# CISC/CMPE452/COGS 400 Multi-layer Perceptrons with Backpropagation

Ch. 3 Text book

Farhana Zulkernine

# Multilayer Networks

- Although single-layer perceptron networks can distinguish between any number of classes, they require linear separability of inputs.
- To overcome this limitation, we can use multiple layers of neurons.
- Rosenblatt first suggested this idea in 1961, but he used perceptrons.

# Multilayer Networks (cont…)

- However, their non-differentiable output function led to an inefficient and weak learning algorithm.
- The idea that eventually led to a breakthrough was the use of continuous output functions and gradient descent.

# Multilayer Networks (cont…)

- Multilayer networks introduced the **credit-assignment problem** - how to determine which nodes are responsible for an outcome (so that the corresponding weights can be changed)
- The solution – **backpropagation** (BP) **network** – which computes errors in a node and propagates the correction to the weight values backwards through the hidden nodes
- **Backpropagation algorithm** was popularized by Rumelhart, Hinton, and Williams (1986).

# Architecture

- BP networks consist of multiple layers with the hidden and output layers processing the inputs.
- One dummy input (=1) is used often to be able to treat the bias as a modifiable weight value.
- Output layer can contain multiple output nodes.

# Output Function – Sigmoid

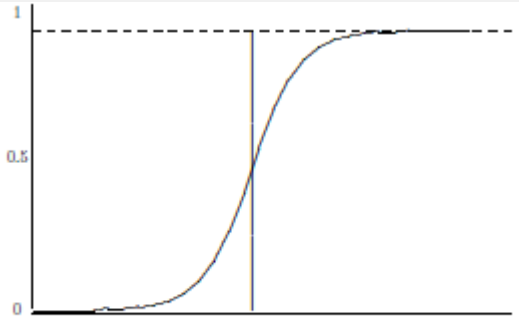



Figure 1.2: A sigmoid function

Each hidden and output node applies a sigmoid function to its net input.

The main reasons motivating the use of an S-shaped sigmoidal function are that it is continuous, monotonically increasing, invertible, every-where differentiable, and asymptotically approaches its saturation values as $net \rightarrow \pm\infty$.

The basic properties of the sigmoidal function are more important than the specific sigmoidal function chosen in our presentation bellow, viz.

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

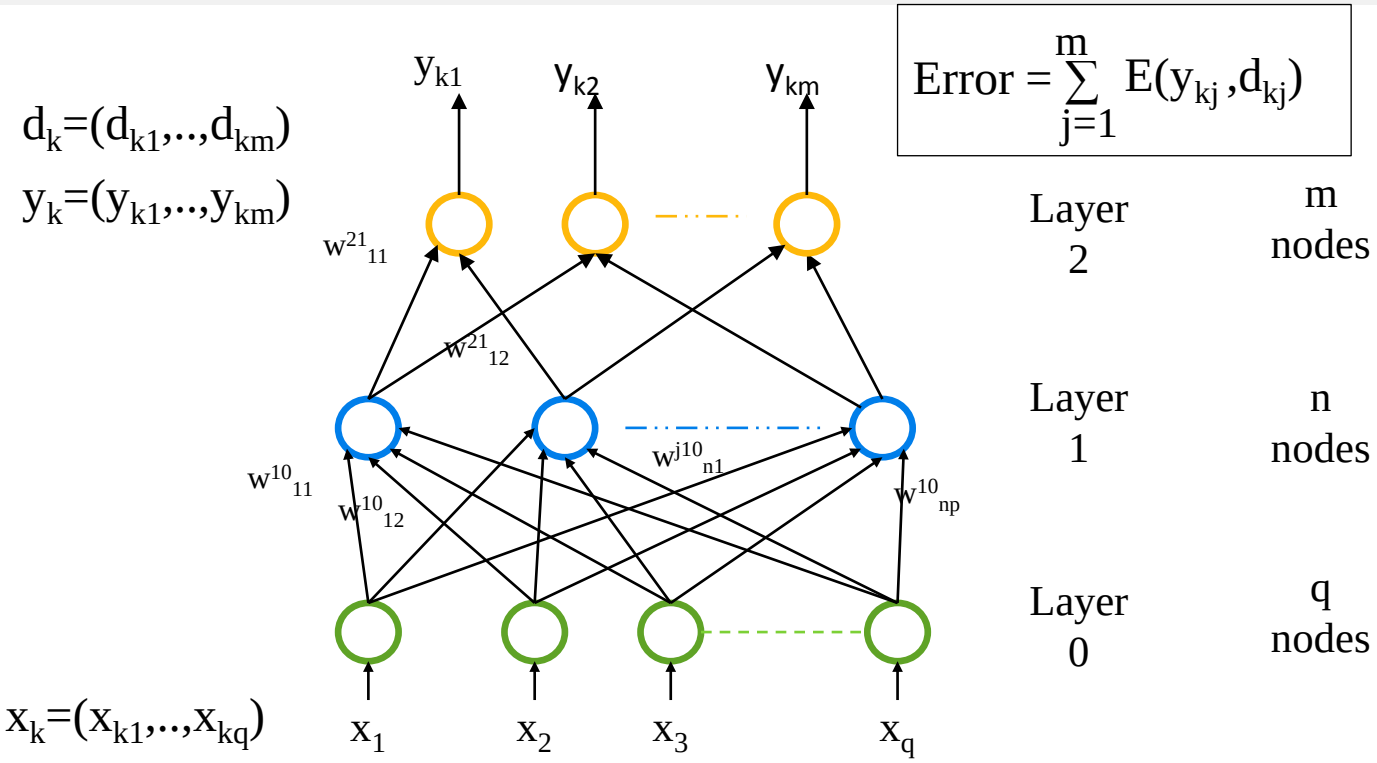
# Supervised Learning

The algorithm is supervised learning algorithm throned using P input patterns. For each input vector $x_p$, we have the corresponding desired K-dimensional output vector.

$$d_p = (d_{p,1}, d_{p,2}, \ldots, d_{p,K})$$

for $1 \leq p \leq P$. This collection of input-ouput pairs constitutes the training set $\{(x_p, d_p) : p = 1, \ldots, P\}$. The length of the input vector $x_p$ is equal to the number of inputs for the given application. The length of the output vector $d_p$ is equal to the number of outputs of the given application.

