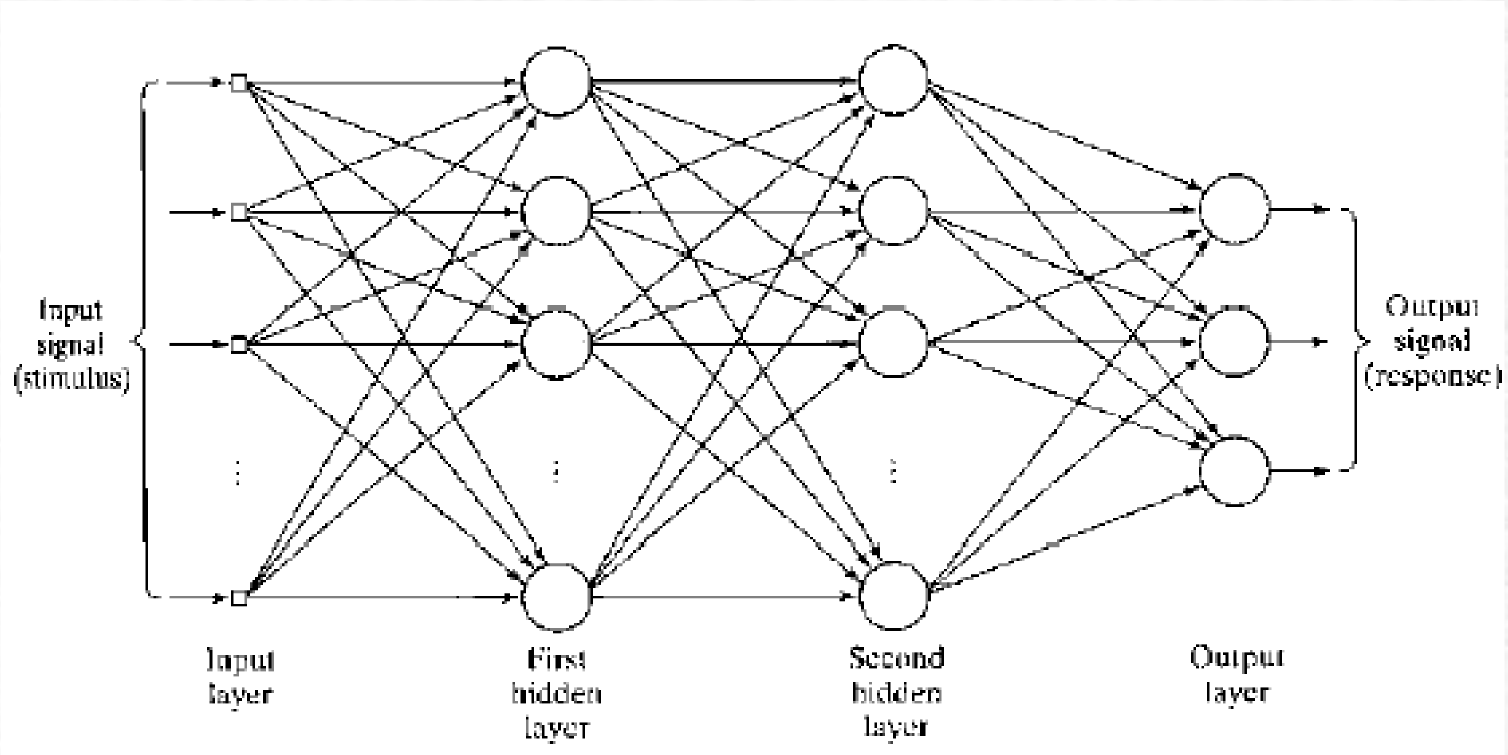
Chapter.V. Multilayer Perceptrons

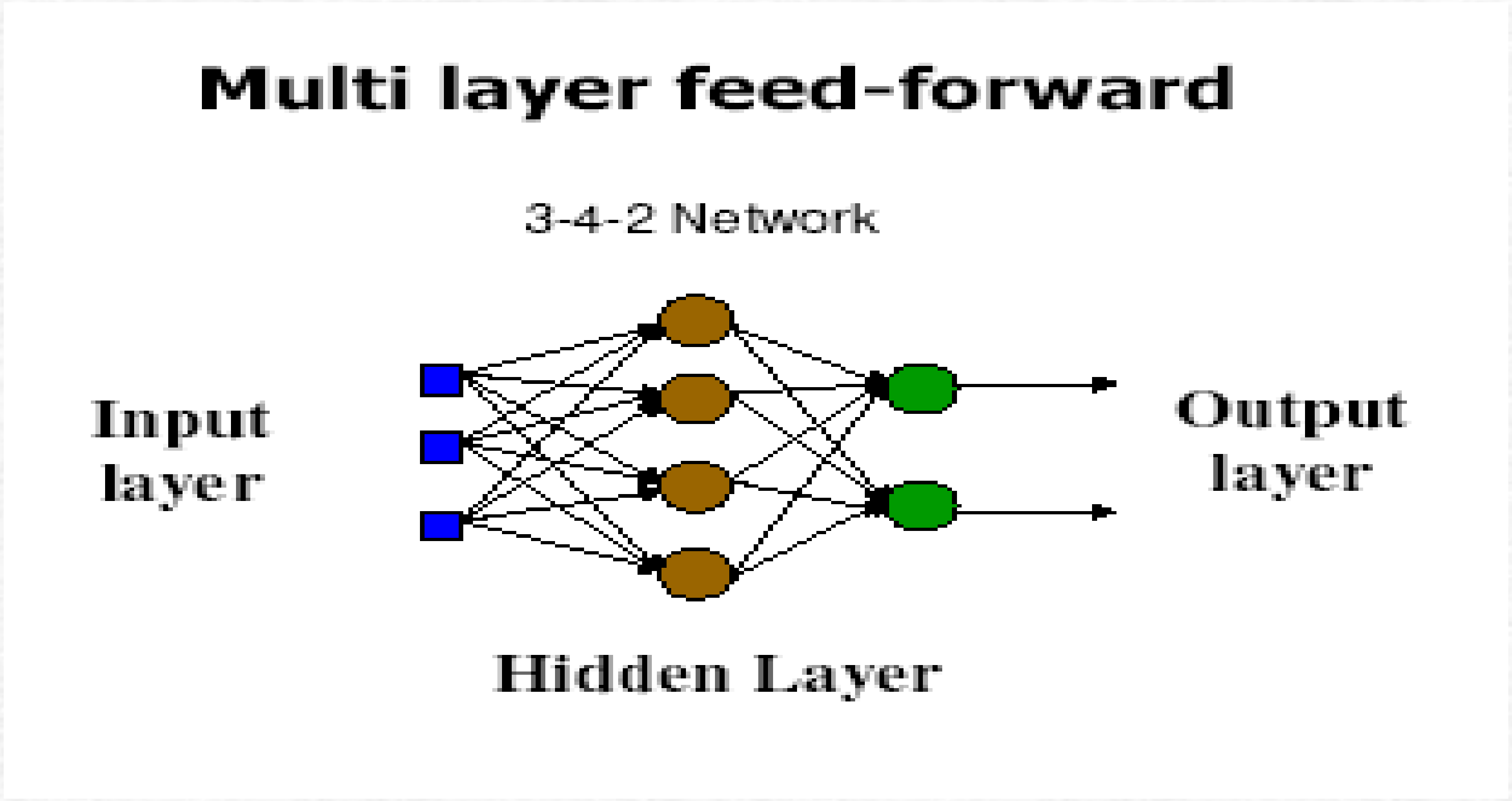
Practical Considerations

- Do we need to pre-process the training data? How?
- How do we choose the initial weights from which we start the training?
- How do we choose the learning rate?
- Do we change the weights after each training pattern, or after the whole set?
- Are some activation functions better than others?
- How do we avoid flat spots in the error function?
- How do we avoid local minima in the error function?
- When do we stop training?

5.1. Structure- Feedforward Multilayer Perceptron



A multilayer Perceptron is a feedforward networks with one or more layers of nodes between the input and output

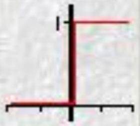
