Multilayer Perceptrons

Introduction

- Multilayer feedforward networks
  - Input layer (a set or source nodes)
  - One or more hidden layers (computation nodes)
  - Output layer (computation nodes)
- Signal propagates through network in a forward direction layer by layer
- Such networks are called multilayer perceptrons (MLPs)
- They have been successfully applied to solve diverse difficult problems by training them in a supervised manner using *error back-propagation algorithm*

Chapter Organization

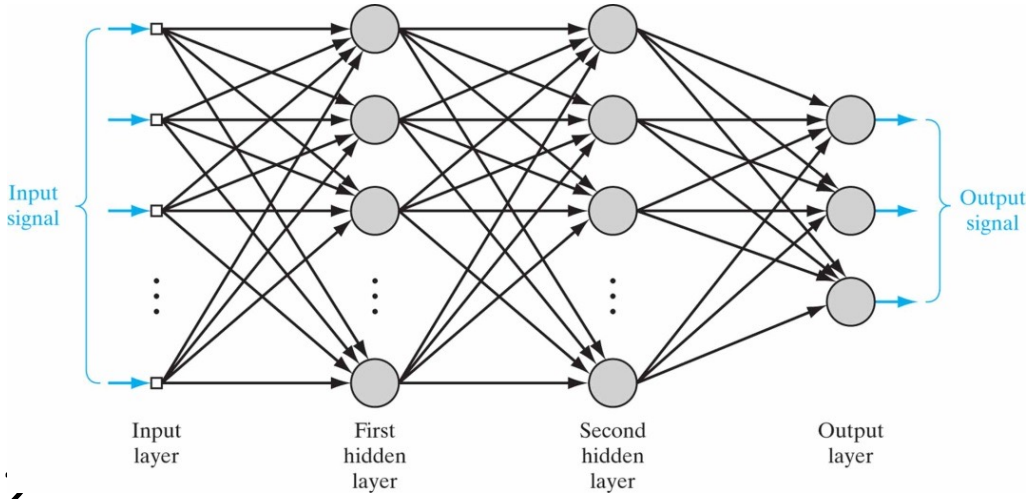
- Back Propagation Learning
- MLPs as Pattern Recognizers
- Error Surface
- Performance
- Back Propagation Learning (revisited)
- Supervised Learning as an optimization problem
- Convolutional Networks

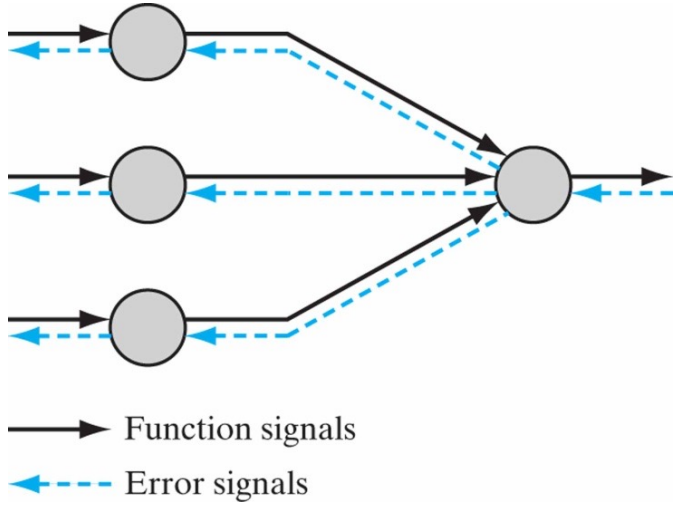
Chapter Organization

- **Back Propagation Learning**
- MLPs as Pattern Recognizers
- Error Surface
- Performance
- Back Propagation Learning (revisited)
- Supervised Learning as an optimization problem
- Convolutional Networks

Back Propagation Learning
Some Preliminaries

- 2 kinds of signals
  - Function signals
  - Error signals
- Each neuron is involved in 2 calculations
  - Computation of function signal
  - Computation of an estimate of the gradient vector (needed for backward pass)





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

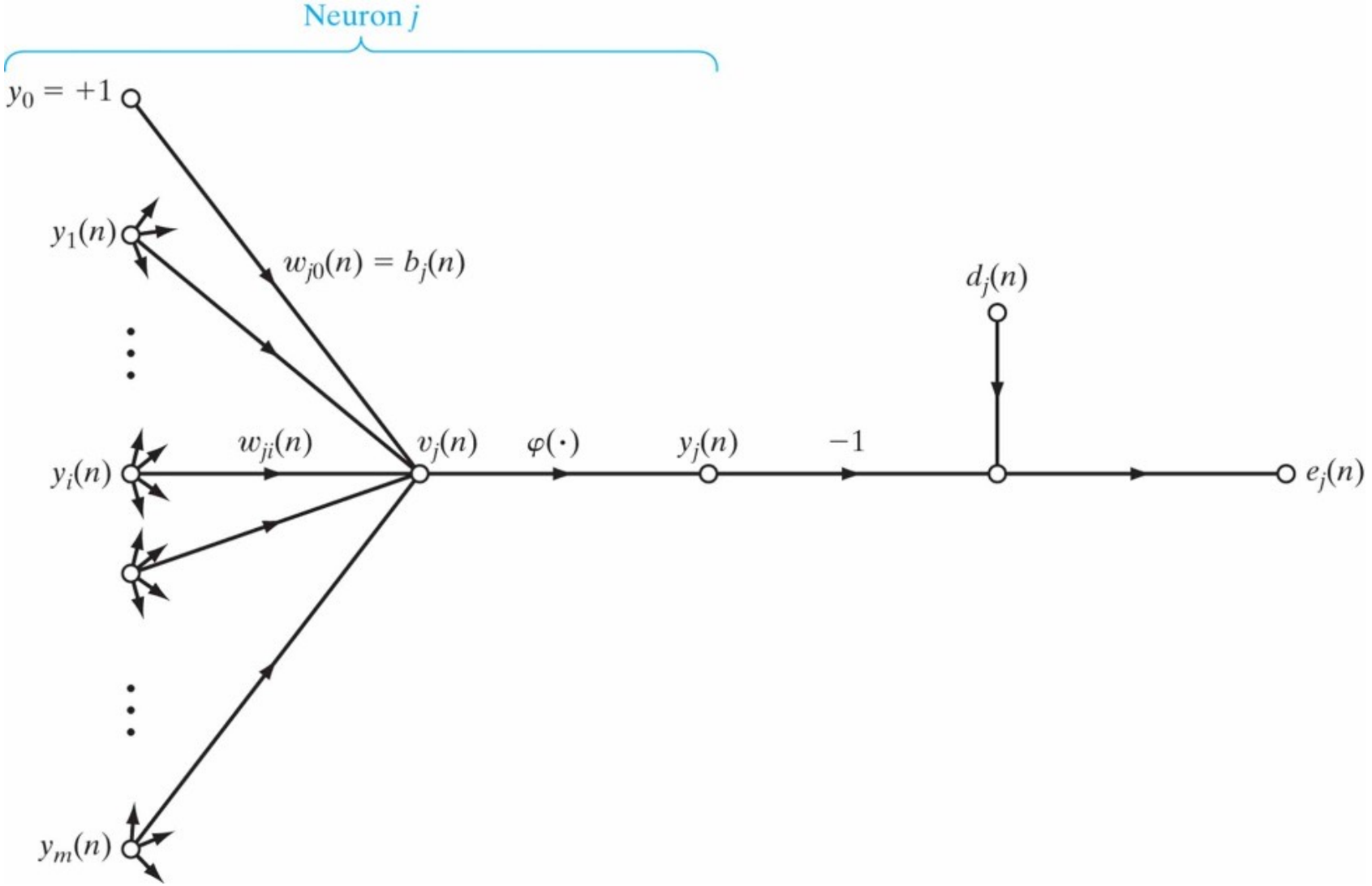
- $\mathcal{E}(n)$ and $\mathcal{E}_{av}$ are functions of all free parameters (synaptic weights and bias levels)

