1 Basic concepts of Neural Networks and Fuzzy Logic Systems

Inspirations based on course material by Professors Heikki Koiovo http://www.control.hut.fi/Kurssit/AS-74.115/Material/ and David S. Touretzki http://www-2.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15782-s04/slides/ are duly acknowledged

Neural networks and Fuzzy Logic Systems are often considered as a part of **Soft Computing** area:

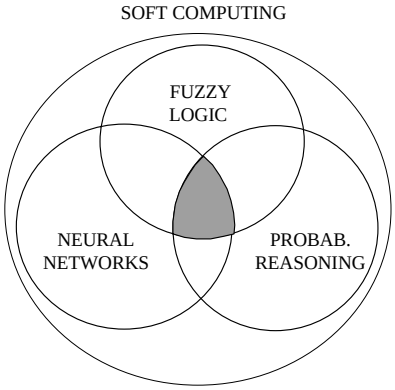


Figure 1–1: Soft computing as a composition of fuzzy logic, neural networks and probabilistic reasoning.

Intersections include

- neuro-fuzzy systems and techniques,
- probabilistic approaches to neural networks (especially classification networks) and fuzzy logic systems,
- and Bayesian reasoning.

Neuro-Fuzzy systems

- We may say that neural networks and fuzzy systems try to emulate the operation of human brain.
- Neural networks concentrate on the structure of human brain, i.e., on the "hardware" emulating the basic functions, whereas fuzzy logic systems concentrate on "software", emulating fuzzy and symbolic reasoning.

Neuro-Fuzzy approach has a number of different connotations:

- The term "Neuro-Fuzzy" can be associated with hybrid systems which act on two distinct subproblems: a neural network is utilized in the first subproblem (e.g., in signal processing) and a fuzzy logic system is utilized in the second subproblem (e.g., in reasoning task).
- We will also consider system where both methods are closely coupled as in the Adaptive Neuro-Fuzzy Inference Systems (anfis)

Neural Network and Fuzzy System research is divided into two basic schools

- Modelling various aspects of human brain (structure, reasoning, learning, perception, etc)
- Modelling artificial systems and related data: pattern clustering and recognition, function approximation, system parameter estimation, etc.

Neural network research in particular can be divided into

- Computational Neuroscience
  - Understanding and modelling operations of single neurons or small neuronal circuits, e.g. minicolumns.
  - Modelling information processing in actual brain systems, e.g. auditory tract.
  - Modelling human perception and cognition.
- Artificial Neural Networks used in
  - Pattern recognition
  - adaptive control
  - time series prediction, etc.

Areas contributing to Artificial neural networks:

- Statistical Pattern recognition
- Computational Learning Theory
- Computational Neuroscience
- Dynamical systems theory
- Nonlinear optimisation

We can say that in general

- Neural networks and fuzzy logic systems are parameterised computational nonlinear algorithms for numerical processing of data (signals, images, stimuli).
- These algorithms can be either implemented of a general-purpose computer or built into a dedicated hardware.
- Knowledge is acquired by the network/system through a learning process.
- The acquired knowledge is stored in internal parameters (weights).

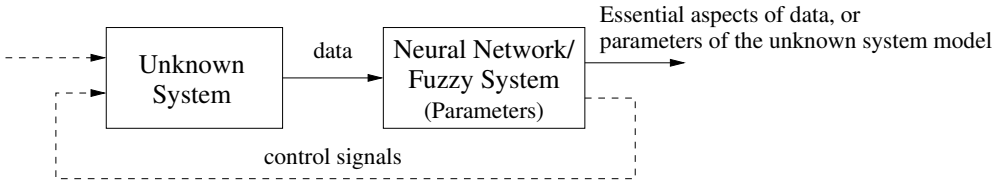
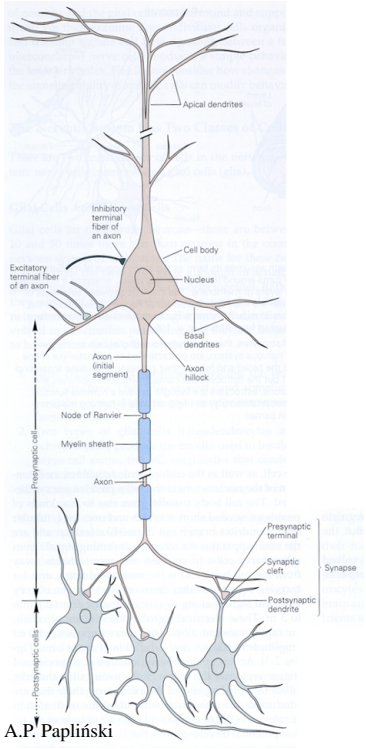


Figure 1–2: System as a data generator

1.1 Biological Fundamentals of Neural Networks

from Kandel, Schwartz and Jessel, *Principles of Neural Science*

A typical neuron, or nerve cell, has the following structure:

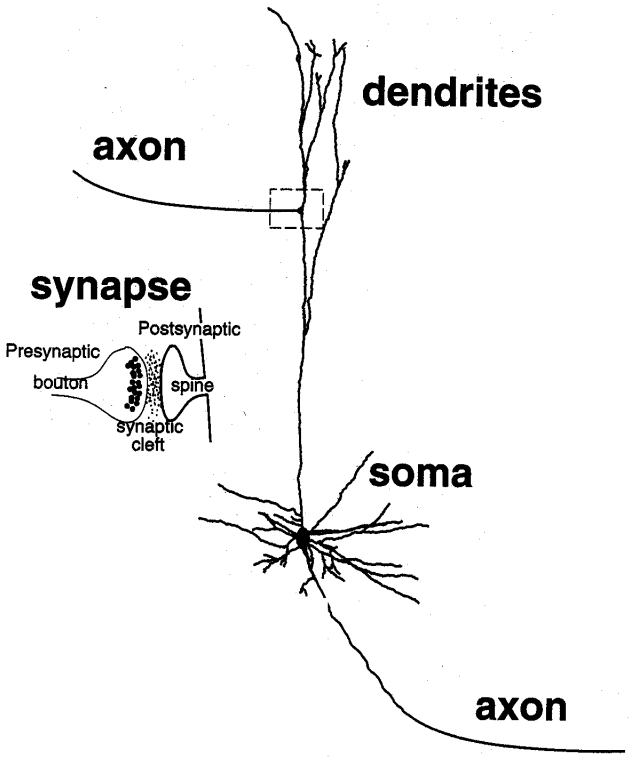


- Most neurons in the vertebrate nervous system have several main features in common.
- The **cell body** contains the nucleus, the storehouse of genetic information, and gives rise to two types of cell processes, **axons** and **dendrites**.
- Axons, the transmitting element of neurons, can vary greatly in length; some can extend more than 3m within the body. Most axons in the central nervous system are very thin ($0.2 \ldots 20$ μm in diameter) compared with the diameter of the cell body (50 μm or more).
- Many axons are insulated by a fatty sheath of myelin that is interrupted at regular intervals by the nodes of Ranvier.
- The **action potential**, the cell's conducting signal, is initiated either at the **axon hillock**, the initial segment of the axon, or in some cases slightly farther down the axon at the first nod of Ranvier.
- Branches of the axon of one neuron (the presynaptic neuron) transmit signals to another neuron (the postsynaptic cell) at a site called the **synapse**.
- The branches of a single axon may form synapses with as many as 1000 other neurons.
- Whereas the axon is the output element of the neuron, the **dendrites** (apical and basal) are input elements of the neuron. Together with the cell body, they receive synaptic contacts from other neurons.

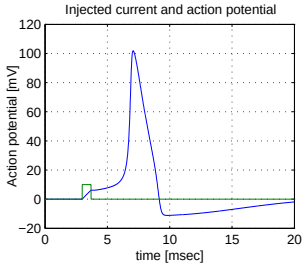
Simplified functions of these very complex in their nature "building blocks" of a neuron are as follow:

- The synapses are elementary signal processing devices.
  - A synapse is a biochemical device which converts a pre-synaptic electrical signal into a chemical signal and then back into a post-synaptic electrical signal.
  - The input pulse train has its amplitude modified by parameters stored in the synapse. The nature of this modification depends on the type of the synapse, which can be either inhibitory or excitatory.
- The postsynaptic signals are aggregated and transferred along the dendrites to the nerve cell body.
- The cell body generates the output neuronal signal, activation potential, which is transferred along the axon to the synaptic terminals of other neurons.
- The frequency of firing of a neuron is proportional to the total synaptic activities and is controlled by the synaptic parameters (weights).
- The pyramidal cell can receive 10^4 synaptic inputs and it can fan-out the output signal to thousands of target cells — the connectivity difficult to achieve in the artificial neural networks.

Another microscopic view of typical neuron of the mammalian cortex (a pyramidal cell):



- Note the cell body or soma, dendrites, synapses and the axon.
- Neuro-transmitters (information-carrying chemicals) are released pre-synaptically, floats across the synaptic cleft, and activate receptors postsynaptically.
- According to Calaj's "neuron-doctrine" information carrying signals come into the dendrites through synapses, travel to the cell body, and activate the axon. Axonal signals are then supplied to synapses of other neurons.



1.2 A simplistic model of a biological neuron

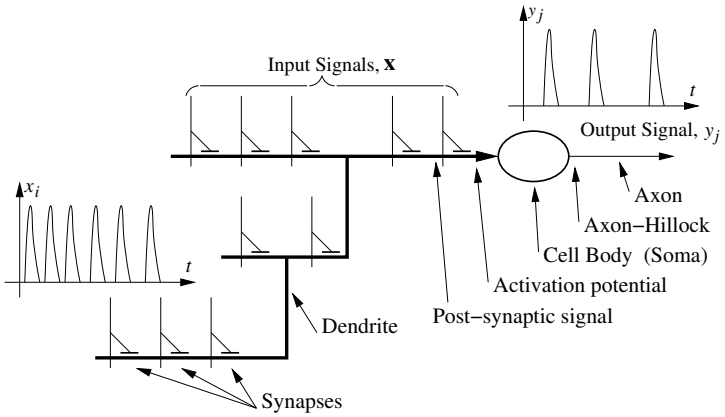


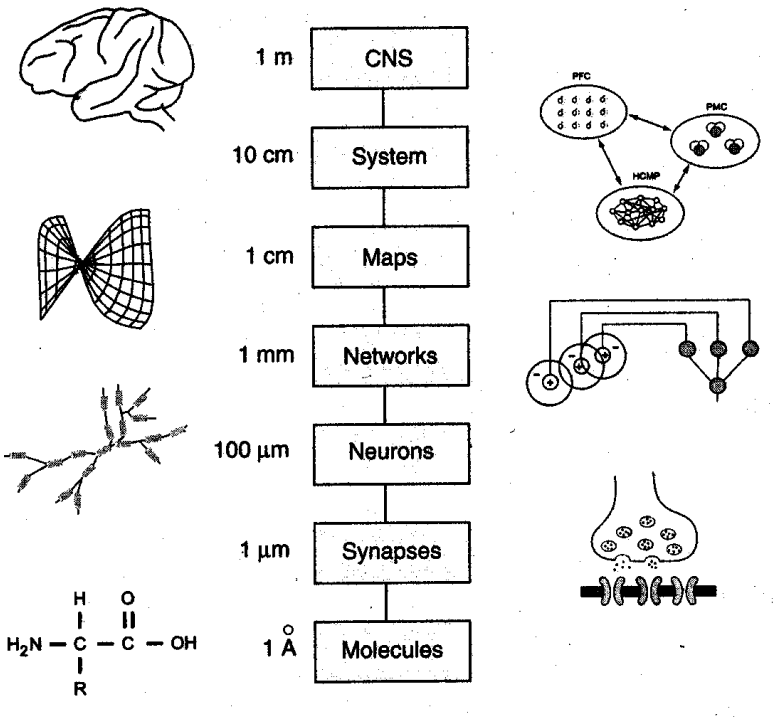
Figure 1–3: Conceptual structure of a biological neuron

Basic characteristics of a biological neuron:

- data is coded in a form of instantaneous frequency of pulses
- synapses are either excitatory or inhibitory
- Signals are aggregated ("summed") when travel along dendritic trees
- The cell body (neuron output) generates the output pulse train of an average frequency proportional to the total (aggregated) post-synaptic activity (activation potential).

Levels of organization of the nervous system

from T.P.Trappenberg, *Fundamentals of Computational Neuroscience*, Oxford University Press, 2002



A few good words about the brain

Adapted from S. Haykin, *Neural Networks – a Comprehensive Foundation,* Prentice Hall, 2nd ed., 1999

The brain is a highly complex, non-linear, parallel information processing system. It performs tasks like pattern recognition, perception, motor control, many times faster than the fastest digital computers.

- Biological neurons, the basic building blocks of the brain, are slower than silicon logic gates. The neurons operate in milliseconds which is about six–seven orders of magnitude slower that the silicon gates operating in the sub-nanosecond range.
- The brain makes up for the slow rate of operation with two factors:
  - a huge number of nerve cells (neurons) and interconnections between them. The number of neurons is estimated to be in the range of 10^{10} with $60 \cdot 10^{12}$ synapses (interconnections).
  - A function of a biological neuron seems to be much more complex than that of a logic gate.
- The brain is very energy efficient. It consumes only about 10^{-16} joules per operation per second, comparing with 10^{-6} J/oper·sec for a digital computer.

Brain plasticity:

- At the early stage of the human brain development (the first two years from birth) about 1 million synapses (hard-wired connections) are formed per second.
- Synapses are then modified through the learning process (plasticity of a neuron).
- In an adult brain plasticity may be accounted for by the above two mechanisms: creation of new synaptic connections between neurons, and modification of existing synapses.

What can you do with Artificial Neural Networks (aNN)

Adapted from Neural Nets FAQ: ftp://ftp.sas.com/pub/neural/FAQ.html

- In principle, NNs can compute any computable function, i.e., they can do everything a normal digital computer can do.
- In practice, NNs are especially useful for **classification** and **function approximation/**mapping problems which are tolerant of some imprecision, which have lots of training data available, but to which hard and fast rules (such as those that might be used in an expert system) cannot easily be applied.
- Almost any mapping between vector spaces can be approximated to arbitrary precision by feedforward NNs (which are the type most often used in practical applications) if you have enough data and enough computing resources.
- To be somewhat more precise, feedforward networks with a single hidden layer, under certain practically-satisfiable assumptions are statistically consistent estimators of, among others, arbitrary measurable, square-integrable regression functions, and binary classification devices.
- NNs are, at least today, difficult to apply successfully to problems that concern manipulation of symbols and memory. And there are no methods for training NNs that can magically create information that is not contained in the training data.

Who is concerned with NNs?

- Computer scientists want to find out about the properties of non-symbolic information processing with neural nets and about learning systems in general.
- Statisticians use neural nets as flexible, nonlinear regression and classification models.
- Engineers of many kinds exploit the capabilities of neural networks in many areas, such as signal processing and automatic control.
- Cognitive scientists view neural networks as a possible apparatus to describe models of thinking and consciousness (high-level brain function).
- Neuro-physiologists use neural networks to describe and explore medium-level brain function (e.g. memory, sensory system, motorics).
- Physicists use neural networks to model phenomena in statistical mechanics and for a lot of other tasks.
- Biologists use Neural Networks to interpret nucleotide sequences.
- Philosophers and some other people may also be interested in Neural Networks for various reasons.

1.3 Taxonomy of neural networks

- From the point of view of their **active** or **decoding** phase, artificial neural networks can be classified into **feedforward** (static) and **feedback** (dynamic, recurrent) systems.
- From the point of view of their **learning** or **encoding** phase, artificial neural networks can be classified into **supervised** and **unsupervised** systems.
- Practical terminology often mixes up the above two aspects of neural nets.

Feedforward supervised networks

This networks are typically used for function approximation tasks. Specific examples include:

- Linear recursive least-mean-square (LMS) networks
- Multi Layer Perceptron (MLP) aka Backpropagation networks
- Radial Basis networks

Feedforward unsupervised networks

This networks are used to extract important properties of the input data and to map input data into a "representation" domain. Two basic groups of methods belong to this category

- Hebbian networks performing the **Principal Component Analysis** of the input data, also known as the Karhunen-Loeve Transform.
- Competitive networks used to performed **Learning Vector Quantization**, or tessellation/clustering of the input data set.
- Self-Organizing Kohonen Feature Maps also belong to this group.