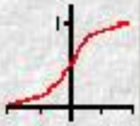
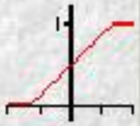
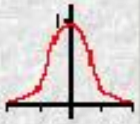


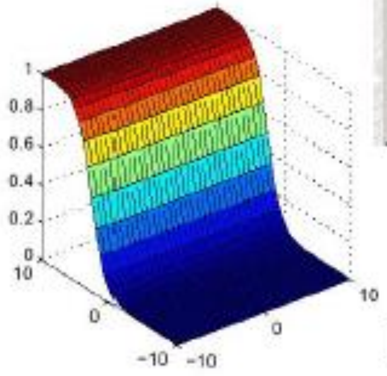
The Characteristics

- The nonlinearity
- Contains one or more hidden neurons
- High degree of connectivity
- The learning process is more difficult due to a much larger space

Activation Functions

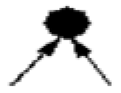
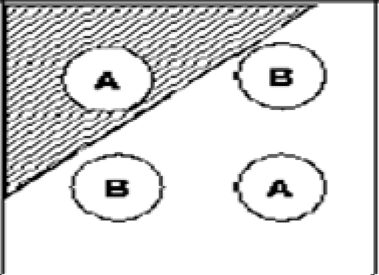
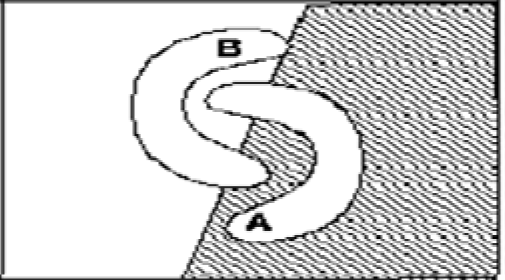
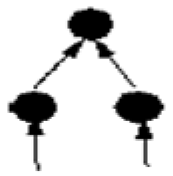
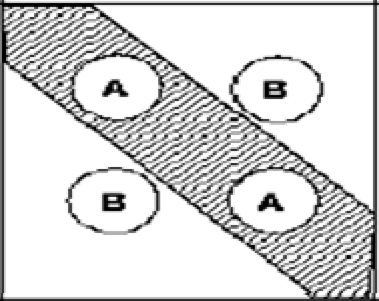
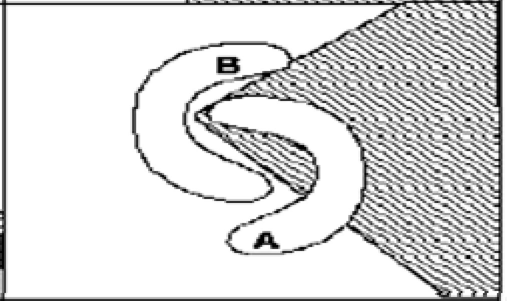
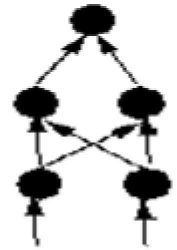
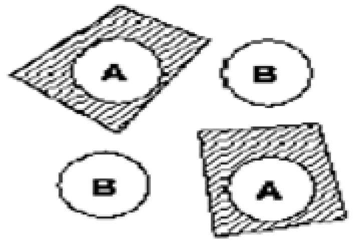
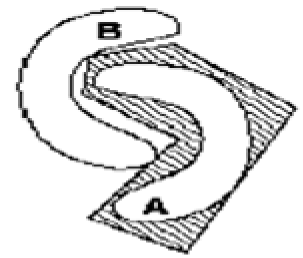
Activation functions

