OVER 40 YEARS OF ACADEMIC WISDOM

GAIN MORE KNOWLEDGE
REACH GREATER HEIGHTS

PRESIDENCY UNIVERSITY

CSE3016 – NEURAL NETWORK & FUZZY LOGIC

Department of Computer Science Engineering

School of Engineering

Limitations of Rosenblatt's Perceptron

- The output of a perceptron can only be a binary number (0 or 1) due to the hard limit transfer function.
- Perceptron can only be used to classify the linearly separable sets of input vectors. If input vectors are non-linear, it is not easy to classify them properly.

Introduction to XOR Problem

- The early enthusiasm over neural networks received a severe blow when Minsky and Papert (1969) demonstrated that perceptrons are unable to implement an elementary function like a 2-input XOR.
- Research community lost interest in the subject and no further development took place for several years.
- Discovery of multilayered perceptron (also referred to as multilayered networks) independently by several researchers (Rumelhart, Ivilliams, McClelland etc.) eventually restored interest in this field.
- The limitation of a single layer perceptron is overcome by multilayer neural nets.
- It is proved that a multilayer feedforward net can be made to learn any continuous function to any extent of desired accuracy.

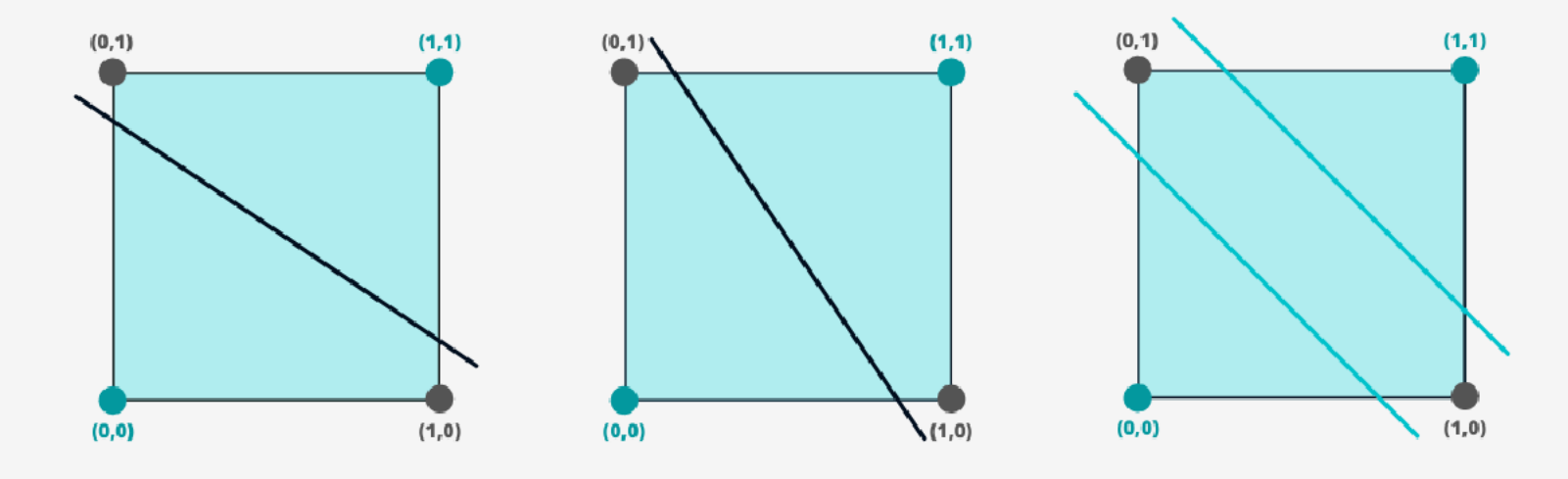
The XOR Problem

- The output of the XOR function has only a true value if the two inputs are different.
- If the two inputs are identical, the XOR function returns a false value.
- The following table shows the inputs and outputs of the XOR function:

Inputs		Outputs
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

The XOR Problem

- The XOR problem cannot be linearly separated by a single boundary line.
- Let's have a look at the following images:



- Two boundary lines are needed to solve the problem.