Recurrent networks

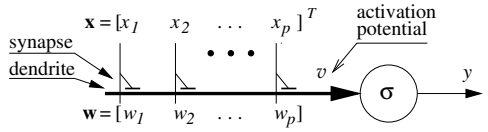
These networks are used to learn or process the temporal features of the input data and their internal state evolves with time. Specific examples include:

- Hopfield networks
- Associative Memories
- Adaptive Resonance networks

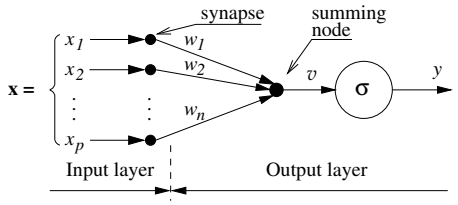
1.4 Models of artificial neurons

Three basic graphical representations of a single p-input (p-synapse) neuron:

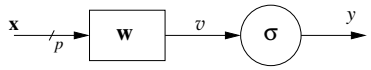
a. **Dendritic representation**



b. **Signal flow representation**



c. **Block−diagram representation**



$$v = w_1 \cdot x_1 + \cdots + w_p \cdot x_p = \mathbf{w} \cdot \mathbf{x}$$

$$y = \sigma(v)$$

Artificial neural networks are nonlinear information (signal) processing devices which are built from interconnected elementary processing devices called neurons.

An artificial neuron is a p-input single-output signal processing element which can be thought of as a simple model of a non-branching biological neuron.

From a **dendritic representation** of a single **neuron** we can identify p **synapses** arranged along a linear **dendrite** which aggregates the synaptic activities, and a neuron body or axon-hillock generating an output signal.

The **pre-synaptic activities** are represented by a p-element **column vector** of input (afferent) signals

$$\mathbf{x} = [x_1 \ \ldots \ x_p]^T$$

In other words the space of input patterns is p-dimensional.

Synapses are characterised by adjustable parameters called weights or synaptic strength parameters. The weights are arranged in a p-element **row vector**:

$$\mathbf{w} = [w_1 \ \ldots \ w_p]$$

- In a **signal flow** representation of a neuron p synapses are arranged in a layer of input **nodes**. A dendrite is replaced by a single summing node. Weights are now attributed to **branches** (connections) between input nodes and the summing node.
- Passing through synapses and a dendrite (or a summing node), input signals are aggregated (combined) into the **activation potential**, which describes the total **post-synaptic activity**.
- The activation potential is formed as a linear combination of input signals and synaptic strength parameters, that is, as an **inner product** of the weight and input vectors:

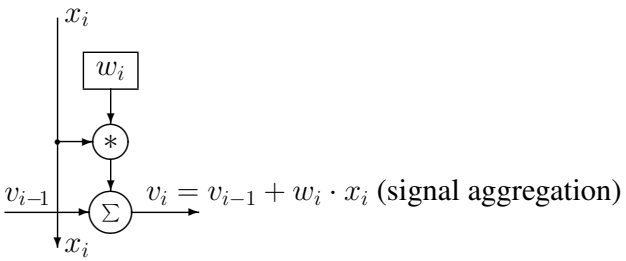
$$v = \sum_{i=1}^{p} w_i x_i = \mathbf{w} \cdot \mathbf{x} = \begin{bmatrix} w_1 & w_2 & \cdots & w_p \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix} \tag{1.1}$$

- Subsequently, the activation potential (the total post-synaptic activity) is passed through an **activation function**, $\sigma(\cdot)$, which generates the output (efferent) signal:

$$y = \sigma(v) \tag{1.2}$$

- The activation function is typically a saturating function which normalises the total post-synaptic activity to the standard values of output (axonal) signal.
- The **block-diagram** representation encapsulates basic operations of an artificial neuron, namely, aggregation of pre-synaptic activities, eqn (1.1), and generation of the output signal, eqn (1.2)

A **single synapse** in a dendritic representation of a neuron can be represented by the following block-diagram:



In the synapse model we can identify:

- a storage for the synaptic weight,
- augmentation (multiplication) of the pre-synaptic signal with the weight parameter, and
- the dendritic aggregation of the post-synaptic activities.

A synapse is classified as

- **excitatory**, if a corresponding weight is positive, $w_i > 0$, and as
- **inhibitory**, if a corresponding weight is negative, $w_i < 0$.

It is sometimes convenient to add an additional parameter called **threshold**, θ or **bias** $b = -\theta$. It can be done by fixing one input signal to be constant. Than we have

$$x_p = +1 \ , \text{ and } \quad w_p = b = -\theta$$

With this addition, the activation potential is calculated as:

$$\hat{v} = \sum_{i=1}^{p} w_i x_i = v - \theta \ , \qquad v = \sum_{i=1}^{p-1} w_i x_i$$

where $\hat{v}$ is the augmented activation potential.

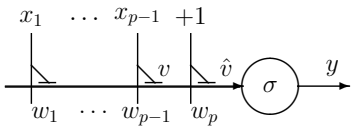


Figure 1–4: A single neuron with a biasing input

1.5 Types of activation functions

Typically, the activation function generates either **unipolar** or **bipolar** signals.

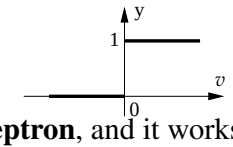
A linear function: $y = v$.

Such linear processing elements, sometimes called ADALINEs, are studied in the theory of linear systems, for example, in the "traditional" signal processing and statistical regression analysis.

A step function

unipolar:

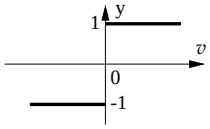
$$y = \sigma(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{if } v < 0 \end{cases}$$



Such a processing element is traditionally called **perceptron**, and it works as a threshold element with a binary output.

bipolar:

$$y = \sigma(v) = \begin{cases} +1 & \text{if } v \geq 0 \\ -1 & \text{if } v < 0 \end{cases}$$

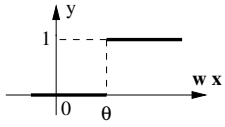


A step function with bias

The bias (threshold) can be added to both, unipolar and bipolar step function. We then say that a neuron is "fired", when the synaptic activity exceeds the threshold level, θ. For a unipolar case, we have:

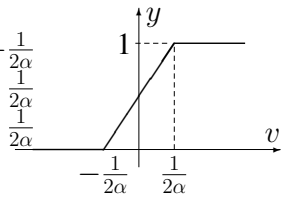
$$y = \sigma(v) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} \geq \theta \\ 0 & \text{if } \mathbf{w} \cdot \mathbf{x} < \theta \end{cases}$$

(The McCulloch-Pitts perceptron — 1943)



A piecewise-linear function

$$y = \sigma(v) = \begin{cases} 0 & \text{if } v \leq -\frac{1}{2\alpha} \\ \alpha v + \frac{1}{2} & \text{if } |v| < \frac{1}{2\alpha} \\ 1 & \text{if } v \geq \frac{1}{2\alpha} \end{cases}$$

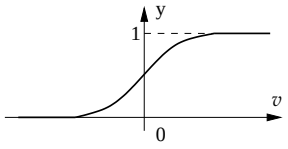


- For small activation potential, v, the neuron works as a linear combiner (an ADALINE) with the gain (slope) α.
- For large activation potential, v, the neuron saturates and generates the output signal either) or 1.
- For large gains $\alpha \to \infty$, the piecewise-linear function is reduced to a step function.

Sigmoidal functions

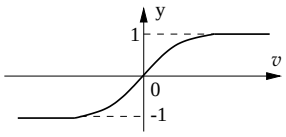
unipolar:

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



bipolar:

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



The parameter β controls the slope of the function.

The hyperbolic tangent (bipolar sigmoidal) function is perhaps the most popular choice of the activation function specifically in problems related to function mapping and approximation.

Radial-Basis Functions

- Radial-basis functions arise as optimal solutions to problems of interpolation, approximation and regularisation of functions. The optimal solutions to the above problems are specified by some integro-differential equations which are satisfied by a wide range of nonlinear differentiable functions (S. Haykin, *Neural Networks – a Comprehensive Foundation*, Ch. 5).
- Typically, Radial-Basis Functions $\varphi(\mathbf{x}; \mathbf{t}_i)$ form a family of functions of a p-dimensional vector, $\mathbf{x}$, each function being centered at point $\mathbf{t}_i$.
- A popular simple example of a Radial-Basis Function is a symmetrical multivariate Gaussian function which depends only on the distance between the current point, $\mathbf{x}$, and the center point, $\mathbf{t}_i$, and the variance parameter σ_i:

$$\varphi(\mathbf{x}; \mathbf{t}_i) = G(||\mathbf{x} - \mathbf{t}_i||) = \exp\left(-\frac{||\mathbf{x} - \mathbf{t}_i||^2}{2\sigma_i^2}\right)$$

  where $||\mathbf{x} - \mathbf{t}_i||$ is the norm of the distance vector between the current vector $\mathbf{x}$ and the centre, $\mathbf{t}_i$, of the symmetrical multidimensional Gaussian surface.
- The spread of the Gaussian surface is controlled by the variance parameter σ_i.

Two concluding remarks:

- In general, the smooth activation functions, like sigmoidal, or Gaussian, for which a continuous derivative exists,

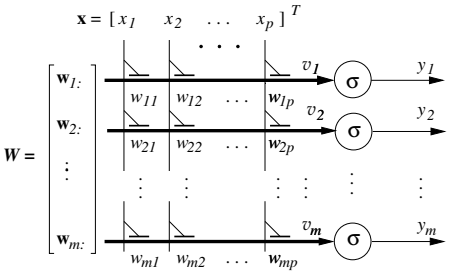
  are typically used in networks performing a function approximation task,

  whereas the step functions are used as parts of pattern classification networks.
- Many learning algorithms require calculation of the derivative of the activation function (see the assignments/practical 1).

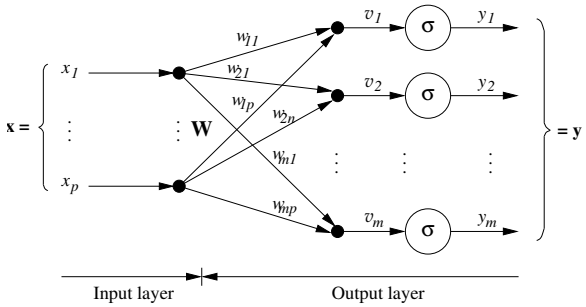
1.6 A layer of neurons

- Neurons as in sec. 1.4 can be arrange into a **layer** of neurons.
- A **single layer neural network** consists of m neurons each with the same p input signals.
- Similarly to a single neuron, the neural network can be represented in all **three** basic forms: **dendritic**, **signal-flow**, and **block-diagram** form
- From the **dendritic representation** of the neural network it is readily seen that a layer of neurons is described by a $m \times p$ matrix W of synaptic weights.
- Each row of the **weight matrix** is associated with one neuron.

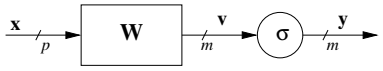
a. Dendritic graph



b. Signal–flow graph



c. Block–diagram



Operations performed by the network can be described as follows:

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix} = \begin{bmatrix} w_{11} & \cdots & w_{1p} \\ w_{21} & \cdots & w_{2p} \\ \vdots & \cdots & \vdots \\ w_{m1} & \cdots & w_{mp} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_p \end{bmatrix} \; ; \quad \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} \sigma(v_1) \\ \sigma(v_2) \\ \vdots \\ \sigma(v_m) \end{bmatrix}$$

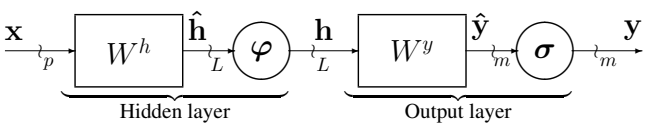
$\mathbf{v} = W \cdot \mathbf{x}$; $\mathbf{y} = \boldsymbol{\sigma}(W \cdot \mathbf{x}) = \boldsymbol{\sigma}(\mathbf{v})$; $\mathbf{v}$ is a vector of activation potentials.

From the **signal-flow graph** it is visible that each weight parameter w_{ij} (synaptic strength) is now related to a connection between nodes of the input layer and the output layer.
Therefore, the name **connection strengths** for the weights is also justifiable.

The **block-diagram** representation of the single layer neural network is the most compact one, hence most convenient to use.

1.7 Multi-layer feedforward neural networks

Connecting in a serial way layers of neurons presented in sec. 1.6 we can build multi-layer feedforward neural networks also known as **Multilayer Perceptrons** (MLP).
The most popular neural network seems to be the one consisting of **two layers of neurons** as in the following block-diagram



p input signals
L hidden neurons
m output neurons

In order to avoid a problem of counting an input layer, the two-layer architecture is referred to as a **single hidden layer** neural network.

Input signals, $\mathbf{x}$, are passed through synapses of the hidden layer with connection strengths described by the **hidden weight matrix**, W^h, and the L **hidden activation signals**, $\hat{\mathbf{h}}$, are generated.

The hidden activation signals are then normalised by the functions $\boldsymbol{\psi}$ into the L **hidden signals**, $\mathbf{h}$.

Similarly, the hidden signals, $\mathbf{h}$, are first, converted into m **output activation signals**, $\hat{\mathbf{y}}$, by means of the output weight matrix, W^y, and subsequently, into m **output signals**, $\mathbf{y}$, by means of the functions $\boldsymbol{\sigma}$. Hence

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

If needed, one of the input signals and one of the hidden signals can be constant to form a biasing signals.

Functions $\boldsymbol{\varphi}$ and $\boldsymbol{\sigma}$ can be identical.

1.8 Static and Dynamic Systems — General Concepts

Static systems

Neural networks considered in previous sections belong to the class of **static** systems which can be fully described by a set of m-functions of p-variables as in Figure 1–5.

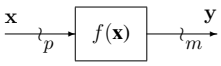


Figure 1–5: A static system: $\mathbf{y} = f(\mathbf{x})$

The defining feature of the static systems is that they are **time-independent** — current outputs depends only on the current inputs in the way specified by the mapping function, f.
Such a function can be very complex.

Dynamic systems — Recurrent Neural Networks

In the dynamic systems, the current output signals depend, in general, on current and past input signals.

There are two equivalent classes of dynamic systems: continuous-time and discrete-time systems.

The dynamic neural networks are referred to as **recurrent neural networks**.

1.9 Continuous-time dynamic systems

- Continuous-time dynamic systems operate with signals which are functions of a continuous variable, t, interpreted typically as **time**. A **spatial variable** can be also used.
- Continuous-time dynamic systems are described by means of **differential equations**. The most convenient yet general description uses only **first-order** differential equations in the following form:

$$\dot{\mathbf{y}}(t) = f(\mathbf{x}(t), \mathbf{y}(t)) \tag{1.3}$$

where

$$\dot{\mathbf{y}}(t) \stackrel{\text{df}}{=} \frac{d\mathbf{y}(t)}{dt}$$

is a vector of time derivatives of output signals.

- In order to model a dynamic system, or to obtain the output signals, the **integration** operation is required. The dynamic system of eqn (1.3) is illustrated in Figure 1–6.

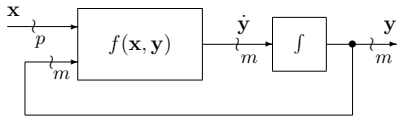


Figure 1–6: A continuous-time dynamic system: $\dot{\mathbf{y}}(t) = f(\mathbf{x}(t), \mathbf{y}(t))$

- It is evident that **feedback** is inherent to dynamic systems.

1.9.1 Discrete-time dynamic systems

- Discrete-time dynamic systems operate with signals which are functions of a discrete variable, n, interpreted typically as time, but a discrete spatial variable can be also used.
- Typically, the discrete variable can be thought of as a **sampled** version of a continuous variable:

$$t = n \cdot t_s \,; \quad t \in \mathcal{R} \,, \quad n \in \mathcal{N}$$

and t_s is the **sampling time**

- Analogously, discrete-time dynamic systems are described by means of **difference equations**.
- The most convenient yet general description uses only **first-order** difference equations in the following form:

$$\mathbf{y}(n+1) = f(\mathbf{x}(n), \mathbf{y}(n)) \tag{1.4}$$

where $\mathbf{y}(n+1)$ and $\mathbf{y}(n)$ are the predicted (future) value and the current value of the vector $\mathbf{y}$, respectively.

- In order to model a discrete-time dynamic system, or to obtain the output signals, we use the **unit delay** operator, $D = z^{-1}$ which originates from the z-transform used to obtain analytical solutions to the difference equations.
- Using the delay operator, we can re-write the first order difference equation into the following operator form:

$$z^{-1}\mathbf{y}(n+1) = \mathbf{y}(n)$$

which leads to the structure as in Figure 1–7.

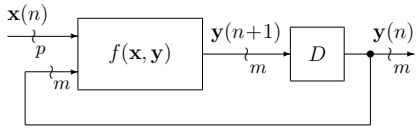


Figure 1–7: A discrete-time dynamic system: **y**(n+1) = f(**x**(n), **y**(n))

- Notice that **feedback** is also present in the discrete dynamic systems.