- 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 $\mathcal{E}_{av}$
- 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



Back-Propagation Algorithm

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

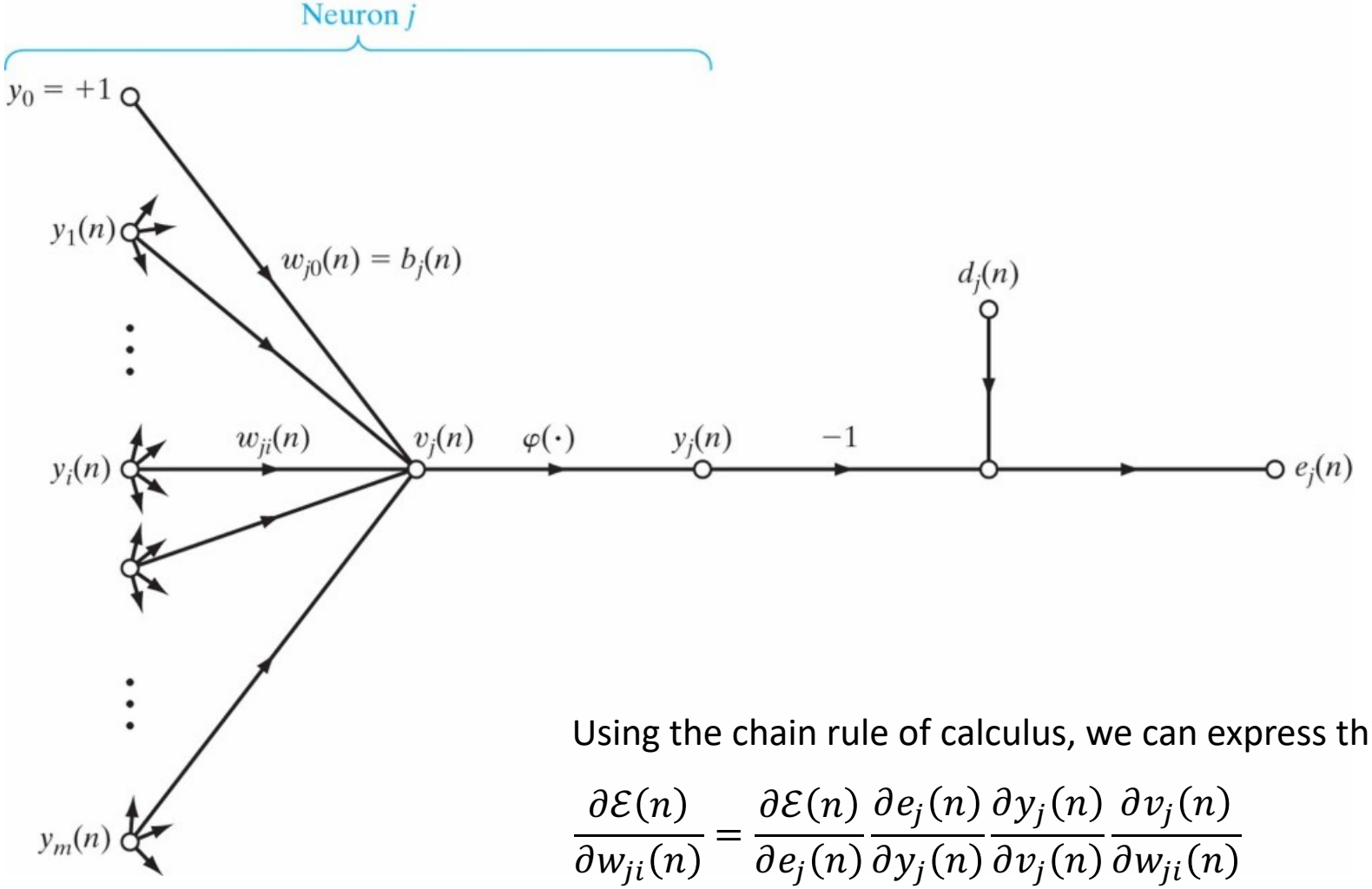
$$y_j(n) = \varphi_j\left(v_j(n)\right) \qquad \text{(Eq. 4)}$$

- 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



Using the chain rule of calculus, we can express this with

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

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. $w_{ji}(n)$

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

$e_j(n) = d_j(n) - y_j(n)$, neuron j is an output node (Eq. 1)

