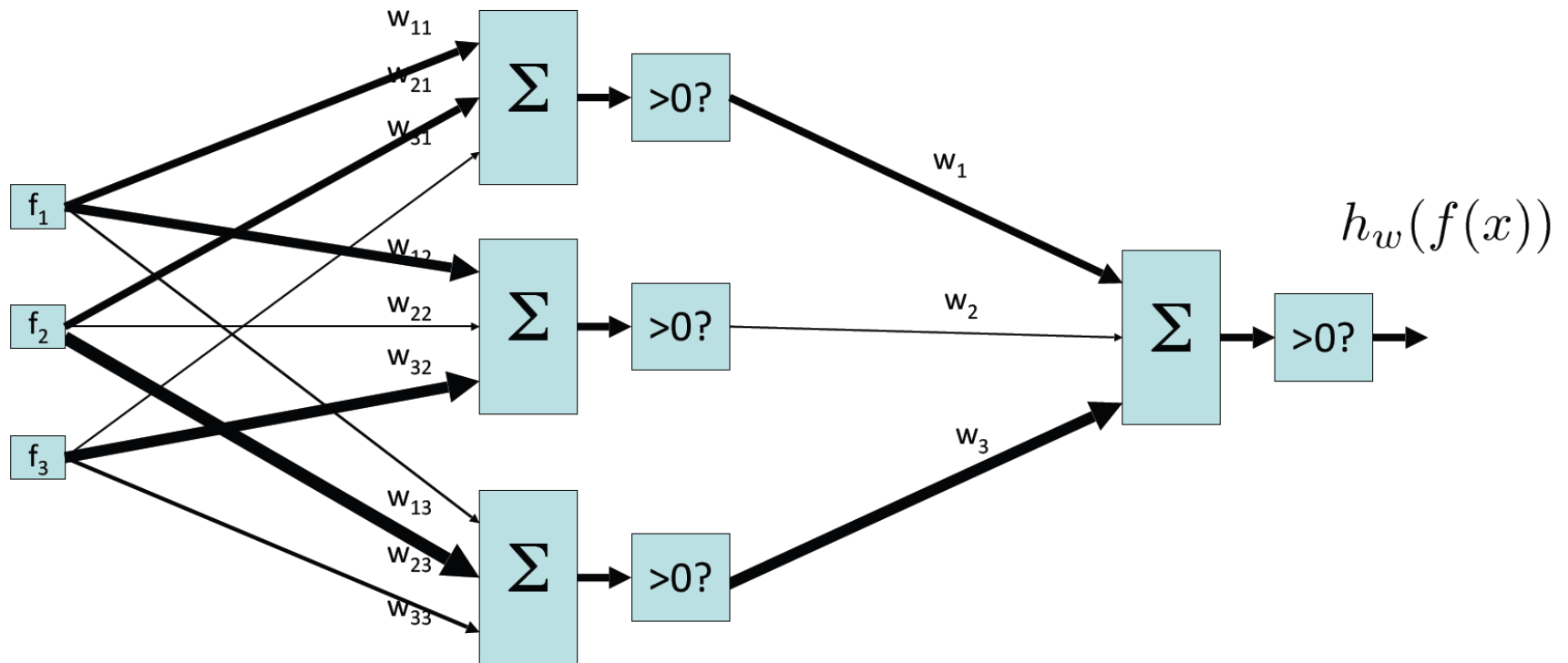
Spring Semester '2022

# Multi-Layer Perceptron

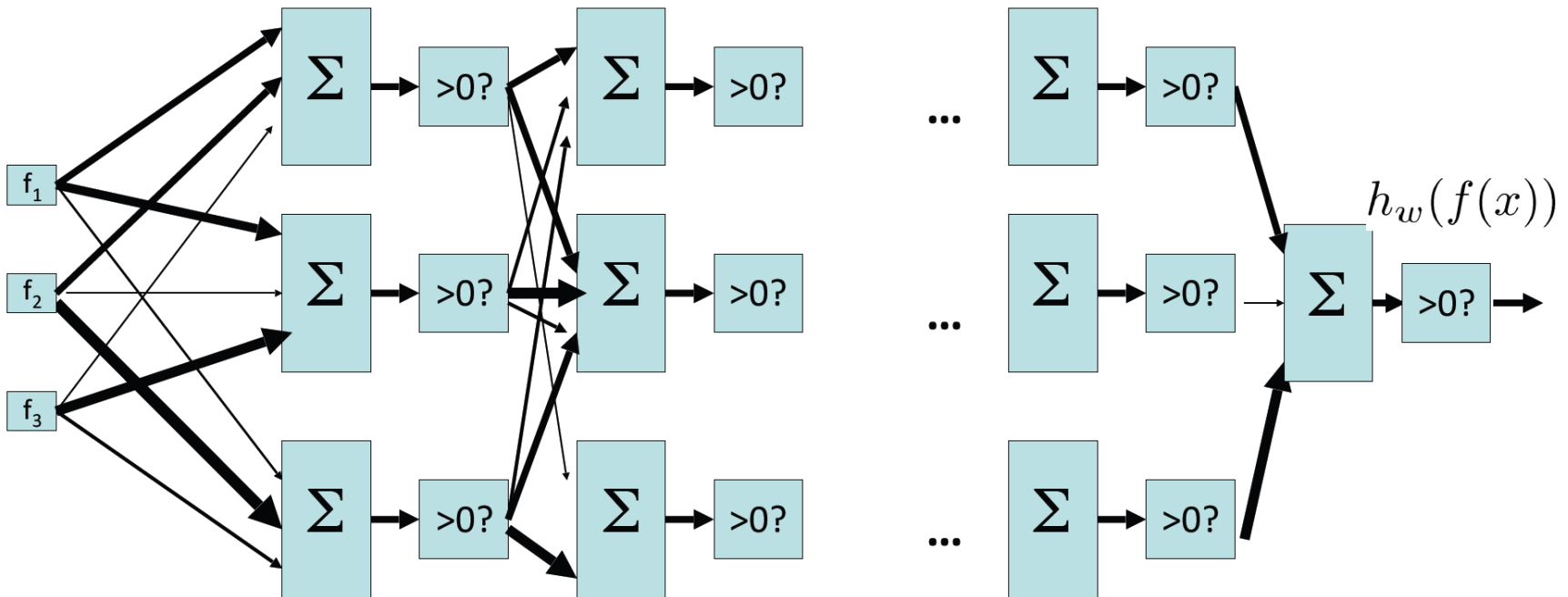
- Higher representational power

Structure	Regions	XOR	Meshed regions
single layer 	Half plane bounded by hyper-plane		
two layer 	Convex open or closed regions		
three layer 	Arbitrary (limited by # of nodes)		

