4. Multilayer Perceptrons

4.1 Introduction

- A multilayer feedforward network consists of an input layer, one or more hidden layers, and an output layer.
- Computations take place in the hidden and output layers only.
- The input signal propagates through the network in a forward direction, layer-by-layer.
- Such neural networks are called *multilayer perceptrons* (MLPs).
- They have been successfully applied to many difficult and diverse problems.
- Multilayer perceptrons are typically trained using so-called error *back-propagation algorithm.*
- This is a supervised error-correction learning algorithm.

- It can be viewed as a generalization of the LMS algorithm.
- Back-propagation learning consists of two passes through the different layers of a MLP network.
- In the *forward pass*, the output (response) of the network to an input vector is computed.
- Here all the synaptic weights are kept fixed.
- During the *backward pass*, the weights are adjusted using an error-correction rule.
- The error signal is propagated backward through the network.
- After adjustment, the output of the network should have moved closer to the desired response in a statistical sense.

Properties of Multilayer Perceptron

- Each neuron has a *smooth* (differentiable everywhere) *nonlinear activation function*.
- This is usually a sigmoidal nonlinearity defined by the *logistic function*

$$y_j = \frac{1}{1 + \exp(-v_j)}$$

where v_j is the local field (weighted sum of inputs plus bias).

- Nonlinearities are important: otherwise the network could be reduced to a linear single-layer perceptron.
- The network contains hidden layer(s), enabling learning complicated tasks and mappings.
- The network has a high connectivity.

- These properties give the multilayer perceptron its computational power.

- On the other hand, distributed nonlinearities make the theoretical analysis of a MLP network difficult.

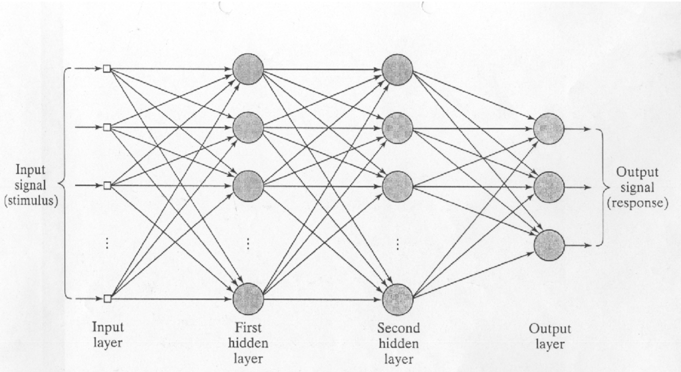
- Back-propagation learning is more difficult and in its basic form slow because of the hidden layer(s).

Contents of the Chapter 4

- The chapter contains 100 pages (including notes and exercises) divided in seven major parts:
  - Back-propagation learning (Sections 4.2-4.6)
  - Multilayer perceptrons in pattern recognition (4.7-4.9)
  - Error surface (4.10-4.11)
  - Performance of a MLP trained using backpropagation (4.12-4.15)
  - Advantages, drawbacks, and heuristics for backpropagation learning (4.16-4.17)
  - Improved learning methods based on optimization (4.18)
  - Convolutional multilayer perceptron (4.19)
- In this basic course, we shall skip less important or too advanced topics.

4.2 Some preliminaries

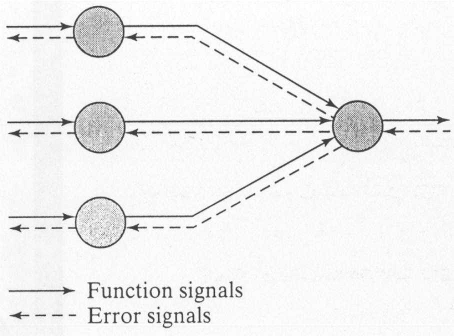
- An architectural graph of a multilayer perceptron with two hidden layers and an output layer.



- Recall that in the input layer, no computations take place; the input vector is only fed in componentwise.

- The network is *fully connected*.

- Two kinds of signals appear in the MLP network:



1. *Function Signals.* Input signals propagating forward through the network, producing in the last phase output signals.
2. *Error signals.* Originate at output neurons, and propagate layer by layer backward through the network.

- Each hidden or output neuron performs two computations:

    1. The computation of the function signal appearing at its output. This is a nonlinear function of the input signal and synaptic weights of that neuron.
    2. The computation of an estimate of the gradient vector, needed in the backward pass.

- The derivation of the back-propagation algorithm is rather involved.

