4 Feedforward Multilayer Neural Networks — part I

- Feedforward multilayer neural networks (introduced in sec. 1.7) with supervised error correcting learning are used to **approximate** (**synthesise**) a non-linear input-output mapping from a set of training patterns.

Consider the following mapping $f(X)$ from a p-dimensional domain X into an m-dimensional output space D:

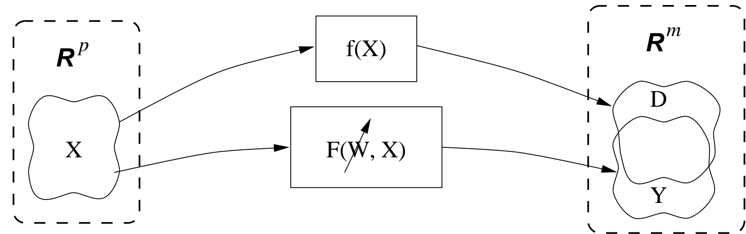


Figure 4–1: Mapping from a p-dimensional domain into an m-dimensional output space

- A function: $f : X \to D$, 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)

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

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

$$J = J(W, \|D - Y\|) \quad \text{is minimised.}$$

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 Σ_i are the centre and covariance matrix of the i-th RBF representing **adjustible parameters** (weights) of the network.

- **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:

$$F(W, \mathbf{x}) = \sigma(W_1 \cdot \ldots \sigma(W_l \cdot \mathbf{x}) \ldots)$$

  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.

4.1 Multilayer perceptrons (MLPs)

- 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 (4–2).

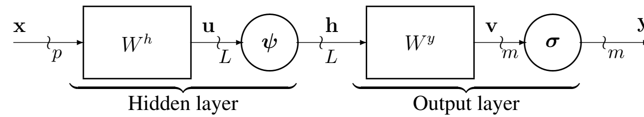


Figure 4–2: A block-diagram of a single-hidden-layer feedforward neural network

- The structure of each layer has been discussed in sec. 1.6.
- Nonlinear functions used in the hidden layer and in the output layer can be different.
- The output function can be linear.
- 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:

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

or simply as

$$\mathbf{y}(n) = \boldsymbol{\sigma}\left(W^y \cdot \boldsymbol{\psi}(W^h \cdot \mathbf{x}(n))\right) \tag{4.1}$$

- 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:

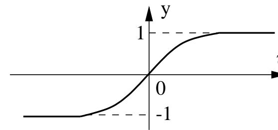
$$y = \sigma(v) = \frac{1}{1+e^{-\beta v}} = \frac{1}{2}(\tanh(\beta v/2) - 1)$$

The derivative of the unipolar sigmoidal function:

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

Sigmoidal bipolar:

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



The derivative of the bipolar sigmoidal function:

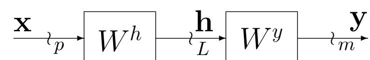
$$y' = \frac{d\sigma}{dv} = \frac{4\beta e^{2\beta v}}{(e^{2\beta v}+1)^2} = \beta(1-y^2)$$

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:

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

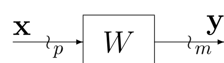
After substitution we have:

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

where

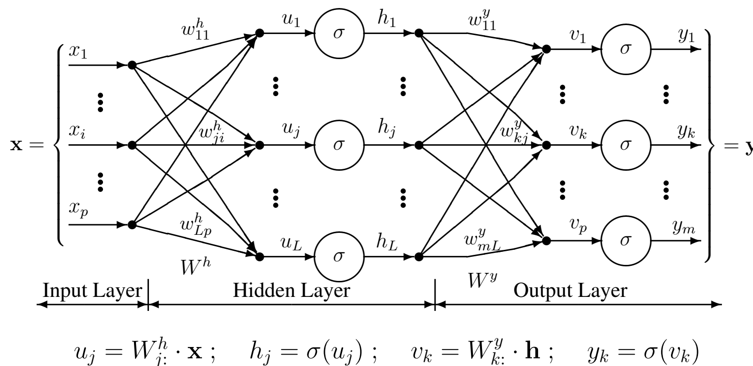
$$W = W^y \cdot W^h$$

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



4.2 Detailed structure of a Two-Layer Perceptron — the most commonly used feedforward neural network

Signal-flow diagram:



$$u_j = W^h_{j:} \cdot \mathbf{x}\ ; \quad h_j = \sigma(u_j)\ ; \quad v_k = W^y_{k:} \cdot \mathbf{h}\ ; \quad y_k = \sigma(v_k)$$

Dendritic diagram:

We can have:

$$x_p = 1$$

and possibly

$$h_L = 1$$

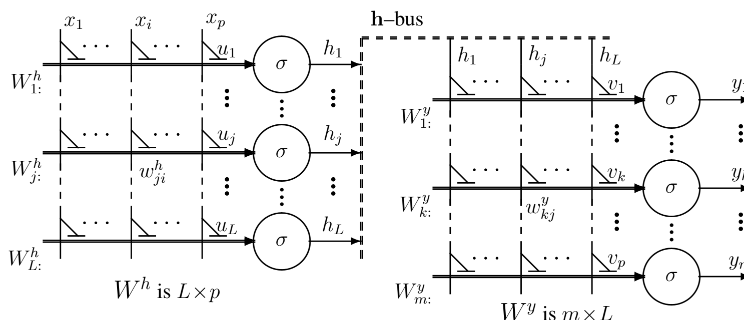
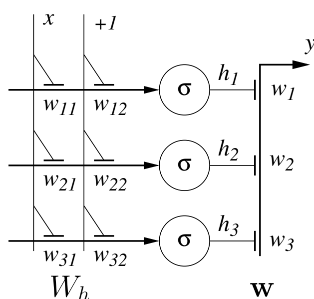
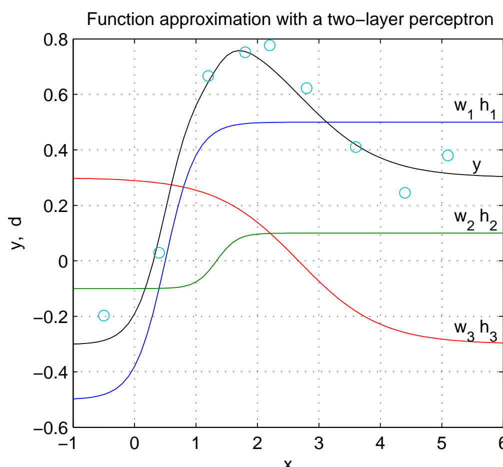


Figure 4–3: Various representations of a Two-Layer Perceptron

4.3 Example of function approximation with a two-layer perceptron

Consider a single-variable function approximated by the following two-layer perceptron:



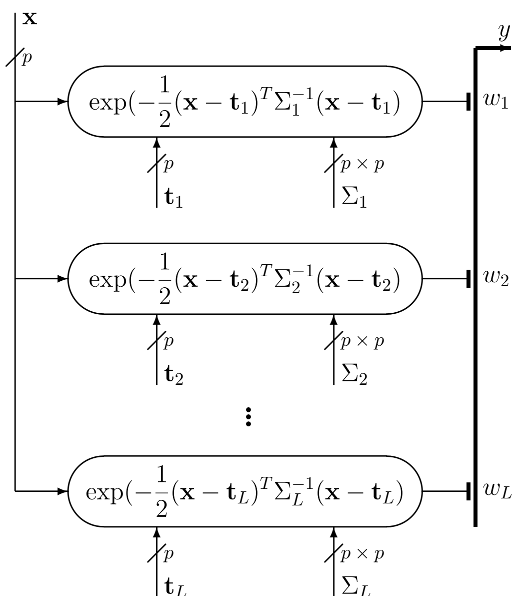


The neural net generates the following function:

$$\begin{aligned} y &= \mathbf{w} \cdot \mathbf{h} = \mathbf{w} \cdot \tanh\left(W_h \cdot \begin{bmatrix} x \\ 1 \end{bmatrix}\right) = w_1 \cdot h_1 + w_2 \cdot h_2 + w_3 \cdot h_3 \\ &= w_1 \cdot \tanh(w_{11} \cdot x + w_{12}) + w_2 \cdot \tanh(w_{21} \cdot x + w_{22}) + w_3 \cdot \tanh(w_{31} \cdot x + w_{32}) \\ &= 0.5 \cdot \tanh(2x - 1) + 0.1 \cdot \tanh(3x - 4) - 0.3 \cdot \tanh(0.75x - 2) \end{aligned}$$

4.4 Structure of a Gaussian Radial Basis Functions (RBF) Neural Network

An RBF neural network is similar to a two-layer perceptron with the linear output layer:



Let us consider a single output case, that is, approximation of a single function of p variables:

$$y = F(\mathbf{x}; \mathbf{t}_1, \dots, \mathbf{t}_L, \Sigma_1, \dots, \Sigma_L, \mathbf{w})$$
$$= \sum_{l=1}^{L} w_l \cdot \exp(-\frac{1}{2}(\mathbf{x} - \mathbf{t}_l)^T \Sigma_l^{-1} (\mathbf{x} - \mathbf{t}_l))$$

where

$\mathbf{t}_l$, p-element vectors, are the centers of the Gaussian functions, and

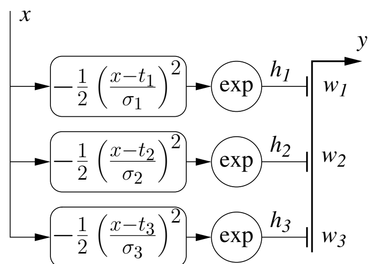
Σ_l, $p \times p$ covariance matrices, specified a shape of the Gaussian functions

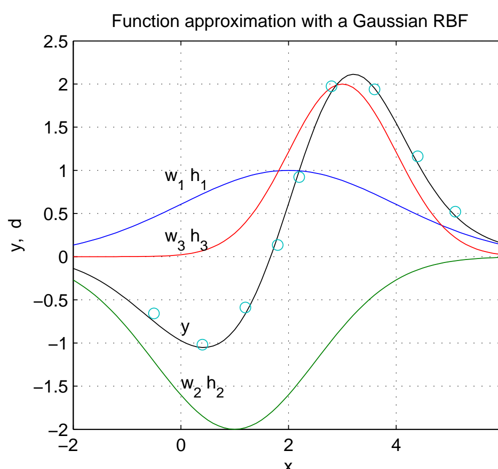
All parameters: $(\mathbf{t}_1, \dots, \mathbf{t}_L, \Sigma_1, \dots, \Sigma_L, \mathbf{w})$ are adjusted during learning procedure.

When the covariance matrix is diagonal, the axes of the Gaussian shape are aligned with the coordinate system axes.
If, in addition, all diagonal elements are identical, the Gaussian function is symmetrical in all directions.