# Two-layer perceptron network

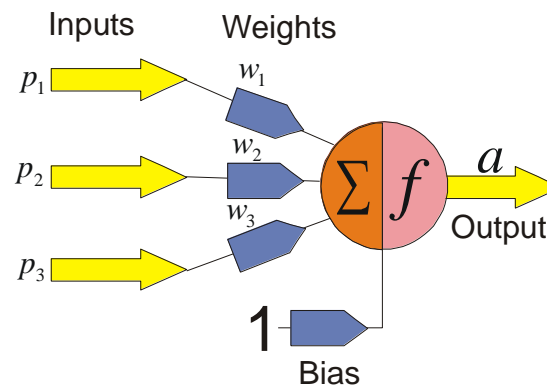


# N-layer perceptron network



# The Key Elements of Neural Networks

- Neural computing requires a number of **neurons**, to be connected together into a **neural network**. Neurons are arranged in layers.

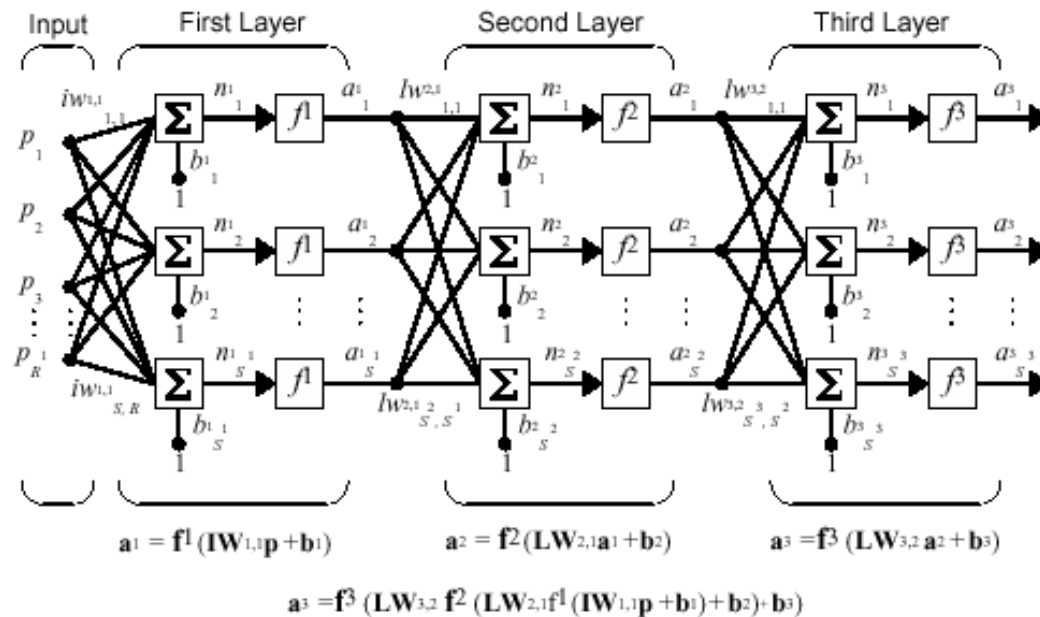
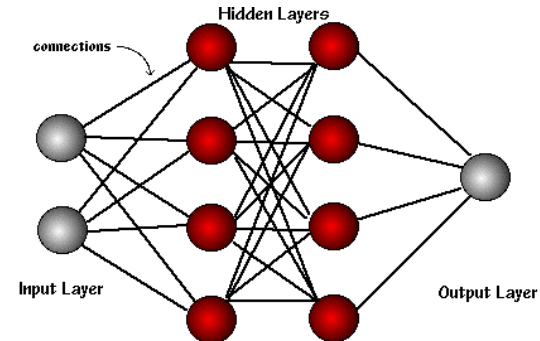


