

## 5 Feedforward Multilayer Neural Networks

Feedforward multilayer neural networks were introduced in sec. 2. Such neural networks with supervised error correcting learning are used to **approximate (synthesise)** a non-linear input-output mapping from a set of training patterns.

Consider a mapping  $f(X)$  from a  $p$ -dimensional domain  $X$  into an  $m$ -dimensional output space  $D$  as in Figure 5–1.

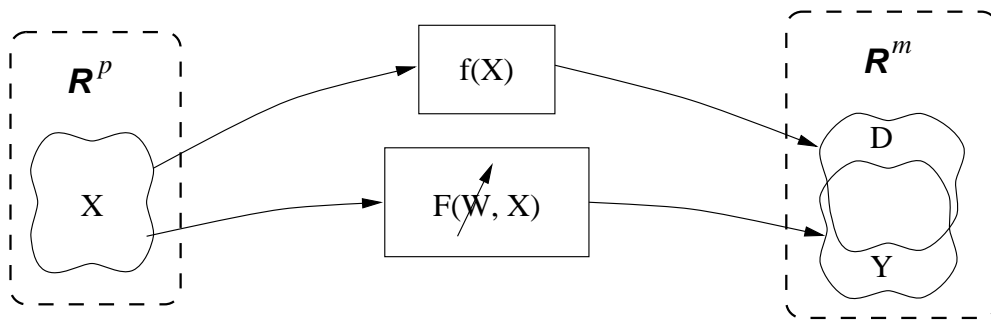


Figure 5–1: Mapping from a  $p$ -dimensional domain into an  $m$ -dimensional output space

A function

$$f : X \rightarrow D, \text{ or } \mathbf{d} = f(\mathbf{x}); \mathbf{x} \in X \subset \mathcal{R}^p, \mathbf{d} \in D \subset \mathcal{R}^m$$

is assumed to be unknown, but it is specified by a set of training examples,  $\{X; D\}$ . This function is approximated by a fixed, parameterised function (a neural network)

$$F : \mathcal{R}^p \times \mathcal{R}^M \rightarrow \mathcal{R}^m, \text{ or } \mathbf{y} = F(W, \mathbf{x}); \mathbf{x} \in \mathcal{R}^p, \mathbf{d} \in \mathcal{R}^m, W \in \mathcal{R}^M$$

Approximation is performed in such a way that some **performance index**,  $J$ , typically a function of errors between  $D$  and  $Y$ ,

$$J = J(W, \|D - Y\|)$$

is minimised

Basic types of NETWORKS for APPROXIMATION:

- **Linear Networks — Adaline**

$$F(W, \mathbf{x}) = W \cdot \mathbf{x}$$

- **“Classical” approximation schemes**

Consider a set of suitable **basis functions**  $\{\phi\}_{i=1}^n$   
then

$$F(W, \mathbf{x}) = \sum_{i=1}^n w_i \phi_i(\mathbf{x})$$

Popular examples: power series, trigonometric series, splines, Radial Basis Functions. The Radial Basis Functions guarantee, under certain conditions, an optimal solution of the approximation problem.

A special case: Gaussian Radial Basis Functions:

$$\phi_i(\mathbf{x}) = \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{t}_i)^T \Sigma_i^{-1}(\mathbf{x} - \mathbf{t}_i)\right)$$

where  $\mathbf{t}_i$  and  $\Sigma_i$  are the centre and covariance matrix of the  $i$ -th RBF.

- **Multilayer Perceptrons — Feedforward neural networks**

Each layer of the network is characterised by its matrix of parameters, and the network performs composition of nonlinear operations as follows:

$$F(W, \mathbf{x}) = \varphi(W_1 \cdot \dots \varphi(W_l \cdot \mathbf{x}) \dots)$$

A feedforward neural network with two layers (one hidden and one output) is very commonly used to approximate unknown mappings. If the output layer is linear, such a network may have a structure similar to an RBF network.

## 5.1 Multilayer perceptrons

Multilayer perceptrons are commonly used to approximate complex nonlinear mappings. In general, it is possible to show that two layers are sufficient to approximate any nonlinear function. Therefore, we restrict our considerations to such two-layer networks.

The structure of the decoding part of the two-layer back-propagation network is presented in Figure (5–2).

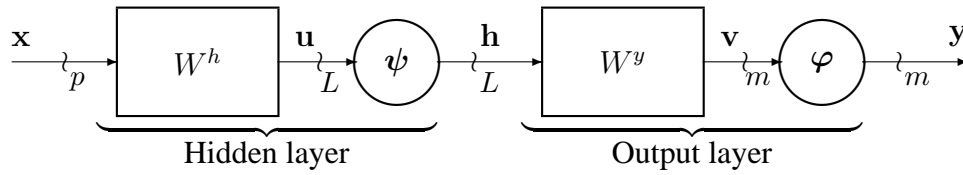


Figure 5–2: A block-diagram of a single-hidden-layer feedforward neural network

The structure of each layer has been depicted in Figure ???. Nonlinear functions used in the hidden layer and in the output layer can be different. There are two weight matrices: an  $L \times p$  matrix  $W^h$  in the hidden layer, and an  $m \times L$  matrix  $W^y$  in the output layer. The working of the network can be described in the following way:

$$\begin{aligned} \mathbf{u}(n) &= W^h \cdot \mathbf{x}(n); \quad \mathbf{h}(n) = \psi(\mathbf{u}(n)) \quad \text{— hidden signals;} \\ \mathbf{v}(n) &= W^y \cdot \mathbf{h}(n); \quad \mathbf{y}(n) = \varphi(\mathbf{v}(n)) \quad \text{— output signals.} \end{aligned}$$

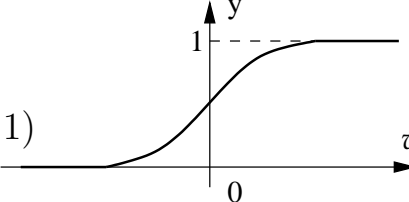
or simply as

$$\mathbf{y}(n) = \varphi(W^y \cdot \psi(W^h \cdot \mathbf{x}(n))) \quad (5.1)$$

Typically, sigmoidal functions (hyperbolic tangents) are used, but other choices are also possible. The important condition from the point of view of the learning law is for the function to be differentiable.