4.5 Example of function approximation with a Gaussian RBF network

Consider a single-variable function approximated by the following Gaussian RBF network:





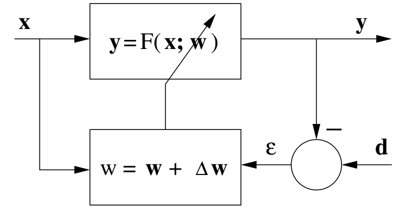
The neural net generates the following function:

$$y = \mathbf{w} \cdot \mathbf{h} = w_1 \cdot h_1 + w_2 \cdot h_2 + w_3 \cdot h_3$$
$$= w_1 \cdot \exp(-\frac{1}{2}\left(\frac{x - t_1}{\sigma_1}\right)^2) + w_2 \cdot \exp(-\frac{1}{2}\left(\frac{x - t_2}{\sigma_2}\right)^2) + w_3 \cdot \exp(-\frac{1}{2}\left(\frac{x - t_3}{\sigma_3}\right)^2)$$

4.6 Error-Correcting Learning Algorithms for Feedforward Neural Networks

Error-correcting learning algorithms are **supervised** training algorithms that modify the parameters of the network in such a way to minimise that error between the desired and actual outputs.

The vector $\mathbf{w}$ represents all the adjustable parameters of the network.



- Training data consists of N p-dimensional vectors $\mathbf{x}(n)$, and N m-dimensional desired output vectors, $\mathbf{d}(n)$, that are organized in two matrices:

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

- For each input vector, $\mathbf{x}(n)$, the network calculates the actual output vector, $\mathbf{y}(n)$, as

$$\mathbf{y}(n) = F(\mathbf{x}(n); \mathbf{w}(n))$$

- The output vector is compared with the desired output $\mathbf{d}(n)$ and the error is calculated:

$$\boldsymbol{\varepsilon}(n) = [\varepsilon_1(n) \ldots \varepsilon_m(n)]^T = \mathbf{d}(n) - \mathbf{y}(n) \quad \text{is an } m \times 1 \text{ vector,} \tag{4.2}$$

- In a pattern training algorithm, at each step the weight vector is updated

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \Delta\mathbf{w}(n)$$

so that the total error is minimised.

- More specifically, we try to minimise a **performance index**, typically the **mean-squared error (mse)**, $J(\mathbf{w})$, specified as an averaged sum of instantaneous squared errors at the network output:

$$J(\mathbf{w}) = \frac{1}{m\,N} \sum_{n=1}^{N} E(\mathbf{w}(n)) \tag{4.3}$$

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

$$E(\mathbf{w}(n)) = \frac{1}{2} \sum_{k=1}^{m} \varepsilon_k^2(n) = \frac{1}{2}\, \boldsymbol{\varepsilon}^T(n) \cdot \boldsymbol{\varepsilon}(n) \tag{4.4}$$

and $\boldsymbol{\varepsilon}$ is an $m \times 1$ vector of instantaneous errors as in eqn (4.2).

- To consider possible minimization algorithm we can expand $J(\mathbf{w})$ into the Taylor power series:

$$\begin{aligned} J(\mathbf{w}(n+1)) = J(\mathbf{w} + \Delta\mathbf{w}) &= J(\mathbf{w}) + \Delta\mathbf{w} \cdot \nabla J(\mathbf{w}) + \frac{1}{2}\Delta\mathbf{w} \cdot \nabla^2 J(\mathbf{w}) \cdot \Delta\mathbf{w}^T + \cdots \\ &= J(\mathbf{w}) + \Delta\mathbf{w} \cdot \mathbf{g}(\mathbf{w}) + \frac{1}{2}\Delta\mathbf{w} \cdot H(\mathbf{w}) \cdot \Delta\mathbf{w}^T + \cdots \end{aligned} \tag{4.5}$$

(n) has been omitted for brevity.

where: $\nabla J(\mathbf{w}) = \mathbf{g}$ is the gradient vector of the performance index,

$\nabla^2 J(\mathbf{w}) = H$ is the Hessian matrix (matrix of second derivatives).

- As for the Adaline, we infer that in order for the performance index to be reduced, that is

$$J(\mathbf{w} + \Delta\mathbf{w}) < J(\mathbf{w})$$

the following condition must be satisfied:

$$\Delta\mathbf{w} \cdot \nabla J(\mathbf{w}) < 0$$

where the higher order terms in the expansion (4.5) have been ignored.

- This condition describes the **steepest descent method** in which the weight vector is modified in the direction opposite to the gradient vector:

$$\Delta\mathbf{w} = -\eta\, \nabla J(\mathbf{w}) \tag{4.6}$$

- However, the gradient of the total error is equal to the sum of its components:

$$\nabla J(\mathbf{w}) = \frac{1}{m\,N} \sum_{n=1}^{N} \nabla E(\mathbf{w}(n)) \tag{4.7}$$

- Therefore, in the pattern training, at each step the weight vector can be modified in the direction opposite to the **gradient of the total instantaneous squared error**:

$$\Delta\mathbf{w}(n) = -\eta\, \nabla E(\mathbf{w}(n)) \tag{4.8}$$

- The **gradient of the total instantaneous squared error (tise)** is a K-component vector, where K is the total number of weights, of all partial derivatives of $E(\mathbf{w})$ with respect to $\mathbf{w}$:

$$\nabla E(\mathbf{w}) = \frac{\partial E}{\partial \mathbf{w}} = \left[\frac{\partial E}{\partial w_1} \cdots \frac{\partial E}{\partial w_K} \right]$$

(n) has been omitted for brevity.