$$a = f(p_1w_1 + p_2w_2 + p_3w_3 + b) = f\left(\sum p_iw_i + b\right)$$

- Each neuron within the network is usually a simple processing unit which takes one or more inputs and produces an output. At each neuron, every input has an associated **weight** which modifies the strength of each input. The neuron simply adds together all the inputs and calculates an output to be passed on.

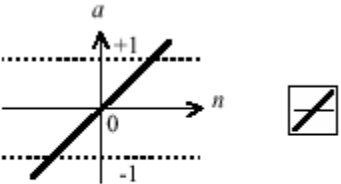
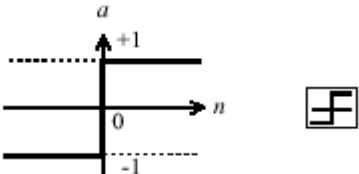
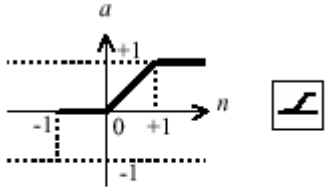
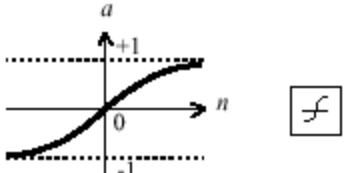
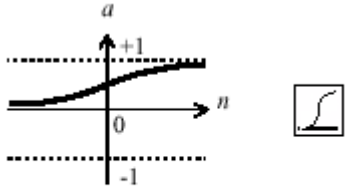
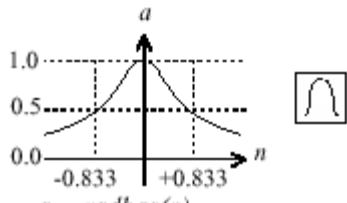
# Feedforward NNs

- The basic structure of a feedforward Neural Network
- The **learning rule** modifies the weights according to the input patterns that it is presented with. In a sense, NNs **learn by example** as do their biological counterparts.
- When the desired output are known we have **supervised learning** or learning with a teacher.



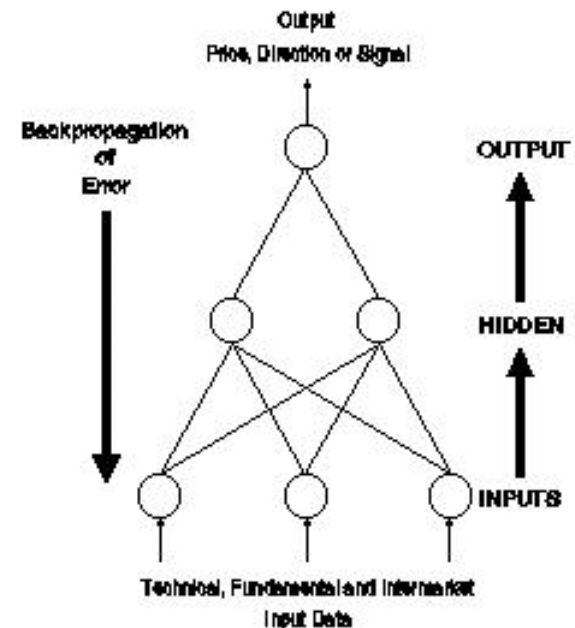
# Activation functions

- The activation function is generally non-linear.

 <p><math>a = \text{purelin}(n)</math></p> <p>Linear Transfer Function</p>	 <p><math>a = \text{hardlims}(n)</math></p> <p>Symmetric Hard Limit Trans. Funct.</p>
 <p><math>a = \text{satlin}(n)</math></p> <p>Satlin Transfer Function</p>	 <p><math>a = \text{tansig}(n)</math></p> <p>Tan-Sigmoid Transfer Function</p>
 <p><math>a = \text{logsig}(n)</math></p> <p>Log-Sigmoid Transfer Function</p>	 <p><math>a = \text{radbas}(n)</math></p> <p>Radial Basis Function</p>