Typical non-linear functions and their derivatives used in multi-layer perceptrons:

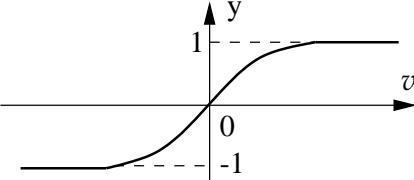
**Sigmoidal unipolar:**

$$y = \varphi(v) = \frac{1}{1 + e^{-\beta v}} = \frac{1}{2}(\tanh(\beta v/2) + 1)$$


The derivative of the unipolar sigmoidal function:

$$y' = \frac{d\varphi}{dv} = \beta \frac{e^{-\beta v}}{(1 + e^{-\beta v})^2} = \beta y(1 - y)$$

**Sigmoidal bipolar:**

$$\varphi(v) = \tanh(\beta v)$$


The derivative of the bipolar sigmoidal function:

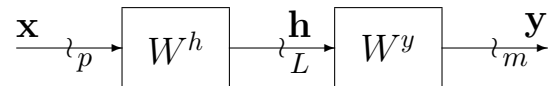
$$y' = \frac{d\varphi}{dv} = \frac{4\beta e^{2\beta v}}{(e^{2\beta v} + 1)^2} = \beta(1 - y^2)$$

Note that

- Derivatives of the sigmoidal functions are always non-negative.
- Derivatives can be calculated directly from output signals using simple arithmetic operations.
- In saturation, for big values of the activation potential,  $v$ , derivatives are close to zero.
- Derivatives of used in the error-correction learning law.

### Comments on multi-layer linear networks

Multi-layer feedforward **linear** neural networks can be always replaced by an equivalent single-layer network. Consider a linear network consisting of two layers:



The hidden and output signals in the network can be calculated as follows:

$$\mathbf{h} = W^h \cdot \mathbf{x} , \quad \mathbf{y} = W^y \cdot \mathbf{h}$$

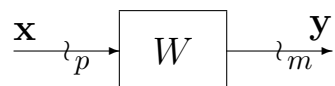
After substitution we have:

$$\mathbf{y} = W^y \cdot W^h \cdot \mathbf{x} = W \cdot \mathbf{x}$$

where

$$W = W^y \cdot W^h$$

which is equivalent to a single-layer network described by the weight matrix,  $W$ :



## 5.2 Back-propagation of error

The back-propagation learning law is a very popular and conceptually important learning rule applied to multilayer feedforward neural networks. Because of this popularity, multilayer perceptrons are often referred to as the **back-propagation neural networks**.

The back-propagation learning law is a supervised error-correction rule in which the output error, that is, the difference between the desired and the actual output is **propagated back** to the hidden layers. Now, if the error at the output of each layer can be determined, it is possible to apply any method which minimises the performance index to each layer sequentially.

Consider a single layer of nonlinear neurons as in Figure 5–3

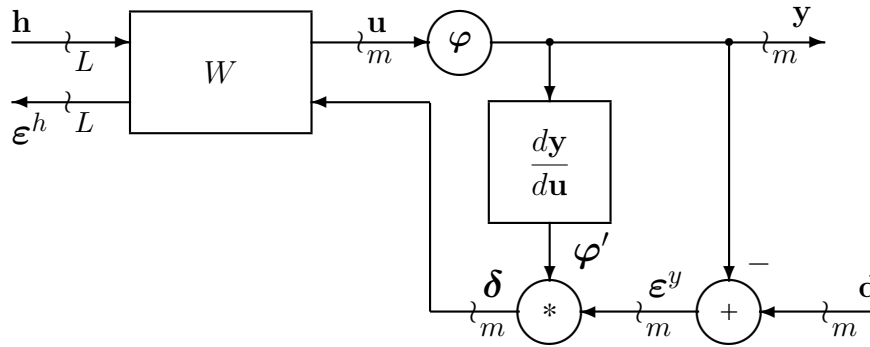


Figure 5–3: Back-propagation of error

If the error measured at the neurons output is

$$\boldsymbol{\varepsilon}^y = \mathbf{d} - \mathbf{y}$$

then an equivalent input error,  $\boldsymbol{\varepsilon}^h$ , that is, an input perturbation which would cause the output error,  $\boldsymbol{\varepsilon}^y$ , can be determined as:

$$\boldsymbol{\varepsilon}^h = W^T \cdot \boldsymbol{\delta} ; \quad \boldsymbol{\delta} = [\delta_1 \dots \delta_m]^T$$

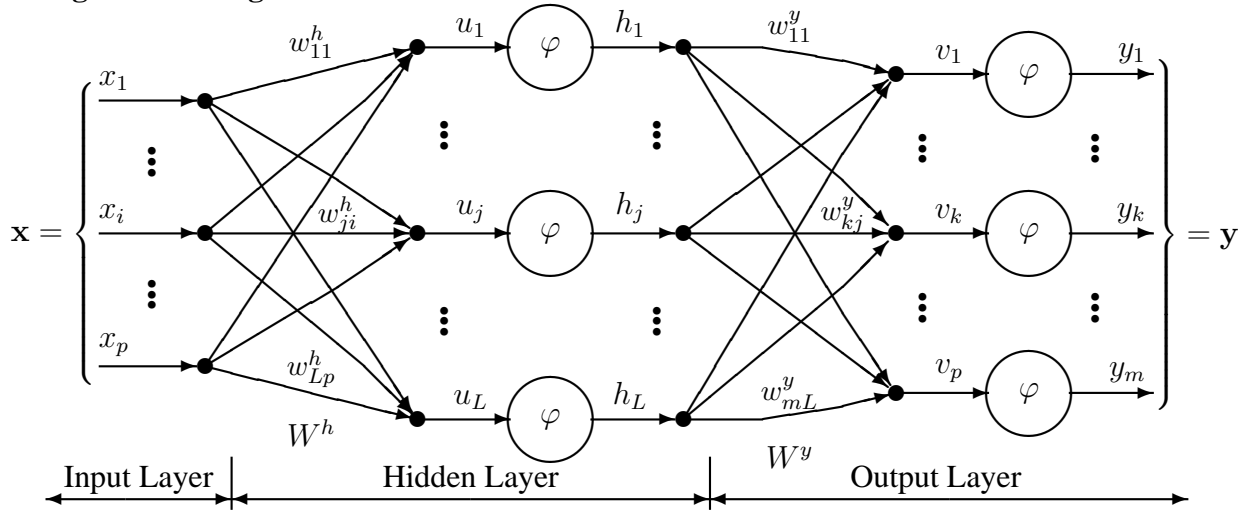
where the  $k$ -th **delta signal** is

$$\delta_k = \varphi'_k \cdot \varepsilon_k^y = \frac{dy_k}{du_k} \cdot \varepsilon_k^y$$