- Using eqns (4.4) and (4.2) we have

$$\nabla E(\mathbf{w}) = \frac{\partial E}{\partial \mathbf{w}} = \boldsymbol{\varepsilon}^T \cdot \frac{\partial \boldsymbol{\varepsilon}}{\partial \mathbf{w}} = -\boldsymbol{\varepsilon}^T \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{w}} \tag{4.9}$$

where

$\frac{\partial \mathbf{y}}{\partial \mathbf{w}} = \left\{ \frac{\partial y_j}{\partial w_k} \right\}_{m \times K}$ is a $m \times K$ matrix of all derivatives $\frac{\partial y_j}{\partial w_k}$ (**Jacobian** matrix).

- Details of calculations of the gradient of the total instantaneous squared error (gradient of tise), in particular the Jacobian matrix will be different for a specific type of neural nets, that is, for a specific mapping function

$$\mathbf{y} = F(\mathbf{x}; \mathbf{w})$$

4.7 Steepest Descent Backpropagation Learning Algorithm for a Multi-Layer Perceptron

- The **steepest descent backpropagation** learning algorithm is the simplest **error-correcting algorithms** for a multi-layer perceptron.
- For simplicity we will derive it for a two-layer perceptron as in Figure 4–3 where the vector of output signals is calculated as:

$$\mathbf{y} = \boldsymbol{\sigma}(\mathbf{v}) \; ; \;\; \mathbf{v} = W^y \cdot \mathbf{h} \; ; \;\; \mathbf{h} = \boldsymbol{\psi}(W^h \cdot \mathbf{x}) \tag{4.10}$$

- We start with re-shaping two weight matrices into one weight vector of the following structure:

$$\mathbf{w} = \mathrm{scan}(W^h, W^y) = [\underbrace{w^h_{11} \ldots w^h_{1p} \ldots w^h_{Lp}}_{\text{hidden weights}} \,|\, \underbrace{w^y_{11} \ldots w^y_{1L} \ldots w^y_{mL}}_{\text{output weights}}] = \left[\; \mathbf{w}^h \;\; \mathbf{w}^y \;\right]$$

The size of $\mathbf{w}^h$ is $L \cdot p$ and $\mathbf{w}^y$ is $m \cdot L$, total number of weights being equal to $K = L(p + m)$.

- Now, the **gradient of tise** has components associated with the hidden layer weights, W^h, and the output layer weights, W^y, which are arranged in the following way:

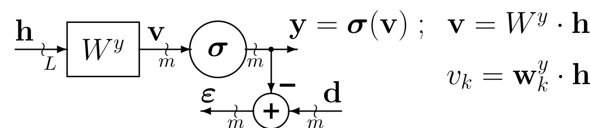
$$\nabla E(\mathbf{w}) = \left[\frac{\partial E}{\partial \mathbf{w}^h} \;\; \frac{\partial E}{\partial \mathbf{w}^y}\right] = \left[\frac{\partial E}{\partial w^h_{11}} \cdots \frac{\partial E}{\partial w^h_{ji}} \cdots \frac{\partial E}{\partial w^h_{Lp}} \; \frac{\partial E}{\partial w^y_{11}} \cdots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

- Using eqns (4.9) and (4.10) the gradient of **tise** can be further expressed as:

$$\nabla E(\mathbf{w}) = \frac{\partial E}{\partial \mathbf{w}} = -\boldsymbol{\varepsilon}^T \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{w}} = -\boldsymbol{\varepsilon}^T \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{v}} \cdot \frac{\partial \mathbf{v}}{\partial \mathbf{w}} \quad \text{(the chain rule)} \tag{4.11}$$

4.7.1 Output layer

- The vector of the output signals is calculated as:



$$\mathbf{y} = \boldsymbol{\sigma}(\mathbf{v}) \; ; \;\; \mathbf{v} = W^y \cdot \mathbf{h}$$
$$v_k = \mathbf{w}^y_k \cdot \mathbf{h}$$

- The first Jacobian matrix of eqn (4.11) can be evaluated as follows:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{v}} = \mathrm{diag}(\boldsymbol{\sigma}') = \mathrm{diag}(\left[\; \frac{\partial y_1}{\partial v_1} \;\; \cdots \;\; \frac{\partial y_m}{\partial v_m} \;\right]) \tag{4.12}$$

which is a diagonal matrix of derivatives of the activation function $\sigma'_k = \dfrac{\partial y_k}{\partial v_k}$.

- The products of errors ε_k and derivatives of the activation function, σ'_k are known as the **delta errors**:

$$\boldsymbol{\delta}^T = \boldsymbol{\varepsilon}^T \cdot \mathrm{diag}(\boldsymbol{\sigma}') = \left[\; \varepsilon_1 \cdot \sigma'_1 \;\; \cdots \;\; \varepsilon_m \cdot \sigma'_m \;\right] \tag{4.13}$$

is a vector of **delta errors**.