1.9.2 Example: A continuous-time generator of a sinusoid

- As a simple example of a continuous-time dynamic system let us consider a linear system which generates a sinusoidal signal.
- Eqn (1.3) takes on the following form:

$$\dot{\mathbf{y}}(t) = A \cdot \mathbf{y}(t) + B \cdot \mathbf{x}(t) \tag{1.5}$$

where

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}; \quad A = \begin{bmatrix} 0 & \omega \\ -\omega & 0 \end{bmatrix}; \quad B = \begin{bmatrix} 0 \\ b \end{bmatrix}; \quad \mathbf{x} = \delta(t)$$

$\delta(t)$ is the unit impulse which is non-zero only for $t = 0$ and is used to describe the initial condition.

- In order to show that eqn (1.5) really describes the sinusoidal generator we re-write this equation for individual components. This yields:

$$\begin{aligned} \dot{y}_1 &= \omega\, y_2 \\ \dot{y}_2 &= -\omega\, y_1 + b\, \delta(t) \end{aligned} \tag{1.6}$$

- Differentiation of the first equation and substitution of the second one gives the second-order linear differential equation for the output signal y_1:

$$\ddot{y}_1 + \omega^2 y_1 = \omega\, b\, \delta(t)$$

- Taking the Laplace transform and remembering that $\mathcal{L}\delta(t) = 1$, we have:

$$y_1(s) = b\frac{\omega}{s^2 + \omega^2}$$

- Taking the inverse Laplace transform we finally have

$$y_1(t) = b\sin(\omega t)$$

- The internal structure of the generator can be obtained from eqns (1.7) and illustrated using the dendritic representation as in Figure 1–8.

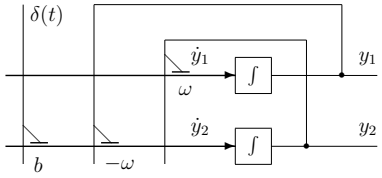


Figure 1–8: A continuous-time sinusoidal generator

- The generator can be thought of as a simple example of a **linear recurrent neural network** with the fixed weight matrix of the form:

$$W = [B\ A] = \begin{bmatrix} 0 & 0 & \omega \\ b & -\omega & 0 \end{bmatrix}$$

- The weights were designed appropriately rather than "worked out" during the learning procedure.

1.9.3 Example: A discrete-time generator of a sinusoid

- It is possible to build a discrete-time version of the sinusoidal generator using difference equations of the general form as in eqn (1.4):

$$\mathbf{y}(n+1) = A \cdot \mathbf{y}(n) + B \cdot \mathbf{x}(n) \tag{1.7}$$

where

$$A = \begin{bmatrix} \cos\Omega & \sin\Omega \\ -\sin\Omega & \cos\Omega \end{bmatrix}\ ; \quad B = \begin{bmatrix} 0 \\ b \end{bmatrix}\ ; \quad \mathbf{x}(n) = \delta(n)$$

- This time we take the z-transform directly of eqn (1.7), which gives:

$$(zI - A)\mathbf{y}(z) = B\ ; \quad \text{where } I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \text{and } \mathcal{Z}\delta(n) = 1$$

Hence

$$\mathbf{y}(z) = (zI - A)^{-1}B$$

and subsequently

$$\mathbf{y}(z) = (\begin{bmatrix} z - \cos\Omega & \sin\Omega \\ -\sin\Omega & z - \cos\Omega \end{bmatrix} \begin{bmatrix} 0 \\ b \end{bmatrix})/(z^2 - 2z\cos\Omega + 1)$$

Extracting the first component, we have

$$y_1(z) = \frac{b\sin\Omega}{z^2 - 2z\cos\Omega + 1}$$

Taking the inverse z-transform finally yields

$$y_1(n+1) = b\sin(\Omega n)$$

which means that the discrete-time dynamic system described by eqn (1.7) generates a sinusoidal signal.

- The structure of the generator is similar to the previous one and is presented in Figure 1–9.

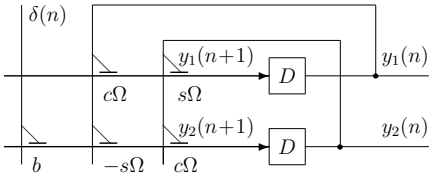


Figure 1–9: A discrete-time sinusoidal generator

- This time the weight matrix is:

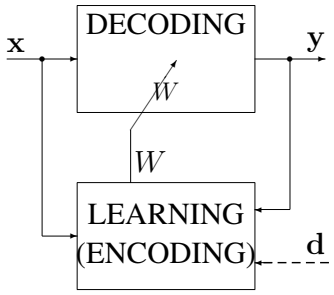
$$W = [B\ A] = \begin{bmatrix} 0 & c\Omega & s\Omega \\ b & -s\Omega & c\Omega \end{bmatrix}$$

where

$$s\Omega = \sin\Omega, \quad \text{and} \quad c\Omega = \cos\Omega$$

1.10 Introduction to learning

- In the previous sections we concentrated on the **decoding** part of a neural network assuming that the **weight matrix**, W, is given.
- If the weight matrix is satisfactory, during the decoding process the network performs some useful task it has been design to do.
- In simple or specialised cases the weight matrix can be pre-computed, but more commonly it is obtained through the **learning** process.
- Learning is a dynamic process which modifies the weights of the network in some desirable way. As any dynamic process learning can be described either in the continuous-time or in the discrete-time framework.
- A neural network with learning has the following structure:



- The learning can be described either by differential equations (continuous-time)

$$\dot{W}(t) = L(\, W(t), \mathbf{x}(t), \mathbf{y}(t), \mathbf{d}(t)\,) \tag{1.8}$$

  or by the difference equations (discrete-time)

$$W(n+1) = L(\, W(n), \mathbf{x}(n), \mathbf{y}(n), \mathbf{d}(n)\,) \tag{1.9}$$

  where $\mathbf{d}$ is an external teaching/supervising signal used in **supervised learning**.

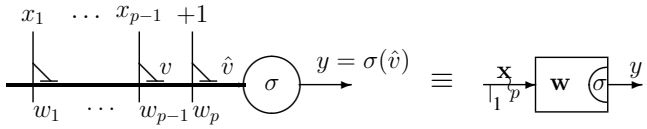
- This signal in not present in networks employing **unsupervised learning**.

- The discrete-time learning law is often used in a form of a **weight update** equation:

$$\begin{aligned} W(n+1) &= W(n) + \Delta W(n) \\ \Delta W(n) &= L(\, W(n), \mathbf{x}(n), \mathbf{y}(n), \mathbf{d}(n)\,) \end{aligned} \tag{1.10}$$

2 Perceptron

- The perceptron was introduced by McCulloch and Pitts in 1943 as an artificial neuron with a **hard-limiting activation function**, σ.
- Recently the term **multilayer perceptron** has often been used as a synonym for the term **multilayer feedforward neural network**.



- Input signals, x_i, are assumed to have real values.
- The activation function, σ, is a unipolar step function (sometimes called the Heaviside function), therefore, the output signal is binary, $y \in \{0, 1\}$.
- One input signal is constant $(x_p = 1)$, and the related weight is interpreted as the bias, or threshold, θ:

$$w_p = -\theta$$

- The **input** signals and **weights** are arranged in the following column and row vectors, respectively:

$$\mathbf{x} = \begin{bmatrix} x_1 & \cdots & x_{p-1} & 1 \end{bmatrix}^T \; ; \quad \mathbf{w} = \begin{bmatrix} w_1 & \cdots & w_{p-1} & w_p \end{bmatrix}$$

- Aggregation of the "proper" input signals results in the **activation potential**, v, which can be expressed as the **inner product** of "proper" input signals and related weights:

$$v = \sum_{i=1}^{p-1} w_i x_i = \mathbf{w}_{1:p-1} \cdot \mathbf{x}_{1:p-1}$$

- The augmented activation potential, $\hat{v}$, can be expressed as:

$$\hat{v} = \mathbf{w} \cdot \mathbf{x} = v - \theta$$

- For each input signal, $\mathbf{x}$, the output, y, is determined as

$$y = \sigma(\hat{v}) = \begin{cases} 0 & \text{if} \quad v < \theta \quad (\hat{v} < 0) \\ 1 & \text{if} \quad v \geq \theta \quad (\hat{v} \geq 0) \end{cases}$$

- Hence, a perceptron works as a **threshold element**, the output being "active" if the activation potential exceeds the threshold.

2.1 A Perceptron as a Pattern Classifier

- A single perceptron classifies input patterns, $\mathbf{x}$, into **two classes**.
- A linear combination of signals and weights for which the augmented activation potential is zero, $\hat{v} = 0$, describes a **decision surface** which partitions the input space into two regions.
- The decision surface is a **hyperplane** specified as:

$$\mathbf{w} \cdot \mathbf{x} = 0 \ , \quad x_p = 1 \ , \quad \text{or} \quad v = \theta \tag{2.1}$$

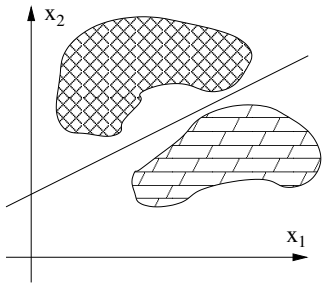
and we say that an input vector (pattern)

$$\mathbf{x} \ \text{ belongs to a class } \left\{ \begin{array}{c} \#1 \\ \#0 \end{array} \right\} \ \text{ if } \ \left\{ \begin{array}{l} y = 1, \ (v \geq \theta) \\ y = 0, \ (v < \theta) \end{array} \right.$$

- The input patterns that can be classified by a single perceptron into two distinct classes are called **linearly separable patterns**.

Example from a two-dimensional space $(p = 3)$.
Equation of the **decision line**:

$$w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 = 0 \ , \quad \text{or} \quad \left[\begin{array}{ccc} w_1 & w_2 & w_3 \end{array} \right] \left[\begin{array}{c} x_1 \\ x_2 \\ 1 \end{array} \right] = 0$$



Useful remarks

- The inner product of two vectors can be also expressed as:

$$\mathbf{w} \cdot \mathbf{x} = ||\mathbf{w}|| \cdot ||\mathbf{x}|| \cdot \cos \alpha$$

where $||.||$ denotes the norm (magnitude, length) of the vector, and α is the angle between vectors.

- The weight vector, $\mathbf{w}$, is orthogonal to the decision plane in the augmented input space, because

$$\mathbf{w} \cdot \mathbf{x} = 0$$

that is, the inner product of the weight and augmented input vectors is zero.

- Note that the weight vector,

$$\mathbf{w}_{1:p-1}$$

is also orthogonal to the decision plane

$$\mathbf{w}_{1:p-1} \cdot \mathbf{x}_{1:p-1} + w_p = 0$$

in the "proper", $(p-1)$–dimensional input space.

- For any pair of input vectors, say, $(\mathbf{x}^a_{1:p-1}, \mathbf{x}^b_{1:p-1})$, for which the decision plane equation is satisfied, we have

$$\mathbf{w}_{1:p-1} \cdot (\mathbf{x}^a_{1:p-1} - \mathbf{x}^b_{1:p-1}) = 0$$

Note that the difference vector $(\mathbf{x}^a_{1:p-1} - \mathbf{x}^b_{1:p-1})$ lies in the decision plane.

Geometric Interpretation

- Let us consider a 3-dimensional augmented input space $(p = 3)$. Input vectors are 2-dimensional, and $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & 1 \end{bmatrix}^T$. The decision plane, is described as follows:

$$w_1 x_1 + w_2 x_2 + w_3 x_3 = 0 \ , \quad x_3 = 1 \tag{2.2}$$

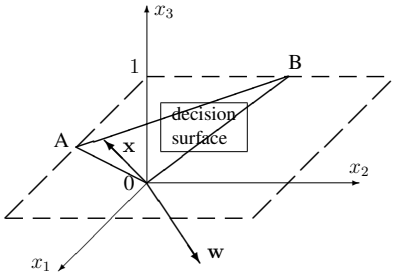


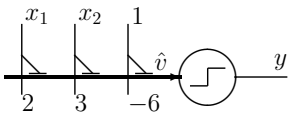
Figure 2–1: Augmented 3-D input space with the decision plane

- The augmented decision plane, OAB, goes through the origin of the augmented input space (a homogeneous equation).
- Each augmented input vector, $\mathbf{x}$, lies in the this plane. The weight vector, $\mathbf{w}$, is orthogonal to the OAB plane hence to each augmented input vector.
- Intersection of the augmented decision plane, OAB, and the horizontal plane, $x_3 = 1$, gives a 2-D "plane" AB (a straight line, in this case) described by eqn (2.2).

2.2 Example — a three-synapse perceptron

Consider a three-input perceptron with weights

$$\mathbf{w} = [2 \quad 3 \quad -6]$$



The total (augmented) activation potential is determine as:

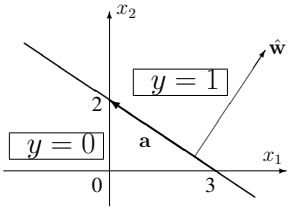
$$\hat{v} = \mathbf{w} \cdot \mathbf{x} = \begin{bmatrix} 2 & 3 & -6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} = 2x_1 + 3x_2 - 6$$

The input space, $[x_1 \ x_2$, is 2-dimensional and the decision plane is reduced to a straight line:

$$2x_1 + 3x_2 - 6 = 0$$

The 2-D weight vector $\hat{\mathbf{w}} = [2 \ \ 3]$ is perpendicular to the decision line, that is, to the vector $\mathbf{a} = [-3 \ \ 2]^T$ which is parallel to (lies in) the decision line, because the inner product $\hat{\mathbf{w}} \cdot \mathbf{a} = 0$, that is

$$[2 \ \ 3] \begin{bmatrix} -3 \\ 2 \end{bmatrix} = 0$$



Note that the weight vector $\hat{\mathbf{w}}$ points in the "positive" direction, that is, to the side of the plane where $y = 1$.

□

2.3 Selection of weights for the perceptron

In order for the perceptron to perform a desired task, its weights must be properly selected.

In general two basic methods can be employed to select a suitable weight vector:

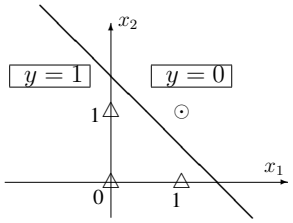
- By off-line calculation of weights. If the problem is relatively simple it is often possible to calculate the weight vector from the specification of the problem.
- By learning procedure. The weight vector is determined from a given (training) set of input-output vectors (exemplars) in such a way to achieve the best classification of the training vectors.

Once the weights are selected (the **encoding process**),
the perceptron performs its desired task (e.g., pattern classification)
(the **decoding process**).

2.3.1 Selection of weights by off-line calculations — Example

Consider the problem of building the **NAND** gate using the perceptron. In this case the desired input-output relationship is specified by the following truth table and related plot:

x_1	0	0	1	1
x_2	0	1	0	1
y	1	1	1	0



The input patterns (vectors) belong to two classes and are marked in the input space (the plane (x_1, x_2)) with $\triangle$ and $\odot$, respectively. The decision plane is the straight line described by the following equation

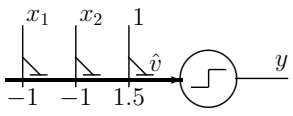
$$x_1 + x_2 = 1.5 \quad \text{or} \quad -x_1 - x_2 + 1.5 = 0$$

In order for the weight vector to point in the direction of $y = 1$ patterns we select it as:

$$\mathbf{w} = \begin{bmatrix} -1 & -1 & 1.5 \end{bmatrix}$$

For each input vector the output signal is now calculated as

$$\hat{v} = \begin{bmatrix} -1 & -1 & 1.5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} \; ; \; y = \begin{cases} 0 & \text{if } \hat{v} < 0 \\ 1 & \text{if } \hat{v} \geq 0 \end{cases}$$



Verify that such a perceptron does implement a two-input NAND gate.

2.4 The Perceptron learning law

- **Learning** is a recursive procedure of modifying weights from a given set of input-output patterns.
- For a single perceptron, the objective of the learning (encoding) procedure is to find the decision plane, (that is, the related weight vector), which separates two classes of given **input-output training vectors**.
- Once the learning is finalised, every input vector will be classified into an appropriate class.
- A single perceptron can classify only the **linearly separable** patterns.
- The perceptron learning procedure is an example of a **supervised error-correcting learning** law.