Multilayer Perceptron

- They are a class of neural networks
  - which consists of a set of source nodes called the input layer,
  - one or more hidden layers of computation nodes
  - an output layer of computation nodes
- Multilayer perceptrons have been applied successfully to solve some difficult and diverse problems by training them in a supervised manner with *the error back-propagation algorithm*.
- This algorithm is based on the error-correction learning rule which may be viewed as a generalization of adaptive filtering algorithm:
  - the least-mean-square (LMS) algorithm

Multilayer Perceptron

- The error back-propagation learning consists of two passes through the different layers of the network:
  - Forward pass
  - Backward pass.
- In the forward pass, an activity pattern (input vector) is applied to the sensory nodes of the network, and its effect propagates through the network layer by layer.
- Finally, a set of outputs is produced as the actual response of the network.
- During the forward pass the synaptic weights of the networks are all fixed.
- During the backward pass, the synaptic weights are all adjusted in accordance with an error-correction rule.

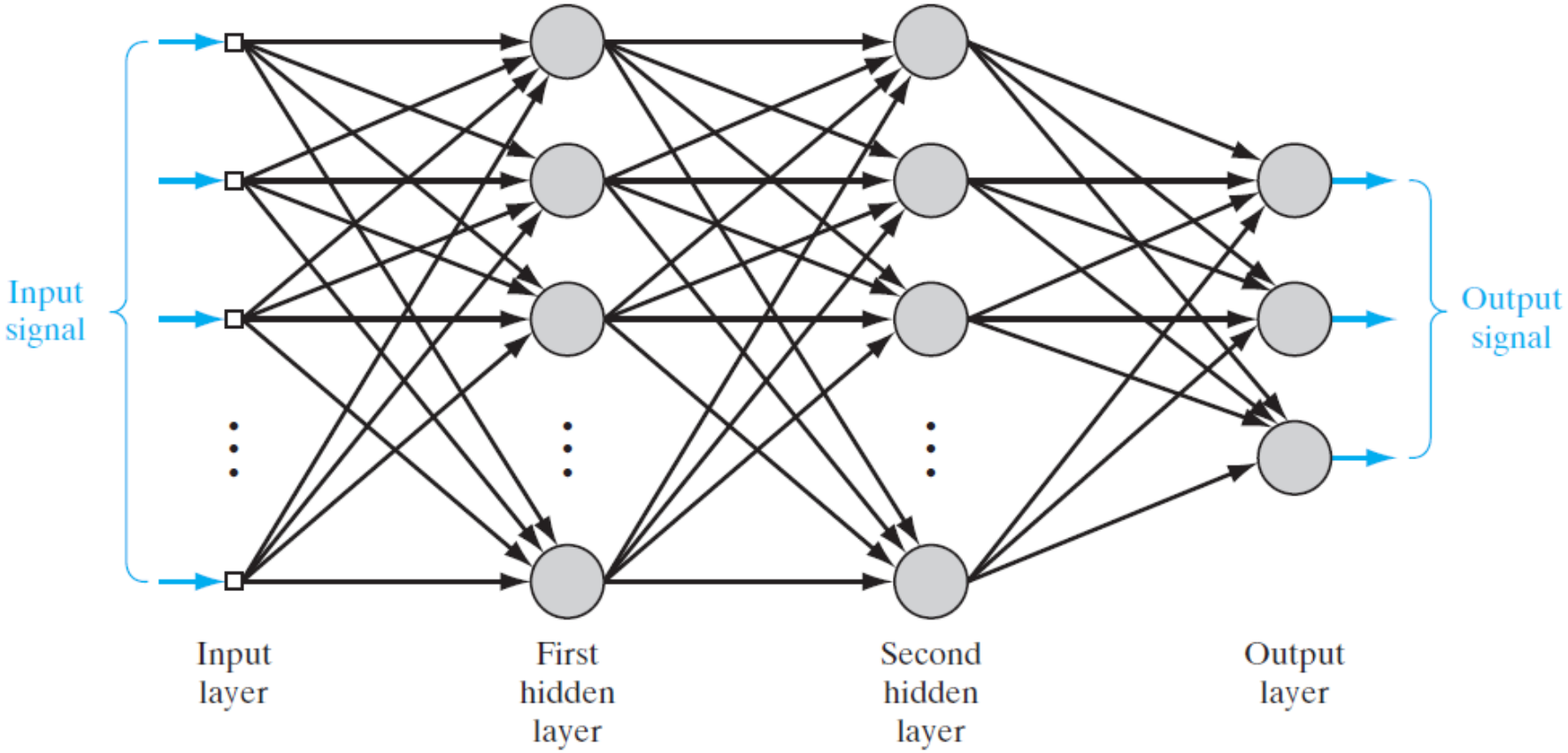
Basic Features of Multilayer Perceptron

- The model of each neuron in the network includes a nonlinear activation function that is *differentiable*.
- The network contains one or more layers that are *hidden* from both the input and output nodes.
- The network exhibits a high degree of *connectivity*, the extent of which is determined by synaptic weights of the network.

Deficiencies of Multilayer Perceptron

- The presence of a distributed form of nonlinearity and the high connectivity of the network make the theoretical analysis of a multilayer perceptron difficult to undertake.