### 5.3 Back-Propagation Algorithm for a Two-Layer Perceptron

The most commonly used feedforward neural network is a two-layer perceptron of the following structure:

**Signal-flow diagram:**



**Dendritic diagram:**

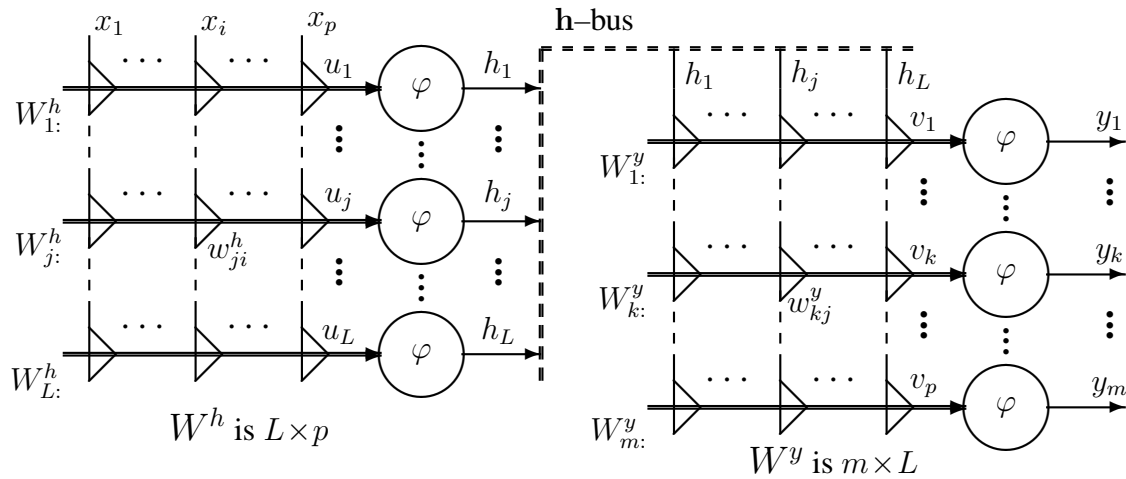


Figure 5–4: Various representations of a Two-Layer Perceptron

Training data is organised as for the linear neurons, that is, we are given  $N$  input vectors,  $\mathbf{x}(n)$ , and  $N$  desired output vectors,  $\mathbf{d}(n)$ , in two matrices:

$$\begin{aligned} X &= [\mathbf{x}(1) \ \dots \ \mathbf{x}(n) \ \dots \ \mathbf{x}(N)] \text{ is } p \times N \text{ matrix,} \\ D &= [\mathbf{d}(1) \ \dots \ \mathbf{d}(n) \ \dots \ \mathbf{d}(N)] \text{ is } m \times N \text{ matrix} \end{aligned}$$

For each input vector,  $\mathbf{x}(n)$ , the network calculates the actual output vector,  $\mathbf{y}(n)$ , as in eqn (5.1). For the whole training epoch the output vectors can be collected in a  $m \times N$  matrix

$$Y = [\mathbf{y}(1) \ \dots \ \mathbf{y}(n) \ \dots \ \mathbf{y}(N)] \text{ is } m \times N \text{ matrix}$$

The complete set of the output and the hidden vectors can be also calculated as:

$$Y = \varphi(W^y \cdot H) ; \quad H = \psi(W^h \cdot X)$$

The output error at the  $n$ th step

$$\boldsymbol{\varepsilon}(n) = [\varepsilon_1(n) \ \dots \ \varepsilon_m(n)]^T = \mathbf{d}(n) - \mathbf{y}(n) \text{ is an } m \times 1 \text{ vector,}$$

each component being equal to:

$$\varepsilon_k(n) = d_k(n) - y_k(n) \quad \text{for } k = 1, \dots, m$$

As for the linear network, the objective is to find a set of weight matrices,  $W^h, W^y$ , such that the errors are as small as possible. For convenience, all weights from the weight matrices can be arranged in one “long” weight vector:

$$\mathbf{w} = \text{scan}(W^h, W^y) = [w_{11}^h \ \dots \ w_{1p}^h \ \dots \ w_{Lp}^h | w_{11}^y \ \dots \ w_{1L}^y \ \dots \ w_{mL}^y]$$

The length of  $\mathbf{w}$  is  $K = L(p + m)$ .



More specifically, we try to minimise a **performance index**, typically the **mean-squared error**,  $J(W^h, W^y)$  specified as an averaged sum of instantaneous squared errors at the network output:

$$J(W^h, W^y) = J(\mathbf{w}) = \frac{1}{m N} \sum_{n=1}^N E(\mathbf{w}, n) \quad (5.2)$$

where the **total instantaneous squared error**,  $E(\mathbf{w}, n)$ , is defined as

$$E(\mathbf{w}, n) = \frac{1}{2} \sum_{k=1}^m \varepsilon_k^2(n) = \frac{1}{2} \boldsymbol{\varepsilon}^T(n) \cdot \boldsymbol{\varepsilon}(n) \quad (5.3)$$

In the **steepest descend** learning algorithms the weight update should be made proportional to the gradient of the **mean-squared error** specified in eqn (5.2) with respect to the total weight vector,  $\mathbf{w}$  formed from all the elements of the weight matrices. This gradient can be calculated as follows:

$$\nabla J(\mathbf{w}) = \frac{1}{m N} \sum_{n=1}^N \nabla E(\mathbf{w}, n) \quad (5.4)$$

where  $\nabla E(\mathbf{w}, n)$  is the gradient vector of the total **instantaneous** error. This instantaneous gradient has components associated with the hidden layer weights,  $W^h$ , and the output layer weights,  $W^y$ , and can be arranged in the following way:

$$\nabla E(\mathbf{w}, n) = \left[ \frac{\partial E}{\partial w_{11}^h} \cdots \frac{\partial E}{\partial w_{ji}^h} \cdots \frac{\partial E}{\partial w_{Lp}^h} \frac{\partial E}{\partial w_{11}^y} \cdots \frac{\partial E}{\partial w_{kj}^y} \cdots \frac{\partial E}{\partial w_{mL}^y} \right]$$

We will consider two versions of the **steepest descend** algorithm, known as the **pattern** and **batch** training algorithms.