- Substituting eqns (4.13) and (4.12) into eqn (4.11) the gradient of **tise** can be expressed as:

$$\nabla E(\mathbf{w}) = \frac{\partial E}{\partial \mathbf{w}} = -\boldsymbol{\delta}^T \cdot \frac{\partial \mathbf{v}}{\partial \mathbf{w}} \tag{4.14}$$

- Now we will separately calculate the output and hidden sections of the gradient of **tise**, namely, $\dfrac{\partial E}{\partial \mathbf{w}^h}$ and $\dfrac{\partial E}{\partial \mathbf{w}^y}$
- From eqn. (4.14) the output section of the gradient can be written as:

$$\frac{\partial E}{\partial \mathbf{w}^y} = -\boldsymbol{\delta}^T \cdot \frac{\partial \mathbf{v}}{\partial \mathbf{w}^y} \tag{4.15}$$

- In order to evaluate the Jacobian $\dfrac{\partial \mathbf{v}}{\partial \mathbf{w}^y}$ note first that

$$\text{since} \quad \mathbf{v} = W^y \cdot \mathbf{h} \quad \text{than} \quad \frac{\partial \mathbf{v}}{\partial W^y} = \mathbf{h}^T$$

- Therefore, after scanning W^y row-wise into $\mathbf{w}^y$ we have

$$\frac{\partial \mathbf{v}}{\partial \mathbf{w}^y} = \begin{bmatrix} \mathbf{h}^T & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \mathbf{h}^T \end{bmatrix} = I \otimes \mathbf{h}^T \tag{4.16}$$

where $\otimes$ denotes the Kronecker product and I is an $m \times m$ identity matrix.

- Combining eqns (4.15) and (4.16), we finally have:

$$\frac{\partial E}{\partial \mathbf{w}^y} = -\boldsymbol{\delta}^T \cdot (I \otimes \mathbf{h}^T) \quad \text{or} \quad \frac{\partial E}{\partial W^y} = -\boldsymbol{\delta} \cdot \mathbf{h}^T \tag{4.17}$$

- Hence, if we reshape the weight vector $\mathbf{w}^y$ back into a weight matrix W^y, the **gradient of the total instantaneous squared error** can be expressed as an **outer product** of $\boldsymbol{\delta}$ and $\mathbf{h}$ vectors.

- Using eqn (4.8), the **steepest descent pattern training** algorithm for minimisation of the performance index with respect to the output weight matrix

$$\Delta W^y(n) = -\eta_y \frac{\partial E(n)}{\partial W^y(n)} = \eta_y \cdot \boldsymbol{\delta}(n) \cdot \mathbf{h}^T(n)\ ; \quad W^y(n+1) = W^y(n) + \Delta W^y(n) \tag{4.18}$$

- In the **batch training** mode 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 gradients of the total instantaneous squared errors:

$$\frac{\partial J}{\partial W^y} = \frac{1}{m\,N} \sum_{n=1}^{N} \frac{\partial E(n)}{\partial W^y(n)} = -\frac{1}{m\,N} \sum_{n=1}^{N} \boldsymbol{\delta}(n) \cdot \mathbf{h}^T(n) \tag{4.19}$$

- If we take into account that the sum of outer products can be replaced by a product of matrices collecting the contributing vectors, then the gradient can be written as:

$$\frac{\partial J}{\partial W^y} = -\frac{1}{m\,N} S \cdot H^T \tag{4.20}$$

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

$$S = [\boldsymbol{\delta}(1) \ \ldots \ \boldsymbol{\delta}(N)] \tag{4.21}$$

and H is the $L \times N$ matrix of the hidden signals:

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

- Therefore, in the **batch training** LMS algorithm, the weight update after k-th epoch can be written as:

$$\Delta W^y(k) = -\eta_y \frac{\partial J}{\partial W^y} = \eta_y \cdot S(k) \cdot H^T(k)\ ; \quad W^y(k+1) = W^y(k) + \Delta W^y(k) \tag{4.22}$$

4.7.2 Hidden layer

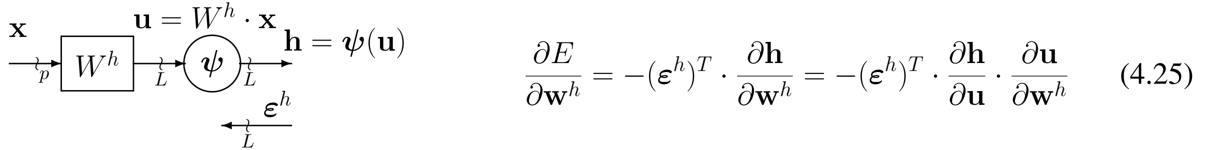
- As for the output layer we start with eqn (4.14) which for the hidden weight can be re-written in a form analogous to eqn (4.15) and then, using a chain rule of differentiation, we can write:

$$\frac{\partial E}{\partial \mathbf{w}^h} = -\boldsymbol{\delta}^T \cdot \frac{\partial \mathbf{v}}{\partial \mathbf{w}^h} = -\boldsymbol{\delta}^T \cdot \frac{\partial \mathbf{v}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{w}^h} \tag{4.23}$$

- Since $\mathbf{v} = W^y \cdot \mathbf{h}$ we have $\frac{\partial \mathbf{v}}{\partial \mathbf{h}} = W^y$ and we can define the **backpropagated error** as:

$$\boldsymbol{\varepsilon}^h = (W^y)^T \cdot \boldsymbol{\delta} \tag{4.24}$$

- The gradient of **tise** with respect to hidden weights can be now written as:



$$\frac{\partial E}{\partial \mathbf{w}^h} = -(\boldsymbol{\varepsilon}^h)^T \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{w}^h} = -(\boldsymbol{\varepsilon}^h)^T \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{u}} \cdot \frac{\partial \mathbf{u}}{\partial \mathbf{w}^h} \tag{4.25}$$

which is structurally identical to eqn (4.9), $\mathbf{w}$, $\boldsymbol{\varepsilon}$ and $\mathbf{y}$ being replaced by $\mathbf{w}^h$, $\boldsymbol{\varepsilon}^h$ and $\mathbf{h}$, respectively.