- The use of hidden neurons makes the learning process harder to visualize.
- The learning process is therefore made more difficult because the search has to be conducted in a much larger space of possible functions, and a choice has to be made between alternative representations of the input pattern.

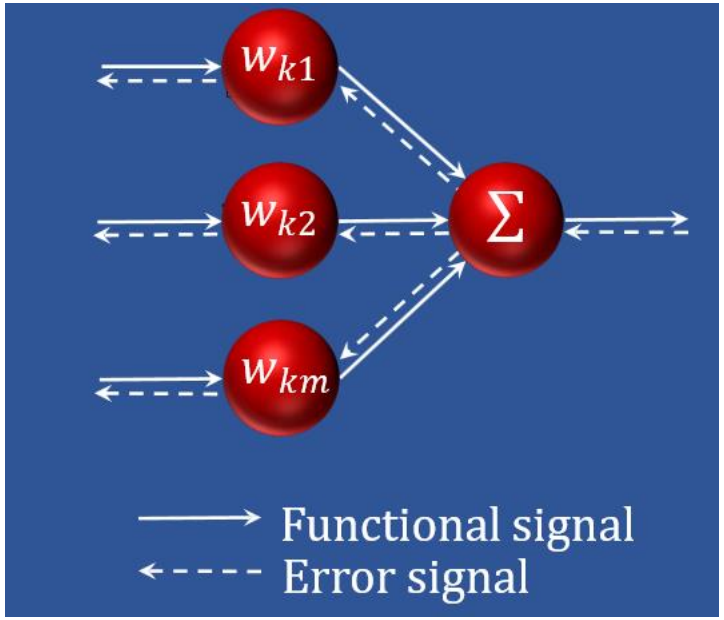
Architecture of a Multilayer Perceptron



Signals in a Multilayer Perceptron

Two kinds of signals are identified in this network:

1. *Function Signals*. A function signal is an input signal (stimulus) that is input to the network, propagates forward (neuron by neuron) through the network, and emerges at the output end of network as an output signal.
2. *Error Signals*. An error signal originates at an output neuron of the network and propagates backward (layer by layer) through the network.



- The output neurons (computational nodes) constitute the output layers of the network.
- Remaining neurons (computational nodes) constitute hidden layers.

Hidden Neurons in a Multilayer Perceptron

- The hidden neurons act as *feature detectors*.
- They play a critical role in the operation of a multilayer perceptron.
- As the learning process progresses across the multilayer perceptron, the hidden neurons begin to gradually "discover" the salient features that characterize the training data.
- They do so by performing a nonlinear transformation on the input data into a new space called the *feature space*.
- In this new space, the classes of interest in a pattern-classification task, for example, may be more easily separated from each other than could be the case in the original input data space.
- Indeed, it is the formation of this feature space through supervised learning that distinguishes the multilayer perceptron from Rosenblatt's perceptron.