Notation

- The indices i, j and k refer to neurons in different layers. Neuron j lies in the layer right to neuron i, and neuron k right to neuron j.
- In iteration n, the nth training vector is presented to the network.
- The symbol $\mathcal{E}(n)$ refers to the instantaneous sum of error squares or error energy at iteration n.
  - $\mathcal{E}_{av}$ is the average of $\mathcal{E}(n)$ over all n.
- $e_j(n)$ is the error signal at the output of neuron j for iteration n.
- $d_j(n)$ is the desired response for neuron j.
- $y_j(n)$ is the function signal at the output of neuron j for iteration n.
- $w_{ji}(n)$ is the weight connecting the output of neuron i to the input of neuron j at iteration n.
  - The correction applied to this weight is denoted by $\Delta w_{ji}(n)$.

- $v_j(n)$ denotes the local field of neuron j at iteration n.
  - It is the weighted sum of inputs plus bias of that neuron.

- The activation function (nonlinearity) associated with neuron j is denoted by $\varphi_j(.)$.

- b_j denotes the bias applied to neuron j, corresponding to the weight $w_{j0} = b_j$ and a fixed input $+1$.

- $x_i(n)$ denotes the ith element of the input vector.

- $o_k(n)$ denotes the kth element of the overall output vector.

- η denotes the learning-rate parameter.

- m_l denotes the number of neurons in layer l.
  - The network has L layers.
  - For output layer, the notation $m_L = M$ is also used.

4.3 Back-Propagation Algorithm

- The error signal at the output of neuron j at iteration n is defined by

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

- The instantaneous value of the error energy for neuron j is defined by $e_j^2(n)/2$.

- The total instantaneous error energy $\mathcal{E}(n)$ for all the neurons in the output layer is therefore

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

  where the set C contains all the neurons in the output layer.

- Let N be the total number of training vectors (examples, patterns).

- Then the *average squared error* is

$$\mathcal{E}_{av} = \frac{1}{N}\sum_{n=1}^{N} \mathcal{E}(n) \tag{3}$$

- For a given training set, $\mathcal{E}_{av}$ is the *cost function* which measures the learning performance.

- It depends on all the free parameters (weights and biases) of the network.

- The objective is to derive a learning algorithm for minimizing $\mathcal{E}_{av}$ with respect to the free parameters.

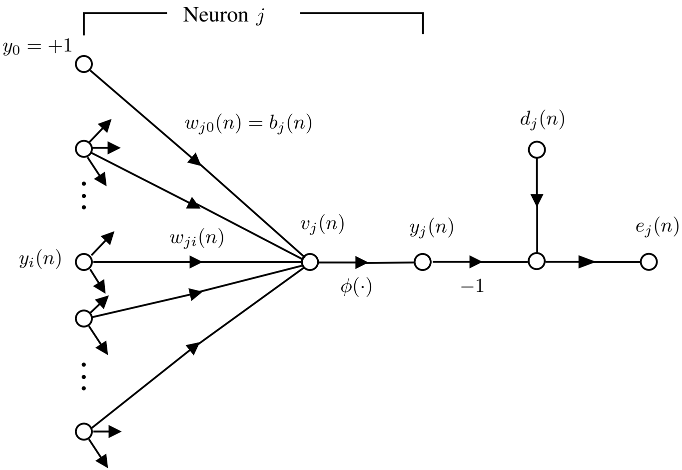
- In the basic back-propagation, a similar training method as in the LMS algorithm is used.

- Weights are updated on a pattern-by-pattern basis during each epoch.

- *Epoch* is one complete presentation of the entire training set.

- In other words, instantaneous stochastic gradient based on a single sample only is used for getting simple adaptive update formulas.

- The average of these updates over one epoch estimates the gradient of $\mathcal{E}_{av}$.



- Neuron j.

- It is fed by a set of function signals produced by a layer of neurons to its left.

- The local field $v_j(n)$ of neuron j is clearly

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \tag{4}$$

- The function signal $y_j(n)$ appearing at the output of neuron j at iteration n is then
$$y_j(n) = \varphi_j(v_j(n)). \tag{5}$$

- The correction $\Delta w_{ji}(n)$ made to the synaptic weight $w_{ji}(n)$ is proportional to the partial derivative $\partial \mathcal{E}(n)/\partial w_{ji}(n)$ of the instantaneous error.

- Using the *chain rule* of calculus, this gradient can be expressed as follows:
$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{6}$$

- The partial derivative $\partial \mathcal{E}(n)/\partial w_{ji}(n)$ represents a *sensitivity factor*.

- It determines the direction of search for the weight $w_{ji}(n)$.