- Following eqn (4.13) we can define the **backpropagated delta errors**:

$$(\boldsymbol{\delta}^h)^T = (\boldsymbol{\varepsilon}^h)^T \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{u}} = (\boldsymbol{\varepsilon}^h)^T \cdot \text{diag}(\boldsymbol{\psi}') = \left[\, \varepsilon_1^h \cdot \psi_1' \;\; \cdots \;\; \varepsilon_L^h \cdot \psi_L' \,\right] \; ; \;\; \psi_j' = \frac{\partial h_j}{\partial u_j} \tag{4.26}$$

- Now, the gradient of **tise** can be expressed in a form analogous to eqn (4.15):

$$\frac{\partial E}{\partial \mathbf{w}^h} = -(\boldsymbol{\delta}^h)^T \cdot \frac{\partial \mathbf{v}}{\partial \mathbf{w}^h} \tag{4.27}$$

- Finally we have:

$$\frac{\partial E}{\partial \mathbf{w}^h} = -(\boldsymbol{\delta}^h)^T \cdot (I \otimes \mathbf{x}^T) \quad \text{or} \quad \frac{\partial E}{\partial W^h} = -\boldsymbol{\delta}^h \cdot \mathbf{x}^T \tag{4.28}$$

- Hence, if we reshape the weight vector $\mathbf{w}^h$ back into a weight matrix W^h, the **gradient of the total instantaneous squared error** can be expressed as an **outer product** of $\boldsymbol{\delta}^h$ and $\mathbf{x}$ vectors.
- Therefore, for the **pattern training** algorithm, the update of the hidden weight matrix for the n–the training pattern now becomes:

$$\Delta W^h(n) = -\eta_h \frac{\partial E(n)}{\partial W^h(n)} = \eta_h \cdot \boldsymbol{\delta}^h(n) \cdot \mathbf{x}^T(n) \; ; \;\; W^h(n+1) = W^h(n) + \Delta W^h(n) \tag{4.29}$$

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

- For the **batch training**, 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:

$$\frac{\partial J}{\partial W^h} = \frac{1}{m\,N} \sum_{n=1}^{N} \frac{\partial E(n)}{\partial W^h(n)} = -\frac{1}{m\,N} \sum_{n=1}^{N} \boldsymbol{\delta}^h(n) \cdot \mathbf{x}^T(n) \tag{4.30}$$

- 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

$$\frac{\partial J}{\partial W^y} = -\frac{1}{m\,N} S^h \cdot X^T \tag{4.31}$$

where S^h is the $L \times N$ matrix of hidden delta errors: $S^h = \left[\boldsymbol{\delta}^h(1)\ \dots\ \boldsymbol{\delta}^h(N)\right]$ (4.32)

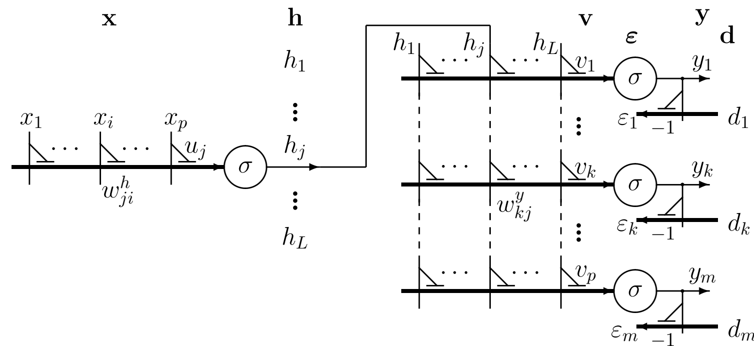
and X is the $p \times N$ matrix of the input signals.

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

$$\Delta W^h(k) = -\eta_h \frac{\partial J}{\partial W^h} = \eta_h \cdot S^h(k) \cdot X^T \ ; \quad W^h(k+1) = W^h(k) + \Delta W^h(k) \tag{4.33}$$

4.7.3 Alternative derivation

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



- Good to remember

$$\begin{aligned}
E &= \frac{1}{2} \sum_{k=1}^{m} \varepsilon_k^2\ , \quad \varepsilon_k = d_k - y_k \\
y_k &= \sigma(\mathbf{w}_k^y \cdot \mathbf{h}) = \sigma(\sum_{j=1}^{L} w_{kj}^y \cdot h_j) \\
h_j &= \sigma(\mathbf{w}_j^h \cdot \mathbf{x}) = \sigma(\sum_{i=1}^{p} w_{ji}^y \cdot x_i)
\end{aligned}$$

The gradient component related to a synaptic weight w^y_{kj} for the n–th training vector can be calculated as follows:

$$\begin{aligned}
\frac{\partial E(n)}{\partial w^y_{kj}} &= -\varepsilon_k \frac{\partial y_k}{\partial w^y_{kj}} && E = \tfrac{1}{2}(\ldots + \varepsilon_k^2 + \ldots)\ ,\ y_k = \sigma(v_k) \\
&= -\varepsilon_k \frac{\partial y_k}{\partial v_k}\frac{\partial v_k}{\partial w^y_{kj}} && v_k = W^y_{k:} \cdot \mathbf{h} = \ldots + w^y_{kj} h_j + \ldots \\
&= -\varepsilon_k \cdot \sigma'_k \cdot h_j && \sigma'_k = \frac{\partial y_k}{\partial v_k} \\
&= -\delta_k \cdot h_j && \delta_k = \varepsilon_k \cdot \sigma'_k
\end{aligned}$$

where the **delta error**, δ_k, is the output error, ε_k, modified with the **derivative of the activation function**, σ'_k.