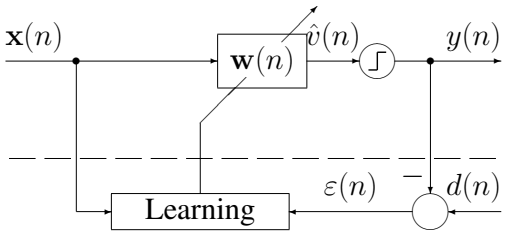


Figure 2–2: Error-Correcting, Supervised Learning in perceptron

More specifically, the supervised learning procedure can be described as follows

- given the set of N **training patterns** consisting of input signals $\mathbf{x}(n)$ and the related **desired** output signals, $d(n)$:

$$\mathbf{x}(n) = \begin{bmatrix} x_1(n) \\ \vdots \\ x_{p-1}(n) \\ 1 \end{bmatrix}, \quad d(n), \quad \text{for } n = 1, 2, \ldots, N$$

- obtain the correct decision plane specified by the weight vector $\mathbf{w}$.
- The training patterns are arranged in a **training set** which consists of a $p \times N$ input matrix, X, and an N-element output vector, D:

$$X = \begin{bmatrix} x_1(1) & x_1(2) & \cdots & x_1(N) \\ x_2(1) & x_2(2) & \cdots & x_2(N) \\ \vdots & \vdots & & \vdots \\ x_{p-1}(1) & x_{p-1}(2) & \cdots & x_{p-1}(N) \\ 1 & 1 & \cdots & 1 \end{bmatrix}$$

$$D = \begin{bmatrix} d(1) & d(2) & \cdots & d(N) \end{bmatrix}$$

- We can assume that an **untrained perceptron** can generate an incorrect output $y(n) \neq d(n)$ due to incorrect values of weights.
- We can think about the **actual output** $y(n)$ as an "**estimate**" of the **correct**, **desired** output $d(n)$.

The **perceptron learning law** proposed by Rosenblatt in 1959 can be described as follows:

1. weights are initially set to "small" random values

$$\mathbf{w}(0) = \text{ rand}$$

2. for $n = 1, 2, \ldots, N$ training input vectors, $\mathbf{x}(n)$, are presented to the perceptron and the **actual output** $y(n)$, is compared with the **desired output**, $d(n)$, and the **error** $\varepsilon(n)$ is calculated as follows

$$y(n) = \sigma(\mathbf{w}(n) \cdot \mathbf{x}(n)) \ , \quad \varepsilon(n) = d(n) - y(n)$$

Note that because

$$d(n),\ y(n) \in \{0, 1\}, \quad \text{then} \quad \varepsilon(n) \in \{-1, 0, +1\}$$

3. at each step, the weights are adapted as follows

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta\, \varepsilon(n)\, \mathbf{x}(n) \tag{2.3}$$

where $0 < \eta < 1$ is a learning parameter (gain), which controls the adaptation rate. The gain must be adjusted to ensure the convergence of the algorithm.

Note that the weight update

$$\Delta\mathbf{w} = \eta\, \varepsilon(n)\, \mathbf{x}(n) \tag{2.4}$$

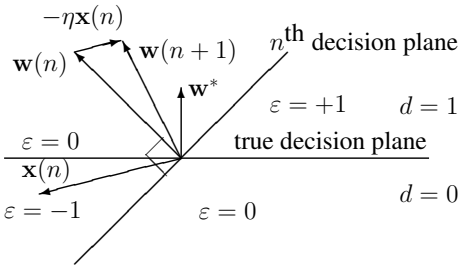
is zero if the error $\varepsilon(n) = 0$.

The tabular form of the weight updates:

$$\Delta\mathbf{w} = \eta\, \varepsilon(n)\, \mathbf{x}(n)$$

$d(n)$	$y(n)$	$\varepsilon(n)$	$\Delta\mathbf{w}(n)$
0	0	0	0
1	1	0	0
0	1	-1	$-\eta\, \mathbf{x}(n)$
1	0	$+1$	$+\eta\, \mathbf{x}(n)$

Geometric interpretation of the perceptron learning law:



- Identify a current weight vector, $\mathbf{w}(n)$, the next weight vector, $\mathbf{w}(n+1)$, and the correct weight vector, $\mathbf{w}^*$.
- Related decision planes are orthogonal to these vectors and are depicted as straight lines.
- During the learning process the current weight vector $\mathbf{w}(n)$ is modified in the direction of the current input vector $\mathbf{x}(n)$, if the input pattern is misclassified, that is, if the error is non-zero.

- Presenting the perceptron with enough training vectors, the weight vector $\mathbf{w}(n)$ will tend to the correct value $\mathbf{w}^*$.
- Rosenblatt proved that if input patterns are linearly separable, then the perceptron learning law converges, and the hyperplane separating two classes of input patterns can be determined.
- The structure of the perceptron and its learning law are presented in the following block-diagram

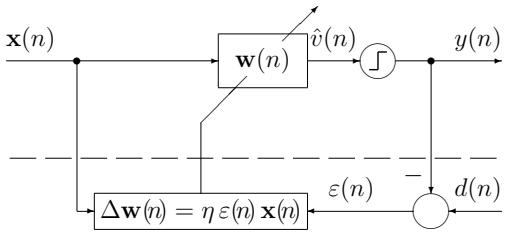


Figure 2–3: The structure of a perceptron with its learning part.

- Note the **decoding part** of the perceptron is a static, feedforward system, whereas the **encoding part** is a discrete-time dynamic system described by the difference equation (2.3).

2.5 Implementation of the perceptron learning law in MATLAB — Example

The following numerical example demonstrates the perceptron learning law. In the example we will demonstrate three main steps typical to most of the neural network applications, namely:

Preparation of the training patterns. Normally, the training patterns are specific to the particular application.

In our example we generate a set of vectors and partition this set with a plane into two linearly separable classes.

The set of vectors is split into two halves, one used as a **training** set, the other as a **validation** set.

Learning. Using the training set, we apply the perceptron learning law as described in sec. 2.4

Validation. Using the validation set we verify the quality of the learning process.

Calculations are performed in MATLAB.

Description of a demo script perc.m

`p = 5;` % dimensionality of the augmented input space
`N = 50;` % number of training patterns — size of the training epoch

PART 1: Generation of the training and validation sets

`X = 2*rand(p-1,2*N)-1;` % a $(p-1) \times 2N$ matrix of uniformly distributed random numbers from the interval $[-1, +1]$
`nn = randperm(2*N); nn = nn(1:N);` % generation of N random integer numbers from the range $[1..2N]$.
`X(:,nn) = sin(X(:,nn));` % Columns of the matrix X pointed to by `nn` are "coloured" with the function 'sin', in order to make the training patterns more interesting.
`X = [X; ones(1,2*N)];` % Each input vector is appended with a constant 1 to implement biasing.

The resulting matrix of input patterns, X, may be of the following form:

X =	-0.02	0.83	-0.72	...	-0.68	-0.04
	0.33	0.73	-0.10	...	0.45	-0.09
	0.36	0.78	0.98	...	-0.41	0.04
	-0.57	0.09	-0.57	...	-0.90	-0.11
	1.00	1.00	1.00	...	1.00	1.00

where only the three initial and two final columns of the matrix X are shown.

Specification of an arbitrary separation plane in the augmented input space:
`wht = 3*rand(1,p)-1; wht = wht/norm(wht);` % This is a unity length vector orthogonal to the augmented separation plane. It is also the target weight vector.
The result may be of the form:

```
wht = 0.38 0.66 -0.14 0.61 0.14
```

`D = (wht*X >= 0);` % Classification of every point from the input space with respect to the class number, 0 or 1. The three initial and two terminal values of the output vector may be of the following form:

```
D = 0 1 1 ...  0 0
```

`Xv = X(:, N+1:2*N) ;` % The validation set: `Xv` is $p \times N$
`Dv = D(:, N+1:2*N) ;` % `Dv` is $1 \times N$ `X = X(:, 1:N) ;` % The training set: `X` is $p \times N$
`D = D(:, 1:N) ;` % `D` is $1 \times N$

Visualisation of the input-output patterns.

The input space is p-dimensional, hence difficult to visualise.
We will plot projections of input patterns on a 2-D plane, say $(x_1 - x_3)$ plane.
The projection is performed by extracting rows specified in `pr` from the X and D matrices.

```
pr = [1, 3]; Xp = X(pr, :);
wp = wht([pr p]);  % projection of the weight vector
c0 = find(D==0); c1 = find(D==1);  % c0 and c1 are vectors of indices to input patterns
```

X belonging to the class 0 or 1, respectively.

```
figure(1), clf reset
plot(Xp(1,c0),Xp(2,c0),'o',Xp(1,c1),Xp(2,c1),'x')  % The input patterns are
        plotted on the selected projection plane.
        Patterns belonging to the class 0, or 1 are marked with 'o', or 'x', respectively.
```

Setting up the plot and freezing its axes:

```
axis(axis),  axis('square')
title('Projection of the input space and a decision plane')
xlabel(['x_', num2str(pr(1))])
ylabel(['x_', num2str(pr(2))])
hold on
```

Superimposition of the projection of the separation plane on the plot. The projection is a straight line:

$$x_1 \cdot w_1 + x_3 \cdot w_3 + w_5 = 0$$

To plot this line we need at least two points. However, in order to account for all possible situations we will use four co-linear points which are intersections with the four lines limiting the square as presented in sec. 2.7.

```
L = [-1 1] ;
S = -diag([1 1]./wp(1:2))*(wp([2,1])'*L +wp(3)) ;
plot([S(1,:) L], [L S(2,:)]), grid, drawnow  % plotting the line
```

The relevant plot is in Figure 2–5.

PART 2: Learning

- The training input-output patterns are stored in matrices X $(p \times N)$ and D $(1 \times N)$.
- We will start the learning/training process with a randomly selected weight vector (thus related separation/decision plane).
- During training procedure the weight vector should converge to the weight vector specifying the correct separation plane.

```
eta = 0.5;  % The training gain.
wh = 2*rand(1,p)-1;  % Random initialisation of the weight vector with values from
        the range [−1, +1]. An example of an initial weight vector follows
       wh = 0.36 -0.34 -0.89 0.97 0.08
```

Projection of the initial decision plane which is orthogonal to `wh` is plotted as previously:

```
wp = wh([pr p]); % projection of the weight vector
S = -diag([1 1]./wp(1:2))*(wp([2,1])'*L +wp(3)) ;
plot([S(1,:) L], [L S(2,:)]), grid on, drawnow
```

- In what follows, the internal loop controlled by the variable n goes through N training exemplars (one epoch).
- The loop is repeated until the performance index (error) E is small but not more than C times (C epochs).
- The projection of the current decision surface is plotted after the previous projection has been erased.

```
C = 50;  % Maximum number of training epochs
E = [C+1, zeros(1,C)];  % Initialization of the vector of the total sums of squared errors
        over an epoch.
WW = zeros(C*N,p);  % The matrix WW will store all weight vectors wh, one weight vector
        per row of the matrix WW
c = 1 ;     % c is an epoch counter
cw = 0 ;  % cw total counter of weight updates
while (E(c)>1)|(c==1)
  c = c+1 ;
  plot([S(1,:)  L], [L S(2,:)], 'w')  % At the beginning of each internal loop
        the former projection of the decision plane is erased (option 'w')
  for n = 1:N  % The internal loop goes once through all training exemplars.
    eps = D(n) - ((wh*X(:,n)) >= 0);  % ε(n) = d(n) − y(n)
    wh =wh+eta*eps*X(:,n)';  % The Perceptron Learning Law
    cw=cw+1;
    WW(cw,:)=wh/norm(wh);  % The updated and normalised weight vector is stored in WW
        for feature plotting
    E(c)=E(c)+abs(eps);  % |ε| = ε^2
  end;
  wp=wh([pr p]); % projection of the weight vector
  S = -diag([1 1]./wp(1:2))*(wp([2,1])'*L +wp(3)) ;
  plot([S(1,:) L], [L S(2,:)], 'g'), drawnow
end;
```


After every pass through the set of training patterns, the projection of the current decision plane, which is determined by the current weight vector, is plotted after the previous projection has been erased.

```
WW = WW(1:cw, pr);
E = E(2:c+1)
```

An example of the performance index E and the final value of the weight vector is as follows:

E = 10 6 4 3 1 0

wt = 0.29 0.73 –0.18 0.61 0.14

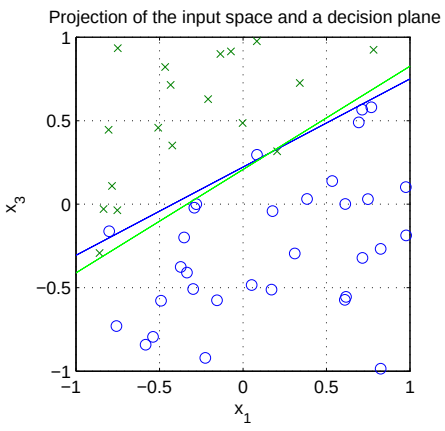


Figure 2–4: Projection of the input space onto a 2–D plane.

Validation

- The final decision (separation) planes differs slightly from the correct one.
- Therefore, if we use input signals from the validation set which were not used in training, it may be expected that some patterns will be misclassified and the following total sum of squared errors will be non-zero.

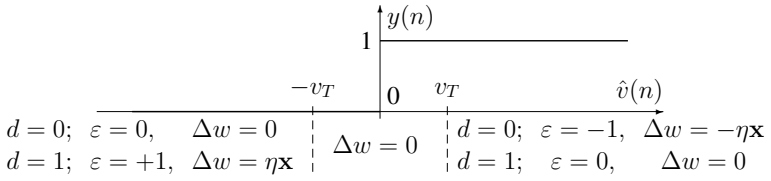
```
Ev = sum(abs(Dv - (wh*Xv >= 0))')
```

The values of `Ev` over many training sessions were between 0 and 5 (out of $N = 50$).

2.6 A modified perceptron learning rule

- In the basic version of the perceptron learning rule, the weight update occurs for each input vector, $\mathbf{x}(n)$, which generates output, $y(n)$, different from desired $d(n)$, that is, when the error $\varepsilon(n)$ is non-zero.
- In the modified version of the perceptron learning rule, in addition to misclassification, the input vector $\mathbf{x}(n)$ must be located far enough from the decision surface, or, alternatively, the net activation potential $\hat{v}(n)$ must exceed a preset margin, v_T.
- The reason for adding such a condition is to avoid an erroneous classification decision in a situation when $\hat{v}(n)$ is very small, and due to presence of noise, the sign of $\hat{v}(n)$ cannot be reliably established.
- The modified perceptron learning rule can be illustrated in the following tabular and graphical forms:

$d(n)$	$y(n)$	$\varepsilon(n)$	$\hat{v}(n)$	update if	$\Delta w(n)$
0	0	0	$v < 0$	—	0
1	1	0	$v \geq 0$	—	0
0	1	-1	$v \geq 0$	$v > v_T$	$-\eta\mathbf{x}(n)$
1	0	$+1$	$v \geq 0$	$v < -v_T$	$+\eta\mathbf{x}(n)$



The additional update condition:

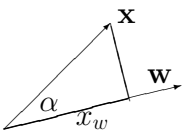
$$|\hat{v}(n)| > v_T \tag{2.5}$$

can be re-written as

$$|\mathbf{w}(n) \cdot \mathbf{x}(n)| > v_T$$

Knowing that the absolute value of the inner product can be expressed as:

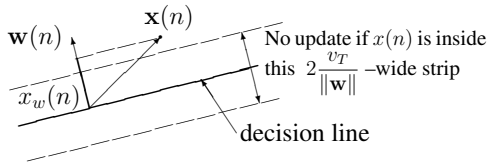
$$|\mathbf{w} \cdot \mathbf{x}| = \|\mathbf{w}\| \cdot |\,\|\mathbf{x}\| \cdot \cos\alpha| = \|\mathbf{w}\| \cdot |\hat{x}_w|$$



where $\|.\|$ denotes the length (norm) of the vector, and $x_w = \|\mathbf{x}\| \cdot \cos\alpha$ is the projection of the input vector, $\mathbf{x}$ on the weight vector, $\mathbf{w}$.