Back Propagation Learning Summary of Notation

- Indices i, j, k refer to different neurons in the network
- n = iteration
- $\mathcal{E}(n)$ refers to the instantaneous sum of error squares (error energy) at iteration n. ($\mathcal{E}_{av}$ average of $\mathcal{E}(n)$ over all n)
- $e_j(n)$ error signal at the output of neuron j at iteration n
- $d_j(n)$ desired response for neuron j (used to calculate $e_j(n)$)
- $y_j(n)$ function signal at the output of neuron j
- $w_{ji}(n)$ synaptic weight connecting output of neuron i to neuron j (the correction applied to this weight is shown as $\Delta w_{ji}(n)$
- $v_j(n)$ activation potential (induced local field) of neuron j
- $\varphi_j(\cdot)$activation function of neuron j
- $w_{j0}(n) = b_j(n)$ bias applied to neuron j
- $x_i(n)$, the ith element of input vector
- $o_k(n)$, the kth element of overall output vector
- η learning-rate parameter
- $m_l$size (number of nodes) in layer l, $l \in \{0,1, \dots, L\}$ where L is the depth of network (m_0 size of input layer, m_1 size of first hidden layer, …, m_L (or M) size of output layer)

Back-Propagation Algorithm

- Error signal at the output of neuron j at iteration n (i.e., presentation of the nth training example) is

$$e_j(n) = d_j(n) - y_j(n), \text{ neuron } j \text{ is an output node} \qquad \text{(Eq. 1)}$$

- Instantaneous value of the error energy for neuron j

$$\frac{1}{2} e_j^2(n)$$

- For total instantaneous error energy, sum over all output neurons

$$\mathcal{E}(n) = \frac{1}{2} \sum_{j \in C} e_j^2(n) \qquad \text{(Eq. 2)}$$

where set C denotes all output neurons

- Let N be the number of examples in the training set. Then *average squared error energy* is

$$\mathcal{E}_{av} = \frac{1}{N} \sum_{n \in N} \mathcal{E}(n)$$

Back-Propagation Algorithm

- $\mathcal{E}(n)$ and $\mathcal{E}_{av}$ are functions of all free parameters (synaptic weights and bias levels)
- For a given training set, $\mathcal{E}_{av}$ represents the *cost function* as a measure of learning performance
- The objective during the learning process is to adjust the free parameters to minimize $\mathcal{E}_{av}$
- To do this, we will use an approximation similar to the one we used in LMS algorithm
- Weights will be adjusted after each example in the training set
- The arithmetic average of these individual adjustments over the training set is an *estimate* of the true change that would happen if we modified the weights based on minimizing $\mathcal{E}_{av}$ over the entire set.

Back-Propagation Algorithm

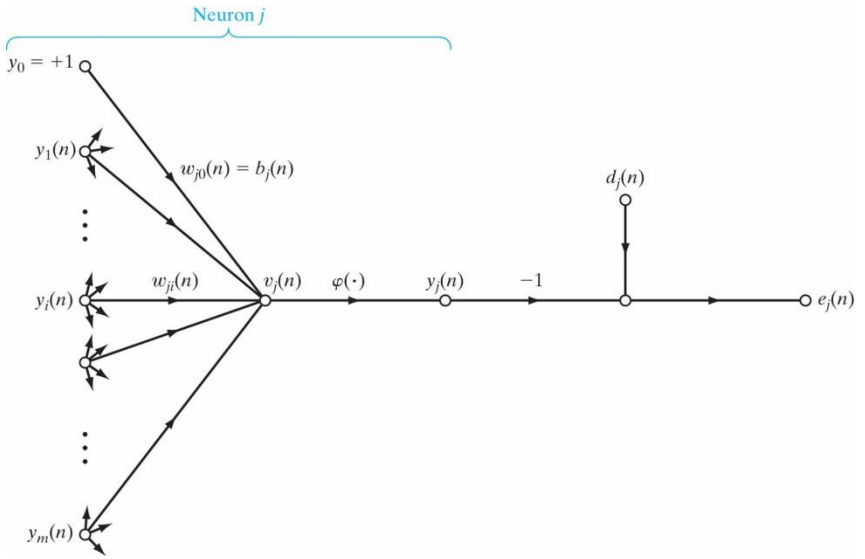
$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \qquad \text{(Eq. 3)}$$