In the **pattern training** steepest descend algorithm which is equivalent to the LMS algorithms the weight update is made proportional to the gradient of the total instantaneous squared error, that is:

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \eta \nabla E(\mathbf{w}, n) \quad (5.5)$$

In the pattern training algorithm the weights are modified after the presentation of every training pattern.

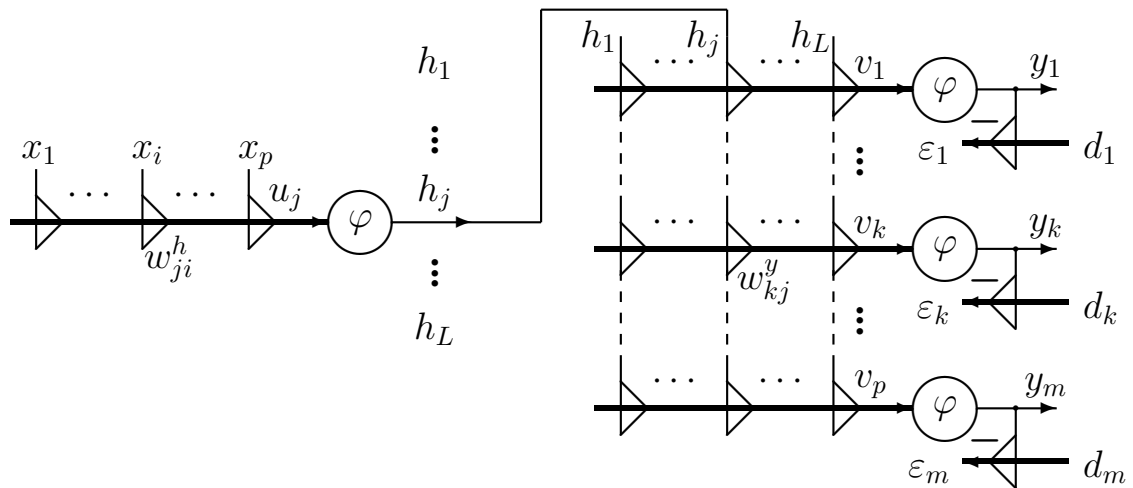
In the **batch training** steepest descend algorithm the weight update is made proportional to the gradient of the performance index, that is, the mean-squared error calculated for all training patterns:

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \eta \nabla J(\mathbf{w}) \quad (5.6)$$

In the batch training algorithm the weights are modified after the presentation of all training patterns, that is, after each epoch.

We will calculate first a component of the instantaneous gradient vector with respect to an **output** weight,  $w_{kj}^y$ , and then with respect to a **hidden** weight,  $w_{ji}^h$ .

The basic signal flow referred to in the above calculations is as follows:



### 5.3.1 The output layer

The gradient component related to a synaptic weight  $w_{kj}^y$  for the  $n$ -th training vector can be calculated as follows:

$$\begin{aligned}
 \frac{\partial E(n)}{\partial w_{kj}^y} &= -\varepsilon_k \frac{\partial y_k}{\partial w_{kj}^y} \\
 &= -\varepsilon_k \frac{\partial y_k}{\partial v_k} \frac{\partial v_k}{\partial w_{kj}^y} \\
 &= -\varepsilon_k \cdot \varphi'_k \cdot h_j \\
 &= -\delta_k \cdot h_j
 \end{aligned}
 \left\| \begin{aligned}
 E &= \frac{1}{2}(\dots + \varepsilon_k^2 + \dots), \quad y_k = \varphi(v_k) \\
 v_k &= W_{k:}^y \cdot \mathbf{h} = \dots + w_{kj}^y h_j + \dots \\
 \varphi'_k &= \frac{\partial y_k}{\partial v_k} \\
 \delta_k &= \varepsilon_k \cdot \varphi'_k
 \end{aligned} \right.$$

where the **delta error**,  $\delta_k$ , is the output error,  $\varepsilon_k$ , modified with the derivative of the activation function,  $\varphi'_k$ .

Alternatively, the gradient components related to the weight vector,  $W_{k:}^y$  of the  $k$ th output neuron can be calculated as:

$$\begin{aligned}
 \frac{\partial E(n)}{\partial W_{k:}^y} &= -\varepsilon_k \frac{\partial y_k}{\partial W_{k:}^y} \\
 &= -\varepsilon_k \frac{\partial y_k}{\partial v_k} \frac{\partial v_k}{\partial W_{k:}^y} \\
 &= -\varepsilon_k \cdot \varphi'_k \cdot \mathbf{h}^T \\
 &= -\delta_k \cdot \mathbf{h}^T
 \end{aligned}
 \left\| \begin{aligned}
 y_k &= \varphi(v_k) \\
 v_k &= W_{k:}^y \cdot \mathbf{h}
 \end{aligned} \right.$$

In the above expression, each component of the gradient is a function of the delta error for the  $k$ -th output,  $\delta_k$ , and respective output signal from the hidden layer,  $\mathbf{h}^T = [h_1 \dots h_L]$ .

Finally, the gradient components related to the complete weight matrix of the output layer,  $W^y$ , can be collected in an  $m \times L$  matrix, as follows:

$$\frac{\partial E(n)}{\partial W^y} = -\boldsymbol{\delta}(n) \cdot \mathbf{h}^T(n) \quad (5.7)$$

where

$$\boldsymbol{\delta} = \begin{bmatrix} \delta_1 \\ \vdots \\ \delta_m \end{bmatrix} = \begin{bmatrix} \varepsilon_1 \cdot \varphi'_1 \\ \vdots \\ \varepsilon_m \cdot \varphi'_m \end{bmatrix} = \boldsymbol{\varepsilon} \odot \boldsymbol{\varphi}', \quad (5.8)$$

and ‘ $\odot$ ’ denotes an ‘element-by-element’ multiplication, and the hidden signals are calculated as follows:

$$\mathbf{h}(n) = \boldsymbol{\psi}(W^h(n) \cdot \mathbf{x}(n))$$

Employing the **pattern training** (LMS) algorithm for minimisation of the performance index, we obtain an expression for the instantaneous, based on the  $n$ -th training pattern, update for the output weight matrix.