Unit Step		$f(x) = \begin{cases} 0 \text{ if } 0 > x \\ 1 \text{ if } x \geq 0 \end{cases}$
Sigmoid		$f(x) = \dfrac{1}{1+e^{-\beta x}}$
Piecewise Linear		$f(x) = \begin{cases} 0 & \text{if } x \leq x_{min} \\ mx+b & \text{if } x_{max} > x > x_{min} \\ 1 & \text{if } x \geq x_{max} \end{cases}$
Gaussian		$f(x) = \dfrac{1}{\sqrt{2\pi\sigma}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$



Logistic

Structures and Their Performance

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)		

Credit Assignment Problem

- In the output layer we have direct access to the visible neurons. It is a straightforward matter to calculate the error signals in that layer, hence assign credit or blame.
- But what about the hidden neurons, where we do not have direct access? How do we assign credit or blame for their operation during training?

2. Derivation of the BP Algorithm

- Cost function: $$E(n) = \frac{1}{2}\sum_j e_j^2(n)$$

$$\underbrace{e_j(n)}_{\text{error signal}} = \underbrace{d_j(n)}_{\text{desired response applied to neuron } j} - \underbrace{y_j(n)}_{\text{actual output of neuron } j}$$

Two modes of training

- Sequential mode (Stochastic mode, on-line mode, pattern mode)
- Batch mode

Batch versus Online

- When we add up the weight changes for all the training patterns like this and apply them in one go, it is called ***Batch Training***. (Complete presentation of the entire training set).
- An alternative is to update all the weights after processing each training pattern. This is called ***On-line Training*** (or ***Sequential Training***).

Learning Process

Two cases to be considered:

Case 1: Neuron *j* is an output neuron

Case 2: Neuron *j* is a hidden neuron

η= learning-rate parameter

$$\underbrace{w_{ji}(n+1)}_{\text{update value}} = \underbrace{w_{ji}(n)}_{\text{old value}} + \underbrace{\Delta w_{ji}(n)}_{\text{correction}}$$

$$\Delta w_{ji}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}}$$

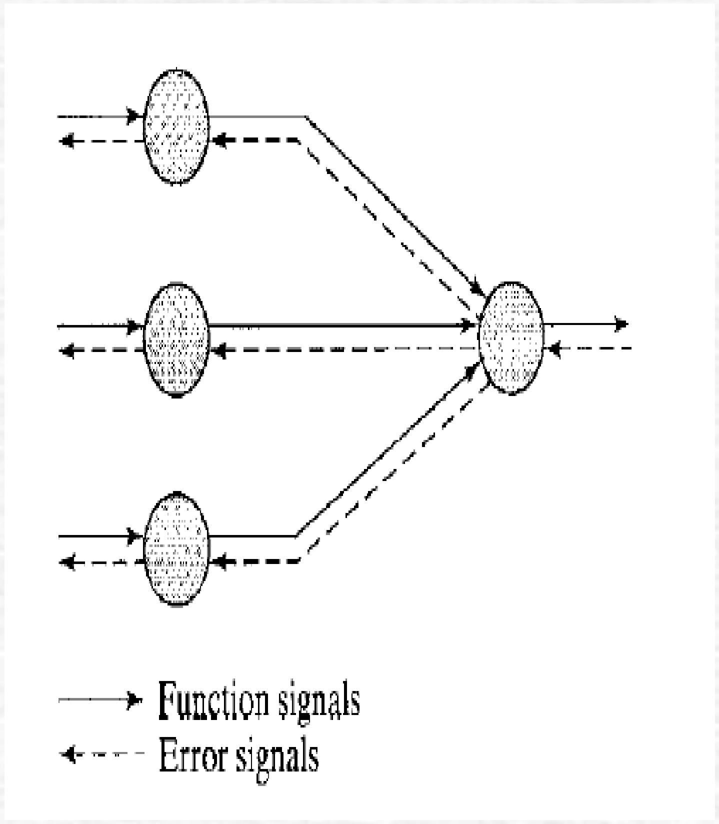
Derivation of the BackPropagation Algorithm

Two passes:

- Forward pass to compute the outputs and the error signals
- Backward pass to “propagate” the error signals to adjust synaptic weights
- Generalization of error correction learning

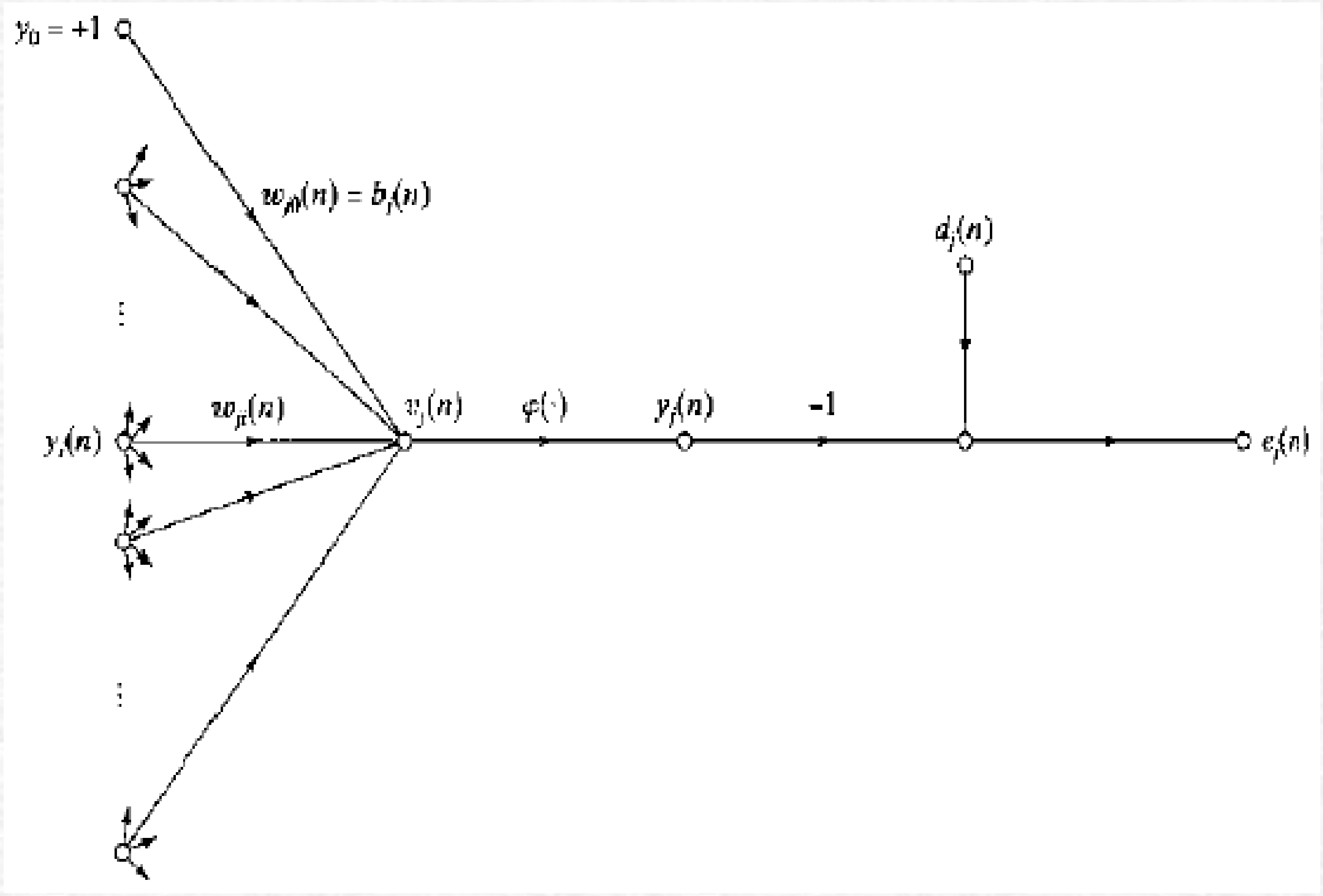
and LMS algorithm

Forward and Backward Passes



- Iteratively use the outputs from each layer as inputs to the next layer to compute the outputs from that layer
- Compute the error signal at the output
- Use the error signal to compute the gradient
- Use the gradient to compute the change in the weights for this layer
- Going backward one step at a time:
- Compute the gradients at successive layers
- Use the gradients to compute the change in the weights for this layer
- This process constitutes a form of credit assignment

Signal Flow



Case 1: Neuron *j* is Output Neuron

$$\underbrace{v_j(n)}_{\text{induced local field}} = \sum_{i=0}^{p} \underbrace{w_{ji}(n)}_{i\text{th synaptic weight of neuron } j} \underbrace{y_i(n)}_{\text{output of neuron } i \text{ in preceding layer}}$$

- $y_j(n)$ = output of neuron j = $\varphi(v_j(n))$
- $\varphi(.)$ = activation function

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \underbrace{\frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}}_{-\delta_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