$$y_j(n) = \varphi_j\left(v_j(n)\right) \qquad \text{(Eq. 4)}$$

- Similar to LMS, back-propagation algorithm applies a correction $\Delta w_{ji}(n)$ to weight $w_{ji}(n)$ proportional to partial derivative $\partial \mathcal{E}(n)/\partial w_{ji}(n)$. Using the chain rule of calculus, we can express this with

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

- Differentiate Eq.s 1-4 and put them into the above equation (next page)



Back-Propagation Algorithm

- Differentiate Eq. 2 w.r.t. $e_j(n)$

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

- Differentiate Eq. 1 w.r.t. $y_j(n)$

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

- Differentiate Eq. 4 w.r.t. $v_j(n)$

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'\left(v_j(n)\right)$$

- Differentiate Eq. 3 w.r.t. $w_{ji}(n)$

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

- Put these all into the equation on the previous page

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'\left(v_j(n)\right)y_i(n)$$

- The correction $\Delta w_{ji}(n)$ applied to $w_{ji}(n)$ is

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

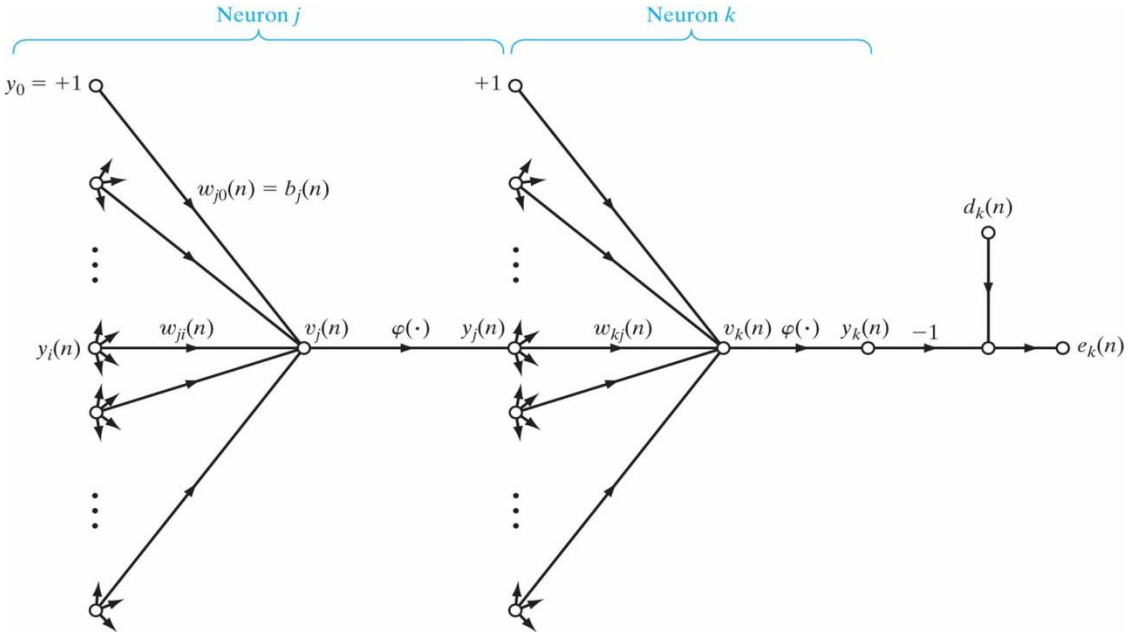
where $\delta_j(n) = e_j(n)\varphi_j'\left(v_j(n)\right)$ is local gradient