# An overview of the backpropagation

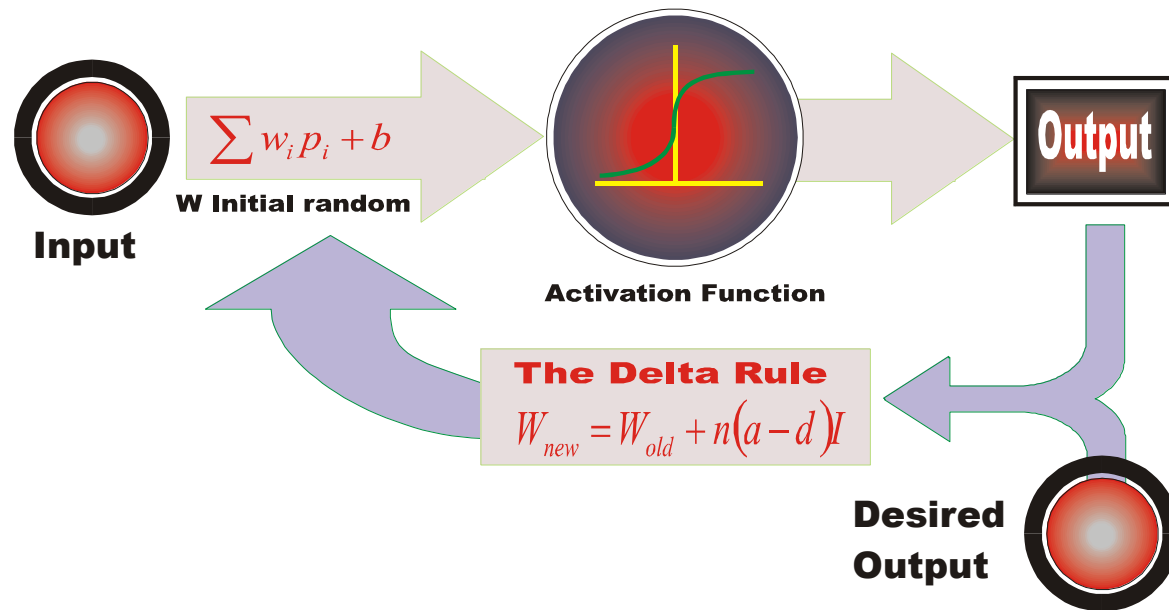
1. Problem statement defined
  - input into the network and desired output from the network
2. Set of training samples assembled
3. Input data entered into network via the input layer.
4. Each neuron processes input data with resultant values steadily "percolating" through the network, layer by layer, until result is generated by the output layer
5. Actual output of the network compared to expected output for that particular input
  - Results in *error value*
6. Connection weights gradually adjusted working backwards from output layer, through hidden layer, and to input layer, until correct output is produced, i.e. the network *learns*.





# The Learning Rule

- The **delta rule** is often utilized by the most common class of NNs.



# Learning to approximate

Error measure:

$$E = \frac{1}{N} \sum_{t=1}^N (F(x_t; W) - y_t)^2$$

Rule for changing the synaptic weights:

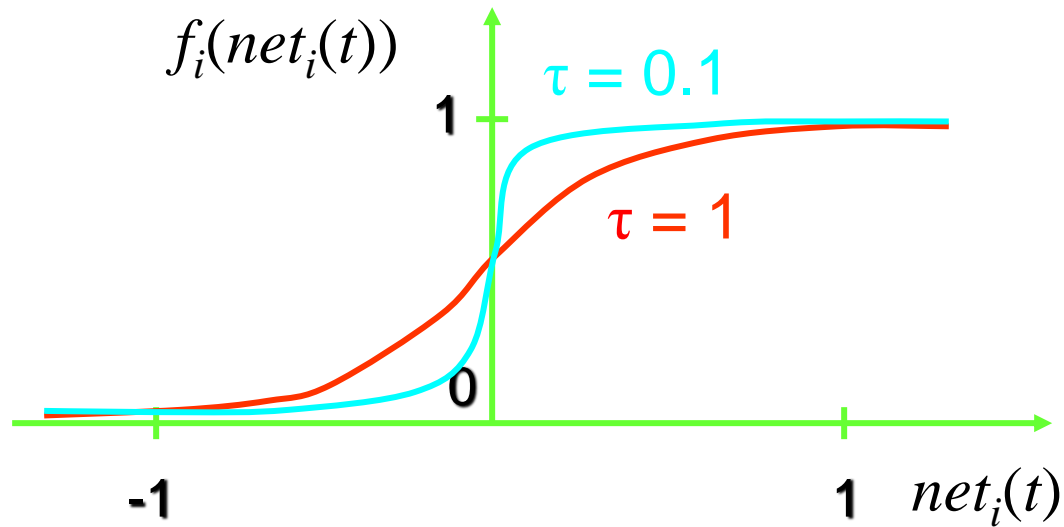
$$\Delta w_i^j = -c \cdot \frac{\partial E}{\partial w_i^j} (W)$$

$$w_i^{j, new} = w_i^j + \Delta w_i^j$$

$c$  is the learning parameter (usually a constant)

# Sigmoidal Neurons

$$f_i(net_i(t)) = \frac{1}{1 + e^{-(net_i(t) - \theta)/\tau}}$$



- In backpropagation networks, we typically choose  $\tau = 1$  and  $\theta = 0$ .