$\delta_j(n) =$ local gradient of neuron j

$$\frac{\partial E(n)}{\partial e_j(n)} = e_j(n)$$

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1$$

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi'(v_j(n))$$

$$\frac{\partial v_j(n)}{\partial w_j(n)} = y_i(n)$$

Local gradient of neuron j

$$\delta_j(n) = + \; e_j(n)\varphi'(v_j(n))$$

Delta Rule

Delta rule for output neuron j:

$$\Delta w_{ji}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}(n)}$$

$$= \eta \ \delta_j(n) y_i(n)$$

$$\delta_j(n) = + e_j(n)\varphi'(v_j(n))$$

$$= \begin{pmatrix} \textit{learning} - \\ \textit{rate} \\ \textit{parameter} \end{pmatrix} \begin{pmatrix} \textit{local} \\ \textit{gradient} \\ \textit{of} \\ \textit{neuronj} \end{pmatrix} \begin{pmatrix} \textit{output} \\ \textit{of} \\ \textit{neuron} \ \ i \\ \textit{input} \\ \textit{applied} \\ \textit{to} \ \ \textit{neuron} \ \ j \end{pmatrix}$$

Case 2: Neuron *j* is Hidden Neuron

Redefine local gradient of neuron j:

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \dashrightarrow \varphi'(v_j(n))$$

Redefine the cost function:

$$E(n) = \frac{1}{2}\sum_k e_k^2(n) \qquad \text{neuron } k \text{ is output neuron}$$

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_k e_k(n) \frac{\partial e_k(n)}{\partial y_j(n)}$$

$$= \sum_k e_k(n) \frac{\partial}{\partial y_j(n)} (d_k(n) - y_k(n))$$

$$= -\sum_k e_k(n) \frac{\partial y_k(n)}{y_j(n)}$$

$$= -\sum_k e_k(n) \frac{\partial y_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)}$$

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \boxed{\frac{\partial y_j(n)}{\partial v_j(n)}} \dashrightarrow \varphi'(v_j(n))$$

$$= -\sum_k e_k(n) \frac{\partial y_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)}$$

$$v_k(n) = \sum_{j=0}^{p} w_{kj}(n) y_j(n)$$

$$\therefore \frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n)$$

$$v_k(n) = \varphi(v_k(n))$$

$$\therefore \frac{\partial y_k(n)}{\partial v_k(n)} = \varphi'(v_k(n))$$

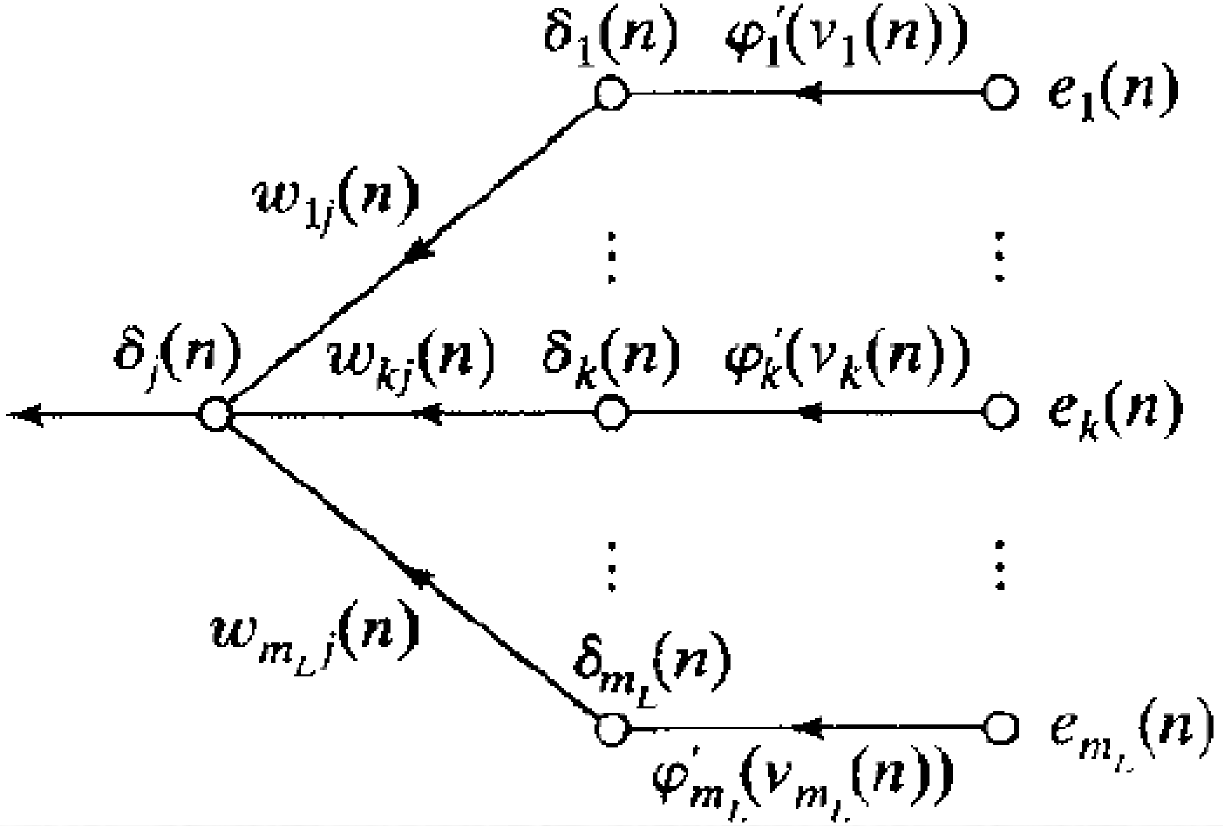
$$\delta_j(n) = + \varphi_j'(v \ (n)) \sum_k e_k(n) \varphi'(v_k(n)) w_{kj}(n)$$

$$+ e_k(n)(\varphi' v_k(n)) = \delta_j(n)$$

$$\delta_j(n) = \varphi'(v_k(n)) \sum_k \delta_k(n) w_{kj}(n)$$

- Back-propagate the local gradients, and reapply delta rule

Error signals are propagated back one layer at a time:



Function Differentiation

$$\varphi_j(v_j(n)) = \frac{1}{1+\exp(-av_j(n))}$$

$$\varphi_j^{'}(v_j(n)) = \frac{a\exp(-av_j(n))}{[1+\exp(-av_j(n))]^2}$$

$$\varphi_j^{'}(v_j(n)) = a\frac{[1+\exp(-av_j(n))-1]}{[1+\exp(-av_j(n))]^2}$$

$$\varphi_j^{'}(v_j(n)) = a\left[\frac{1+\exp(-av_j(n))}{[1+\exp(-av_j(n))]^2} - \frac{1}{[1+\exp(-av_j(n))]^2}\right]$$

$$\varphi_j^{'}(v_j(n)) = a[y(n) - y^2(n)] = ay(n)[1-y(n)]$$

$$\varphi_j(v_j(n)) = a\tanh(bv_j(n))$$

$$\varphi_j^{'}(v_j(n)) = ab\sec h^2(bv_j(n))$$

$$\varphi_j^{'}(v_j(n)) = ab(1-\tanh^2(bv_jn))$$

$$\varphi_j^{'}(v_j(n)) = \frac{b}{a}(a^2 - a^2\tanh^2(bv_jn))$$

$$\varphi_j^{'}(v_j(n)) = \frac{b}{a}(a - a\tanh(bv_j(n))(a + a\tanh(bv_j(n))$$

$$\varphi_j^{'}(v_j(n)) = \frac{b}{a}(a - y_j(n))(a + y_j(n))$$

Summary of BP Algorithm

- 1. Initialization: Select initial weights for synaptic weights from uniformly distributed set of random numbers .
- 2. Presentation of examples: Pick an example at random from training sample and apply it to MLP.
- 3. Forward phase: Propagate input signals through MLP, layer by layer.

$$v_j^{(l)}(n) = \sum_{i=0}^{m_0} w_{ji}^{(l)}(n) y_i^{(l-1)}(n)$$

$$y_j^{(l)} = \varphi_j(v_j(n))$$

$$y_j^{(L)} = o_j(n)$$

$$e_j(n) = d_j(n) - o_j(n)$$

4. Backward Phase

- Calculate error signals, and propagate them through MLP in the backward direction.

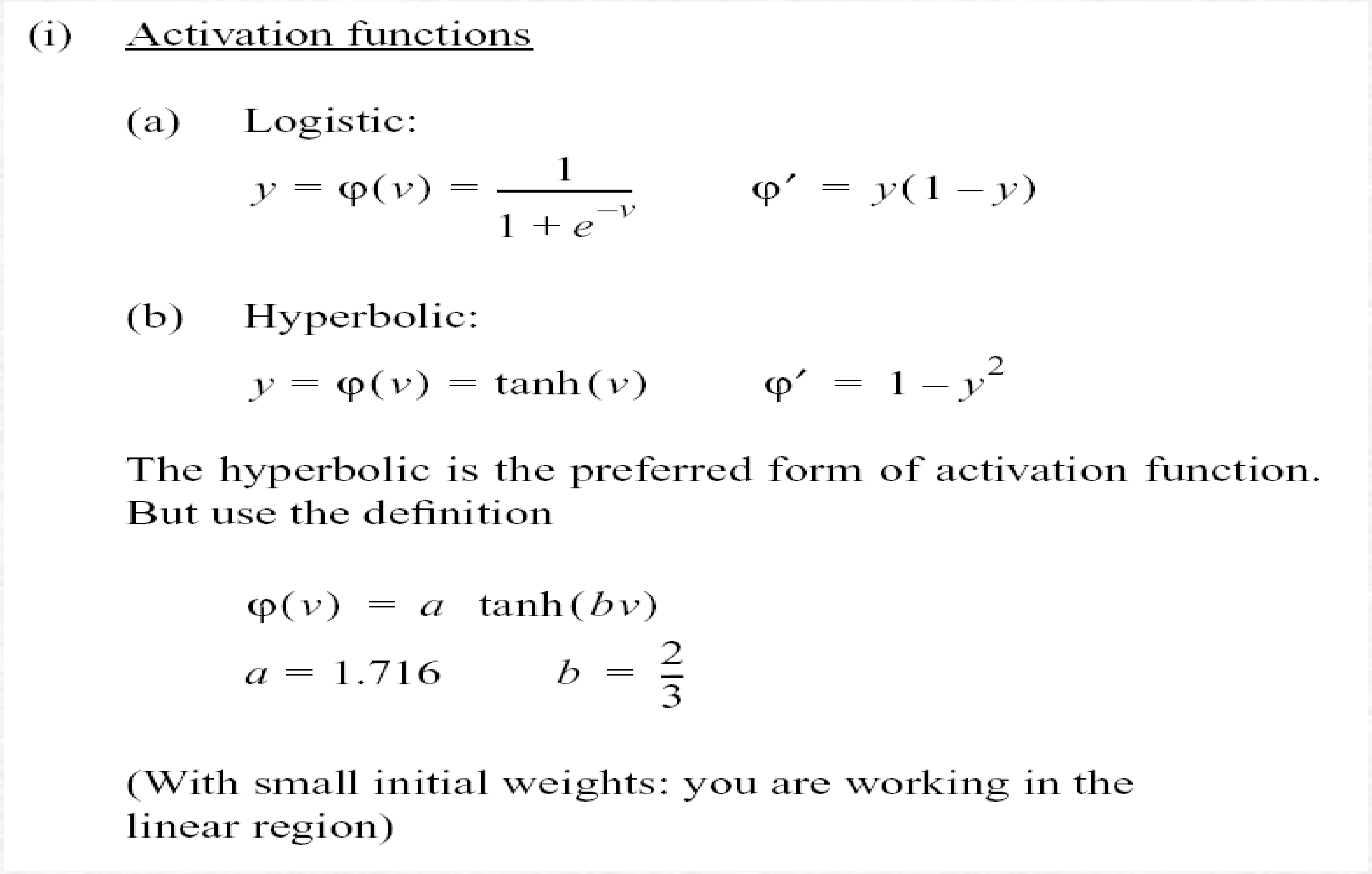
$$\delta_j^{(l)}(n) = \begin{cases} e_j^{(L)}(n)\,\varphi_j'(v_j^{(L)}(n)) & \text{for neuron } j \text{ in output layer } L \\ \varphi_j'(v_j^{(l)}(n)) \sum_k \delta_k^{(l+1)}(n)\, w_{kj}^{(l+1)}(n) & \text{for neuron } j \text{ in hidden layer } l \end{cases}$$