Back-Propagation Algorithm

- So, a key factor in calculating the weight adjustments is the error signal at the output of neuron *j*
- There are 2 possibilities (depending on where neuron *j* is)
  - Case 1: neuron *j* is an output neuron (simple because each output neuron is supplied with a desired response)
  - Case 2: neuron *j* is a hidden neuron. Although hidden neurons are not directly visible, they are still responsible for the error made at the network output. The problem here is how to penalize or reward hidden neurons for their share of responsibility. This is actually credit-assignment problem. This problem will be solved by back-propagating the error signals through the network
- For case 1, solution is trivial
- For case 2, next page

Back-Propagation Algorithm

- When neuron *j* is a hidden neuron, we cannot provide a desired response
- An error signal must be determined from the error signals of all neurons to which this neuron is directly connected (this is where development of back-propagation algorithm gets complicated)

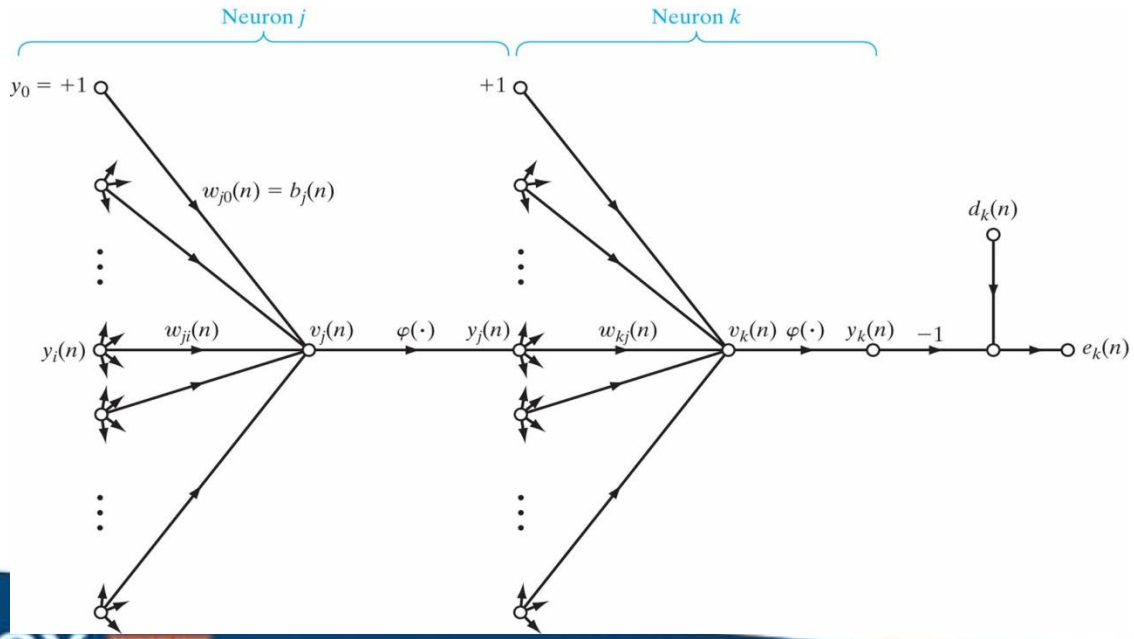


Back-Propagation Algorithm

- We can redefine $\delta_j(n)$ for hidden neuron j as

$$\delta_j(n) = -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}$$

$$= -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \varphi_j'\left(v_j(n)\right) \qquad \text{(Eq. 5)}$$