Alternatively, the gradient components related to the complete weight vector, $W^y_{k:}$ of the kth output neuron can be calculated as:

$$\begin{aligned}
\frac{\partial E(n)}{\partial W^y_{k:}} &= -\varepsilon_k \frac{\partial y_k}{\partial W^y_{k:}} && y_k = \sigma(v_k) \\
&= -\varepsilon_k \frac{\partial y_k}{\partial v_k}\frac{\partial v_k}{\partial W^y_{k:}} && v_k = W^y_{k:} \cdot \mathbf{h} \\
&= -\varepsilon_k \cdot \sigma'_k \cdot \mathbf{h}^T \\
&= -\delta_k \cdot \mathbf{h}^T
\end{aligned}$$

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

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:

$$\frac{\partial E(n)}{\partial W^y} = -\boldsymbol{\delta}(n) \cdot \mathbf{h}^T(n) \qquad (4.34)$$

where

$$\boldsymbol{\delta} = \begin{bmatrix} \delta_1 \\ \vdots \\ \delta_m \end{bmatrix} = \begin{bmatrix} \varepsilon_1 \cdot \sigma'_1 \\ \vdots \\ \varepsilon_m \cdot \sigma'_m \end{bmatrix} = \boldsymbol{\varepsilon} \odot \boldsymbol{\sigma}'\ , \qquad (4.35)$$

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

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

- Gradient components related to the weight matrix of the hidden layer:

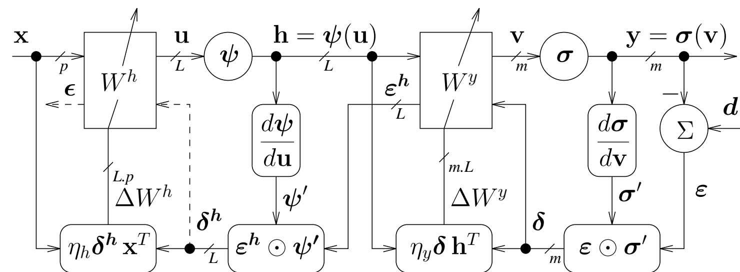
$$\begin{aligned}
\frac{\partial E(n)}{\partial w^h_{ji}} &= -\sum_{k=1}^{m} \varepsilon_k \frac{\partial y_k}{\partial w^h_{ji}} && y_k = \sigma(v_k) \\
&= -\sum_{k=1}^{m} \varepsilon_k \frac{\partial y_k}{\partial v_k}\frac{\partial v_k}{\partial w^h_{ji}} && v_k = \ldots + w^y_{kj} h_j + \ldots \\
&= -(\sum_{k=1}^{m} \delta_k\, w^y_{kj}) \frac{\partial h_j}{\partial w^h_{ji}} && \delta_k = \varepsilon_k \cdot \sigma'_k \\
&= -\, W^{yT}_{:j}\ \boldsymbol{\delta}\ \frac{\partial h_j}{\partial w^h_{ji}} && W^{yT}_{:j} \boldsymbol{\delta} = \boldsymbol{\delta}^{\boldsymbol{T}} W^y_{:j} = \sum_{k=1}^{m} \delta_k\, w^y_{kj} = \varepsilon^h_j
\end{aligned}$$

- Note, that the j–th column of the output weight matrix, $W^y_{:j}$, is used to modify the delta errors to create the **equivalent hidden layer errors**, known as **back-propagated errors** which are specified as follows:
$$\varepsilon^h_j = W^{yT}_{:j} \cdot \boldsymbol{\delta} \;, \quad \text{for} \quad j = 1, \ldots, L$$
- Using the back-propagated error, we can now repeat the steps performed for the output layer, with ε^h_j and h_j replacing ε_k and y_k, respectively.
$$\begin{aligned} \frac{\partial E(n)}{\partial w^h_{ji}} &= -\,\varepsilon^h_j \frac{\partial h_j}{\partial w^h_{ji}} && \Bigg\| \; h_j = \psi(u_j)\;,\; u_j = W^h_{j:} \cdot \mathbf{x} \\ &= -\,\varepsilon^h_j \cdot \psi'_j \cdot x_i && \Bigg\| \; \psi'_j = \frac{\partial \psi_j}{\partial u_j} \\ &= -\,\delta^h_j \cdot x_i && \Bigg\| \; \delta^h_j = \varepsilon^h_j \cdot \psi'_j \end{aligned}$$
where the back-propagated error has been used to generate the delta-error for the hidden layer, δ^h_j.
- 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 (4.34):

$$\frac{\partial E(n)}{\partial W^h} = -\,\boldsymbol{\delta}^h \cdot \mathbf{x}^T \;,\; \text{where} \;\; \boldsymbol{\delta}^h = \begin{bmatrix} \varepsilon^h_1 \cdot \psi'_1 \\ \vdots \\ \varepsilon^h_L \cdot \psi'_L \end{bmatrix} = \boldsymbol{\varepsilon}^h \odot \boldsymbol{\psi}' \;\; \text{and} \;\; \boldsymbol{\varepsilon}^h = W^{yT} \cdot \boldsymbol{\delta} \qquad (4.36)$$

4.7.4 The structure of the two-layer back-propagation network with learning

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 $\mathbf{x}$, through $\mathbf{u}$, $\mathbf{h}$, $\boldsymbol{\psi}'$, $\mathbf{v}$ to $\mathbf{y}$ and $\boldsymbol{\sigma}'$.

  backward pass in which the signals of the learning part are determined starting from $\mathbf{d}$, through $\boldsymbol{\varepsilon}$, $\boldsymbol{\delta}$, ΔW^y, $\boldsymbol{\varepsilon^h}$, $\boldsymbol{\delta^h}$ and ΔW^h.
- From Figure 4.7.4 and the relevant equations note that, in general, the weight update is proportional to the synaptic input signals ($\boldsymbol{x}$, or $\boldsymbol{h}$) and the delta signals ($\boldsymbol{\delta^h}$, or $\boldsymbol{\delta}$).
- The delta signals, in turn, are proportional to the derivatives the activation functions, $\boldsymbol{\psi}'$, or $\boldsymbol{\sigma}'$.