# Sigmoidal Neurons

- This leads to a simplified form of the sigmoid function:

$$S(net) = \frac{1}{1 + e^{(-net)}}$$

We do not need a modifiable threshold  $\Theta$ , because we will use “dummy” inputs as we did for perceptrons.

The choice  $\tau = 1$  works well in most situations and results in a very simple derivative of  $S(net)$ .

# Sigmoidal Neurons

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

$$S'(x) = \frac{dS(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2}$$

$$= \frac{1 + e^{-x} - 1}{(1 + e^{-x})^2} = \frac{1}{1 + e^{-x}} - \frac{1}{(1 + e^{-x})^2}$$

$$= S(x)(1 - S(x))$$

This result will be very useful when we develop the backpropagation algorithm.

# Backpropagation Learning

The goal of the Backpropagation learning algorithm is to modify the network's weights so that its output vector

$$\mathbf{o}_p = (o_{p,1}, o_{p,2}, \dots, o_{p,K})$$

is as close as possible to the desired output vector

$$\mathbf{d}_p = (d_{p,1}, d_{p,2}, \dots, d_{p,K})$$

for  $K$  output neurons and input patterns  $p = 1, \dots, P$ .

The set of input-output pairs (exemplars)

$\{(\mathbf{x}_p, \mathbf{d}_p) \mid p = 1, \dots, P\}$  constitutes the training set.

# Backpropagation Learning

Also, we need a cumulative error function that is to be minimized:

$$Error = \sum_{p=1}^P Err(\mathbf{o}_p, \mathbf{d}_p)$$

We can choose the mean square error (MSE) once again (but the  $1/P$  factor does not matter):

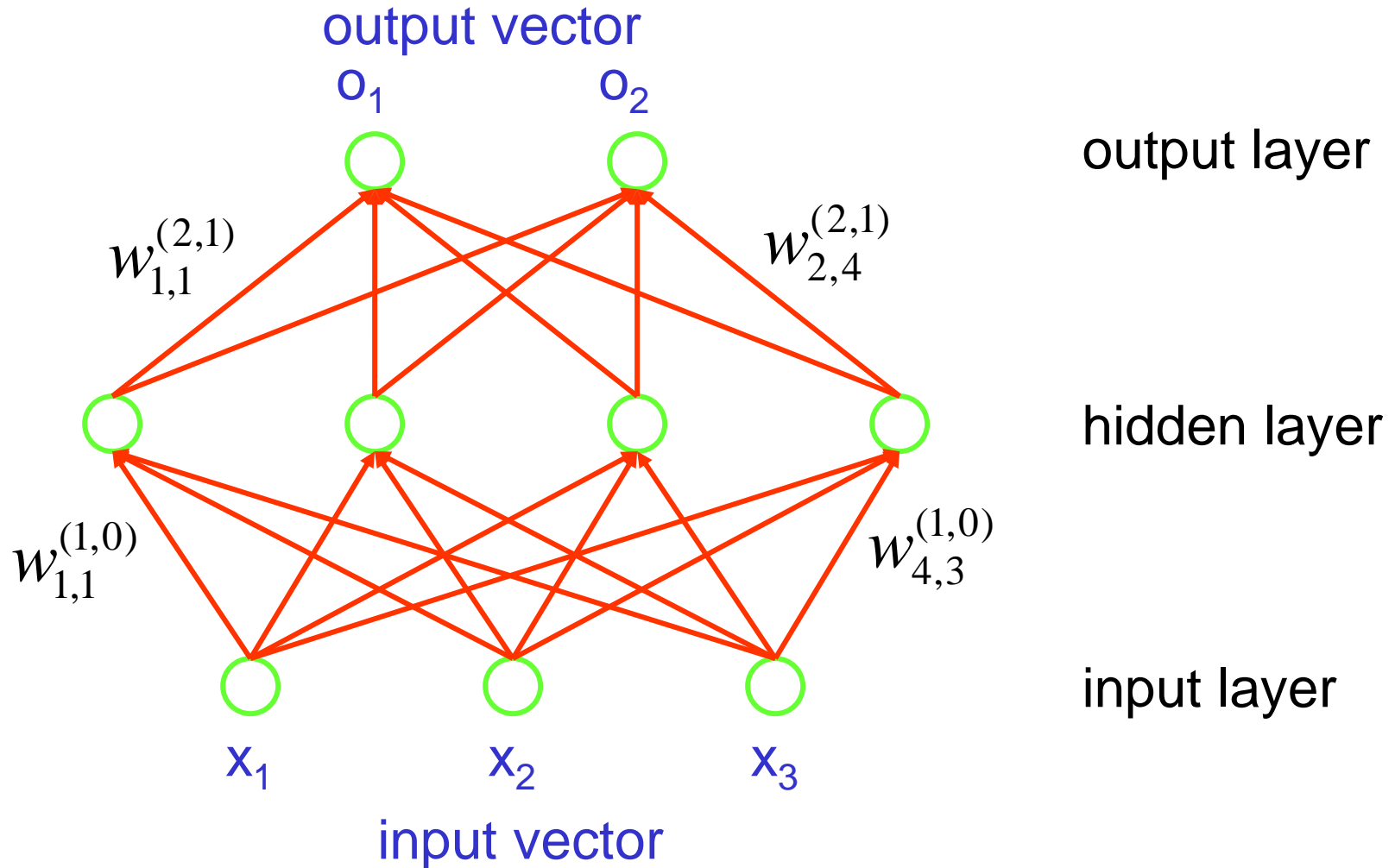
$$MSE = \frac{1}{P} \sum_{p=1}^P \sum_{j=1}^K (l_{p,j})^2$$