# 3 - Layer Network




$$\text{Error} = \sum_{j=1}^{m} E(y_{kj}, d_{kj})$$

We consider P items in the training dataset as

$\{(x_p, d_p): p= 1,\ldots,P\}$ i.e., $(x_1, d_1), (x_2, d_2),\ldots, (x_P, d_P)$

where $x_k=(x_{k1},..,x_{kq})$ and $d_k=(d_{k1},..,d_{km})$

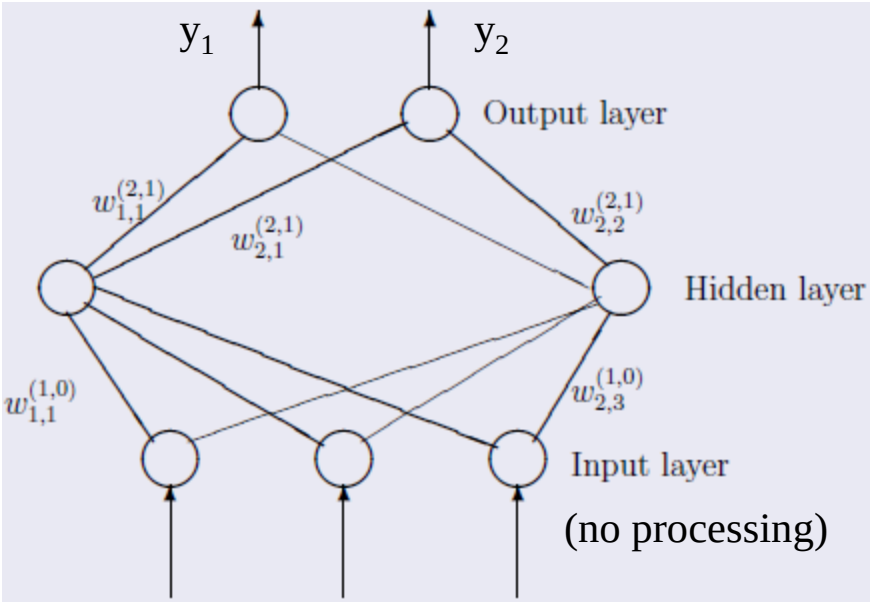
$w^{ll'}{}_{ji}$ is the weight to node j in layer *l* from $i^{th}$ input in layer *l′*

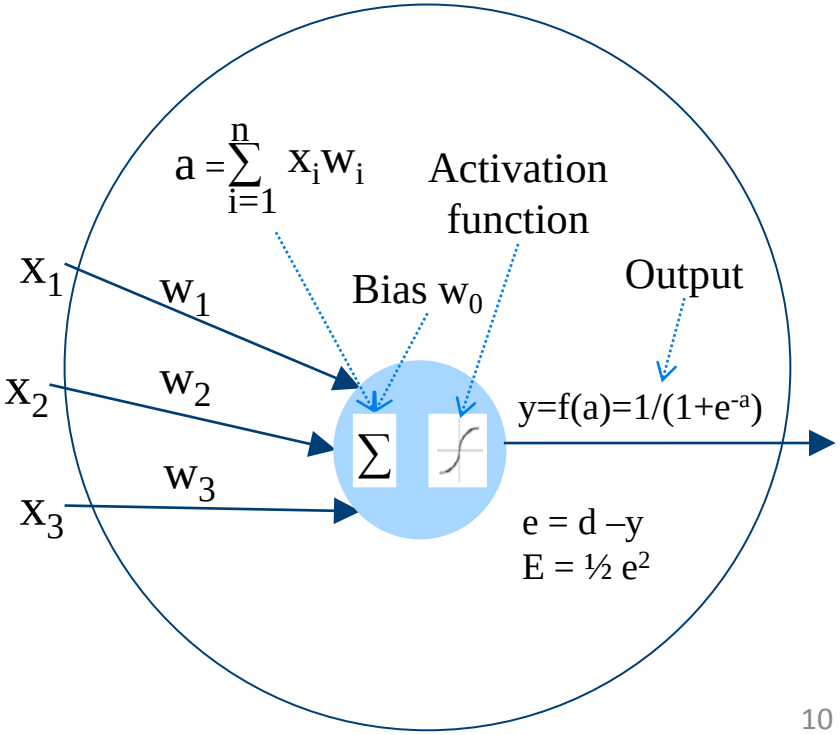
# Backpropagation Algorithm

- The BP algorithm assumes a **feedforward** neural network architecture where nodes are partitioned into layers {0, L}.
- The input layer is layer 0 and the output layer is L.
- BP addresses networks for which $L \geq 2$, containing "hidden layers" numbered 1 to L – 1.

# Backpropagation Networks

- We consider a BP network with 2 processing layers. The layers are output, hidden, and input (which is just fan-in).