$$\Delta W^y(n) = -\eta_y \frac{\partial E(n)}{\partial W^y(n)} = \eta_y \cdot \boldsymbol{\delta}(n) \cdot \mathbf{h}^T(n) \quad (5.9)$$

The pattern weight update rule based on the LMS algorithm states that the weight modification should be proportional to the **outer product** of the vector of the modified output error,  $\boldsymbol{\delta}$ , and the vector of the input signals to the output layer (hidden signals),  $\mathbf{h}$ , that is,

$$W^y(n+1) = W^y(n) + \eta_y \cdot \boldsymbol{\delta}(n) \cdot \mathbf{h}^T(n) \quad (5.10)$$

The gradient of the total performance index,  $J(W^h, W^y)$ , related to the output weight matrix,  $W^y$ , can be obtained by summing the instantaneous gradients:

$$\frac{\partial J}{\partial W^y} = \frac{1}{m N} \sum_{n=1}^N \frac{\partial E(n)}{\partial W^y(n)} = -\frac{1}{m N} \sum_{n=1}^N \boldsymbol{\delta}(n) \cdot \mathbf{h}^T(n) \quad (5.11)$$

If we take into account that the sum of outer products can be replaced by a product of matrices collecting the contributing vectors, then we finally have

$$\frac{\partial J}{\partial W^y} = -\frac{1}{m N} S \cdot H^T \quad (5.12)$$

where  $S$  is the  $m \times N$  matrix of output delta errors:

$$S = [\boldsymbol{\delta}(1) \ \dots \ \boldsymbol{\delta}(N)] \quad (5.13)$$

and

$$H = \boldsymbol{\psi}(W^h \cdot X)$$

is the  $L \times N$  matrix of the hidden signals.

In the **batch training** steepest descent algorithm, the weight update after  $k$ -th epoch can be written as:

$$\Delta W^y(k) = -\hat{\eta}_y \frac{\partial J}{\partial W^y} = \eta_y \cdot S \cdot H^T \quad (5.14)$$

or simply:

$$W^y(k+1) = W^y(k) + \eta_y \cdot S \cdot H^T \quad (5.15)$$

### 5.3.2 The hidden layer

Calculations of the gradient components related to the weight matrix of the hidden layer are slightly more complicated, mainly, because we have to back-propagate error from the output layer.

We will demonstrate that by back-propagating the output error to the hidden layer it is possible to obtain a similar update rule for the hidden weight matrix.

Firstly, we calculate the gradient component related to a single weight of the hidden layer,  $w_{ji}^h$ , adding up error contributions from all output neurons as in figure in page 5–10.

$$\begin{aligned}
 \frac{\partial E(n)}{\partial w_{ji}^h} &= - \sum_{k=1}^m \varepsilon_k \frac{\partial y_k}{\partial w_{ji}^h} & \left\{ \begin{array}{l} y_k = \varphi(v_k) \\ v_k = \dots + w_{kj}^y h_j + \dots \\ \delta_k = \varepsilon_k \cdot \varphi'_k \\ W_{:j}^{yT} \boldsymbol{\delta} = \boldsymbol{\delta}^T W_{:j}^y = \sum_{k=1}^m \delta_k w_{kj}^y \end{array} \right. \\
 &= - \sum_{k=1}^m \varepsilon_k \frac{\partial y_k}{\partial v_k} \frac{\partial v_k}{\partial w_{ji}^h} \\
 &= - \left( \sum_{k=1}^m \delta_k w_{kj}^y \right) \frac{\partial h_j}{\partial w_{ji}^h} \\
 &= - W_{:j}^{yT} \boldsymbol{\delta} \frac{\partial h_j}{\partial w_{ji}^h}
 \end{aligned}$$

Note, that the  $j$ -th column of the output weight matrix,  $W_{:j}^y$ , is used to modify, we say, **back-propagate** the delta errors to create the equivalent hidden layer errors which are specified as follows:

$$\varepsilon_j^h = W_{:j}^{yT} \cdot \boldsymbol{\delta}, \quad \text{for } j = 1, \dots, L$$

Using the back-propagated error, we can now repeat the steps performed for the output layer, with  $\varepsilon_j^h$  and  $h_j$  replacing  $\varepsilon_k$  and  $y_k$ , respectively.

$$\begin{aligned}
\frac{\partial E(n)}{\partial w_{ji}^h} &= -\varepsilon_j^h \frac{\partial h_j}{\partial w_{ji}^h} \quad \left\| \begin{array}{l} h_j = \psi(u_j), \quad u_j = W_{j:}^h \cdot \mathbf{x} \\ \psi_j' = \frac{\partial \psi_j}{\partial u_j} \\ \delta_j^h = \varepsilon_j^h \cdot \psi_j' \end{array} \right. \\
&= -\varepsilon_j^h \cdot \psi_j' \cdot x_i \\
&= -\delta_j^h \cdot x_i
\end{aligned}$$

where the back-propagated error has been used to generate the delta-error for the hidden layer,  $\delta_j^h$ .

All gradient components related to the hidden weight matrix,  $W^h$ , can now be calculated in a way similar to that for the output layer as in eqn (5.7):

$$\frac{\partial E(n)}{\partial W^h} = -\boldsymbol{\delta}^h \cdot \mathbf{x}^T \quad (5.16)$$

where

$$\boldsymbol{\delta}^h = \begin{bmatrix} \varepsilon_1^h \cdot \psi_1' \\ \vdots \\ \varepsilon_L^h \cdot \psi_L' \end{bmatrix} = \boldsymbol{\varepsilon}^h \odot \boldsymbol{\psi}' \quad (5.17)$$

and the back-propagated (hidden) error

$$\boldsymbol{\varepsilon}^y = W^{yT} \cdot \boldsymbol{\delta}.$$

For the **pattern training** algorithm, the update of the hidden weight matrix for the  $n$ -the training pattern now becomes:

$$\Delta W^h(n) = -\eta_h \frac{\partial E_n}{\partial W^h(n)} = \eta_h \cdot \boldsymbol{\delta}^h \cdot \mathbf{x}^T \quad (5.18)$$

It is interesting to note that the weight update rule is identical in its form for both the output and hidden layers, that is,

$$W^h(n+1) = W^h(n) + \eta_h \cdot \delta^h(n) \cdot \mathbf{x}^T(n) \quad (5.19)$$

The gradient of the total performance index,  $J(W^h, W^y)$ , related to the hidden weight matrix,  $W^h$ , can be obtained by summing the instantaneous gradients:

$$\frac{\partial J}{\partial W^h} = \frac{1}{mN} \sum_{n=1}^N \frac{\partial E(n)}{\partial W^h(n)} = -\frac{1}{mN} \sum_{n=1}^N \delta^h(n) \cdot \mathbf{x}^T(n) \quad (5.20)$$

If we take into account that the sum of outer products can be replaced by a product of matrices collecting the contributing vectors, then we finally have

$$\frac{\partial J}{\partial W^h} = -\frac{1}{mN} S^h \cdot X^T \quad (5.21)$$

where  $S^h$  is the  $L \times N$  matrix of hidden delta errors:

$$S^h = [\delta^h(1) \ \dots \ \delta^h(N)] \quad (5.22)$$

and  $X$  is the  $p \times N$  matrix of the input signals.