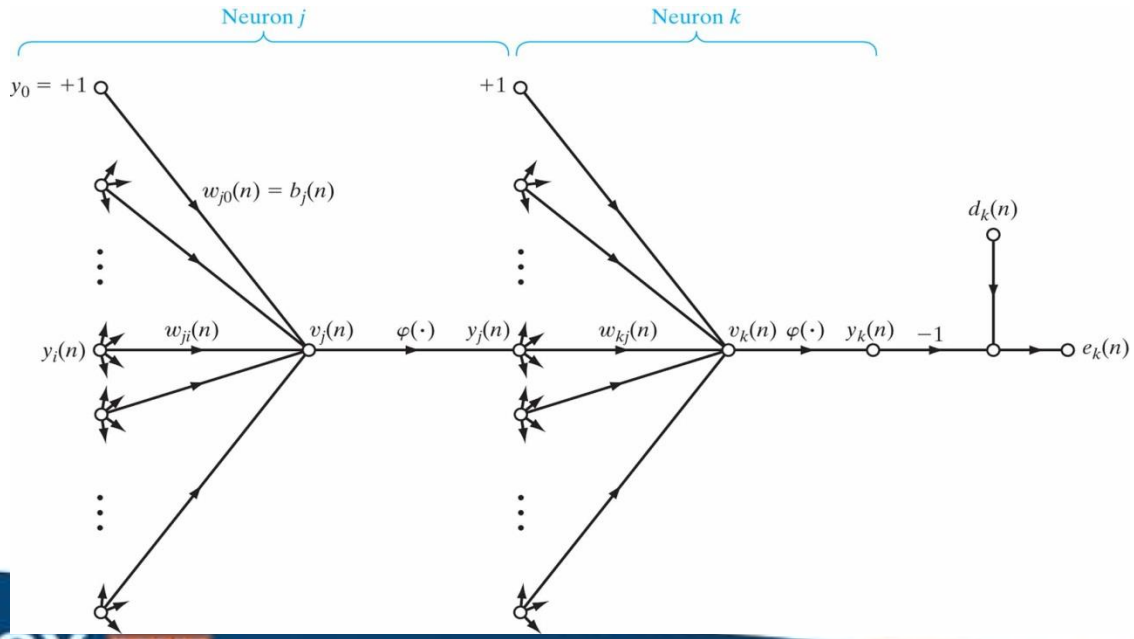
Back-Propagation Algorithm

- To calculate $\frac{\partial \mathcal{E}(n)}{\partial y_j(n)}$, we see that for neuron k

$$\mathcal{E}(n) = \frac{1}{2}\sum_{k \in C} e_k^2(n), \qquad \text{neuron } k \text{ is an output node}$$

- Differentiate this w.r.t. $y_j(n)$

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



Back-Propagation Algorithm

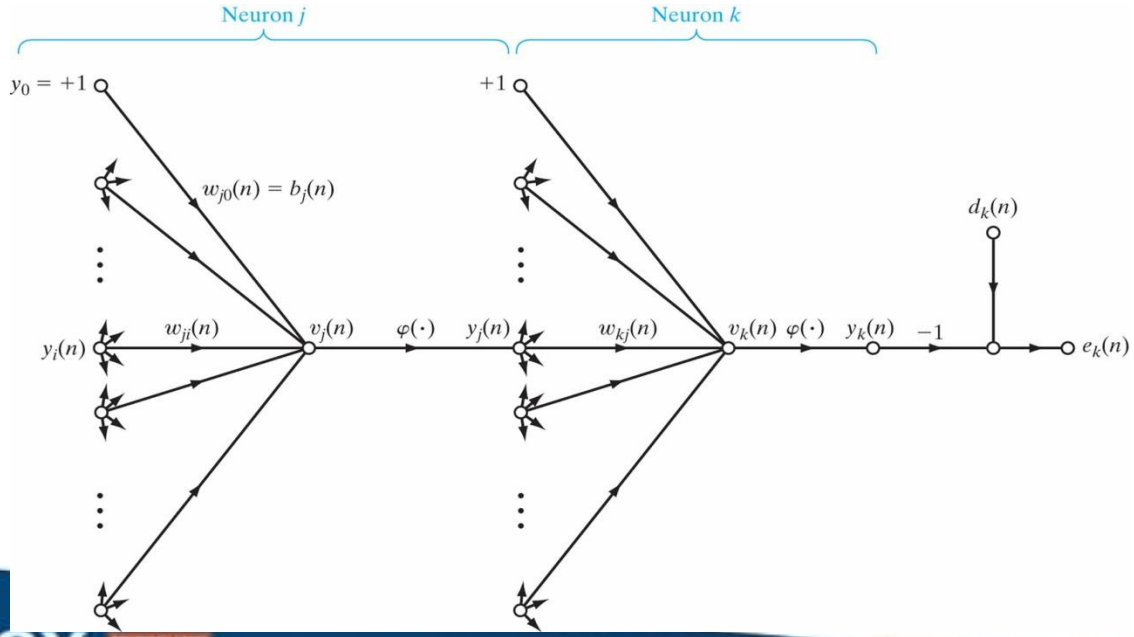
- Using the chain rule

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

- We know that $e_k(n) = d_k(n) - y_k(n) = d_k(n) - \varphi_k\big(v_k(n)\big)$, so