The additional update condition (2.5) can be now written as $\|\mathbf{w}\| \cdot |x_w| > v_T$ or as $|x_w| > \dfrac{v_T}{\|\mathbf{w}\|}$

The condition says that in order the update to occur the current input vector, $\hat{x}$ must lie outside the $2v_T/\|\hat{\mathbf{w}}(n)\|$ strip surrounding the decision plane, as illustrated for the 2–D case in the following figure:



2.7 Intersection of a cube by a plane. A 2-D case.

Consider a straight line described by the following equation

$$\mathbf{w} \cdot \mathbf{x} = 0 \tag{2.6}$$

where

$$\mathbf{w} = [w_1\ w_2\ w_3]\ ;\ \ \mathbf{x} = [x_1\ x_2\ 1]^T$$

Given a square limited by the following four edge lines:

$$x_1 = l_{11}\ ;\ \ x_1 = l_{12}$$
$$x_2 = l_{21}\ ;\ \ x_2 = l_{22}$$

we would like to calculate all four intersection points of the square edges by the straight line:

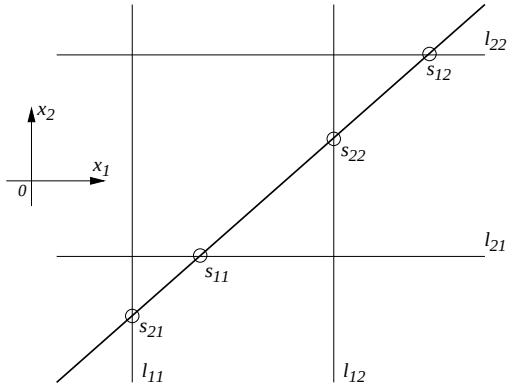


Figure 2–5: A straight line intersecting a square

The coordinates of the four intersection points can be arranged as follows:

$$\left[\begin{array}{c||cc|cc} x_1 & l_{11} & l_{12} & s_{11} & s_{12} \\ x_2 & s_{21} & s_{22} & l_{21} & l_{22} \end{array}\right] \tag{2.7}$$

For each column of table (2.7) we can write eqn (2.6) as

$$[w_1\ w_2] \cdot \begin{bmatrix} s_{11} \\ l_{21} \end{bmatrix} = -w_3\ , \quad [w_1\ w_2] \cdot \begin{bmatrix} s_{12} \\ l_{22} \end{bmatrix} = -w_3$$
$$[w_1\ w_2] \cdot \begin{bmatrix} l_{11} \\ s_{21} \end{bmatrix} = -w_3\ , \quad [w_1\ w_2] \cdot \begin{bmatrix} l_{12} \\ s_{22} \end{bmatrix} = -w_3$$

Grouping the unknown s_{ij} on the left-hand side we have

$$\begin{aligned} w_1[s_{11}\ s_{12}] &= -w_2[l_{21}\ l_{22}] - w_3[1\ 1] \\ w_2[s_{21}\ s_{22}] &= -w_1[l_{11}\ l_{12}] - w_3[1\ 1] \end{aligned}$$

or in a matrix form

$$S = -\begin{bmatrix} 1/w_1 & 0 \\ 0 & 1/w_2 \end{bmatrix} \left(\begin{bmatrix} w_2 & 0 \\ 0 & w_1 \end{bmatrix} L + w_3 \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \right) \tag{2.8}$$

where

$$S = \begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{bmatrix}\ ;\ \ L = \begin{bmatrix} l_{21} & l_{22} \\ l_{11} & l_{12} \end{bmatrix}\ ;$$

2.8 Design Constraints for a Multi-Layer Perceptron

In this section we consider a design procedure for implementation of combinational (logic) circuits using a multi-layer perceptron. In such a context, the perceptron is also referred to as a Linear Threshold Gate (LTG).

Consider a single-hidden-layer perceptron as in Figure 2–6 described by the following equations

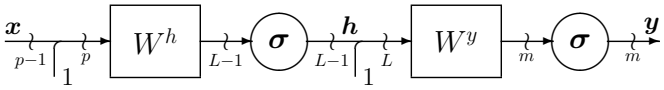


Figure 2–6: A multi-layer perceptron

$$\boldsymbol{h} = \boldsymbol{\sigma}\left(W^h \cdot \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}\right)\ ,\ \ \boldsymbol{y} = \boldsymbol{\sigma}\left(W^y \cdot \begin{bmatrix} \boldsymbol{h} \\ 1 \end{bmatrix}\right) \tag{2.9}$$

where

$$\sigma(x) = \begin{cases} 0 & \text{for } x \le 0 \\ 1 & \text{for } x > 0 \end{cases} \qquad \text{is a step function (a hard limiter).}$$

Note that the step function has been shifted so that for a binary signal

$$x \in \{0, 1\}\ ,\ \ \sigma(x) = x$$

$\boldsymbol{x} = [x_1 \ldots x_{p-1}]^T$ is a $(p-1)$–element input vector,

$\boldsymbol{h} = [h_1 \dots h_{L-1}]^T$ is an $(L-1)$–element hidden vector,

$\boldsymbol{y} = [y_1 \dots y_m]^T$ is an m-element output vector,

W^h is an $(L-1)\times p$ hidden weight matrix, and

W^y is an $m \times L$ output weight matrix.

The input and hidden signals are augmented with constants:

$$x_p = 1\ , \quad \text{and } h_L = 1$$

The total number of adjustable weights is

$$n_w = (L-1)p + m \cdot L = L(p+m) - p \tag{2.10}$$

Design Procedure

Assume that we are given the set of N points

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}(1) & \dots & \boldsymbol{x}(N) \\ \boldsymbol{y}(1) & \dots & \boldsymbol{y}(N) \end{bmatrix} \tag{2.11}$$

to be mapped by the perceptron so that

$$\boldsymbol{y}(n) = f(\boldsymbol{x}(n)) \ \ \forall n = 1, \dots, N. \tag{2.12}$$

Combining eqns (2.9), (2.11), and (2.12) we obtain a set of **design constraints** (equations) which must be satisfied by the perceptron in order for the required mapping to be achieved:

$$\boldsymbol{\sigma}\left(\left[W^y_{:,1:L-1} \cdot \boldsymbol{\sigma}\left(\left[W^h_{:,1:p-1} \cdot X\right] \oplus_c W^h_{:,p}\right)\right] \oplus_c W^h_{:,L}\right) = Y \tag{2.13}$$

where

$$M \oplus_c \mathbf{c} \stackrel{def}{=} M + (\mathbf{1} \otimes \mathbf{c})$$

describe addition of a column vector $\mathbf{c}$ to every column of the matrix M ($\otimes$ denotes the Kronecker product, and $\mathbf{1}$ is a row vector of ones). Eqn (2.13) can be re-written as a set of N individual constraints, one for each output, $\boldsymbol{y}(n)$ where $n = 1, \dots N$.

For a binary case, the eqn (2.11) represent a truth table of m Boolean functions of $(p-1)$ variables. The size of this table is

$$N = 2^{p-1}$$

The constraints (2.13) can now be simplify to

$$\left[W^y_{:,1:L-1} \cdot \boldsymbol{\sigma}\left(\left[W^h_{:,1:p-1} \cdot X\right] \oplus_c W^h_{:,p}\right)\right] \oplus_c W^h_{:,L} = Y \tag{2.14}$$

Conjecture 1 *Under appropriate assumptions the necessary condition for a solution to the design problem as in eqn (2.13) or (2.14) to exist is that the number of weights must be greater that the of points to be mapped, that is,*

$$L(p+m) - p > N$$

Hence, the number of hidden signals

$$L > \frac{N+p}{p+m} \tag{2.15}$$

For a single output $(m = 1)$ binary mapping when $N = 2^{p-1}$ eqn (2.15) becomes

$$L > \frac{2^{p-1}+p}{p+1} \tag{2.16}$$

Once the structural parameters p, L, m are known the weight matrices W_h, W_y that satisfy the constraint equation (2.13), or (2.14) can be selected.

Example

Consider the XOR function for which

$$\left[\frac{X}{Y}\right] = \left[\begin{array}{cccc} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ \hline 0 & 1 & 1 & 0 \end{array}\right], \quad p = 3\,, \quad N = 2^{p-1} = 4 \tag{2.17}$$

From eqn (2.16) we can estimate that

$$L > \frac{N+p}{p+1}\,, \qquad \text{Let} \qquad L = 3$$

The dendritic structure of the resulting two-layer perceptron is depicted in Figure 2–7.

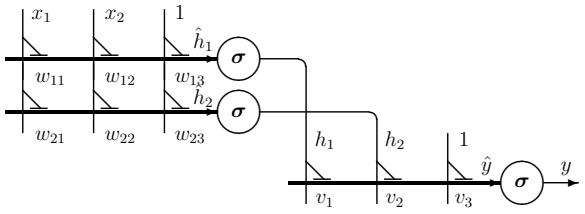


Figure 2–7: A dendritic diagram of a two-layer perceptron which implements the XOR function

A known combination of weights which implements the XOR function specified in eqn (2.17) can be obtained from the following plots in the input and hidden spaces presented in Figure 2–8.
Equations of the straight lines illustrated in Figure 2–8 and related equations for activation potentials ("hat" signals) for all individual perceptrons are as follows

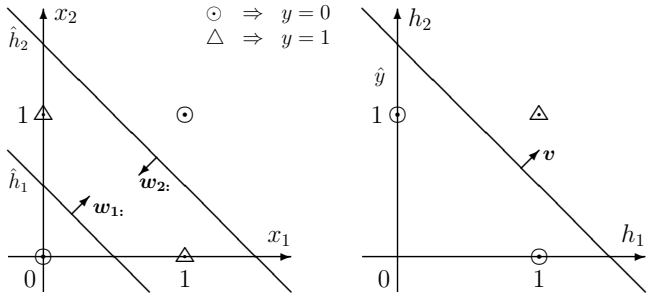


Figure 2–8: A possible configuration of the input and hidden spaces for the XOR perceptron

equations of lines: activation signals:

$$\begin{aligned} x_1 + x_2 = 0.5 &\implies \hat{h}_1 = x_1 + x_2 - 0.5 \\ x_1 + x_2 = 1.5 &\implies \hat{h}_2 = -x_1 - x_2 + 1.5 \\ h_1 + h_2 = 1.5 &\implies \hat{y} = h_1 + h_2 - 1.5 \end{aligned}$$

From the above equations the weight matrices can be assembled as follows

$$W^h = \begin{bmatrix} 1 & 1 & -0.5 \\ -1 & -1 & 1.5 \end{bmatrix}, \quad W^y = \boldsymbol{v} = \begin{bmatrix} 1 & 1 & -1.5 \end{bmatrix} \tag{2.18}$$

Note that:

$$\left[\frac{X}{H}\right] = \left[\begin{array}{cccc} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ \hline 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{array}\right] \tag{2.19}$$

Constraint Equations for the XOR perceptron

In order to calculate other combinations of weights for the XOR perceptron we can re-write the constraint equation (2.14) using the structure as in Figure **??**. The result is as follows

$$\begin{bmatrix} v_1 & v_2 \end{bmatrix} \cdot \boldsymbol{\sigma}\left(\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \oplus_c \begin{bmatrix} w_{13} \\ w_{23} \end{bmatrix} \right) + v_3 = \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix} \tag{2.20}$$

Eqn (2.20) is equivalent to the following four scalar equations, the total number of weights to be selected being 9.

$$\begin{aligned} &v_1 \cdot \sigma(w_{13}) + v_2 \cdot \sigma(w_{23}) + v_3 = 0 \\ &v_1 \cdot \sigma(w_{11} + w_{13}) + v_2 \cdot \sigma(w_{21} + w_{23}) + v_3 = 1 \\ &v_1 \cdot \sigma(w_{12} + w_{13}) + v_2 \cdot \sigma(w_{22} + w_{23}) + v_3 = 1 \\ &v_1 \cdot \sigma(w_{11} + w_{12} + w_{13}) + v_2 \cdot \sigma(w_{21} + w_{22} + w_{23}) + v_3 = 0 \end{aligned}$$

It can easily be verified that the weights specified in eqn (2.18) satisfy the above constraints.

3 ADALINE — The Adaptive Linear Element

- The Adaline can be thought of as the smallest, linear building block of the artificial neural networks.
- This element has been extensively used in science, statistics (in the linear **linear regression analysis**), engineering (the **adaptive signal processing**, **control systems**), and many other areas.

In general, the Adaline is used to perform

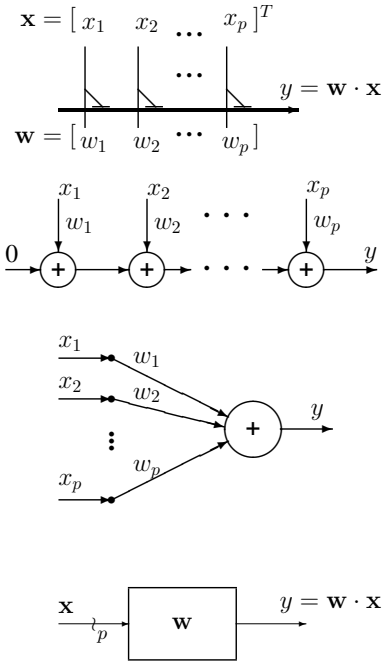
linear approximation of a "small" segment of a nonlinear hyper-surface, which is generated by a p–variable function, $y = f(\mathbf{x})$.

In this case, the bias is usually needed, hence, $w_p = 1$.

linear filtering and prediction of data (signals)

pattern association, that is, generation of m–element output vectors (using m Adalines) associated with respective p–element input vectors.

Various representations of the Adaline:



3.1 Linear approximation of a p-variable function

- A function of p variables, $y = f(\mathbf{x})$, can be interpreted as a hyper-surface in a $(p+1)$–dimensional space, $[x_1, \ldots x_p, y]$.
- In this section, we will discuss methods of approximating such a surface by a hyperplane using an Adaline. It is also known as a linear regression problem: Given

  $$d = f(\mathbf{x}), \text{ we find a hyperplane } y = \mathbf{w} \cdot \mathbf{x} \text{ such that the error } \varepsilon = |d - y| \text{ is small for all } \mathbf{x}.$$

- We start with a bit more general problem, namely, approximation of m such functions using m p-input Adalines.
- Let the functions to be linearly approximated be known at N points, $\mathbf{x}(n)$, $\mathbf{d}(n)$ being a vector of values of functions.
- N points (training patterns) can be arranged, as previously, in the following two matrices:

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

- In order to approximate the above function let us consider a p-input m-output Adaline characterised by an $m \times p$ weight matrix, W, each row related to a single neuron.
- For each input vector, $\mathbf{x}(n)$, the Adaline calculates the actual output vector

  $$\mathbf{y}(n) = W \cdot \mathbf{x}(n) \,. \qquad (3.1)$$

- All output vectors can also be arranged in an output matrix:

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

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

$$Y = W\cdot X \tag{3.2}$$

- Typically, the actual output vector, $\mathbf{y}(n)$ differs from the desired output vector, $\mathbf{d}(n)$, and the **pattern error**:

$$\boldsymbol{\varepsilon}(n) = \mathbf{d}(n) - \mathbf{y}(n)\quad\text{is a}\;\; m\times 1\;\;\text{vector,} \tag{3.3}$$

each component being equal to:

$$\varepsilon_j(n) = d_j(n) - y_j(n)\;. \tag{3.4}$$

- The problem of approximation of the surfaces specified by D by the hyper-planes specified by weight vectors stored in the weight matrix, W, is to select the weights so that the errors are as small as possible.
- The total measure of the goodness of approximation, or the **performance index**, can be specified by the **mean-squared error** over m neurons and N training vectors:

$$J(W) = \frac{1}{2mN}\sum_{n=1}^{N}\sum_{j=1}^{m}\varepsilon_j^2(n) \tag{3.5}$$

- Defining the total **instantaneous** error over m neurons as:

$$E(n) = \frac{1}{2}\sum_{j=1}^{m}\varepsilon_j^2(n) = \frac{1}{2}\boldsymbol{\varepsilon}^T(n)\cdot\boldsymbol{\varepsilon}(n) \tag{3.6}$$

the performance index can be expressed as

$$J(W) = \frac{1}{2mN}\sum_{n=1}^{N}E(n) = \frac{1}{2mN}\,\mathbf{e}^T\cdot\mathbf{e} \tag{3.7}$$

where $\mathbf{e}$ is a $mN\times 1$ vector consisting of all errors which can be calculated as:

$$\mathcal{E} = D - Y\;;\quad \mathbf{e} = \mathcal{E}(:)$$

where ':' is the MATLAB column-wise scan operator.

- The performance index, $J(W)$, is a non-negative scalar function of $(m\cdot p)$ weights (a quadratic surface in the **weight space**).
- To solve the approximation problem, we will determine the weight matrix which minimises the performance index, that is, the **mean-squared error**, $J(W)$.
- For simplicity, solution to the approximation problem will be given for a **single-neuron** case (single output), when $m = 1$. Now, $\mathbf{e}^T = \mathcal{E} = D - Y$.
- The weight matrix, W, becomes the $1\times p$ vector, $\mathbf{w}$ and the **mean-squared error**, $J(\mathbf{w})$, can now be calculated in the following way:

$$J(\mathbf{w}) = \frac{1}{2N}(D-Y)(D-Y)^T = \frac{1}{2N}(D\cdot D^T - D\cdot Y^T - Y\cdot D^T + Y\cdot Y^T)$$

where D and $Y = \mathbf{w}\cdot X$ are now $1\times N$ row-matrices.