- Differentiating both sides of Eq. (2) with respect to $e_j(n)$, we get
$$\frac{\partial \mathcal{E}(n)}{\partial e_j(n)} = e_j(n) \tag{7}$$

- Differentiating Eq. (1) with respect to $y_j(n)$ yields

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{8}$$

- Differentiating Eq. (5) with respect to $v_j(n)$, we get

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)) \tag{9}$$

where φ_j' denotes the derivative of φ_j.

- Finally, differentiating (4) with respect to $w_{ji}(n)$ yields

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n). \tag{10}$$

- Inserting these partial derivatives into (6) yields

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n) \tag{11}$$

- The correction $\Delta w_{ji}(n)$ applied to the weight $w_{ji}(n)$ is defined by the *delta rule*:

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} \tag{12}$$

  where η is the learning-rate parameter of the back-propagation algorithm.

- The minus sign comes from using *gradient descent* in learning for minimizing the error $\mathcal{E}(n)$.

- Inserting (11) into (12) yields

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_j(n) \tag{13}$$

  where the *local gradient* is defined by

$$\delta_j(n) = -\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} = e_j(n) \varphi_j'(v_j(n)) \tag{14}$$

- We note that a key factor in the calculation of the weight adjustment $\Delta w_{ji}(n)$ is the error signal $e_j(n)$ at the output of neuron j.

- This error signal depends on the location of the neuron in the MLP network.

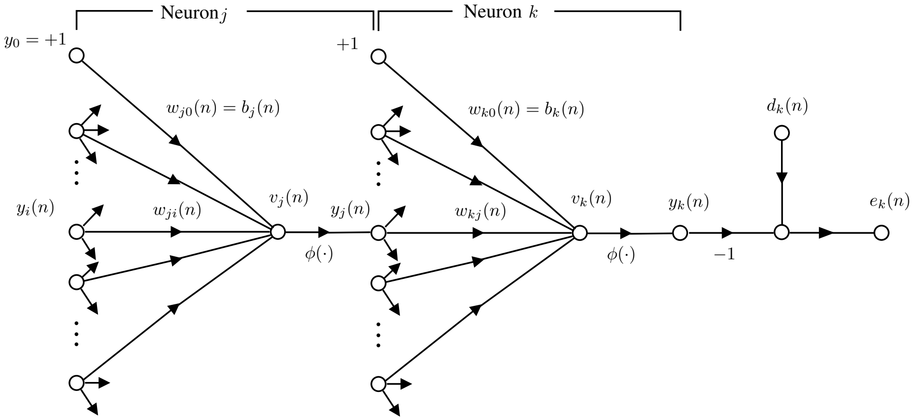
Case 1: Neuron j is an Output Node

- Computation of the error $e_j(n)$ is straightforward in this case.
- The desired response $d_j(n)$ for the neuron j is directly available.
- One can use the previous formulas (13) and (14).

Case 2: Neuron j is a Hidden Node

- Now there is no desired response available for neuron j.
- Question: how to compute the responsibility of this neuron for the error made at the output?

- This is the *credit-assignment problem* discussed earlier.
- The error signal for a hidden neuron must be determined recursively in terms of the error signals of all neurons connected to it.
- Here the development of the back-propagation algorithm gets complicated.



- Using Eq. (14), we may redefine the local gradient $\delta_j(n)$ for hidden neuron j as follows:

$$\delta_j(n) = -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} = -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \varphi_j'(v_j(n)) \qquad (15)$$

- The partial derivative $\partial \mathcal{E}(n)/\partial y_j(n)$ may be calculated as follows.

- From the figure we see that

$$\mathcal{E}(n) = \frac{1}{2} \sum_{k \in C} e_k^2(n), \text{ neuron k is an output node} \qquad (16)$$

- Differentiating this with respect to the function signal $y_j(n)$ and using the chain rule we get

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \qquad (17)$$

- From the figure we note that when the neuron k is an output node

$$e_k(n) = d_k(n) - y_k(n) = d_k(n) - \varphi_k(v_k(n)) \qquad (18)$$

so that

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k'(v_k(n)) \tag{19}$$

- Figure shows also that the local field of neuron k is

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \tag{20}$$

where the bias term is again included as the weight $w_{k0}(n)$.