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

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

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

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

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

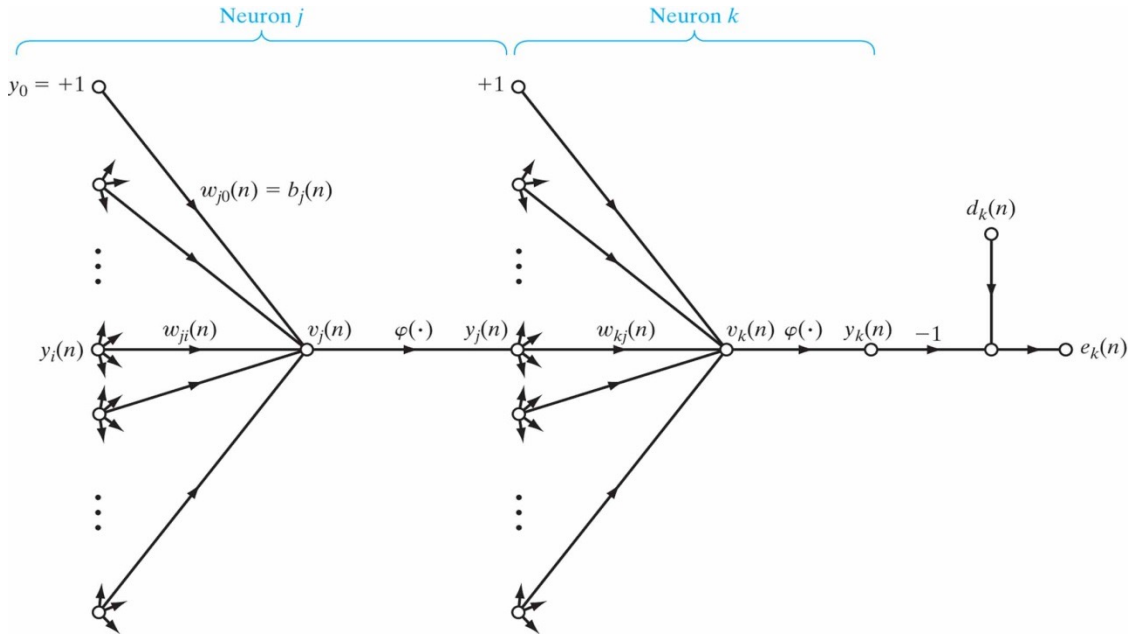
where $\delta_j(n) = e_j(n)\varphi_j'\left(v_j(n)\right)$ is local gradient

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 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)