# Goal for Weight Adjustment

- Initially, the weights are assigned random values.
- As in the case of perceptrons and Adalines, the goal of training is to modify the weights in the network so that the network's output vector

$$y_p = (y_{p,1}, y_{p,2}, \ldots\ldots, y_{p,m})$$

  is as close as possible to the desired output vector $d_p$, when an input vector $x_p$ is presented to the network.
- Hence our goal will be to find a set of weights that minimize the *Sum Square Error*, E, for all training data where

$$E = \sum_{k=1}^{P} \sum_{j=1}^{m} E(y_{kj}, d_{kj}) = \sum_{k=1}^{P} \sum_{j=1}^{m} (y_{kj} - d_{kj})^2$$

# 2-Layer Network

- We use **gradient descent** to determine the direction of change of weights which is:

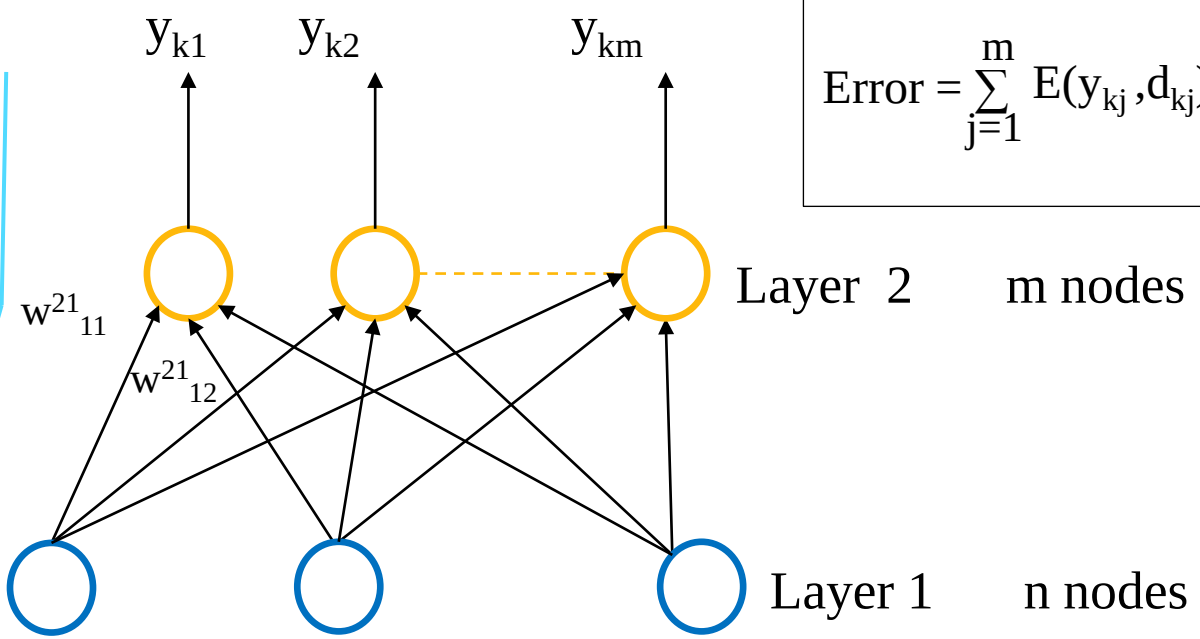
  $\Delta w = -\partial E/\partial w$.
- For all layers, we need to apply adjustments to each weight $w^{\ell\ell'}_{ji}$, which is the weight to node *j* in layer $\ell$ from input *i* in layer $\ell'$.
- $w^{\ell\ell'}_{ji}$ only affects output from node j or $y_j$. Hence, for computing $\partial E$ it is sufficient to compute $\partial E/\partial y_j$ and then differentiate $y_j$ with respect to $w_{ji}$
- Error correction propagates backwards.
- For a 2-layer network we start from final output layer 2 towards layer 0. Therefore, $\ell=2, \ell'=1$.

# 2-Layer Network

$d_k=(d_{k1},..,d_{km})$

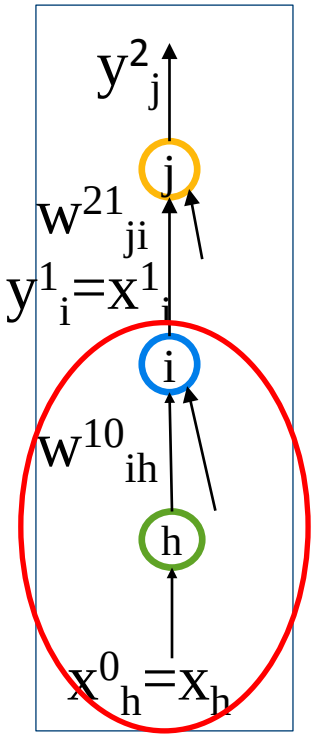
$y_k=(y_{k1},..,y_{km})$

**Error Correction** $\Delta w^{21}_{ji}$



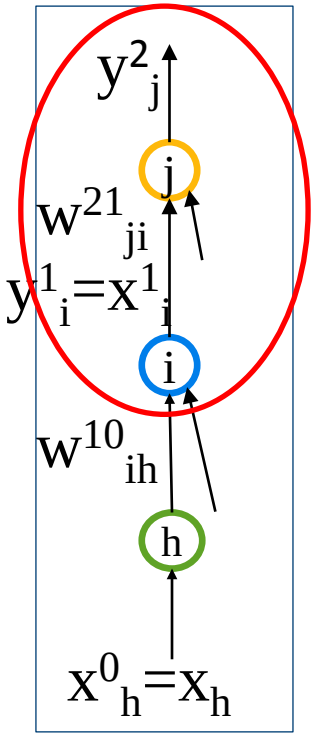

$$\text{Error} = \sum_{j=1}^{m} E(y_{kj}, d_{kj})$$

# Inputs/Outputs




- $h^{th}$ node in the input layer holds a value of $x_{p,h}$ for the $p^{th}$ pattern
- The net input to the $i^{th}$ node in the hidden layer: $a^1_i = \sum_{h=0}^{n} w^{10}_{ih} x_h$
- The output from $i^{th}$ node in the hidden layer: $y^1_i = x^1_i = f(a^1_i) = S(\sum_{h=0}^{n} w^{10}_{ih} x_h)$

# Inputs/Outputs (cont…)




- Net input to the $j^{th}$ node in the output layer for the $p^{th}$ pattern: $a^2_j = \sum_{i=0}^{n} w^{21}_{ji} x_i$
- The output from $j^{th}$ node in the output layer using Sigmoid function:

  $y^2_j = f(a^2_j) = S(\sum_{i=0}^{n} w^{21}_{ji} x_i)$
- Desired output of $j^{th}$ node in the output layer for the $p^{th}$ input pattern is: $d_{p,j}$
- The corresponding squared error is:

  $E_j = | d_{p,j} - y_{p,j} |^2$

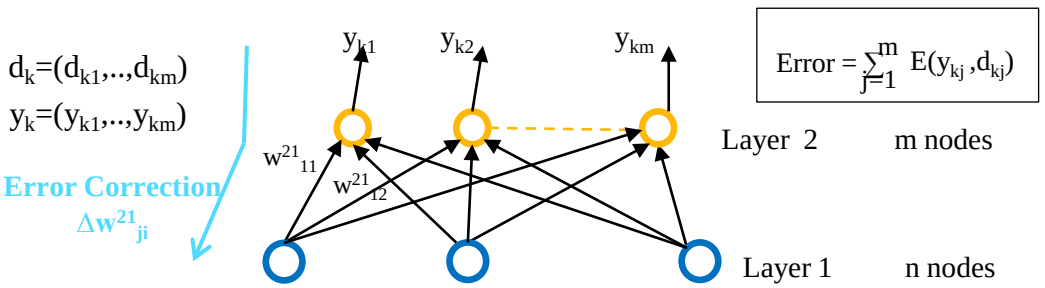
# Weight Adjustment for $\Delta w^{21}_{ji}$

- For layer 2 ignoring $\ell\ell'$ ($\ell$=2, $\ell'$=1):

  $\Delta w_{ji} = -c.\partial E/\partial w_{ji}$ c is small constant learning rate

  1. $E = \frac{1}{2} \sum_j e_j^2$ where E is total error at the layer
  2. $e_j = (d_j - y_j)$ $e_j$ is error at node j
  3. $y_j = f(a^2_j)$ $y_j$ actual output at node j
  4. $a_j = \sum_i (w_{ji} x^1_i)$ activation at layer 2

  $x^\ell_i$ = input to layer ($\ell$+1) and output from layer $\ell$



# $\Delta w^{21}_{ji}$ (cont...)

1. $E = \frac{1}{2} \sum_j e_j^2$ 2. $e_j = (d_j - y_j)$ 3. $y_j = f(a^2_j)$ 4. $a^2_j = \sum_i (w_{ji} x^1_i)$

$\Delta w_{ji} = -c.\partial E/\partial w_{ji}$

Using chain rule

$\Delta w_{ji} = -c.\ \partial E/\partial e_j\ .\ \partial e_j/\partial y_j.\ \partial y_j/\partial a^2_j\ .\ \partial a^2_j/\partial w_{ji}$

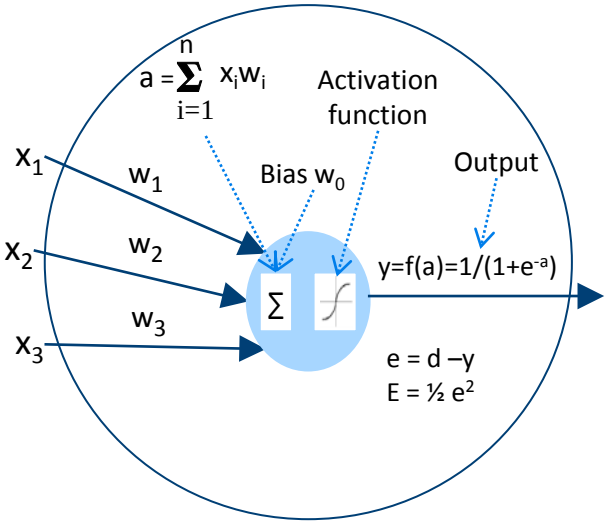
$= -c.\ (d_j - y_j).\ (-1).\ f'(a^2_j).\ x^1_i$

$= c.\delta^o_j.x^1_i$

where $x^\ell_i$ = input to layer $(\ell+1)$ and output from layer $\ell$

$\delta^o_j = (d_j - y_j).f'(a^2_j)$ for output layer

$= e_j.\ f'(a^2_j)$ --- a more general form




***Note $y_j$ without superscript layer number indicates output layer

# Compute f(a)

- Output function y=f(a) can be any function but it must be differentiable for Backpropagation
- If $y = f(x) = 1/(1+e^{-x})$ , a sigmoidal function then

$$f'(x) = e^{-x}/(1+e^{-x})^2$$

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

$$= f(x)(1 - f(x)) = y\,(1 - y)$$

# Compute $\delta^0_j$

- We defined $\delta^o_j = (d_j - y_j).f'(a^2_j)$ where

$$y_j = f(a^2_j) = S(\sum_{i=0}^{n} w^{21}_{ji} x_i)$$

- Therefore, $\delta^o_j = (d_j - y^2_j).\ y^2_j .(1 - y^2_j)$

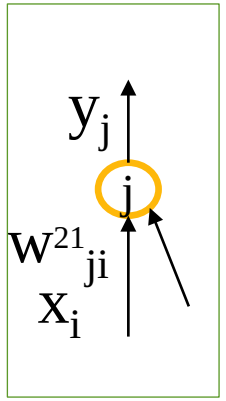
# $\Delta w^{21}_{ji}$ (cont...)