In the **batch training** steepest descent algorithm, the weight update after  $k$ -th epoch can be written as:

$$\Delta W^h(k) = -\hat{\eta}_h \frac{\partial J}{\partial W^h} = \eta_y \cdot S^h \cdot X^T \quad (5.23)$$

or simply:

$$W^h(k+1) = W^h(k) + \eta_h \cdot S^h \cdot X^T \quad (5.24)$$



The structure of the two-layer back-propagation network is shown in Figure 5–5.

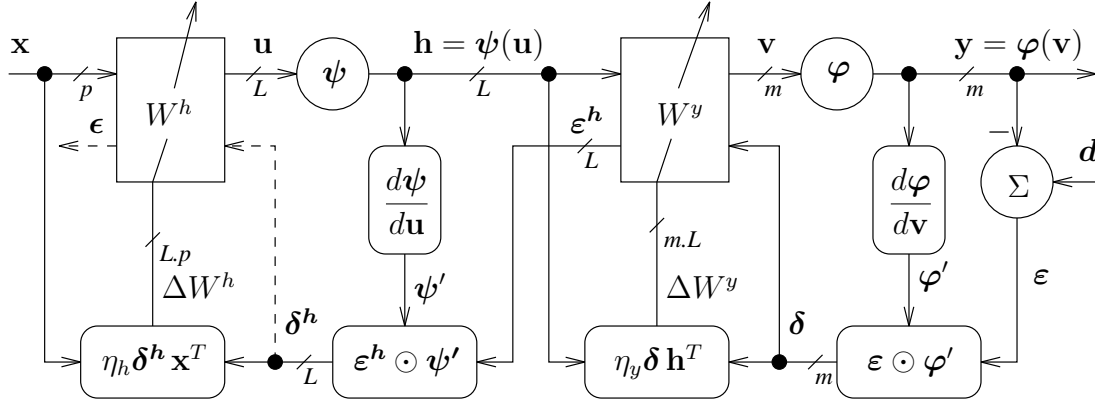


Figure 5–5: The structure of the decoding and encoding parts of the two-layer back-propagation network.

Note the decoding and encoding parts, and the blocks which calculate derivatives, delta signals and the weight update signals.

The process of computing the signals (pattern mode) during each time step consists of the:

**forward pass** in which the signals of the decoding part are determined starting from  $x$ , through  $u$ ,  $h$ ,  $\psi'$ ,  $v$  to  $y$  and  $\varphi'$ .

**backward pass** in which the signals of the learning part are determined starting from  $d$ , through  $\epsilon$ ,  $\delta$ ,  $\Delta W^y$ ,  $\epsilon^h$ ,  $\delta^h$  and  $\Delta W^h$ .

From Figure 5–5 and the relevant equations note that, in general, the weight update is proportional to the synaptic input signals ( $x$ , or  $h$ ) and the delta signals ( $\delta^h$ , or  $\delta$ ). The delta signals, in turn, are proportional to the derivatives the activation functions,  $\psi'$ , or  $\varphi'$ .

## Comments on Learning Algorithms for Multi-Layer Perceptrons.

- The process of training a neural network is monitored by observing the value of the performance index,  $J(W(n))$ , typically the mean-squared error as defined in eqns (5.2) and (5.3).
- In order to reduce the value of this error function, it is typically necessary to go through the set of training patterns (epochs) a number of times as discussed in page 4–17.
- There are two basic modes of updating weights:
  - the **pattern** mode in which weights are updated after the presentation of a single training pattern,
  - the **batch** mode in which weights are updated after each epoch.
- For the basic **steepest descent backpropagation** algorithm the relevant equations are:

### pattern mode

$$\begin{aligned} W^y(n+1) &= W^y(n) + \eta_y \cdot \delta(n) \cdot \mathbf{h}^T(n) \\ W^h(n+1) &= W^h(n) + \eta_h \cdot \delta^h(n) \cdot \mathbf{x}^T(n) \end{aligned}$$

where  $n$  is the pattern index.

### batch mode

$$\begin{aligned} W^y(k+1) &= W^y(k) + \eta_y \cdot S(k) \cdot H^T(k) \\ W^h(k+1) &= W^h(k) + \eta_h \cdot S^h(k) \cdot X^T(k) \end{aligned}$$

where  $k$  is the epoch counter. Definitions of the other variable have been already given.

- **Weight Initialisation**

The weight are initialised in one of the following ways:

- using prior information if available. The Nguyen-Widrow algorithm presented below is a good example of such initialisation.
- to **small** uniformly distributed random numbers.

Incorrectly initialised weights cause that the activation potentials may become large which saturates the neurons. In saturation, derivatives  $\varphi' = 0$  and no learning takes place.

A good initialisation can significantly speed up the learning process.

- **Randomisation**

For the pattern training it might be a good practice to randomise the order of presentation of training examples between epochs.

- **Validation**

In order to validate the process of learning the available data is randomly partitioned into a **training set** which is used for training, and a **test set** which is used for validation of the obtained data model.