- Differentiating this with respect to $y_j(n)$ yields

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n) \tag{21}$$

- Inserting these expressions into (17) we get the desired partial derivative

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = -\sum_k e_k(n)\varphi_k'(v_k(n))w_{kj}(n) = -\sum_k \delta_k(n) w_{kj}(n) \tag{22}$$

- Here again $\delta_k(n)$ denotes the local gradient for neuron k.
- Finally, inserting (22) into (15) yields the *back-propagation formula* for the local gradient $\delta_j(n)$:

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \tag{23}$$

- This holds when neuron j is hidden.
- Let us briefly study the factors in this formula:
  - $\varphi_j'(v_j(n))$ depends solely on the activation function $\varphi_j(.)$ of the hidden neuron j.
  - The local gradients $\delta_k(n)$ require knowledge of the error signals $e_k(n)$ of the neurons in the next (right-hand side) layer.
  - The synaptic weights $w_{kj}(n)$ describe the connections of neuron j to the neurons in the next layer to the right.
- We may summarize the results derived thus far in this section as follows:

- The correction $\Delta w_{ji}(n)$ of the weight connecting neuron i to neuron j is described by Eq. (4.25) in book

- The local gradient $\delta_j(n)$ is computed from Eq. (14) of previous lecture if neuron j lies in the output layer.

- If neuron j lies in the hidden layer, the local gradient is computed from Eq. (23).

Back-Propagation Algorithm: (4.25) in Haykin

$$\begin{pmatrix} \textit{Weight} \\ \textit{correction} \\ \Delta w_{ij}(n) \end{pmatrix} = \begin{pmatrix} \textit{Learning} \\ \textit{parameter} \\ \eta \end{pmatrix} \begin{pmatrix} \textit{Local} \\ \textit{gradient} \\ \delta_j(n) \end{pmatrix} \begin{pmatrix} \textit{Input signal} \\ \textit{of neuron } j \\ y_i(n) \end{pmatrix}$$

- The local gradient is given by

$$\delta_j(n) = e_j(n)\varphi_j'(v_j(n)) \qquad (4.14)$$

when the neuron j is in the output layer.

- In the hidden layer, the local gradient is

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \qquad (4.24)$$

computed recursively from the local gradients of the following layer, back-propagating error

The Two Passes of Computation

- In applying the back-propagation algorithm, two distinct passes of computation are distinguished.

- **Forward pass**

  - The weights are not changed in this phase.
  - The function signal appearing at the output of neuron j is computed as

$$y_j(n) = \varphi(v_j(n)) \tag{24}$$

  - Here the local field $v_j(n)$ of neuron j is

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \tag{25}$$

  - In the first hidden layer, $m = m_0$ is the number of input signals $x_i(n)$, $i = 1, \ldots, m_0$, and in Eq. (25)

$$y_i(n) = x_i(n)$$

- In the output layer, $m = m_L$ is the number of outputs Eq. (24).
- The outputs (components of the output vector) are denoted by

$$y_j(n) = o_j(n)$$

- These outputs are then compared with the respective desired responses $d_j(n)$, yielding the error signals $e_j(n)$.
- In the forward pass, computation starts from the first hidden layer and terminates at the output layer.

- **Backward pass**
  - In the backward pass, computation starts at the output layer, and ends at the first hidden layer.
  - The local gradient δ is computed for each neuron by passing the error signal through the network layer by layer.
  - The delta rule of Eq. (4.25) is used for updating the synaptic weights.
  - The weight updates are computed recursively layer by layer.
- The input vector is fixed through each round-trip (forward pass followed by a backward pass).
- After this, the next training (input) vector is presented to the network.

Activation Function

- The derivative of the activation function $\varphi(.)$ is needed in computing the local gradient δ.

- Therefore, $\varphi(.)$ must be continuous and differentiable.

- In MLP networks, two forms of sigmoidal nonlinearities are commonly used as activation functions.

  1. **Logistic function**

$$\varphi(v) = \frac{1}{1 + \exp(-av)}, \quad a > 0 \text{ and } -\infty < v < \infty$$

For clarity, we have omitted here the neuron index j and the iteration number n.

- The range of $\varphi(v)$ and hence the output $y = \varphi(v)$ always lies in the interval $0 \leq y \leq 1$.

The derivative of $y = \varphi(v)$ can be expressed in terms of the output y as

$$\varphi'(v) = ay(1-y)$$

- This formula allows writing the local gradient $\delta_j(n)$ in somewhat simpler form.

If neuron j is an output node,

$$\delta_j(n) = \ a[d_j(n) - o_j(n)]o_j(n)[1 - o_j(n)]$$

The respective equation for a hidden node is given in Eq. (4.34) in Haykin's book.

2. **Hyperbolic tangent function**

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

where a and b are positive constants.

- In fact, the hyperbolic tangent is just the logistic function rescaled and biased.