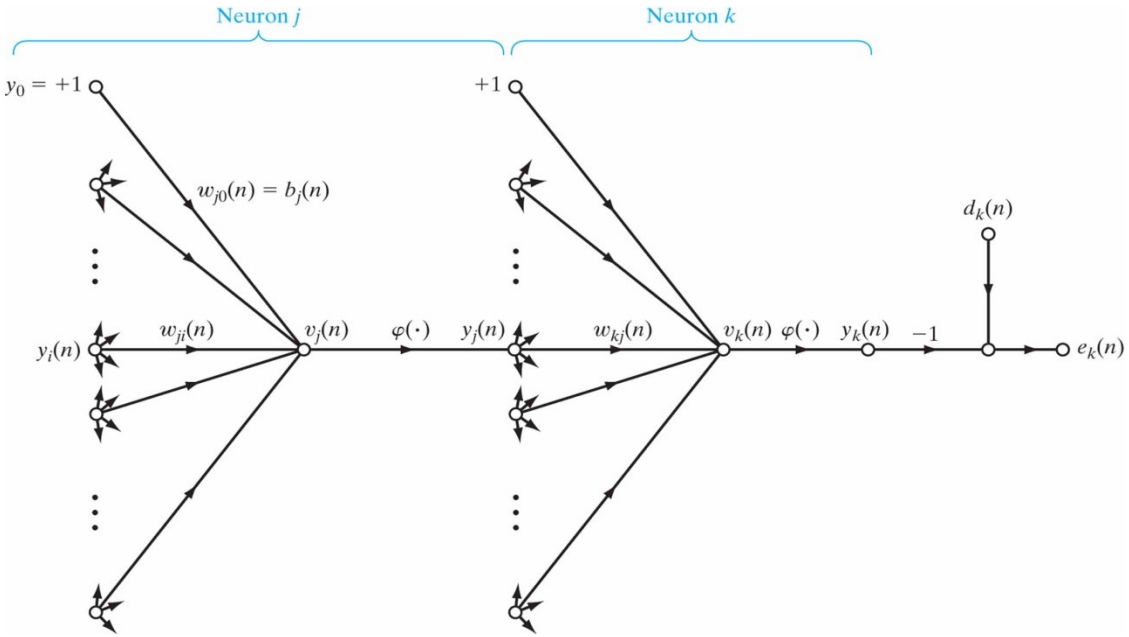


Back-Propagation Algorithm

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

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

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



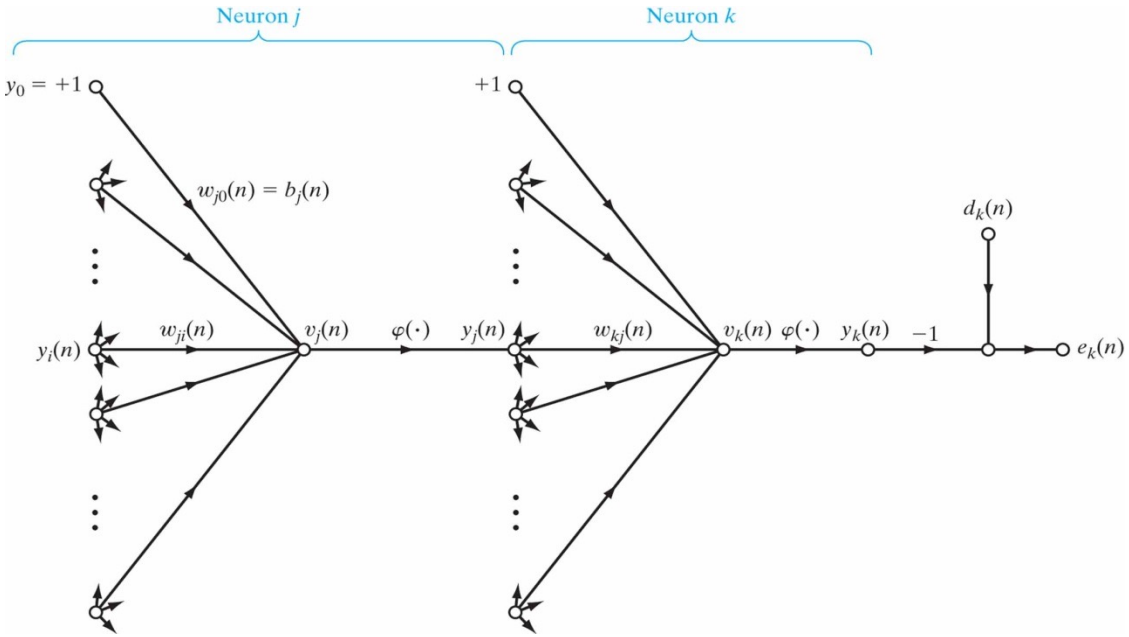
Back-Propagation Algorithm

- 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

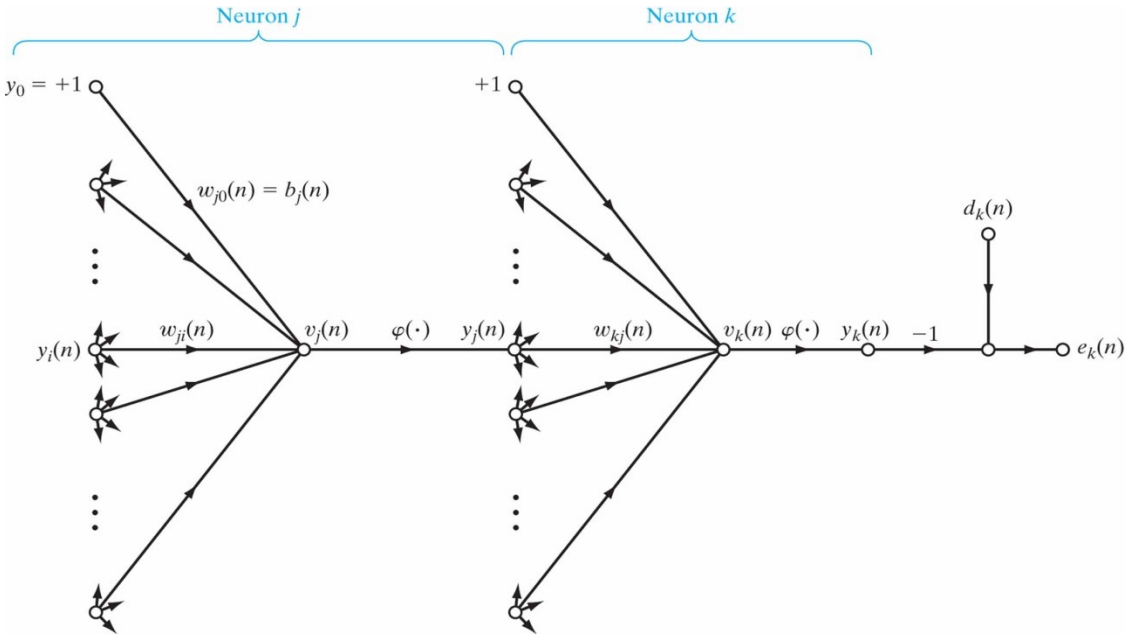
- Using the chain rule

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

- 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

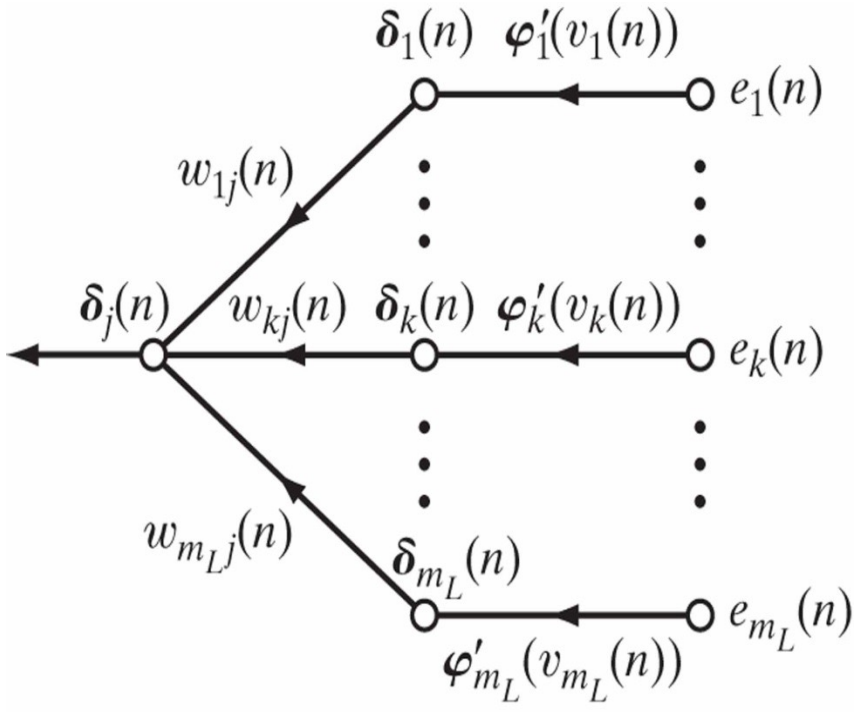
- Putting those on the previous page together

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

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

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

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



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

Back-Propagation Algorithm
Two Passes of Computation

- Back-propagation algorithm works with 2 passes of computation:
  - The first (forward pass):
    - weights remain unchanged and the function signals are computed on a neuron-by-neuron basis.
    - Begins by first hidden layer taking the system inputs and ends at the output layer by computing the error signal for each output neuron
  - The second (backward pass):
    - starts at the output layer by passing error signals leftward through the network

Back-Propagation Algorithm
Activation Function

- To compute δ we need to know the derivative of activation function $\varphi(\cdot)$
- So, function $\varphi(\cdot)$ must be differentiable
- A common continuously differentiable nonlinear activation function is *sigmoidal nonlinearity* and there are 2 versions for it
  - *Logistic function*
  - *Hyperbolic tangent function*