$Y = f(x) = 1/(1+e^{-x})$ $\quad f'(x) = y(1-y)$ $\quad \delta^o_j = (d_j - y_j).y_j.(1-y_j)$

Coming back to $\Delta w^{21}_{ji}$

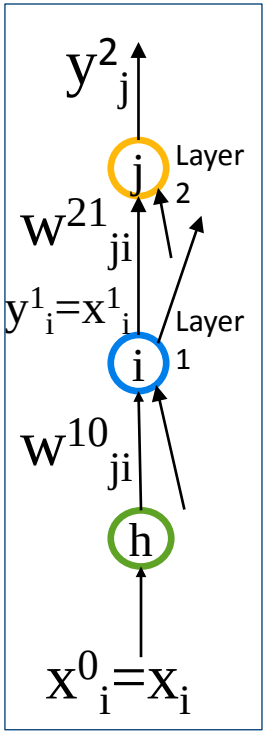
$$\Delta w^{21}_{ji} = c.\delta^o_j.x^1_i$$
$$= c.x^1_i.(d_j - y_j).y_j.(1 - y_j)$$

Where $y_j = f(a^2_j) = S(\sum_{i=0}^{n} w^{21}_{ji} x_i)$




- Very similar to the Perceptron rule that was used for Adaline : constant x error x input
- The difference is that $\delta^o_j$ , the error at output node j, is the *usual error $(d_j - y_j)$, scaled by a factor of $f'(a^2_j)$*

# Weight Adjustment for Hidden Layer $\Delta w^{10}_{ih}$




- For hidden layer 1, $\ell = 1$ and $\ell' = 0$:

  $\Delta w^{10}_{ih} = -c.\sum_{j=1}^{n} \partial E_j / \partial w^{10}_{ih}$
- In this case error E depends on $w^{10}_{ih}$ through $f(a^1_i)$, which in turn affects each $y^2_j$, j=1,…,m.
- $y^2_j = f(a^2_j)$ output from node j in layer 2

  $y^1_i = f(a^1_i)$ output from node i in layer 1

  $a^1_i = \sum_h (w^{10}_{ih} x^0_h)$ activation at hidden layer 1

  $x^\ell_h$ = input to layer $(\ell+1)$ from node h in layer $\ell$

  ( $x^0_h$ input to layer 1 from node h in layer 0)

# Weight Adjustment for Hidden Layer $\Delta w^{10}_{ih}$

1. $E = \frac{1}{2} \sum_j e_j^2$ 2. $e_j = (d_j - y_j)$ 3. $y_j = f(a_j)$ 4. $a^2_j = \sum_i (w^{21}_{ji} x^1_i)$ 5. $a^1_j = \sum_i (w^{10}_{ih} x^0_h)$

$$\Delta w^{10}_{ih} = \sum_{j=1}^{n} \{ - c.\partial E_j / \partial \mathbf{w^{10}_{ih}} \}$$

Using chain rule

$$\Delta w^{10}_{ih} = \sum_{j=1}^{n} \{ - c.\ \partial E_j/\partial e_j \ .\ \partial e_j/\partial y_j \ .\ \partial y_j/\partial a_j \ .\ \partial a_j/\partial w^{10}_{ih} \}$$

Same as before

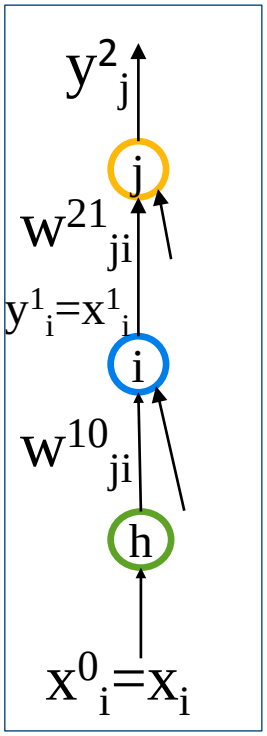
$$= \sum_{j=1}^{n} \{ c \ .\ \delta^o_j \ .\ \partial a^2_j/\partial x^1_i \ .\ \partial y^1_i/\partial a^1_i \ .\ \partial a^1_i/\partial w^{10}_{ih} \}$$ where $x^1_i = y^1_i$

From before

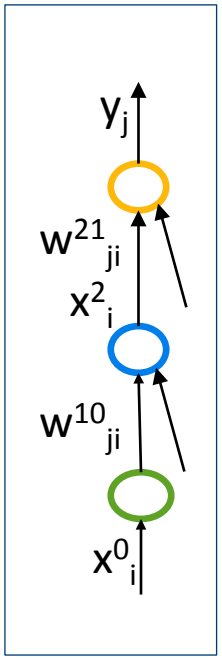
$$= c \ . \sum_{j=1}^{n} \{ \delta^o_j \ .\ w^{21}_{ji} \ .\ f'(a^1_i) \ .\ x^0_h \}$$

$\delta^h_i$

$$= c \ .\ \delta^h_i \ .\ x^0_h$$



# Hidden Layer $\Delta w^{10}_{ji}$ (cont...)




$$\Delta w^{10}_{ih} = c \,.\, \delta^h_i \,.\, x^0_h$$

where

$$\delta^h_i = \sum_{j=1}^{n} \{ \delta^o_j . w^{21}_{ji} \} . f'(a^1_i) \quad \text{and}$$

$$y^1_i = f(a^1_i) \quad \text{and} \quad f'(a^1_i) = y^1_i.(1 - y^1_i) \text{ and}$$

$$\delta^o_j = (d_j - y^2_j) . y^2_j.(1 - y^2_j)$$

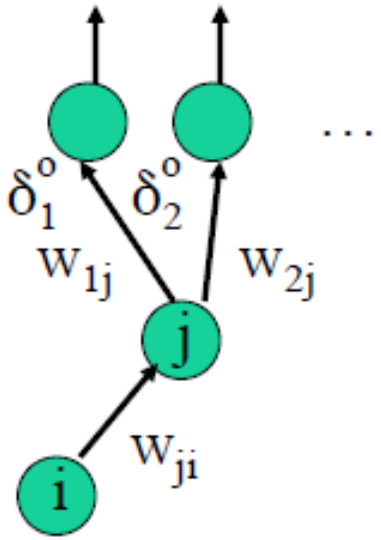
# Backpropagation 2 Layer Network

1. Start with **randomly chosen weights**;
2. **While** MSE is unsatisfactory and computational bounds are not exceeded do
3. **For each input pattern** $x_p$ ,$\{1 \leq p \leq P\}$ do
4. Compute hidden node inputs ($a^1_i$);
5. Compute hidden node outputs ($y^1_i = x^1_i$);
6. Compute inputs to the output nodes ($a^2_j$)
7. Compute the network outputs ($y^2_j$)
8. Modify weights between hidden & output nodes: $\Delta w^{21}_{ji}$
9. Modify weights between input & hidden nodes: $\Delta w^{10}_{ih}$
10. **End for**
11. **End while**

# Problems with Backpropagation

1. It is not generally possible to know whether or not there are any *local minima* in the error surface.
   - Even for very simple problems, it has been shown that Backpropagation will sometimes not find the "correct" solution.
2. The kind of simple error function that is used in deriving the basic Backpropagation equations is usually not appropriate.
   - The derivation using the necessary error function can be very difficult.