where

$$l_{p,j} = o_{p,j} - d_{p,j}$$

# Terminology

- **Example:** Network function  $f: \mathbf{R}^3 \rightarrow \mathbf{R}^2$





# Backpropagation Learning

For input pattern  $p$ , the  $i$ -th input layer node holds  $x_{p,i}$ .

Net input to  $j$ -th node in hidden layer:  $net_j^{(1)} = \sum_{i=0}^n w_{j,i}^{(1,0)} x_{p,i}$

Output of  $j$ -th node in hidden layer:  $x_{p,j}^{(1)} = S\left(\sum_{i=0}^n w_{j,i}^{(1,0)} x_{p,i}\right)$

Net input to  $k$ -th node in output layer:  $net_k^{(2)} = \sum_j w_{k,j}^{(2,1)} x_{p,j}^{(1)}$

Output of  $k$ -th node in output layer:  $o_{p,k} = S\left(\sum_j w_{k,j}^{(2,1)} x_{p,j}^{(1)}\right)$

Network error for  $p$ :  $E_p = \sum_{k=1}^K (l_{p,k})^2 = \sum_{k=1}^K (d_{p,k} - o_{p,k})^2$

# Backpropagation Learning

- As  $E$  is a function of the network weights, we can use gradient descent to find those weights that result in minimal error.
- For individual weights in the hidden and output layers, we should move against the error gradient (omitting index  $p$ ):

$$\Delta w_{k,j}^{(2,1)} \propto \frac{-\partial E}{\partial w_{k,j}^{(2,1)}}$$

Output layer:  
Derivative easy to calculate

$$\Delta w_{j,i}^{(1,0)} \propto \frac{-\partial E}{\partial w_{j,i}^{(1,0)}}$$

Hidden layer:  
Derivative difficult to calculate

# Backpropagation Learning

When computing the derivative with regard to  $w_{k,j}^{(2,1)}$ , we can disregard any output units except  $o_k$ :

$$E = \sum_{i \neq k} l_i^2 + (d_k - o_k)^2$$
$$\frac{\partial E}{\partial o_k} = -2(d_k - o_k)$$

Remember that  $o_k$  is obtained by applying the sigmoid function  $S$  to  $net_k^{(2)}$ , which is computed by:

$$net_k^{(2)} = \sum_j w_{k,j}^{(2,1)} x_j^{(1)}$$

Therefore, we need to apply the chain rule twice.

# Backpropagation Learning

$$\frac{\partial E}{\partial w_{k,j}^{(2.1)}} = \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial net_k^{(2)}} \frac{\partial net_k^{(2)}}{\partial w_{k,j}^{(2.1)}}$$

We know that:  $\frac{\partial o_k}{\partial net_k^{(2)}} = S'(net_k^{(2)})$

Since  $net_k^{(2)} = \sum_j w_{k,j}^{(2.1)} x_j^{(1)}$

We have:  $\frac{\partial net_k^{(2)}}{\partial w_{k,j}^{(2.1)}} = x_j^{(1)}$

Which gives us:  $\frac{\partial E}{\partial w_{k,j}^{(2.1)}} = -2(d_k - o_k) S'(net_k^{(2)}) x_j^{(1)}$

# Backpropagation Learning

For the derivative with regard to  $w_{j,i}^{(1,0)}$ , notice that  $E$  depends on it through  $net_j^{(1)}$ , which influences each  $o_k$  with  $k = 1, \dots, K$ :

$$o_k = S\left(net_k^{(2)}\right), \quad x_j^{(1)} = S\left(net_j^{(1)}\right), \quad net_j^{(1)} = \sum_i w_{j,i}^{(1,0)} x_i$$