Back-Propagation Algorithm
Activation Function

- *Logistic function*

$$\varphi_j\left(v_j(n)\right) = \frac{1}{1+\exp\left(-a v_j(n)\right)} \qquad a > 0 \text{ and } -\infty < v_j(n) < \infty$$

  - Output is always in range $[0,1]$
  - Differentiating this w.r.t. $v_j(n)$, we get

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

  - With $y_j(n) = \varphi_j\left(v_j(n)\right)$, this can be rewritten as

$$\varphi_j'\left(v_j(n)\right) = a\, y_j(n)\left[1 - y_j(n)\right]$$

  - We can now put this back into the local gradient calculations we wrote before

Back-Propagation Algorithm
Activation Function

- *Hyperbolic tangent function*

$$\varphi_j\left(v_j(n)\right) = a\tanh\left(b\,v_j(n)\right) \qquad (a,b) > 0$$

  - It is actually the logistic function rescaled and biased
  - Differentiating this w.r.t. $v_j(n)$, we get

$$\varphi_j'\left(v_j(n)\right) = a\,b\,\mathrm{sech}^2\left(b\,v_j(n)\right)$$

$$= a\,b\,\left(1-\tanh^2\left(b\,v_j(n)\right)\right)$$

$$= \frac{b}{a}\left[a-y_j(n)\right]\left[a+y_j(n)\right]$$

  - Again, we can now put this back into the local gradient calculations we wrote before

Back-Propagation Algorithm
Rate of Learning

- We have the same tradeoff about the value of the learning rate η that we had before
  - As η becomes smaller and smaller, weights will be changed with smaller values at each iteration and the trajectory in weight space will be smoother, but we will have a slower rate of learning
  - On the other hand, if η is taken too large to make learning faster, there will be large changes to the weights at each iteration and this may cause the network to become unstable (i.e., oscillatory)
- A method to obtain advantages of both of the above is to modify the delta rule by including a momentum term

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

- This is called the *generalized delta rule*
- Another possible modification is that instead of a standard η for all of the network, it can be made connection dependent (η_{ji})

Back-Propagation Algorithm
Sequential/Batch Modes of Training

- In practice, learning happens by presenting a set of training samples to the multilayer perceptron many times
- One complete presentation of the entire set is called an *epoch*
- Learning continues in an epoch-by-epoch basis until weights stabilize and average squared error for the entire set converges to some minimum value
- It is good practice to randomize the presentation order of training examples between epochs
- There are 2 ways back-propagation learning
  - Sequential mode (online, pattern, or stochastic mode): weights are updated after each sample (we assumed this mode when deriving and explaining the algorithm)
  - Batch mode: weight updating is done after all samples are presented (changes to equations explained on the next slide)

Back-Propagation Algorithm
Batch Mode of Training

- When weights are to be changed after all training samples, certain details become different and need explanation
- First of all, for an epoch, the cost function is the average squared error

$$\mathcal{E}_{av} = \frac{1}{2N}\sum_{n=1}^{N}\sum_{j\in C} e_j^2(n)$$

where C represents the set of output neurons

- The adjustment applied to w_{ji} is

$$\Delta w_{ji} = -\eta\frac{\partial \mathcal{E}_{av}}{\partial w_{ji}} = -\frac{\eta}{N}\sum_{n=1}^{N} e_j(n)\frac{\partial e_j(n)}{\partial w_{ji}}$$

Back-Propagation Algorithm
Comparing Sequential/Batch Modes

- Sequential mode requires less local storage for each synaptic connection
- Moreover, if training samples are presented in a random order, our search in weight space becomes stochastic in nature (less likely to be trapped in a local minimum)

- On the other hand, this stochastic nature of sequential mode makes it difficult to establish theoretical conditions for the convergence of the algorithm
- Use of batch mode provides an accurate estimate of the gradient vector therefore, convergence to a local minimum is guaranteed under simple conditions
- Batch mode is easier to parallelize

- When data set is large and redundant (contains several copies of some examples), unlike batch mode, sequential mode takes advantage of this situation

- In summary, despite having several disadvantages, sequential mode is popular for two main reasons
  - It is simple to implement
  - It provides effective solutions to large and difficult problems