- If we take into account that the inner product of vectors is commutative, that is, $\mathbf{u}^T\cdot\mathbf{v} = \mathbf{v}^T\cdot\mathbf{u}$, then we have

$$\begin{aligned} J &= \frac{1}{2N}(\|D\|^2 - 2DY^T + YY^T) \\ &= \frac{1}{2N}(\|D\|^2 - 2DX^T\mathbf{w}^T + \mathbf{w}XX^T\mathbf{w}^T) \end{aligned}$$

- Denote by

$$\mathbf{q} = (D\cdot X^T)/N \quad \text{the } 1\times p \textbf{ cross-correlation} \text{ vector,} \tag{3.8}$$

$$R = (X\cdot X^T)/N \quad \text{the } p\times p \textbf{ input correlation} \text{ matrix.} \tag{3.9}$$

- Then the mean-squared error finally becomes

$$J(\mathbf{w}) = \frac{1}{2}(\|D\|^2/N - 2\mathbf{q}\mathbf{w}^T + \mathbf{w}R\mathbf{w}^T) \tag{3.10}$$

Example

Consider an example of a 2-D performance index when $(p=2)$ and $\mathbf{w} = [w_1\ w_2]$.
Eqn (3.10) is of the following matrix form:

$$J(\mathbf{w}) = \frac{1}{2}\mathbf{w}R\mathbf{w}^T - \mathbf{q}\mathbf{w}^T + c$$

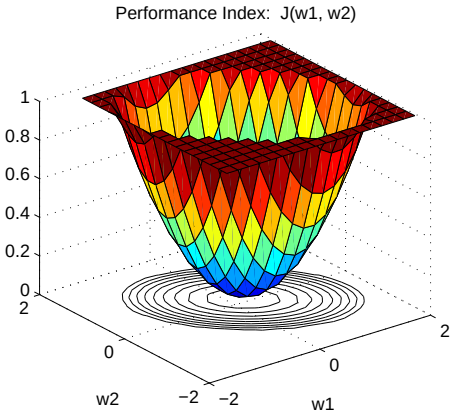
Let

$$J(w_1,w_2) = [w_1\ w_2]\begin{bmatrix} 9 & 4 \\ 4 & 10 \end{bmatrix}\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} - [5\ \ 4]\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} + 1 \tag{3.11}$$

We can re-write eqn (3.11) in the following "scalar" form:

$$J(w_1,w_2) = 9w_1^2 + 8w_1w_2 + 10w_2^2 - 5w_1 - 4w_2 + 1$$

The plot of the normalised performance index, $J(w_1,w_2)$ has been generated by a MATLAB script cJ2x.m:



In order to find the optimal weight vector which minimises the mean-squared error, $J(\mathbf{w})$, we calculate the gradient of J with respect to $\mathbf{w}$:

$$\nabla J(\mathbf{w}) = \frac{\partial J}{\partial \mathbf{w}} = \left[\frac{\partial J}{\partial w_1} \ \cdots \ \frac{\partial J}{\partial w_p}\right] = \frac{1}{2}\nabla(\|D\|^2/N - 2\mathbf{q}\mathbf{w}^T + \mathbf{w}R\mathbf{w}^T) = -\mathbf{q} + \mathbf{w}R^T$$

Taking into account that $R = R^T$ (a symmetric matrix), the gradient of the performance index finally becomes:

$$\nabla J(\mathbf{w}) = -\mathbf{q} + \mathbf{w}R \tag{3.12}$$

The gradient, $\nabla J(\mathbf{w})$, becomes zero for:

$$\mathbf{w}R = \mathbf{q} \tag{3.13}$$

This is a very important equation known as the **normal** or **Wiener–Hopf** equation.

This is a **set of** p **linear equations** for $\mathbf{w} = [w_1 \ \cdots \ w_p]$.

The solution, if exists, can be easily found, and is equal to:

$$\mathbf{w} = \mathbf{q} \cdot R^{-1} = \mathbf{q}/R = DX^T(XX^T)^{-1} \tag{3.14}$$

Using a concept of the **pseudo-inverse** of a matrix X defined as:

$$X^+ \stackrel{def}{=} X^T(XX^T)^{-1} \tag{3.15}$$

the optimal in the least-mean-squared sense weight vector can be also calculated as

$$\mathbf{w} = D \cdot X^+ = D/X \tag{3.16}$$

Example: The performance index of eqn (3.11) attains minimum for

$$\mathbf{w} = [w_1 \ w_2] = \frac{1}{2}\,[5 \ \ 4]\begin{bmatrix} 9 & 4 \\ 4 & 10 \end{bmatrix}^{-1} = [0.23 \ \ 0.11]$$

□

In the **multi-neuron** case, when D is a $m \times N$ matrix, the optimal weight matrix W (which is $m \times p$) can be calculated in a similar way as:

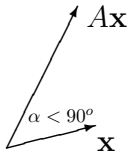
$$W = D \cdot X^+ = D/X \tag{3.17}$$

- In order to check that the above weight vector really minimises the performance index, we calculate the **second derivative** of J which is known as the **Hessian matrix**:

$$H(\mathbf{w}) = \frac{\partial^2 J}{\partial \mathbf{w}^2} = \frac{\partial}{\partial \mathbf{w}}(\nabla J(\mathbf{w})) = R \tag{3.18}$$

- The second derivative is independent of the weight vector and is equal to the input correlation matrix, R.
- R, as a product of X and X^T, can be proved to be a **positive-definite** matrix.
- Moreover, if the number of linearly independent input vectors is at least p, then the R matrix is of full rank.
- This means that the performance index attains minimum for the optimal weight vector, and that the minimum is unique.
- A matrix A is said to be positive-definite if and only if for all non-zero vectors $\mathbf{x}$

$$\mathbf{x}^T \cdot A \cdot \mathbf{x} > 0$$



- It can be shown that the eigenvalues of a positive-definite matrix are real and positive.

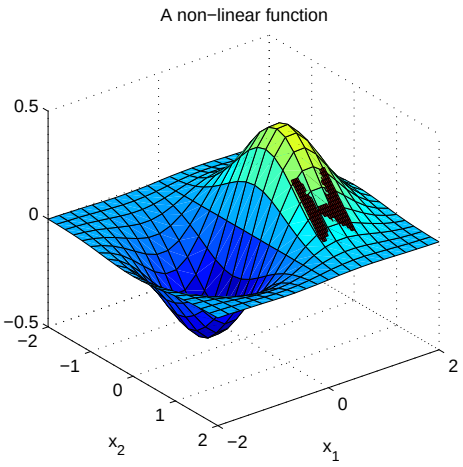
Approximation by a plane — MATLAB example (adln1.m)

In this example, we approximate a small section of a nonlinear 2-D surface with a plane which is specified by a weight vector of a linear neuron.

First, for the 2-D input domain $\mathbf{x} = [x_1, x_2] \in \{-2, 2\}$ we calculate 21×21 points of the Gaussian-like function:

$$y = f(\mathbf{x}) = x_1 e^{(-x_1^2 - x_2^2)}$$

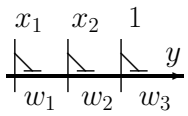
```
N = 20; NN = (0:N)/N ; M = (N+1)^2 ;
x1 = 4*NN - 2 ;
[X1 X2] = meshgrid(x1, x1);
y = X1 .* exp(-X1.^2 - X2.^2);
figure(1), clf reset
surf(x1, x1, y), axis('ij'), hold on
```



Next, we select a small segment of the surface, for $x_1 \in \{0.4 \texttt{ .. } 1\}, x_2 \in \{0.6 \texttt{ .. } 1.4\}$, and form the set of input points (training vectors), X, taken from the points of the 21×21 grid.

The desired output values D are calculated from the equation of the function being approximated, $d = f(\mathbf{x})$

We need also the bias input $x_3 = 1$, so that the equation of the approximating plane and the related Adaline will be:



$$y = w_1 \cdot x_1 + w_2 \cdot x_2 + w_3$$

```
x1 = 0.6*NN+0.4;
x2 = 0.8*NN+0.6;
[X1 X2] = meshgrid(x1, x2);
d = X1 .* exp(-X1.^2 - X2.^2);
D = d(:)';
X = [X1(:)'; X2(:)'; ones(1,M)];
```

The three initial and four last training vectors $\mathbf{x}(k) = [x_1\ x_2\ 1]^T$, and $d(n)$ are of the following form:

```
X(:, [1:3 (M-3):M]) =
    0.40  0.40  0.40 ... 1.00  1.00  1.00  1.00
    0.60  0.64  0.68 ... 1.28  1.32  1.36  1.40
    1.00  1.00  1.00 ... 1.00  1.00  1.00  1.00

D(:, [1:3 (M-3):M]) =
   0.2378 0.2263 0.2147 ... 0.0715 0.0644 0.0579 0.0518
```

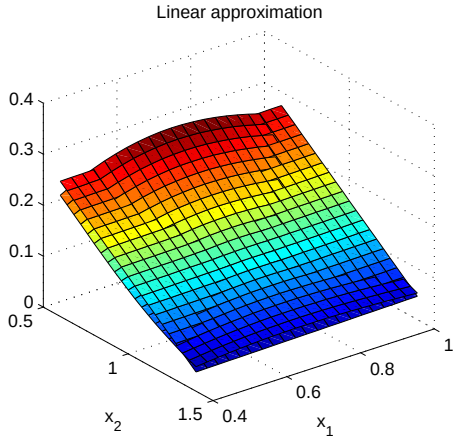
Then we calculate the cross-correlation vector $\mathbf{q}(n)$ and the input correlation matrix R.

```
q = (D*X')/M = 0.1093 0.1388 0.1555
R = (X*X')/M =  0.52  0.70  0.70
                0.70  1.06  1.00
                0.70  1.00  1.00
```


The eigenvalues of the input correlation matrix are real and positive which indicates that R is positive-definite and of full rank:

```
eig(R) = 0.0175 0.0438 2.5203

w =  q/R  = 0.0132  -0.2846  0.4309   % THE  SOLUTION!

Y = w*X;   Y(:, [1:3 438:441]) =
  0.2654  0.2540  0.2426 ... 0.0798  0.0684  0.0570  0.0456
```

```
err = sum(abs(D-Y)) =  1.54
YY = d ; YY(:) = Y;
surf(x1, x2, YY), hold off

figure(2)
surf(x1, x2, d),  axis('ij'), hold on
surf(x1, x2, YY), hold off
```



3.2 Method of steepest descent

- In order to calculate the optimal weights which minimise the approximation error, $J(\mathbf{w})$, we have to
  - calculate the correlation matrices, $\mathbf{q}$ and R, and
  - inverse the autocorrelation matrix, R, as in eqn (3.14).
- The above operations can be computationally intensive for a large number of training patterns, N.
- In addition we might prefer a procedure that finds an optimal weight vector iteratively, in a pattern-by-pattern fashion as for the perceptron.
- We assume that at each step the weight vector is updated by a small update vector:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \Delta\mathbf{w}(n) \tag{3.19}$$

- We can now estimate how the above change in the weight vector is reflected in change of the value of the performance (error) function $J(\mathbf{w})$ defined in eqn (3.5). We have

$$\begin{aligned} J(\mathbf{w}+\Delta\mathbf{w}) &= \frac{1}{2}(\|D\|^2/N - 2\cdot\mathbf{q}\cdot(\mathbf{w}+\Delta\mathbf{w})^T + (\mathbf{w}+\Delta\mathbf{w})\cdot R\cdot(\mathbf{w}+\Delta\mathbf{w})^T) \\ &= J(\mathbf{w}) + \frac{1}{2}(-2\cdot\mathbf{q}\cdot\Delta\mathbf{w}^T + 2\cdot\mathbf{w}\cdot R\cdot\Delta\mathbf{w}^T + \Delta\mathbf{w}\cdot R\cdot\Delta\mathbf{w}^T) \\ &= J(\mathbf{w}) + (-q + \mathbf{w}\cdot R)\cdot\Delta\mathbf{w}^T + \frac{1}{2}\Delta\mathbf{w}\cdot R\cdot\Delta\mathbf{w}^T) \end{aligned}$$

Finally, we have

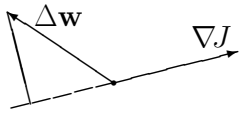
$$J(\mathbf{w}+\Delta\mathbf{w}) = J(\mathbf{w}) + \nabla J(\mathbf{w})\cdot\Delta\mathbf{w}^T + \frac{1}{2}\Delta\mathbf{w}\cdot R\cdot\Delta\mathbf{w}^T \tag{3.20}$$

- In an iterative search for the minimum of the performance index we would like its value to decrease at each step, that is

$$J(\mathbf{w}+\Delta\mathbf{w}) < J(\mathbf{w}) \quad \text{or} \quad J(\mathbf{w}(n+1)) < J(\mathbf{w}(n))$$

- If in eqn (3.20) we neglect the second order term, then the above condition is equivalent to:

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



- This condition means that in order to reduce the value of J we should move in the direction $\Delta\mathbf{w}$ such that its projection on $\nabla J(\mathbf{w})$ is negative.
- The most negative results is obtained when

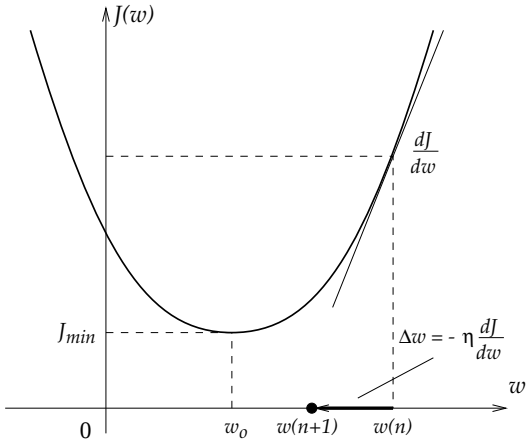
$$\Delta\mathbf{w} = -\eta\,\nabla J(\mathbf{w}) = \eta(q - \mathbf{w}\cdot R) \tag{3.21}$$

- This is the **steepest descent method** that states that in order to reduce J in successive steps, the weight vector should be modified in the direction **opposite to the gradient** of the error function J.
- η is an important parameter known as the **learning gain**, and $\mathbf{q}$ and R are cross- and input correlation matrices defined in eqns (3.8), (3.9).

- The steepest descent learning law of eqn (3.21) is a **batch method**, that is,
  for a complete set of data, N points being stored in X and D,
  the cross-correlation vector $\mathbf{q}$ and
  the input correlation matrix R is calculated and
  iterations are performed until $\Delta\mathbf{w}$ is close to zero.
- When $\Delta\mathbf{w} = 0$, then $\mathbf{w}R = q$, $\nabla J(\mathbf{w}) = 0 \implies \mathbf{w}$ minimises J eqn (3.13)

Illustration of the steepest descent method:

- Moving in the direction opposite to the gradient of the performance index in the weight space takes us towards the minimum of error.
- When the weight vector attains the optimal value for which the gradient is zero (w_0 in the figure), the iterations are stopped.



Optimal learning gain

Optimal/maximum learning gain can be estimated in the following way.

- Using the steepest descent rule of eqn (3.21) the full expression for the weight update becomes:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta(q - \mathbf{w}(n) \cdot R)$$

  or

$$\mathbf{w}(n+1) = \mathbf{w}(n)(I - \eta\, R) + \eta\, q \tag{3.22}$$

  where I is an identity matrix of the same size as R.
- This recursive equation describe a linear dynamic system with weights being its state vector.
- It is known that for such a system to be stable, that is, weights to converge to a fixed value, the eigenvalues of the state matrix $(I - \eta\, R)$ must be located inside the unit circle, that is:

$$|\text{eig}(I - \eta\, R)| < 1 \tag{3.23}$$

- If $(\lambda_i, \mathbf{v}_i)$ are an eigenvalue–eigenvector pair, than we have:

$$(I - \eta\, R) \cdot \mathbf{v}_i = \lambda_i \cdot \mathbf{v}_i$$

- This can be re-written as:

$$R \cdot \mathbf{v}_i = \frac{1 - \lambda_i}{\eta}\mathbf{v}_i\,, \quad \text{or} \quad R \cdot \mathbf{v}_i = \lambda_{Ri}\,\mathbf{v}_i\,, \quad \text{where} \quad \lambda_{Ri} = \frac{1 - \lambda_i}{\eta}$$

  is an eigenvalue of the input correlation matrix R (due to the uniqueness of the eigen-decomposition).

- The condition (3.23) can now be written in terms of the eigenvalues of the correlation matrix as:

$$-1 < 1 - \eta\,\lambda_{Ri} < 1\,, \quad \text{for all} \quad i = 1, \ldots, p$$

  Note that eigenvalues of the correlation matrix are real and positive.