5. Weight Adjustments: Calculate adjustments to synaptic weights in accordance with delta rule.

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha[w_{ji}^{(l)}(n-1)] + \eta\delta_j^{(l)}(n)y_i^{(l-1)}(n)$$

6. Continue until Convergence

Hints for Making BP Work Better



(i) <u>Activation functions</u>

(a) Logistic:

$$y = \varphi(v) = \frac{1}{1 + e^{-v}} \qquad \varphi' = y(1 - y)$$

(b) Hyperbolic:

$$y = \varphi(v) = \tanh(v) \qquad \varphi' = 1 - y^2$$

The hyperbolic is the preferred form of activation function.
But use the definition

$$\varphi(v) = a \ \tanh(bv)$$

$$a = 1.716 \qquad b = \frac{2}{3}$$

(With small initial weights: you are working in the linear region)

Learning-rate parameter

- Local gradient becomes smaller as we move *backward* from the output layer. To equalize the learning speed, learning rate parameter should decrease as we move *forward* from input layer to output layer.
- Larger learning rate speeds up, however the network may become unstable.
- Different learning rate can be used for different layers

Presentation of examples

- One *epoch* consists of the entire training sample.
- As we go from one epoch, shuffle the examples to avoid the network “memorizing”.
- Try to follow the presentation of one example by another with a large contrast between them. Make it “hard” for the network to learn.
- Use normalization as a preprocessing stage so its mean value is close to zero

Size of training set and initialization of weights

W = number of synaptic weights

N = size of training sample

$\in$ = generalization error (error on test)

$$N > \frac{W}{\in}$$

$$\sigma_w = m^{-1/2}$$

m: # of synaptic connections

Use of Momentum

The use of momentum helps the BP algorithm escape local minima. It works best for batch mode of training. Usually not as great with sequential learning. It accelerates the learning in the minimum direction.