Back-Propagation Algorithm
Stopping Criteria

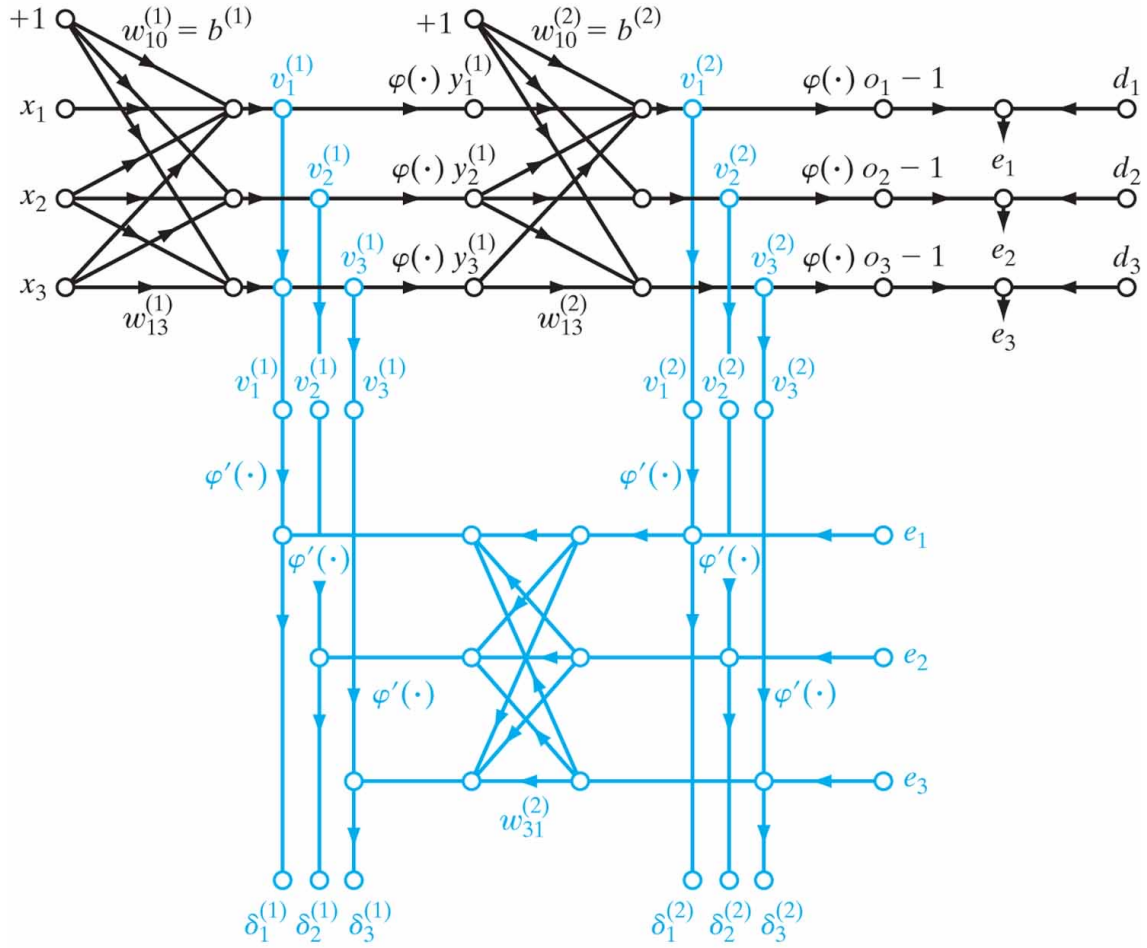
- Back-propagation algorithm cannot be shown to converge
- No well-defined criteria for stopping its operation
- But, we have some reasonable criteria
- Think in terms of the properties of a local or global minimum of the error surface
- Let $\boldsymbol{w}^*$ denote a local or global minimum
- Gradient vector $\boldsymbol{g}(\boldsymbol{w})$ of the error surface w.r.t. $\boldsymbol{w}$ must be 0 at $\boldsymbol{w} = \boldsymbol{w}^*$. So, a sensible criterion is
  - *The back propagation algorithm is considered to have converged when the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold (Kramer and Sangiovanni-Vincentelli, 1989).*
- The drawback of this criterion is that, for successful trials, learning times may be long and it requires computing $\boldsymbol{g}(\boldsymbol{w})$

Back-Propagation Algorithm
Stopping Criteria

- Another property of a minimum is the fact that the cost function or error measure $\mathcal{E}_{av}(w)$ is stationary at the point $\boldsymbol{w} = \boldsymbol{w}^*$. So, a different criterion for convergence can be:
  - The back-propagation algorithm is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small.
- Typically, the rate of change is considered small enough if it is in the range 0.1% to 1% per epoch (Sometimes a value as small as 0.01% is used)
- Unfortunately, this criterion may result in a premature termination
- Another useful and theoretically supporter criterion for convergence is to test the network's generalization performance after each iteration and stop when this performance is adequate or when it has peaked (Details later in this chapter, section 4.14)

Back-Propagation Algorithm Summary

Figure 4.7 Signal-flow graphical summary of back-propagation learning. Top part of the graph: forward pass. Bottom part of the graph: backward pass.

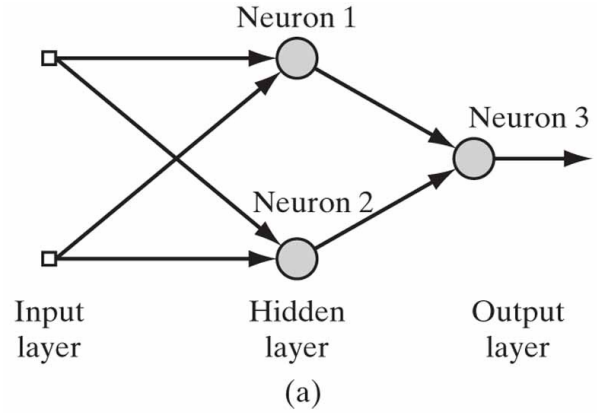


Back-Propagation Algorithm Summary

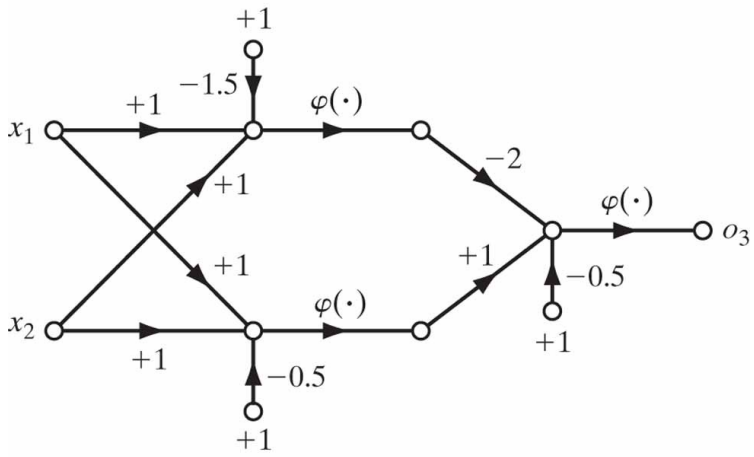
- Sequential mode is generally preferred and for this mode, algorithm cycles through the samples as follows:
  1. Initialization: (With no prior information) pick weights and thresholds from a uniform distribution with zero mean
  2. Presentations of training samples: present the network with an epoch of training samples. For each sample perform forward and backward computations described in 3 and 4 below
  3. Forward computation: using one sample's input values, compute activation potentials and function results going forward layer by layer. Finally, compute the error signal at the outputs.
  4. Backward computation: Compute the local gradients and adjust the weights with the generalized delta rule in a backwards fashion.
  5. Iteration: Iterate 3 and 4 with new epochs until the stopping criterion is met.
- Notes: order of training samples are randomized between epochs. Momentum and learning-rate parameter are typically adjusted (usually decreased) as the number of iterations increases.