- Finally the condition for the **maximum stable learning gain** can be written in the following simple form:

$$\eta_{mx} = \frac{2}{\lambda_{mx}} \tag{3.24}$$

  where λ_{mx}, the largest eigenvalue of the input correlation matrix, represents the maximum curvature of the quadratic function J and

- the condition (3.24) states that the maximum stable learning gain is inversely proportional to this curvature.

Pattern learning

- Often we have a situation when the samples of data $(\mathbf{x}(n), d(n))$ arrive one at a time and we would like to get a weight update specifically for the new data sample.
- This is know as the pattern learning and two incremental learning law, LMS and RMS are discussed in subsequent sections.

3.3 The LMS (Widrow-Hoff) Learning Law

- The **Least-Mean-Square** (LMS) algorithm also known as the Widrow-Hoff Learning Law, or the **Delta Rule** is based on the instantaneous update of the correlation matrices, hence, on the **instantaneous update of the gradient** of the mean-squared error.
- To derive the instantaneous update of the gradient vector we will first express the current values of the correlation matrices in terms of their previous values (at the step $n-1$) and the updates at the step n.
- First observe that the current input vector $\mathbf{x}(n)$ and the desired output signal $d(n)$ are appended to the matrices $\mathbf{d}(n-1)$ and $X(n-1)$ as follows:

$$\mathbf{d}(n) = [\mathbf{d}(n-1)\; d(n)]\,, \;\; \text{and} \;\; X(n) = [X(n-1)\; \mathbf{x}(n)]$$

- Now using definitions of correlation matrices of eqns (3.8) and (3.9) we can write:

$$\begin{aligned}\begin{bmatrix} \mathbf{q}(n) \\ R(n) \end{bmatrix} &= \Big(\begin{bmatrix} \mathbf{d}(n-1) & d(n) \\ X(n-1) & \mathbf{x}(n) \end{bmatrix}\begin{bmatrix} X^T(n-1) \\ \mathbf{x}^T(n) \end{bmatrix}\Big)/n = \Big(\begin{bmatrix} \mathbf{d}(n-1)\,X^T(n-1) + d(n)\,\mathbf{x}^T(n) \\ X(n-1)\,X^T(n-1) + \mathbf{x}(n)\,\mathbf{x}^T(n) \end{bmatrix}\Big)/n \\ &= \mu\begin{bmatrix} \mathbf{q}(n-1) \\ R(n-1) \end{bmatrix} + \begin{bmatrix} \Delta\mathbf{q}(n) \\ \Delta R(n) \end{bmatrix}\end{aligned} \tag{3.25}$$

  where

$$\mu = \frac{n-1}{n} \approx 1 \quad \text{, and}$$

$$\Delta\mathbf{q}(n) = (d(n)\,\mathbf{x}^T(n))/n \quad \text{and} \quad \Delta R(n) = (\mathbf{x}(n)\,\mathbf{x}^T(n))/n \tag{3.26}$$

  are the instantaneous updates of the correlation matrices.

- The gradient of the mean-squared error at the step n can also be expanded into its previous values and the current update.
- From eqn (3.12), we have

$$\nabla J(n) = -\mu(\mathbf{q}(n-1) - \mathbf{w}(n)R(n-1)) - \Delta\mathbf{q}(n) + \mathbf{w}(n)\Delta R(n)$$

or

$$\nabla J(n) = \nabla\hat{J}(n-1) + \Delta\nabla J(n)\,, \quad \text{where} \quad \nabla\hat{J}(n-1) = \mu(-\mathbf{q}(n-1) + \mathbf{w}(n)R(n-1))$$

is the current (step n) estimate of the previous (step $n-1$) value of the gradient vector, and the gradient vector update is:

$$\begin{aligned}\Delta\nabla J(n) &= -\Delta\mathbf{q}(n) + \mathbf{w}(n)\Delta R(n) = -\frac{1}{n}(d(n)\,\mathbf{x}^T(n) - \mathbf{w}(n)\mathbf{x}(n)\,\mathbf{x}^T(n)) \\ &= -\frac{1}{n}(d(n) - \mathbf{w}(n)\mathbf{x}(n))\mathbf{x}^T(n) = -\frac{1}{n}(d(n) - y(n))\mathbf{x}^T(n) = -\frac{1}{n}\varepsilon(n)\mathbf{x}^T(n)\end{aligned}$$

- The **Least-Mean-Square** learning law replaces the gradient of the mean-squared error in eqn (3.21) with the **gradient update** and can be written in following form:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta_n \cdot \varepsilon(n) \cdot \mathbf{x}^T(n) \tag{3.27}$$

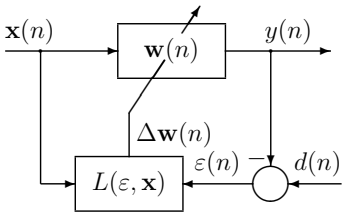
where the output error is

$$\varepsilon(n) = d(n) - y(n)$$

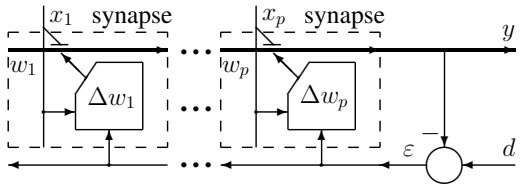
and the learning gain η_n can be either constant or reducible by the factor $1/n$.

Adaline with its error-correcting learning

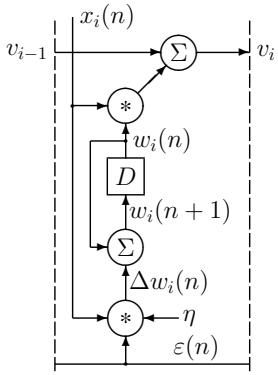
a. Block-diagram structure of an ADALINE



b. Dendritic structure of an ADALINE



c. Detailed structure of an ith synapse implementing the LMS learning



- The LMS weight update for the ADALINE is:

$$\Delta\mathbf{w}(n) = \eta \cdot \varepsilon(n) \cdot \mathbf{x}^T(n)$$

- The weight update for a single synapse is:

$$\Delta w_i(n) = \eta\,\varepsilon(n)\,x_i(n) \tag{3.28}$$

Some comments:

- The LMS learning law works because the instantaneous gradient update, $\Delta\nabla J(n)$ points, on average, in the direction of the gradient, $\nabla J(n)$, of the error function J.
- The LMS learning law of eqn (3.27) can be easily expanded into the case of the multi-neuron Adaline, describes a linear mapping from p-dimensional input space into an m-dimensional output space (m hyper planes).

Stopping criteria of the learning process:

- If it is possible, the learning process goes through all training examples (an epoch) number of times, until a **stopping criterion** is reached.
- The convergence process can be monitored with the plot of the mean-squared error function $J(W(n))$.
- The popular stopping criteria are:
  - the mean-squared error is sufficiently small:
$$J(W(n)) < \epsilon$$
  - The rate of change of the mean-squared error is sufficiently small:
$$\frac{\partial J(W(n))}{\partial n} < \epsilon$$

3.4 A Sequential Regression algorithm

The **sequential regression** algorithm also known as the **recursive least-square** (RLS) algorithm is based on the sequential update of the inverse of the input correlation matrix, $R^{-1}(n)$ for each new input pattern, $\mathbf{x}(n)$.

- Re-call that according to the **Wiener–Hopf** equation (3.13), the optimal weight vector can be calculated as
$$\mathbf{w} = \mathbf{q} \cdot R^{-1} \tag{3.29}$$
- In the sequential regression algorithm, the inverse of the input correlation matrix, R, is calculated iteratively as
$$R^{-1}(n) = \Big(X(n) \cdot X^T(n)\Big)^{-1} = f\left(R^{-1}(n-1), \mathbf{x}(n)\right) \tag{3.30}$$
that is, the current value of the inverse, $R^{-1}(n)$ is calculated from the previous value of the inverse, $R^{-1}(n-1)$, and the input vector, $\mathbf{x}(n)$.
- As a starting point let us rewrite eqn (3.25) and the Wiener–Hopf equation (3.13) in the following forms:
$$\mathbf{q}(n) = \mu\,\mathbf{q}(n-1) + (d(n) \cdot \mathbf{x}^T(n))/n\,, \quad \text{where} \quad \mu = \frac{n-1}{n} \approx 1 \tag{3.31}$$
$$\mathbf{q}(n) = \mathbf{w}(n) \cdot R(n)\,, \quad \mathbf{q}(n-1) = \mathbf{w}(n-1) \cdot R(n-1) \tag{3.32}$$

- Substituting eqns (3.32) into eqn (3.31) we have

$$\mathbf{w}(n) \cdot R(n) = \mu \mathbf{w}(n-1) \cdot R(n-1) + (d(n) \cdot \mathbf{x}^T(n))/n \tag{3.33}$$

- From eqn (3.25), the current value of the input correlation matrix is related to its next value in the following way:

$$\mu R(n-1) = R(n) - (\mathbf{x}(n) \cdot \mathbf{x}^T(n))/n \tag{3.34}$$

- Substitution of eqn (3.34) into eqn (3.32) yields:

$$\mathbf{w}(n) \cdot R(n) = \mathbf{w}(n-1) \cdot R(n) - (\mathbf{w}(n-1) \cdot \mathbf{x}(n) \cdot \mathbf{x}^T(n))/n + (d(n) \cdot \mathbf{x}^T(n))/n$$

- Let us denote the scaled **inverse of the input correlation matrix** as:

$$P(n) = \frac{1}{n} R^{-1}(n)$$

- Post-multiplying eqn (3.4) by the inverse, $P(n)$, gives:

$$\mathbf{w}(n) = \mathbf{w}(n-1) + (d(n) - \mathbf{w}(n-1) \cdot \mathbf{x}(n)) \cdot \mathbf{x}^T(n) \cdot P(n) \tag{3.35}$$

- Denote by

$$\tilde{y}(n) = \mathbf{w}(n-1) \cdot \mathbf{x}(n)$$

the **estimated output signal** based on the previous weight vector, $\mathbf{w}(n)$, and by

$$\varepsilon(n) = d(n) - \tilde{y}(n) = d(n) - \mathbf{w}(n-1) \cdot \mathbf{x}(n) \tag{3.36}$$

the error between the desired and estimated output, and by

$$\mathbf{k}(n) = \mathbf{x}^T(n) \cdot P(n) \tag{3.37}$$

the **update vector** known as the **Kalman gain**.

- Then, from eqns (3.35), (3.36) and (3.37), the sequential **weight update** can be expressed as

$$\mathbf{w}(n) = \mathbf{w}(n-1) + \varepsilon(n) \cdot \mathbf{k}(n) \tag{3.38}$$

- Eqn (3.38) describes the sequential regression algorithms in terms of the output error, $\varepsilon(n)$, and the update vector (Kalman gain), $\mathbf{k}(n)$, which involves calculation of the inverse of the input correlation matrix, $P(n)$.

- In order to derive an iterative expression for this inverse we will need the **matrix inversion lemma**.

According to this lemma, it is possible to show that if R, A, B are appropriately dimensioned matrices (i.e., $p \times p,\ p \times m,\ m \times p$, respectively), then

$$(R + A \cdot B)^{-1} = R^{-1} - R^{-1} \cdot A \cdot (I_m + B \cdot R^{-1} \cdot A)^{-1} \cdot B \cdot R^{-1} \tag{3.39}$$

- Let us re-write eqn (3.34) as

$$(n-1)R(n-1) = n\,R(n) - \mathbf{x}(n) \cdot \mathbf{x}^T(n) \tag{3.40}$$

and apply the matrix inversion lemma to it.

- The scaled inverse of the input correlation matrix can be now written in the following form:

$$P(n) = P(n-1) - \mathbf{r}(n)\left(1 + \mathbf{r}^T(n)\mathbf{x}(n)\right)^{-1}\mathbf{r}^T(n)$$

or

$$P(n) = P(n-1) - \frac{\mathbf{r}(n)\cdot\mathbf{r}^T(n)}{1+\mathbf{r}^T(n)\cdot\mathbf{x}(n)} \tag{3.41}$$

where

$$\mathbf{r}(n) = P(n-1)\cdot\mathbf{x}(n) \tag{3.42}$$

is the **input gain vector** similar to the Kalman gain.

- The update vector (Kalman gain) specified in eqn (3.37) can now be expressed as

$$\mathbf{k}(n) = \mathbf{r}^T(n) - \frac{\mathbf{x}^T(n)\cdot\mathbf{r}(n)\cdot\mathbf{r}^{\mathbf{T}}(n)}{1+\mathbf{r}^T(n)\cdot\mathbf{x}(n)}$$

- or, finally, in the form

$$\mathbf{k}(n) = \frac{\mathbf{r}^T(n)}{1+\mathbf{r}^T(n)\cdot\mathbf{x}(n)} \tag{3.43}$$

- Substitution of eqn (3.43) into eqn (3.41) finally yields equation for the iteration step for the inverse of the input correlation matrix:

$$P(n) = P(n-1) - P(n-1)\cdot\mathbf{x}(n)\cdot\mathbf{k}(n) \tag{3.44}$$

- This equation can, alernatively, be written in the following form

$$P(n) = P(n-1)\left(I_p - \mathbf{x}(n)\cdot\mathbf{k}(n)\right) \tag{3.45}$$

The Sequential Regression (SR) or Recursive Least-Square (RLS) algorithm — Summary

- It can be shown that using the SR algorithm the final value of the estimated input correlation matrix is

$$\hat{R}(N) = R(N) + \frac{P^{-1}(0)}{N}$$

Therefore, the initial value of the inverse of the input correlation matrix should be large to minimise the final error.

- Another problem to consider is that in practical applications we would like the algorithm to work continuously, that is, for large N, but with only the most recent input samples to contribute to the estimate of the correlation matrix.
- This is achieved by introduction of the "forgetting factor", λ in estimation of the correlation matrix.

The practical version of the RLS can be summarised as follows.

Initialisation:

$$\begin{aligned} P(1) &= R^{-1}(1) \quad \text{to be LARGE, e.g.} \quad P(1) = 10^6 I_p \\ \mathbf{w}(1) &= \text{small, random} \end{aligned}$$

an nth iteration step:

- Calculate the input gain vector $(p \times 1)$ as

$$\mathbf{r}(n) = \lambda^{-1} P(n-1) \cdot \mathbf{x}(n) \tag{3.46}$$

  where $0 < \lambda < 1$ is the forgetting factor.

- Calculate the Kalman gain vector $(1 \times p)$

$$\mathbf{k}(n) = \frac{\mathbf{r}^T(n)}{1 + \mathbf{r}^T(n) \cdot \mathbf{x}(n)} \tag{3.47}$$

  (A single neuron case $m = 1$ is assumed)

- Calculate the error signal

$$\varepsilon(n) = d(n) - \mathbf{w}(n) \cdot \mathbf{x}(n) \tag{3.48}$$

- Update the weight vector

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \varepsilon(n) \cdot \mathbf{k}(n) \tag{3.49}$$

- Calculate the next estimate of the **inverse** of the input correlation matrix

$$P(n+1) = \lambda^{-1} P(n) - \mathbf{r}(n) \cdot \mathbf{k}(n) \tag{3.50}$$

  where $\mathbf{r}(n)\mathbf{k}(n)$ is the outer product ($p \times p$ matrix) of the corresponding vectors.

  The forgetting factor, λ, de-emphasises contribution to the estimate of the inverse of the input correlation (covariance) matrix from older input data.

  A similar effect could be achieved by a periodic re-initialisation of the P matrix.

Another interpretation

- The RLS algorithm can be also seen as a way of optimal filtering the true signal, $d(n)$, from the output signal $y(n) = \mathbf{w}(n) \cdot \mathbf{x}(n)$.

- The error equation (3.48) can be re-written as a measurement equation:

$$d(n) = \mathbf{w}(n) \cdot \mathbf{x}(n) + \varepsilon(n)$$

  where $\varepsilon(n)$ is now the observation noise.

- The filtering procedure described above is known as the **Kalman filter**, and eqn (3.50) is known as the **Riccatti difference equation**.