Its derivative with respect to v is

$$\varphi'(v) = ab[1 - \tanh^2(bv)] = \frac{b}{a}[a - y][a + y]$$

Using this, the local gradients of output neurons and hidden neurons can be simplified to Eqs. (4.37) and (4.38).

Rate of Learning

- Back-propagation approximates steepest descent method.
- A small learning-rate parameter η leads to a slow learning rate.
- Generally, basic back-propagation suffers from very slow learning if the network is large (several layers, a lot of nodes).
- On the other hand, choosing too large a learning parameter may lead to oscillatory behavior.
- A simple method of improving the learning speed without oscillatory behavior:
- Use a *generalized delta rule* including a momentum term:

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n) \tag{26}$$

- Here α is a positive *momentum constant*.
- If $\alpha = 0$, the corresponding momentum term vanishes.

- Then Eq. (26) reduces to the standard delta rule derived earlier.

- The effect of the momentum term is analyzed somewhat in Haykin's book.

- The conclusions are:

  1. The momentum constant should be in the interval $0 \leq \alpha < 1$.
  2. The momentum term tends to accelerate descent in steady downhill direction.
  3. In directions where the partial derivative $\partial \mathcal{E}(t)/\partial w_{ji}(t)$ oscillates in sign, the momentum term has a stabilizing effect.

- In deriving the back-propagation algorithm, it was assumed that the learning parameter η is a constant.

- In practice, it is better to use a *connection-dependent* learning parameter η_{ij}.

- This will be discussed later.

Sequential and Batch Modes of Training

- Recall that one complete presentation of the entire training set is called an epoch.
- The learning process is continued over several/many epochs.
- Learning is stopped when the weight values and biases stabilize, and the average squared error converges to some minimum value.
- It is useful to present the training samples in a randomized order during each epoch.
- In back-propagation, one may use either sequential (on-line, stochastic) or batch learning mode.

1. Sequential Mode

- The weights are updated after presenting each training example (input vector).
- The derivation before was given for this mode.

2. Batch Mode

- Here the weights are updated after each epoch only.
- All the training examples are presented once before updating the weights and biases.

- In batch mode, the cost function is the average squared error

$$\mathcal{E}_{av} = \frac{1}{2N} \sum_{n=1}^{N} \sum_{j \in C} e_j^2(n) \tag{27}$$

- The synaptic weight is updated using the batch delta rule

$$\Delta w_{ji} = -\eta \frac{\partial \mathcal{E}_{av}}{\partial w_{ji}} = -\frac{\eta}{N} \sum_{n=1}^{N} e_j(n) \frac{\partial e_j(n)}{\partial w_{ji}} \tag{28}$$

- The partial derivative $\partial e_j(n)/\partial w_{ji}$ may be computed as in the sequential mode.

- *Advantages of sequential mode:*

  - requires less storage
  - less likely to get trapped in a local minimum.

- *Advantages of the batch mode:*

  - Provides an accurate estimate of the gradient vector.
  - Convergence to a local minimum at least is guaranteed.

- The sequential mode of back-propagation has several disadvantages.
- In spite of that, it is highly popular for two important practical reasons:
  - The algorithm is simple to implement.
  - It provides effective solutions to large and difficult problems.

Stopping Criteria

- In general, the back-propagation algorithm cannot be shown to converge.
- There are no well-defined criteria for stopping its operation.
- However, there are reasonable practical criteria for terminating learning.
- Consider first the unique properties of a *local* or *global minimum* of the error surface.
- Denote by $\mathbf{w}^*$ a weight vector at a local or global minimum.
- Necessary condition: the gradient vector $\mathbf{g}(\mathbf{w})$ vanishes at the minimum point $\mathbf{w}^*$.
- This yield the following criterion for the convergence of back-propagation learning:

- *Stop learning when the Euclidean norm $\| \mathbf{g}(\mathbf{w}) \|$ of the gradient vector is below a certain threshold.*

- Drawbacks of this stopping criterion:
  - Learning times may be long.
  - Requires computation of the gradient vector $\mathbf{g}(\mathbf{w})$.

- Another criterion is based on the stationarity of the average squared error measure $\mathcal{E}_{av}$ at the point $\mathbf{w} = \mathbf{w}^*$:

- *Stop learning when the absolute rate of change in the average squared error per epoch is sufficiently small.*

- A small rate of change is usually taken to be 0.1% - 1% per epoch.

- This criterion may result in a premature termination of the learning process.

- Another useful, theoretically sound criterion for convergence: test the generalization performance of the network.

- This is discussed later on in Section 4.14.