XOR Problem

- A single-layer perceptron has no hidden layers and it can only classify linearly separable patterns.
- Non-linearly separable patterns occur commonly. E.g. Exclusive OR problem
- It can be solved by using a single hidden layer (a multilayer perceptron)

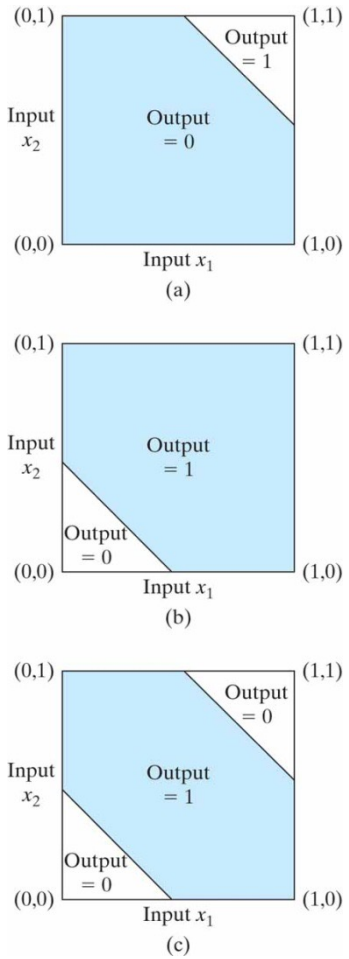


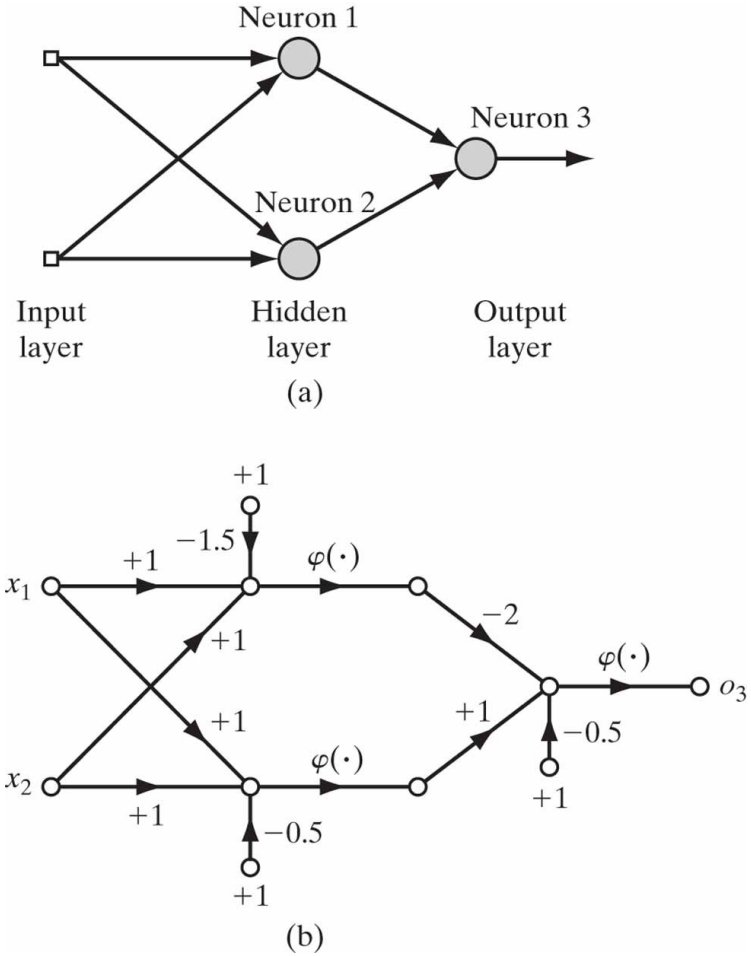
(a)



(b)

XOR Problem


(0,1)
(1,1)
Output = 1
Input x_2
Output = 0
(0,0)
(1,0)
Input x_1
(a)
(0,1)
(1,1)
Input x_2
Output = 1
Output = 0
(0,0)
(1,0)
Input x_1
(b)
(0,1)
(1,1)
Output = 0
Input x_2
Output = 1
Output = 0
(0,0)
(1,0)
Input x_1
(c)


Neuron 1
Neuron 3
Neuron 2
Input layer
Hidden layer
Output layer
(a)
+1
−1.5
+1
$\varphi(\cdot)$
x_1
−2
+1
$\varphi(\cdot)$
o_3
+1
+1
−0.5
$\varphi(\cdot)$
x_2
+1
+1
−0.5
+1
(b)

Heuristics for Back-Propagation Alg.

1. Sequential vs. batch update: sequential mode is faster (especially true when training set is large and highly redundant)
2. Maximizing information content: Every training example presented to back-propagation alg. should ideally be chosen so that its information content is the largest possible for the task at hand.
   - Two ways to achieve this are:
     - Use an example that results in the largest training error
     - Use an example that is radically different from all those previously used
   - With sequential mode, a common simple method is to randomize (i.e., shuffle) the order of examples between epochs
   - A more refined technique (*emphasizing scheme*) is to present the network with more difficult examples. Whether an example is easy or difficult can be determined by the error it produces. 2 problems with this:
     - Example distribution within an epoch is distorted
     - An outlier, or mislabeled example is bad for the performance (learning these degrades the generalization ability of the network)

Heuristics for Back

3. <u>Activation function:</u> A multilayer perceptron trained with back-propagation alg. may learn faster when the sigmoid activation function is antisymmetric than when it is nonsymmetric.
   - $\varphi(v)$ is antisymmetric if $\varphi(-v) = -\varphi(v)$