# Problems (cont…)




$$\delta_j^o = e_j \cdot f'(a_j) \qquad e_j = (d_j - y_j)$$

$$\delta_j^h = \sum \delta_k^o w_{kj} \cdot f'(a_j)$$

3. The **accumulated error at the hidden nodes is a sum of signed values**, and so two very large output node errors (one negative and one positive) *could cancel each other out*, leaving the impression that there is no error that was contributed to by the hidden node.

# Problems (cont…)

4. The *training trials*, the *test trials*, and the *production trials* should all be statistically indistinguishable from one another.
   - One can't control whether the situation will change after the network is trained → performance evaluation helps.
   - When selecting the test trials, one should ensure that they are statistically indistinguishable from the training trials and not just randomly selected.
   - It's **difficult to know how many testing trials to use**. More *training* trials is best, but insufficient testing trials could lead to over-generalization.

# Problems (cont…)

5. Training times can be very long.
   - May require tens of thousands of training trials.
     - Larger the network →more weights→ longer training time required.
6. No exact methods to configure ANN (number of hidden nodes), and trial-and-error tests may be required to find the best possible performance.
7. Backpropagation is **"unbiological"**.
   - There is no evidence of an ability to use synaptic connections "backwards", as is required to develop the error values at the hidden nodes. There are models which attempt to make Backpropagation more plausible by having a set of "satellite" nodes around each hidden node to backpropagate the signals in a way that conforms to the usual neuron models.
   - Even the strictly forward layering is unlike the usual structures of the brain.

# Problems (cont…)

8. Backpropagation cannot usually have its operation enhanced easily with, say, an additional training trial.
   - The relearning that is required can result in a complete restructuring of the values of the weights.
9. Backpropagation may be able to solve a problem, but it is generally not useful in understanding the solution that is being taken.

# K-Class Classification Problem

- Let us denote the k-th class by $C_k$, with $n_k$ exemplars or training samples, forming the sets $T_k$ for k = 1, …, K:

$$T_k = \left\{ \left( x_p^k, d_p^k \right) \middle| \, p = 1, \ldots, n_k \right\}$$

- The complete training set is $T = T_1 \cup \ldots \cup T_K$.
- The goal is to *find a set of weights* such that the output layer nodes respond with the desired output vector $d^k_p$ whenever the corresponding input vector $x^k_p$ is presented.

# K-Class Classification Problem

$$T_k = \left\{ \left( x_p^k, d_p^k \right) \mid p = 1, \ldots, n_k \right\}$$

- If a training set $T_k$ where k = 1, …, K, represents the k-th class $C_k$ as shown above then the complete training set is

$$T = T_1 \cup \ldots \cup T_K$$

- Goal: Find a *set of weights* such that for input vector $x^k_p$ of class k, the ANN should produce an output vector close to the desired output vector $d^k_p$ having 1 at the k-th position and 0 in all other positions: $\mathbf{d}^k = (0,0,\ldots,0,1)$

# Desired Output Value???

- Due to the sigmoid output function,

$$f(\text{net}) = 1 \text{ when net} \rightarrow \infty \text{ and}$$

$$f(\text{net}) = 0 \text{ when net} \rightarrow -\infty$$

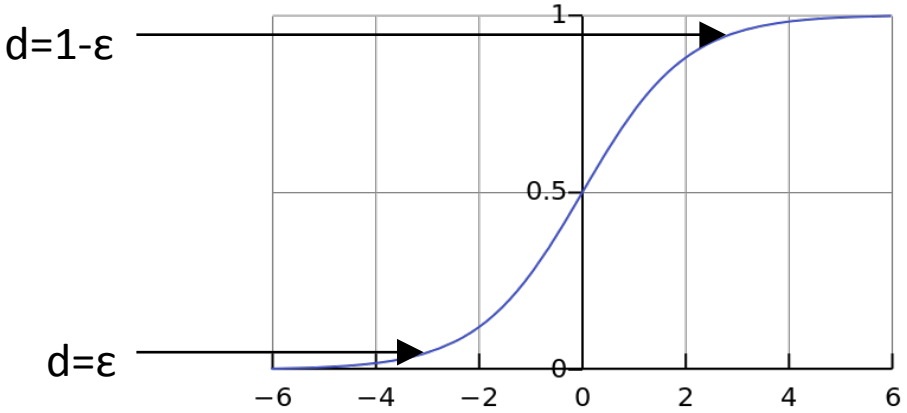
- Because of the shallow slope of the sigmoid function at extreme net inputs, even approaching these values would be very slow.
- To avoid this problem, it is advisable to **use desired outputs ε and (1 - ε) instead of 0 and 1**, respectively where typical values for ε range between 0.01 and 0.1.
- For ε = 0.1, desired output vectors would look like this:

$$\mathbf{d}^k = (0.1, 0.1, \ldots, 0.1, 0.9)$$

# Desired Output Value (cont…)

- To avoid punishment, we can define error $e_{p,j}$ as follows using a reduced range of desired output d:
  - If $d_{p,j} = (1 - \varepsilon)$ and $y_{p,j} \geq d_{p,j}$, then $e_{p,j} = 0$.
  - If $d_{p,j} = \varepsilon$ and $y_{p,j} \leq d_{p,j}$, then $e_{p,j} = 0$.
  - Otherwise,

    $e_{p,j} = | d_{p,j} - y_{p,j}|$



# Desired Output Value (cont…)

- But changing desired values to 0.05 and 0.95 instead of 1.0 and 0.0 **requires an adjustment in the learning algorithm**.
- When calculating e = (d - y), if d = 0.95, and y > 0.95, we still consider e = 0, and similarly for y < 0.05 when d = 0.05, we consider e=0.
- As a result, the greater the values of | d - y |, the more adjustments are made to the weights.