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 (4.3) and (4.4).
- 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 3–21.
- 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

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

  where n is the pattern index.

  batch mode

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

  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 in sec. 5.4 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 $\sigma' = 0$ 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.

4.8 Example of function approximation (fap2D.m)

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

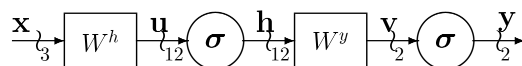
$$\mathbf{y} = \mathbf{f}(\mathbf{x}) \ , \ \text{or} \ y_1 = f_1(x_1, x_2) \ , \ \ y_2 = f_2(x_1, x_2)$$

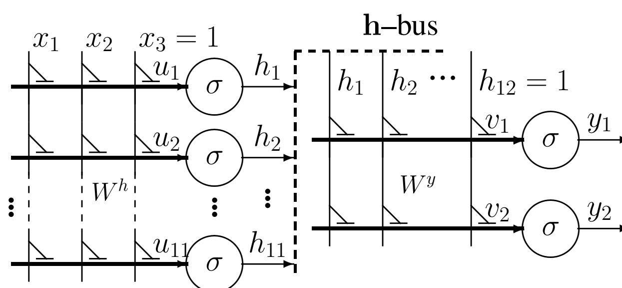
using a two-layer perceptron,

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

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.
Specification of the the neural network (fap2Di.m):

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





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

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

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 16×16 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^2) 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)];
```

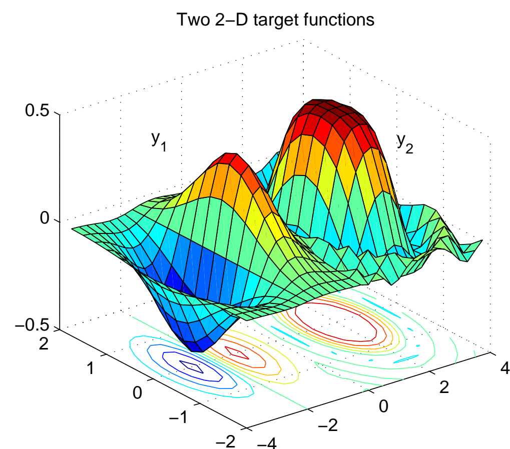
The training exemplars are as follows:

```
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 plotted side-by-side, which distorts the domain which in reality is the same for both functions, namely, $x_1, x_2 \in [-2,\ 2)$.

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



Random initialization of the weight matrices:

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

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

```
for c = 1:C     % The main loop (fap2D.m)
```

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:

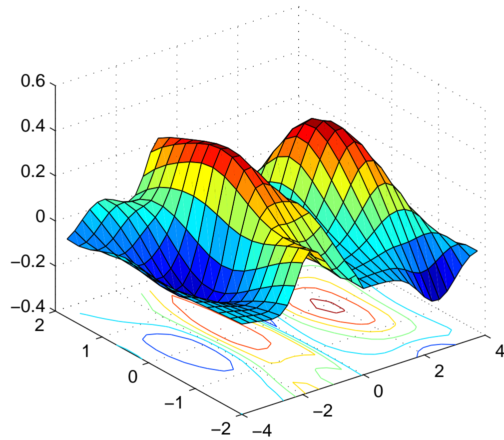
```
   Ey = D - Y;              % The output errors       (m by K)
   JJ = (sum((Ey.*Ey)'))'; % The total error after one epoch
                            % the performance function  m by 1
   delY = Ey.*Yp;           % Output delta signal     (m by K)
   dWy = delY*H';           % Update of the output 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

   Wy = Wy+etay*dWy;  Wh = Wh+etah*dWh; % The batch update of the weights
```

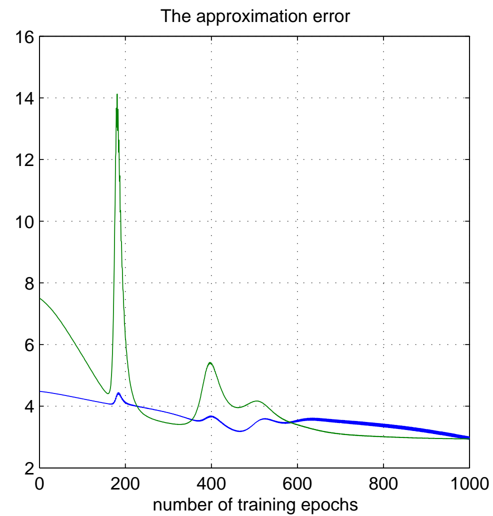
Two 2-D approximated functions are plotted after each epoch.

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

Approximation after 1000 epochs:



The sum of squared errors at the end of each training epoch is collected in a $2 \times C$ matrix. The approximation error for each function at the end of each epoch:



From this example you will note that the backpropagation algorithm is

- painfully slow
- sensitive to the weight initialization
- sensitive to the training gains.

We will address these problems in the subsequent sections.

It is good to know that the best training algorithm can be **two orders of magnitude faster** that a basic backpropagation algorithm.