Generalized delta rule

$$\Delta W_{ji}(n) = \alpha \Delta_{ji}(n-1) + \eta \delta_j(n) y_i(n)$$

$0 \leq \alpha \leq 1$

$\alpha =$ momentum constant

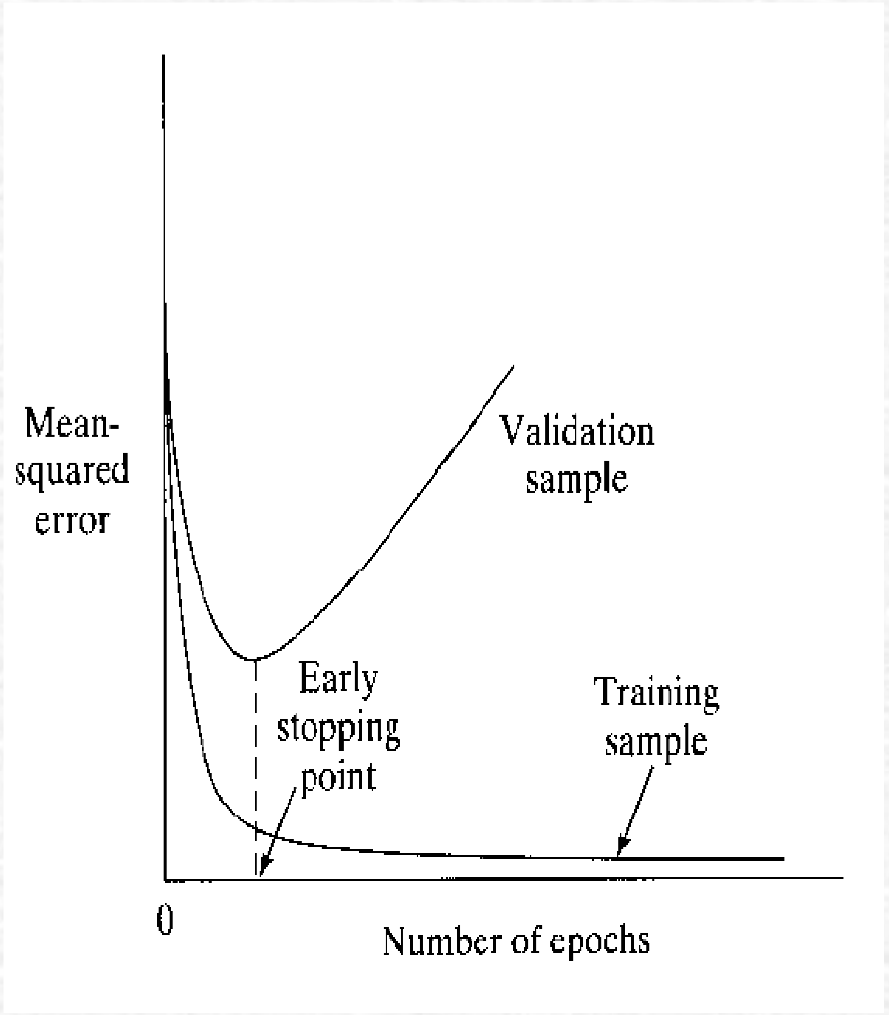
$$\Delta w_{ji}(n) = \eta \sum_{t=0}^{n} \alpha^{n-t} \delta_j(t) y_i(t)$$

Cross-validation

Split the training set into two parts:
- Training set
- Test set

Split the test set into two subparts:
- Modeling (estimation) set
- Validation set

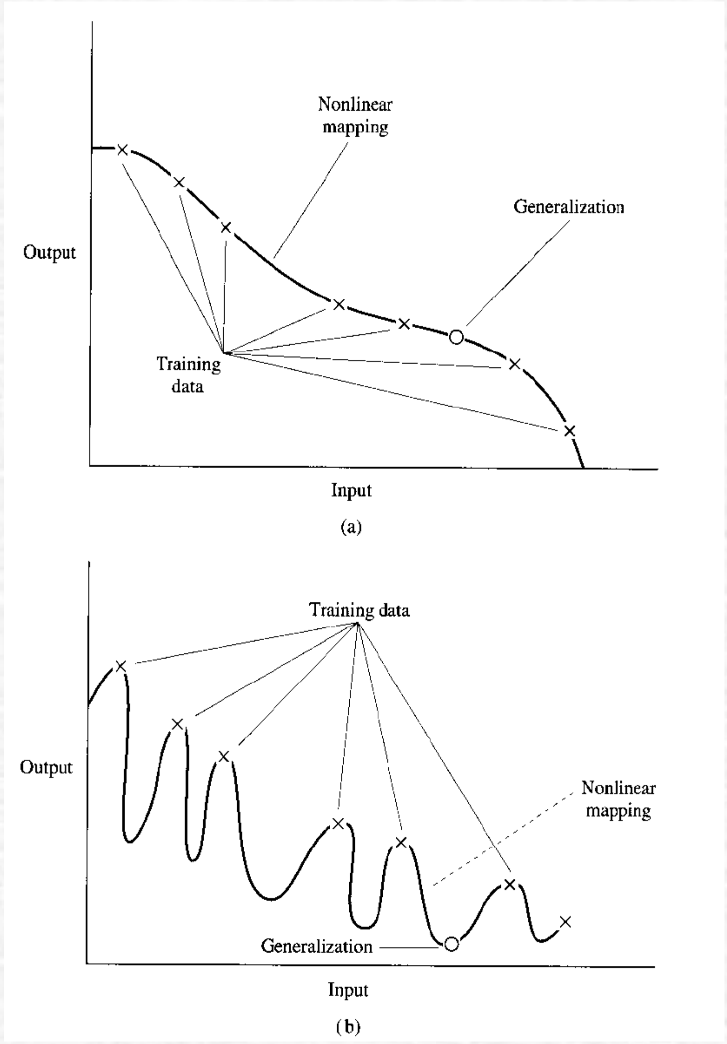
Train the network on the modeling set and after every so many epochs, stop the training and test the network on the validation set.



- To continue the training, the cost function should be on a downward slope when the network is tested on the validation set.

- Stop the training as soon as you see the cost function beginning to increase on the validation set: That is a sign of "overfitting".

- Retrain the network on the whole training set, and then test it on the whole test set.



Summarizing Remarks

- MLP is a *universal approximator*.
- Back-propagation algorithm is a “gradient” technique, simple to compute *locally*.
- Order of complexity is *linear* in the number of neurons.
- It provides a powerful algorithm for computing partial derivatives.
- The computing power of the MLP lies in its hidden neurons that act as *feature detectors*.
- The MLP trained with the back-propagation algorithm (sequential mode) is the *optimum* method for pattern classification.
- For fast training, higher-order information has to be put into the training process: Candidate algorithms:
  - Extended Kalman filter
  - Conjugate gradient algorithm