# Classification of Unknown Pattern

- **After a network is trained**, assign an unknown input data pattern $x_p$ to class using one of the following methods:
  1. Generate output y for a given input pattern and find to which class output d it is the closest, i.e.,

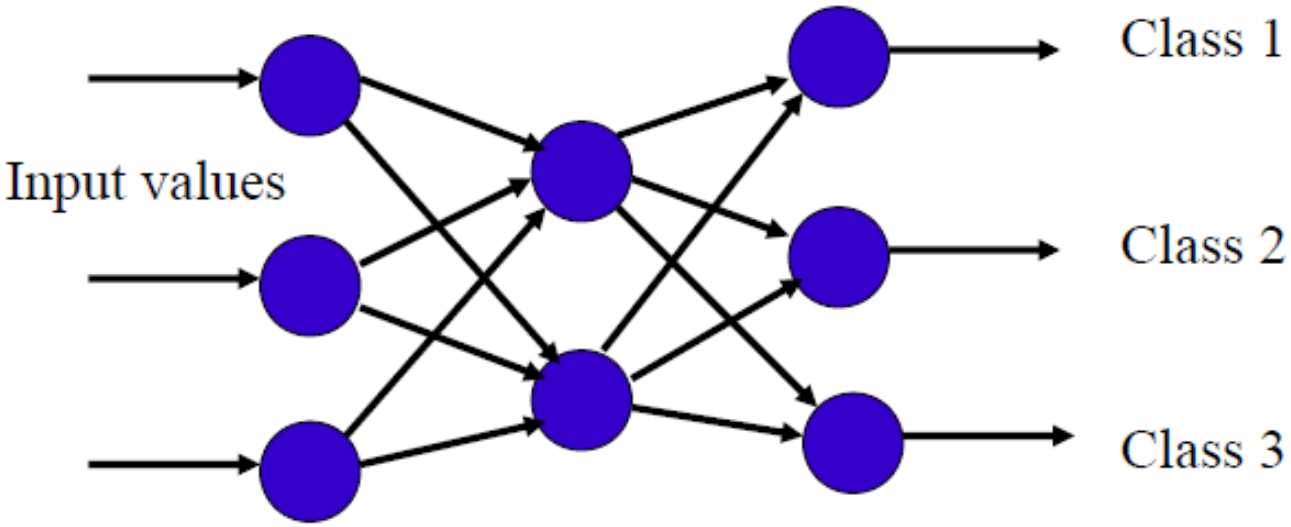
     If $\|d^j - y^j\| \leq \|d^k - y^k\|$, where $j \neq k$ for all k, then assign $x_p$ to class j.
     - $\|d - y\|$ represents Euclidean norm or distance between d and y.
     - If two classes are equidistant, assign the pattern randomly to one of the two.
  2. If $y_j > y_k$ for all $j \neq k$, assign $x_p$ to class j.
     - For a two class problem if y>=0.5 assign it to class $C_1$ and otherwise to class $C_2$.

# Applications

- Classification or pattern association
  - Relating input/output pairs for retrieval (more robust system, with graceful degradation, and with parallel associations)
- Prediction based on training data
  - Can also be used for time series prediction by splitting data in multiple overlapping and sequential time windows.
- Image compression
- Function Approximation
  - Given a function y= f(x) (known or unknown), and examples $\{(x_1,y_1) , \dots , (x_p,y_p)\}$, a neural network can be trained to compute y = f(x)
    - Real estate prediction
    - Image compression

# Applications of ANN

- Classification using Supervised Learning
  - Recognizing characters
  - Loan classification
  - Analyzing radar signals