Using the chain rule of derivatives again:

$$\begin{aligned} \frac{\partial E}{\partial w_{j,i}^{(1,0)}} &= \sum_{k=1}^K \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial net_k^{(2)}} \frac{\partial net_k^{(2)}}{\partial x_j^{(1)}} \frac{\partial x_j^{(1)}}{\partial net_j^{(1)}} \frac{\partial net_j^{(1)}}{\partial w_{j,i}^{(1,0)}} \\ \frac{\partial E}{\partial w_{j,i}^{(1,0)}} &= \sum_{k=1}^K \left[ -2(d_k - o_k) S'(net_k^{(2)}) w_{k,j}^{(2,1)} S'(net_j^{(1)}) x_i \right] \end{aligned}$$

# Backpropagation Learning

This gives us the following weight changes at the output layer:

$$\Delta w_{k,j}^{(2,1)} = \eta \cdot \delta_k \cdot x_j^{(1)} \quad \text{with}$$

$$\delta_k = (d_k - o_k) S'(net_k^{(2)})$$

... and at the inner layer:

$$\Delta w_{j,i}^{(1,0)} = \eta \cdot \mu_j \cdot x_i \quad \text{with}$$

$$\mu_j = \left( \sum_{k=1}^K \delta_k w_{k,j}^{(2,1)} \right) S'(net_j^{(1)})$$

# Backpropagation Learning

As you surely remember from a few minutes ago:

$$S'(x) = S(x)(1 - S(x))$$

Then we can simplify the generalized error terms:

$$\delta_k = (d_k - o_k)S'(net_k^{(2)}) \Rightarrow \delta_k = (d_k - o_k)o_k(1 - o_k)$$

And:

$$\mu_j = \left( \sum_{k=1}^K \delta_k w_{k,j}^{(2,1)} \right) S'(net_j^{(1)})$$

$$\Rightarrow \mu_j = \sum_{k=1}^K \delta_k w_{k,j}^{(2,1)} x_j^{(1)} (1 - x_j^{(1)})$$

# Backpropagation Learning

- The simplified error terms  $\delta_k$  and  $\mu_j$  use variables that are calculated in the feedforward phase of the network and can thus be calculated very efficiently.
- Now let us state the final equations again and reintroduce the subscript  $p$  for the  $p$ -th pattern:

$$\Delta w_{k,j}^{(2,1)} = \eta \cdot \delta_{p,k} \cdot x_{p,j}^{(1)} \quad \text{with}$$

$$\delta_{p,k} = (d_{p,k} - o_{p,k}) o_{p,k} (1 - o_{p,k})$$

$$\Delta w_{j,i}^{(1,0)} = \eta \cdot \mu_{p,j} \cdot x_{p,i} \quad \text{with}$$

$$\mu_{p,j} = \sum_{k=1}^K \delta_{p,k} w_{k,j}^{(2,1)} x_{p,j}^{(1)} (1 - x_{p,j}^{(1)})$$



# Backpropagation Learning

**Algorithm** Backpropagation;

Start with randomly chosen weights;

**while** MSE is above desired threshold and computational bounds are not exceeded, **do**

**for** each input pattern  $\mathbf{x}_p$ ,  $1 \leq p \leq P$ ,

    Compute hidden node inputs;

    Compute hidden node outputs;

    Compute inputs to the output nodes;

    Compute the network outputs;

    Compute the error between output and desired output;

    Modify the weights between hidden and output nodes;

    Modify the weights between input and hidden nodes;

**end-for**

**end-while**

# Improving Backpropagation Performance

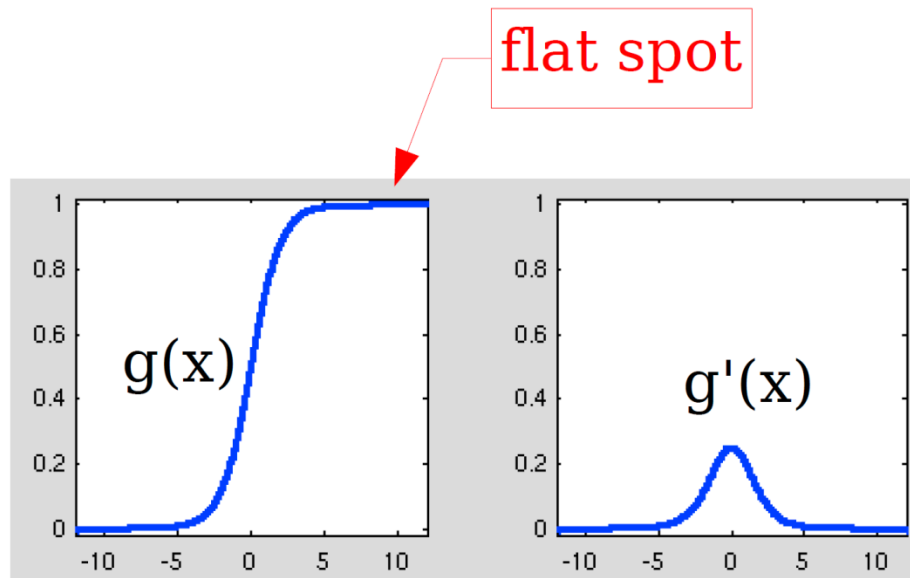
- Avoiding local minima
- Keep derivatives from going to zero
- For classifiers, use reachable targets
- Compensate for error attenuation in deep layers
- Compensate for fan-in effects
- Use momentum to speed learning
- Reduce learning rate when weights oscillate