## 5.4 Example of function approximation (fap2D.m)

In this MATLAB example we approximate two functions of two variables,

$$\mathbf{y} = \mathbf{f}(\mathbf{x}) , \text{ or } y_1 = f_1(x_1, x_2) , \quad y_2 = f_2(x_1, x_2)$$

using a two-layer perceptron,

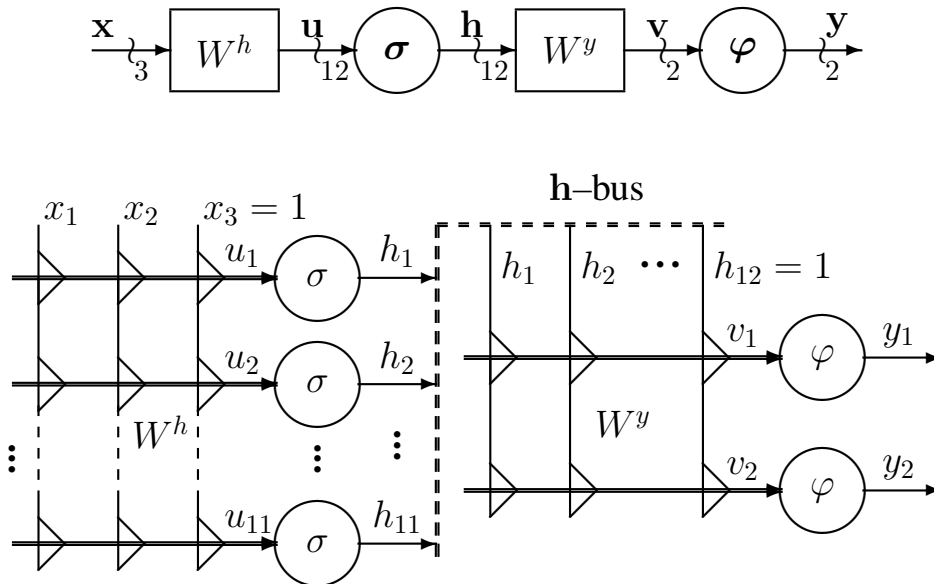
$$\mathbf{y} = \boldsymbol{\sigma}(W^y \cdot \boldsymbol{\varphi}(W^h \cdot \mathbf{x}))$$

The weights of the perceptron,  $W^h, W^y$ , are trained using the basic **back-propagation algorithm** in a **batch mode** as discussed in the previous section.

We begin with the specification of the the neural network (fap2Di.m):

```
p = 3 ; % Number of inputs (2) plus the bias input
L = 12; % Number of hidden signals (with bias)
m = 2 ; % Number of outputs
```

The network structure is as follows



Two functions to be approximated by the two-layer perceptron are as follows:

$$y_1 = x_1 e^{-\rho}, \quad y_2 = \frac{\sin 2\rho}{4\rho}, \quad \text{where } \rho = x_1^2 + x_2^2$$

The domain of the function is a square  $x_1, x_2 \in [-2, 2]$ .

In order to form the training set the functions are sampled on a regular  $16 \times 16$  grid. The relevant MATLAB code to form the matrices  $X$  and  $D$  follows:

```
na = 16; N = na^2; nn = 0:na-1; % Number of training cases
```

Specification of the domain of functions:

```
X1 = nn*4/na-2; % na points from -2 step (4/na)=.25
                    to (2 - 4/na)=1.75
[X1 X2] = meshgrid(X1); % coordinates of the grid vertices
                    X1 and X2 are na by na
R=(X1.^2+X2.^2+1e-5); % R (rho) is a matrix of squares of
                    distances of the grid vertices from the origin.
D1 = X1.*exp(-R); D = (D1(:))';
                    % D1 is na by na, D is 1 by N
D2 = 0.25*sin(2*R)./R ; D = [D ; (D2(:))']';
                    %D2 is na by na, D is a 2 by N matrix of 2-D target vectors
```

The domain sampling points are as follows:

```
X1=-2.00 -1.75 ... 1.50 1.75 X2=-2.00
-2.00 ... -2.00 -2.00
-2.00 -1.75 ... 1.50 1.75 -1.75 -1.75 ... -1.75 -1.75
. . . . . . . . . .
-2.00 -1.75 ... 1.50 1.75 1.50 1.50 ... 1.50 1.50
-2.00 -1.75 ... 1.50 1.75 1.75 1.75 ... 1.75 1.75
```

Scanning  $X1$  and  $X2$  column-wise and appending the bias inputs, we obtain the input matrix  $X$  which is  $p \times N$ :

```
X = [X1(:)'; X2(:)'; ones(1,N)];
```

The training exemplars are as follows:

X =	-2.0000	-2.0000	...	1.7500	1.7500
	-2.0000	-1.7500	...	1.5000	1.7500
	1.0000	1.0000	...	1.0000	1.0000
D =	-0.0007	-0.0017	...	0.0086	0.0038
	-0.0090	0.0354	...	-0.0439	-0.0127

The functions to be approximated are visualised in Figure 5–6. For illustration purposes they are plotted side-by-side, which distorts the domain which in reality is the same for both functions, namely,  $x_1, x_2 \in [-2, 2]$ .

```
surf([X1-2 X1+2], [X2 X2], [D1 D2])
```

Two 2-D target functions

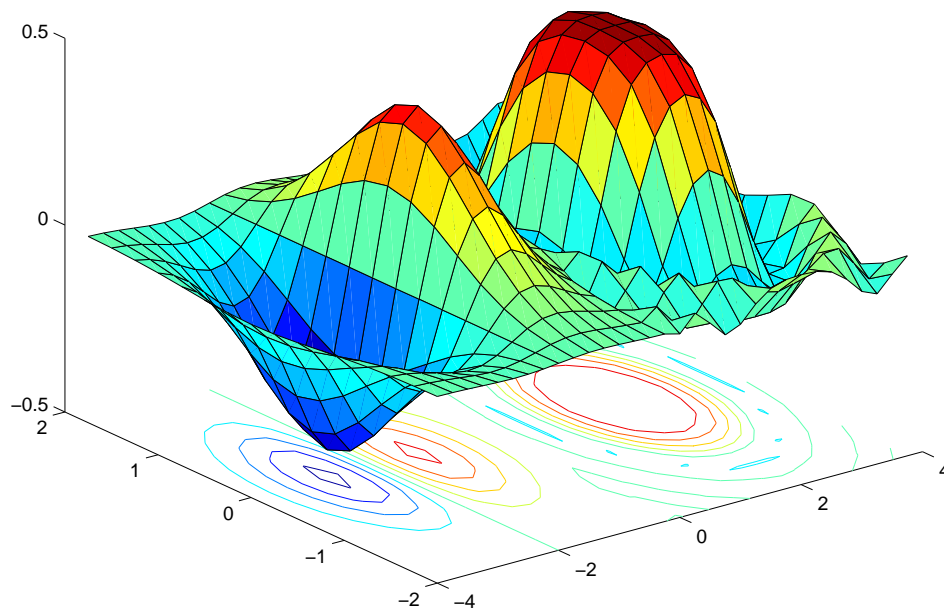


Figure 5–6: Two 2-D functions to be approximated

Random initialization of the weight matrices:

```
Wh = randn(L,p)/p;      the hidden-layer weight matrix  $W^h$  is  $L \times p$ 
Wy = randn(m,L)/L;      the output-layer weight matrix  $W^y$  is  $m \times L$ 

C = 200;                % maximum number of training epochs
J = zeros(m,C);         % Initialisation of the error function
eta = [0.003 0.1];      % Training gains
```