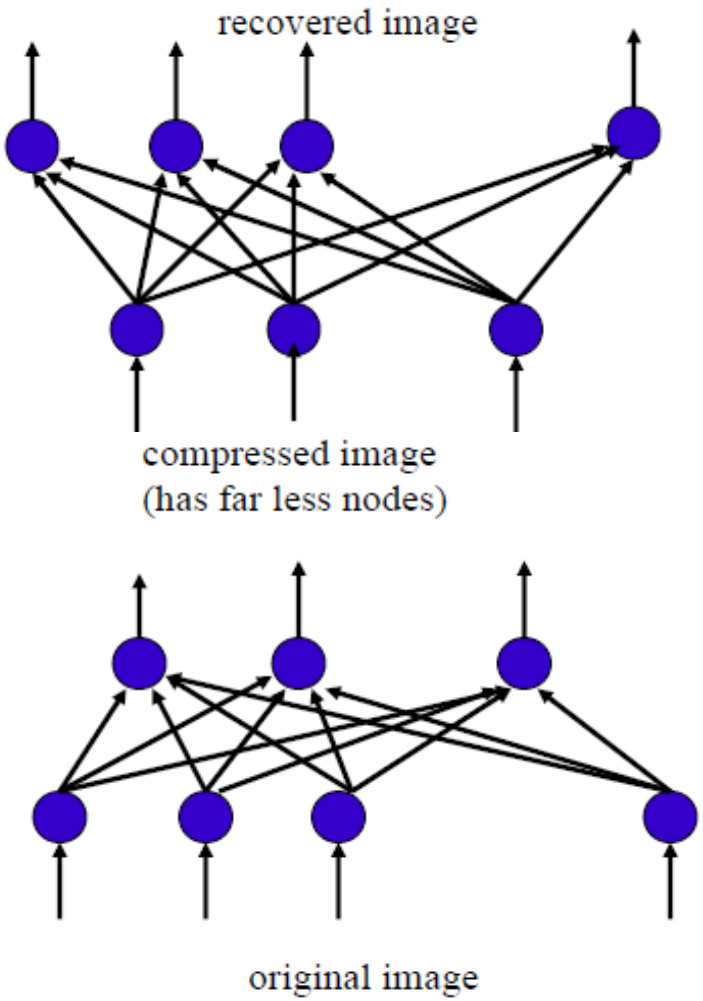


# Time Series Prediction

- If we have a sequence of data (say stock market data for a sequence of days) and we want to predict the next values based on the preceding ones.
- If the data is ($v_1$, $v_2$, $v_3$, … $v_n$), with one $v_i$ or each day, we define a window of some size (say 5) we construct our training trials:
- $x_1 = (v_1, v_2, v_3, v_4, v_5)$ $\quad d_1 = v_6$
- $x_2 = (v_2, v_3, v_4, v_5, v_6,)$ $\quad d_1 = v_7$
- $x_3 = (v_3, v_4, v_5, v_6, v_7)$ $\quad d_1 = v_8$
- So that the input for each training trial is a window into the past for each of the desired values.

# Image Compression

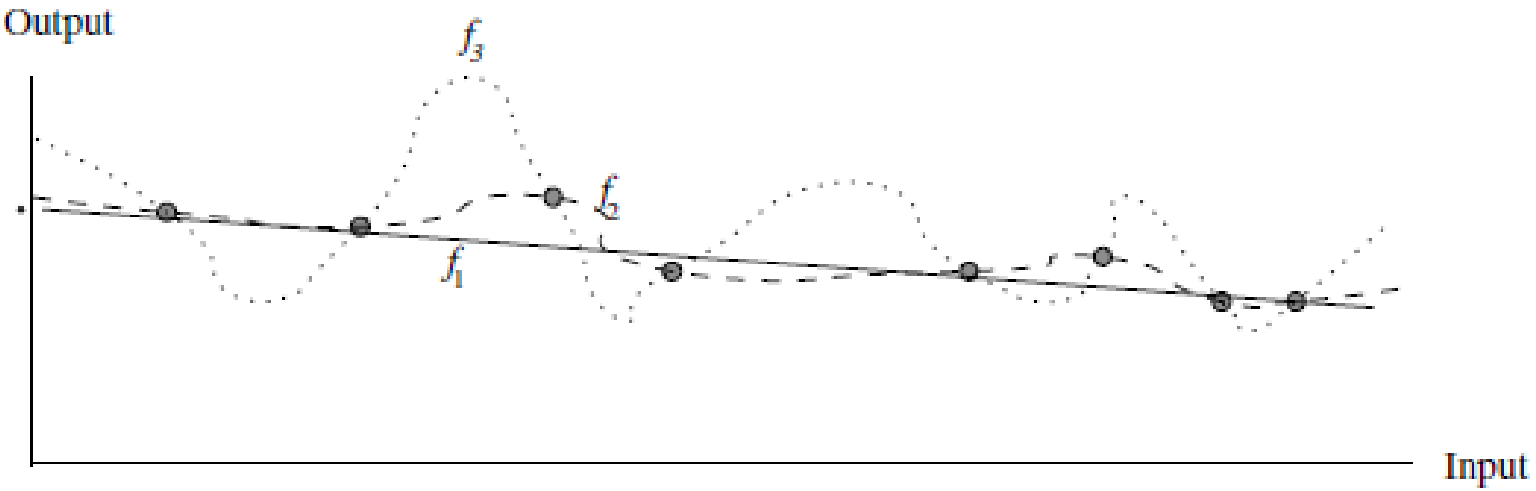
- Train the network to perform the task of just copying the input image to the output.
- Then the first layer of the network is performing image compression while the second layer of the network decompresses the image.



# Function Approximation

- Function Approximation generally refers to the task of approximating mathematical functions, with input vectors whose components are numbers, and a single output that is also a number.
- Since infinitely many functions exist that coincide for a finite set of points, additional criteria are necessary to decide which of these functions are desirable: (i) continuity, (ii) smoothness, (iii) simplicity, (iv) parsimony.

# Function Approximation




- These criteria sometimes oppose the performance criterion of minimal error. In the Figure, the straight line f1 performs reasonably well but f2 and f3 perform best and have zero error.
- Among the latter, f2 is smoother and needs fewer network parameters.

# Book Examples – Ch 3

- See the examples in Ch. 3 of the book on BP networks.
- Classification of Myoelectric signals

# Example 3.7.1 Weaning - Textbook

- Weaning from mechanically assisted ventilation – discontinuing the respiratory support under certain observable conditions.
- Observe how the input data are normalized to lie between 0 and 1 (see in textbook).
- The example network uses momentum to avoid being stuck at a local minima. The learning rate was $\eta = 0.9$ and momentum rate was $\alpha = 0.4$ (doesn't have much effect).

# Example 3.7.2 Myoelectric signals – Textbook

- Electrical signals that correspond to muscle movements in animals - measured on the surface of the skin.
- Classification of such signals into three groups that translate directly into movements of specific parts of the body.
- Problem – Signal measurements contain significant amounts of noise, due to background electrical activity in nerves unrelated to the movement of the relevant muscles.
- Hence perfect classification is impossible.

# Summary

1. Provide an example input x, allow the network to compute in feedforward mode, and produce output y.
2. Calculate the error at each output node j by comparing the desired output d to the actual output y to get $e_j = (d_j - y_j)$. Then $\delta^o_j = e_j . f'(a_j)$
3. Calculate the error at the hidden nodes based on the output node errors (error is propagated back to hidden layer). $\delta^h_i = \sum_{j=1}^{n} \{ \delta^o_j . w^{21}_{ji} \} . f'(a_i)$.
4. Adjust all weights (from *j* to *h*) at output and hidden nodes on the basis of their calculated error.

   $\Delta w_{ji} = c.\delta^o_j.x^1_i$ and $\Delta w_{ih} = c . \delta^h_i . x_h$