# Avoiding Local Minima

- One problem with backprop is that the error surface is no longer bowl-shaped.
- Gradient descent can get trapped in local minima.
  - In practice, this does not usually prevent learning.
- “Noise” can get us out of local minima:
  - Stochastic update (one pattern at a time).
  - Add noise to training data, weights, or activations.
  - Large learning rates can be a source of noise due to overshooting

# Flat Spots

- If weights become large,  $net_j$  becomes large, derivative of  $g()$  goes to zero.
  - Fahlman's trick: add a small constant to  $g'(x)$  to keep the derivative from going to zero. Typical value is 0.1.



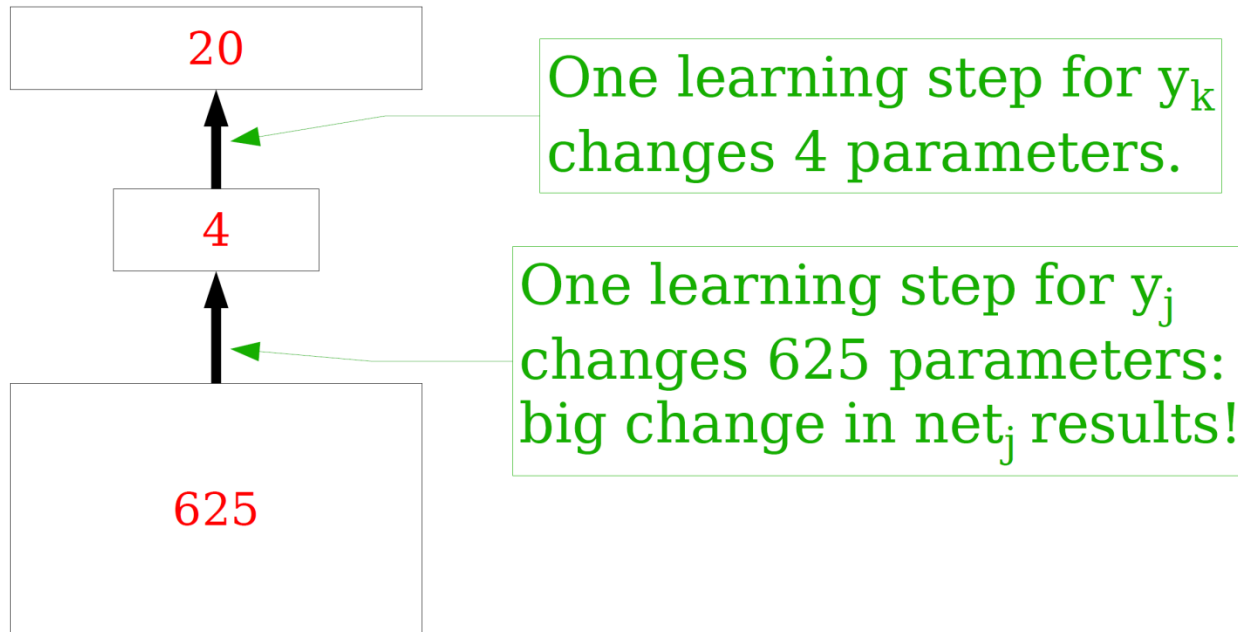
# Reachable Targets for Classifiers

- Targets of 0 and 1 are unreachable by the logistic or tanh functions.
- Weights get large as the algorithm tries to force each output unit to reach its asymptotic value.
- Trying to get a “correct” output from 0.95 up to 1.0 wastes time and resources that should be concentrated elsewhere.
- Solution: use “reachable targets” of 0.1 and 0.9 instead of 0/1. And don't penalize the network for overshooting these targets.

# Error Signal Attenuation

- The error signal gets attenuated as it moves backward through multiple layers.
  - Input-to-hidden weights learn more slowly than hidden-to-output weights.
- Solution: have different learning rates for different layers.
  - Or use Deep Learning

# Fan-In Affects Learning Rate



- Solution: scale learning rate by fan-in.

# Momentum

- Learning is slow if the learning rate is set too low.
- Gradient may be steep in some directions but shallow in others.
- Solution: add a momentum term  $\alpha$ .

$$\Delta w_{ij}(t) = -\eta \frac{\partial E}{\partial w_{ij}(t)} + \alpha \cdot \Delta w_{ij}(t-1)$$

- Typical value for  $\alpha$  is 0.5.
- If the direction of the gradient remains constant, the algorithm will take increasingly large steps.