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k'\big(v_k(n)\big)$$

- Also, $v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n)$ and differentiating this w.r.t. $y_j(n)$ gives $\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n)$



Back-Propagation Algorithm

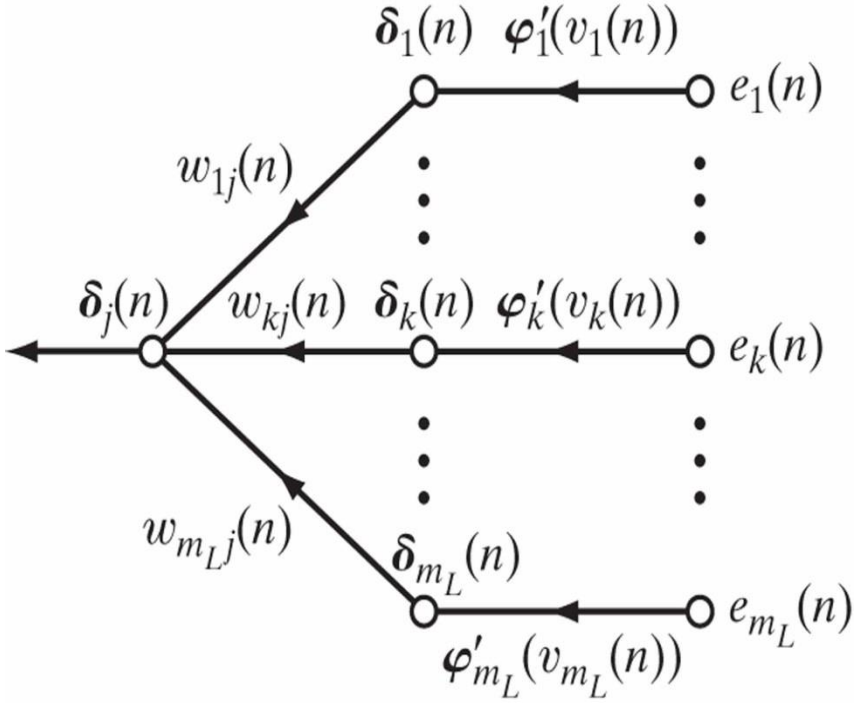
- Putting those on the previous page together

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = -\sum_k e_k \varphi_k'\big(v_k(n)\big) w_{kj}(n)$$

$$= -\sum_k \delta_k(n) w_{kj}(n)$$

- And finally, put this into Eq. 5 to get the back propagation formula

$$\delta_j(n) = \varphi_j'\Big(v_j(n)\Big) \sum_k \delta_k(n) w_{kj}(n)$$



Back-Propagation Algorithm Summary of Derivation

- The correction $\Delta w_{ji}(n)$ applied to synaptic weight connecting neuron i to neuron j is defined by the delta rule

$$\begin{pmatrix} Weight \\ correction \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} learning\ rate \\ parameter \\ \eta \end{pmatrix} \begin{pmatrix} local \\ gradient \\ \delta_j(n) \end{pmatrix} \begin{pmatrix} input\ signal \\ of\ neuron\ j \\ y_i(n) \end{pmatrix}$$

- Local gradient $\delta_j(n)$ depends on whether neuron j is a hidden neuron or output neuron
  - If it is an output neuron $\delta_j(n) = e_j(n)\varphi_j'\left(v_j(n)\right)$
  - If, otherwise, it is a hidden neuron $\delta_j(n) = \varphi_j'\left(v_j(n)\right)\sum_k \delta_k(n)w_{kj}(n)$

Thank You!