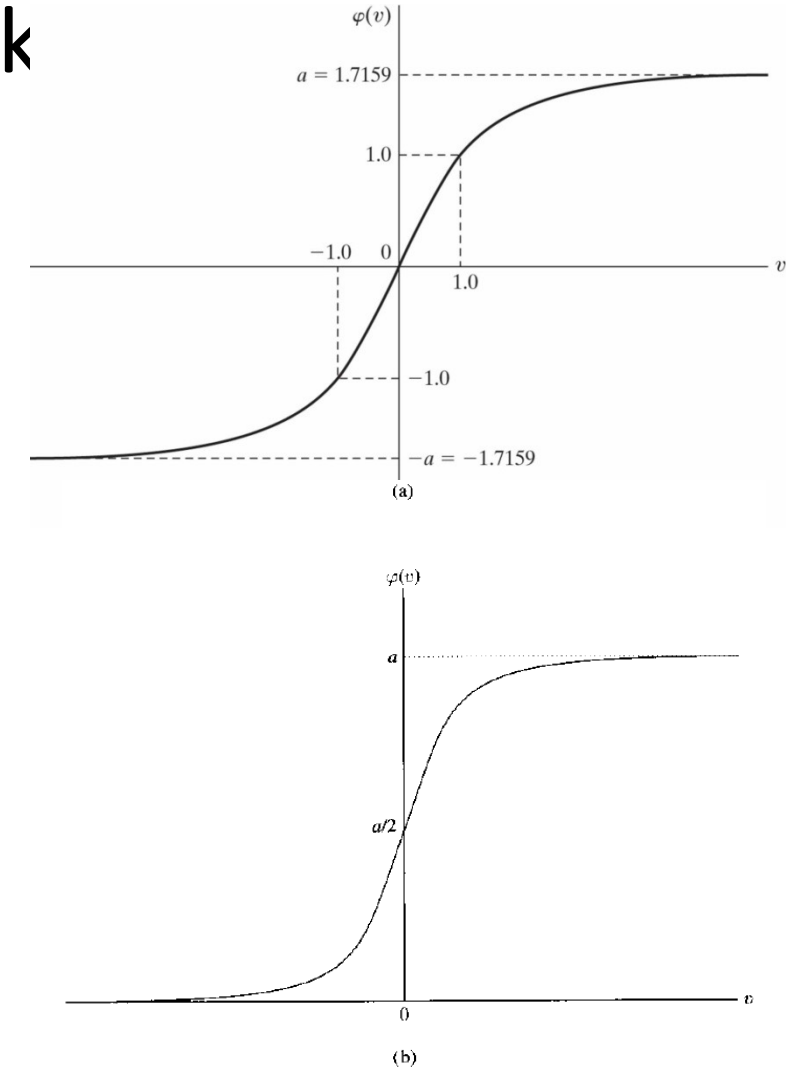
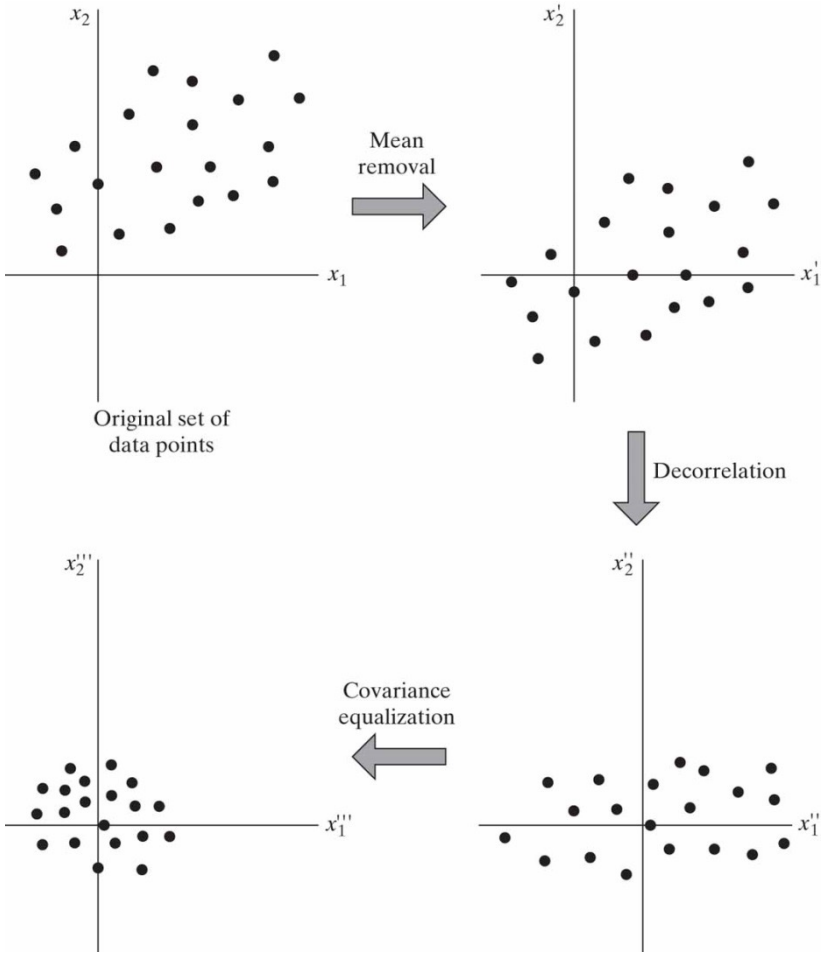


FIGURE 4.10 Antisymmetric activation function. (b) Nonsymmetric activation function.

Heuristics for Back-Propagation Alg.

4. Target values: It is important that the target values (desired responses) are in the range of the sigmoid function.
   - These should be offset by some amount away from the limiting value.
   - Otherwise, back-prop. tends to drive the free parameters to infinity
5. Normalizing the inputs: Each input should be preprocessed so that its mean value (averaged over the entire training set) is close to zero. Also, 2 other measures accelerate back-propagation alg.
   - Input variables should be uncorrelated (can be done using Principal Component Analysis, Chapter 8)
   - Decorrelated input variables should be scaled so that their covariances are approximately equal (This ensures that different synaptic weights learn approximately at the same speed)

Heuristics for Back-Propagation Alg.



Heuristics for Back-Propagation Alg.

6. Initialization: A good choice of initial synaptic weights helps, but what is a good choice?
   - Using both large or small values should be avoided
   - It is desirable for the uniform distribution, from which weights are selected, to have zero mean and a variance $m^{-1/2}$ (m number of synaptic connections)
7. Learning from hints: Learning process may be generalized to include learning from hints (by allowing prior information we may have about the unknown function). These can be invariance properties, symmetries, etc.
8. Learning rates: All neurons should ideally learn at the same rate. Last layers usually have larger local gradients. Hence,
   - last layers should use smaller η than front layers
   - neurons with many inputs should use smaller η compared to those with few inputs

Generalizations

- For the first viewpoint (training set size)
  - There are theoretical formulas for estimation
  - But, there is often a huge gap between these theoretical estimates and actual size needed in practice
  - In practice, a good generalization can be achieved with training set size N satisfying the condition

$$N = O\left(\frac{W}{\in}\right)$$

where W is the total number of free parameters and $\in$ is the fraction of classification errors permitted on test data