The main loop (fap2D.m):

```
for c = 1:C
```

The forward pass:

```
H = ones(L-1,N)./(1+exp(-Wh*X)); % Hidden signals (L-1 by N)
Hp = H.*(1-H);                  % Derivatives of hidden signals
H = [H ; ones(1,N)];           % bias signal appended
Y = tanh(Wy*H);                 % Output signals (m by N)
Yp = 1 - Y.^2;                 % Derivatives of output signals
```

The backward pass:

```
Ey = D - Y;                    % The output errors (m by K)
JJ = (sum((Ey.*Ey)'))';         % The total error after one epoch
                                % the performance function m by 1
dely = Ey.*Yp;                 % Output delta signal (m by K)
dWy = dely*H';                 % Update of the output matrix
                                % dWy is L by m
Eh = Wy(:,1:L-1)'*dely          % The back-propagated hidden error
                                % Eh is L-1 by N
delH = Eh.*Hp;                 % Hidden delta signals (L-1 by N)
dWh = delH*X';                 % Update of the hidden matrix
                                % dWh is L-1 by p
```

The batch update of the weights:

```
Wy = Wy+etay*dWy; Wh = Wh+etah*dWh;
```

Two 2-D approximated functions are plotted after each epoch. See Figure 5–7 for the final approximation.

```
D1(:)=Y(1,:)' ; D2(:)=Y(2,:)' ;
surf([X1-2 X1+2], [X2 X2], [D1 D2]) J(:,c) = JJ ;
end % of the training
```

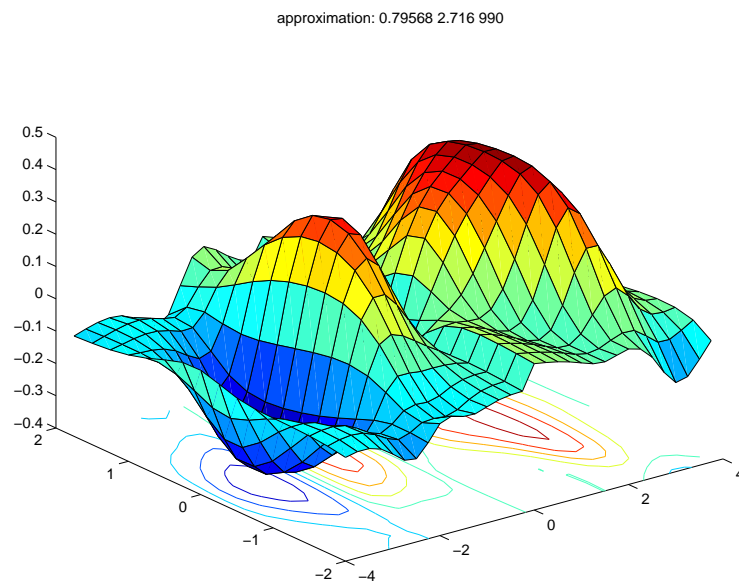


Figure 5–7: Final approximated functions

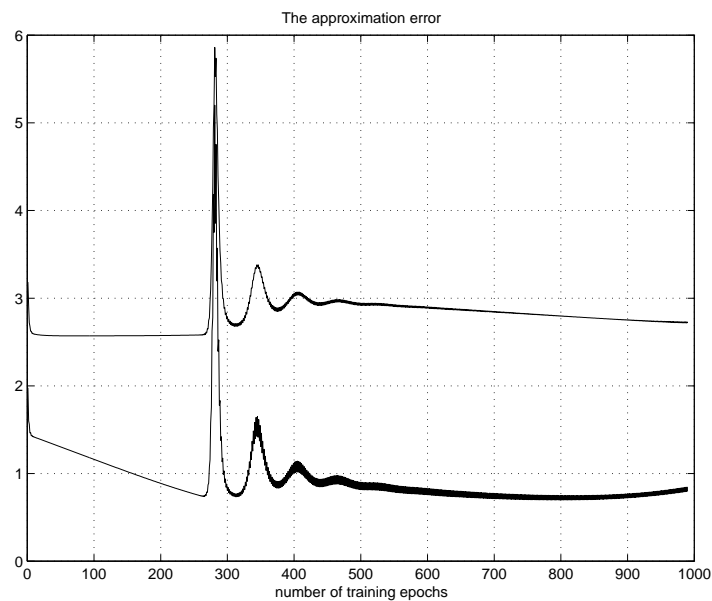


Figure 5–8: Training error for each function at the end of each epoch

The sum of squared errors at the end of each training epoch is collected in a  $2 \times C$  matrix and illustrated in Figure 5–8.



## References

- [1] Simon Haykin. *Neural Networks – a Comprehensive Foundation*. Prentice Hall, New Jersey, 2nd edition, 1999. ISBN 0-13-273350-1.
- [2] H. Demuth and M. Beale. *Neural Network Toolbox. For use with MATLAB. User's Guide*. The MathWorks Inc., 2002. \$MATLAB/help/pdf\_doc/nnet.pdf.
- [3] Martin T. Hagan, H Demuth, and M. Beale. *Neural Network Design*. PWS Publishing, 1996.
- [4] Hertz, Krogh, and Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley, 1991. ISBN 0-201-51560-1.
- [5] W.S. Sarle, editor. *Neural Network FAQ*. Newsgroup: comp.ai.neural-nets, 2002. URL: <ftp://ftp.sas.com/pub/neural/FAQ.html>.
- [6] Mohamad H. Hassoun. *Fundamentals of Artificial Neural Networks*. The MIT Press, 1995. ISBN 0-262-08239-X.