3.5 ADALINE as an adaptive linear filter

- Traditional and very important applications of Linear Neural Networks are in the area one-dimensional adaptive signal processing, digital filtering and **time-series** prediction.
- A **digital filter**, such as an Adaline, is an algorithm executed either on a general purpose computer or specialised Digital Signal Processors (DSPs).
- In real-time signal processing we typically deal with an analog 1-D signal, $x(t)$, generated by some physical devices, for example, a microphone.
- The analog signal is passed through an Analog-to-Digital converter which does two operations: samples the analog signal with a given frequency, $f_s = 1/t_s$, and converts the samples into b-bit numbers, $x(n)$

$$x(t) \stackrel{t=n\cdot t_s}{\Longrightarrow} x(n)$$

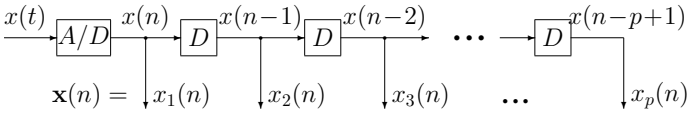
- Typically, more than one, say p, samples of the signal at each processing step are required.
- These samples are arranged into a p-element vector of input signals, $\mathbf{x}(n)$, supplied to a neural network:

$$\mathbf{x}(n) = [x(n)\ x(n-1) \ldots x(n-p+1)]^T \tag{3.51}$$

This vector of the **current and past samples** of the 1-D signal is created by a **tapped delay line** as illustrated below:

Conversion of a 1-D analog signal into digital samples supplied to a neural network using an Analog/Digital converter and a tapped delay line.

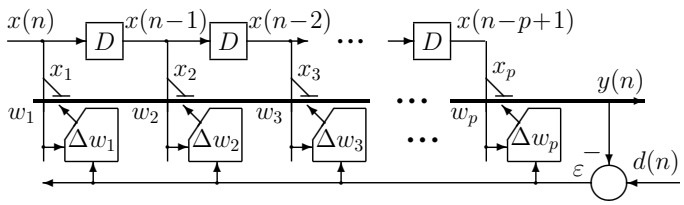
Note the difference between: $x(t), x(n)$, and $\mathbf{x}(n)$



Adaline as an adaptive FIR filter:

- If we connect outputs from the delay elements to the synapses of an Adaline as in the figure below, it will result in a signal processing structure known as an FIR (**Finite-Impulse-Response**) pth-order digital linear filter.
- In time-series processing such a system is called an MA (**Moving-Average**) model.

a. Detailed dendritic/synaptic structure:



b. Block-diagram:

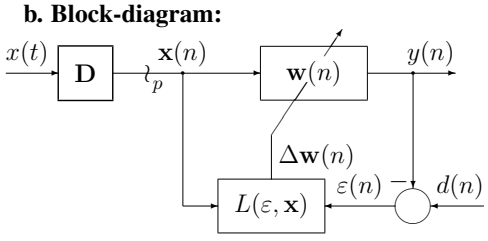


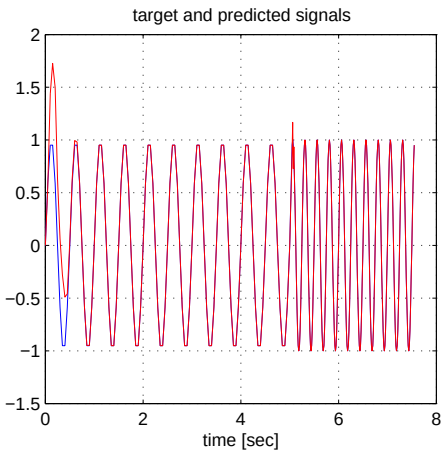
Figure 3–1: Adaline as an adaptive FIR filter

- If, in addition, the desired output signal, $d(n)$, is given, the filter's weights can be adapted using any of the previously discussed learning algorithms, so that the filter's output, $y(n)$ will track the desired output, $d(n)$.

3.5.1 Adaptive Prediction with Adaline — Example (adlpr.m)

- In this example an Adaline configured as in Figure 3–1 is used to predict a 1-D signal (time series).
- To predict the next value of the input signal, p samples of it are sent to the Adaline.
- The input signal is also used as the target/desired signal.
- The LMS learning law as in eqn (3.27) is used to adapt the weight vector at each step.
- We start with specification of a sinusoidal signal of frequency 2kHz sampled every 50μsec.
- After 5sec the frequency of the signal quadruples with the sampling time being also reduced to 12.5μsec.

```
f1 = 2 ; % kHz
ts = 1/(40*f1); % 12.5usec - sampling time,
                %  fs = 80kHz
N = 100 ;       % number of time steps
t1 = (0:N)*4*ts ;             % 1st time segment
t2 = (0:2*N)*ts + 4*(N+1)*ts; % 2nd time segment
t = [t1 t2] ;     % 0 to 7.5 sec total time
N = size(t, 2) ;  % N = 302
xt = [sin(2*pi*f1*t1) sin(2*pi*2*f1*t2)];
plot(t, xt), grid on,
title('Signal to be predicted')
```



- The 1-D signal (time series) must be converted into a collection of input vectors, $\mathbf{x}(n)$, as in eqn (3.51) and stored in a $p \times N$ matrix X:

$$X = \begin{bmatrix} x(0) & x(1) & x(2) & x(3) & x(4) & \dots \\ 0 & x(0) & x(1) & x(2) & x(3) & \dots \\ 0 & 0 & x(0) & x(1) & x(2) & \dots \\ 0 & 0 & 0 & x(0) & x(1) & \dots \end{bmatrix}$$

- It can be observed that the matrix X is a **convolution matrix** (a Sylvester's resultant matrix) associated with a time series, $x(t)$. The relevant MATLAB function is called convmtx.
- Try `convmtx(1:8, 5)` to clarify the operation.

```
p = 4 ;         % Number of synapses
X = convmtx(xt, p) ; X = X(:, 1:N) ;
d = xt ;  % The target signal is equal to the input signal
y  = zeros(size(d)) ;    % memory allocation for y
eps = zeros(size(d)) ;   % memory allocation for eps
eta = 0.4   ;            % learning rate/gain
w = rand(1, p) ; % Initialisation of the weight vector
for n = 1:N                % LMS learning loop
  y(n) = w*X(:,n) ;        % predicted output signal
  eps(n) = d(n) - y(n) ;   % error signal
  w = w + eta*eps(n)*X(:,n)'; % weight update
end
```

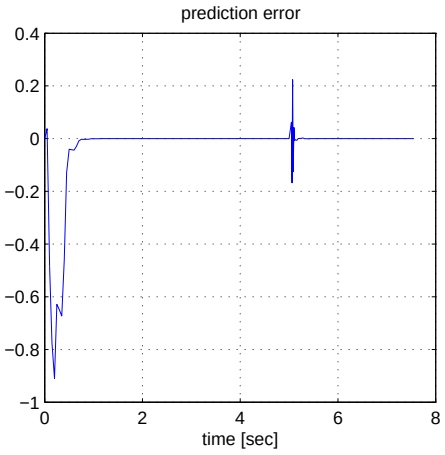


Figure 3–2: The Prediction error

- The resulting weight vectors can be as follows:

  Initial weight vector: `w = 0.9218 0.7382 0.1763 0.4057`

  Weight vectors at 5msec: `w = 0.7682 0.3797 -0.2394 0.0047`

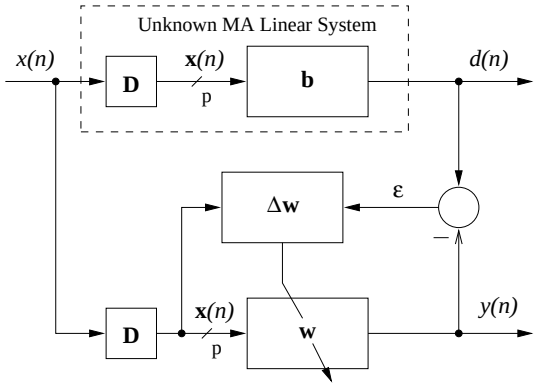
  Final weight vector: `w = 0.7992 0.3808 -0.1987 -0.0011`

- Results vary from run to run not only because of a random initialisation, but because such a simple signal requires less than four parameter to be correctly predicted.

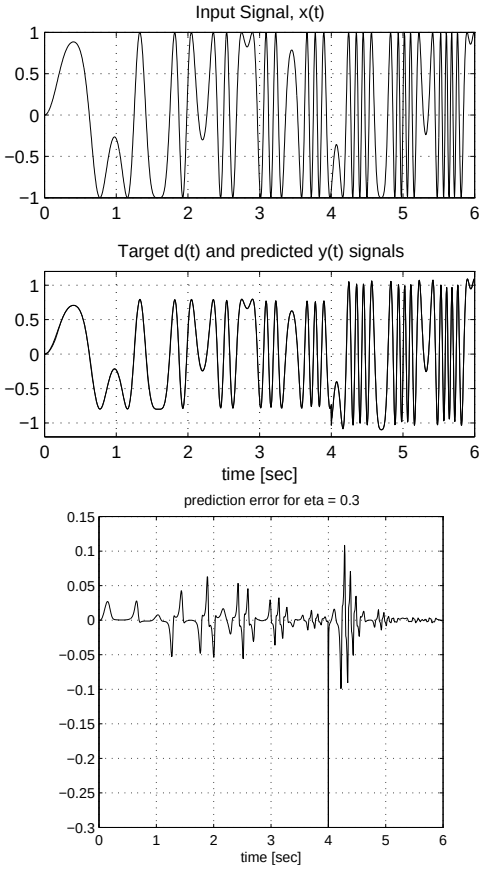
3.5.2 Adaptive System Identification

- Consider a discrete-time signal (time series), $x(n)$, which is processed by an unknown Moving-Average system.
- Such a system is an Adaline with parameters (weights) being a p-element vector b.
- It is assumed that the parameter vector is unknown.
- It is now possible to use another Adaline to observe inputs and outputs from the system and to adapt its weights using previously discussed learning algorithms so that the weight vector, $\mathbf{w}$, approximates the unknown parameter vector, $\mathbf{b}$: $\mathbf{w}(n) \longrightarrow \mathbf{b}$

Block diagram of the adaptive system identification:



Example – adsid.m




```
% Input signal x(t)
f = 0.8 ;       %  Hz
ts = 0.005; % 5 msec - sampling time
N1 = 800 ; N2 = 400 ; N = N1 + N2 ;
t1 = (0:N1-1)*ts ;  % 0 to 4 sec
t2 = (N1:N-1)*ts ;  % 4 to 6 sec
t = [t1 t2] ;       % 0 to 6 sec
xt = sin(3*t.*sin(2*pi*f*t)) ;

p = 3 ; % Dimensionality of the system
b1 = [ 1  -0.6 0.4]; % unknown system parameters during t1
b2 = [0.9 -0.5 0.7]; % unknown system parameters during t2
% formation of the input matrix X of size  p by N
 X = convmtx(xt, p) ; X = X(:, 1:N) ;
% The target signal
 d = [b1*X(:,1:N1) b2*X(:,N1+1:N)] ;
 y  = zeros(size(d)) ;   % memory allocation for y
eps = zeros(size(d)) ;   % memory allocation for eps
eta = 0.2  ; % learning rate/gain
w = 2*(rand(1,p)-0.5); % Initialisation of the weight vector
for n = 1:N  %  learning loop
   y(n) = w*X(:,n) ;      % predicted output signal
   eps(n) = d(n) - y(n) ; % error signal
   w = w + eta*eps(n)*X(:,n)' ;
   if n == N1-1, w1 = w ; end
end
```

Estimated system parameters:

```
b1 = 1.0000 -0.6000  0.4000    b2 = 0.9000 -0.5000  0.7000
w1 = 0.9463 -0.5375  0.3908    w2 = 0.8690 -0.4369  0.6677
```


3.5.3 Adaptive Noise Cancelation

Consider a system as in Figure 3–3:

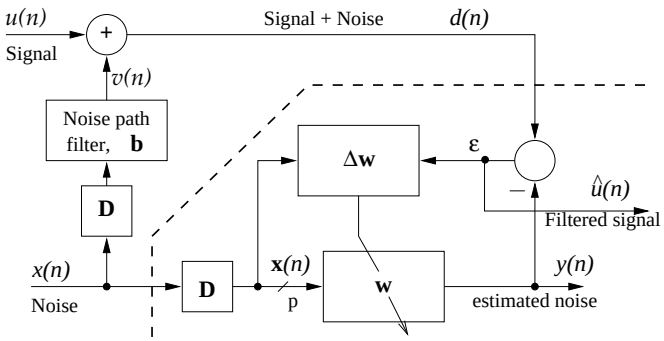


Figure 3–3: Adaptive Noise Cancelation

- A useful signal, $u(n)$, for example, voice of a pilot of an aircraft, is disturbed by a noise, $x(n)$, originated for example from an engine.
- The noise is coloured by an unknown FIR filter specified by an unknown vector $\mathbf{b}$ before it mixes with the signal. As a result, the observed signal is equal to:

$$d(n) = u(n) + v(n)$$

and the problem is to filter out the noise in order to obtain an estimate $\hat{u}(n)$ of the original signal $u(n)$.

Example – adlnc.m

With reference to Figure 3–3 we specified first the useful input signal, $u(n)$ and the noise, $x(t)$.

- The input signal is a sinusoidal signal modulated in frequency and amplitude:

$$u(t) = (1 + 0.2\sin(\omega_a)t) \cdot \sin(\omega \cdot (1 + 0.2\cos(\omega_m t) \cdot t)$$

where $\omega = 2\pi f$ is the fundamental signal frequency,
$\omega_m = 2\pi f_m$ is the frequency of the frequency modulation,
$\omega_a = 2\pi f_a$ is the frequency of the amplitude modulation, and
$t = nt_s,\ t_s$ being the sampling time.

```
f  = 4e3 ;          % 4kHz signal frequency
fm = 300 ;          % 300Hz frequency modulation
fa = 200 ;          % 200Hz amplitude modulation
ts = 2e-5 ;         % 0.2 msec sampling time
N  =  400 ;         % number of sampling points
t = (0:N-1)*ts ; %  discrete time from 0 to 8 msec
ut=(1+.2*sin(2*pi*fa*t)).*sin(2*pi*f*(1+.2*cos(2*pi*fm*t)).*t);
```

- The noise $x(n)$ is a triangular signal of frequency $f_n = 1$kHz.
- This noise is coloured by a linear FIR filter (an Adaline with fixed weights specified by a vector b).
- The resulting signal, $v(n)$, is added to the input noise signal, $x(t)$ to form the corrupted signal, $d(n)$ — see Figure 3–4


```
fn = 1e3 ;
xt = sawtooth(2*pi*1e3*t, 0.7); % the noise
b = [1  -0.6 -0.3]  ;  % noise path filter
vt = filter(b, 1, xt); % coloured noise
dt = ut + vt ;      % corrupted input signal
```

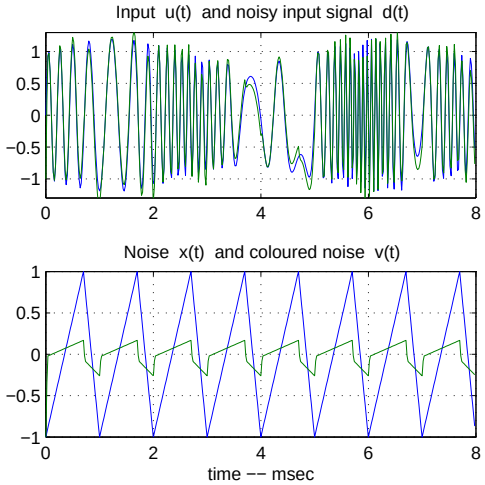


Figure 3–4: Input signal, $u(t)$, the corrupted-by-noise input signal, $d(t)$, noise, $x(t)$, and the coloured noise, $v(t)$

- It is assumed that the parameters of the noise colouring filter, b are unknown.
- The idea of noise cancellation is to estimate parameters of this noise colouring filter, thus to estimate the noise which corrupts the the input signal.
- This noise estimate, $y(n) \approx v(n)$, is available at the output of the Adaline.

- The difference between the corrupted signal, $d(n)$, and the noise estimate, $y(n)$ is the estimate of the original clean input signal:

$$\hat{u}(n) = \varepsilon(n) = d(n) - y(n) \approx u(n)$$

```
p = 4 ; % dimensionality of the input space
% formation of the input matrix X of size  p by N
X = convmtx(xt, p) ; X = X(:, 1:N) ;
y   = zeros(1,N) ;      % memory allocation for y
eps = zeros(1,N) ;      % memory allocation for uh = eps
eta = 0.03  ;           % learning rate/gain
w = 2*(rand(1,p)-0.5); % weight vector initialisation
```

- Note that the number of synapses in the adaptive Adaline, $p = 4$, is different that the order of the noise colouring filter, which is assumed to be unknown.
- Selection of the learning rate, η is very critical to good convergence of the LMS algorithm.
- In order to improve results, the learning loop which goes through all signal samples is repeated four time with diminishing values of the learning gain. Such a repetition is, of course, not possible for the real-time learning.

```
for c = 1:4
  for n = 1:N  %  learning loop
    y(n) = w*X(:,n) ;       % predicted output signal
    eps(n) = dt(n) - y(n) ; % error signal
    w = w + eta*eps(n)*X(:,n)' ;
```



```
  end
  eta = 0.8*eta ;
end
```

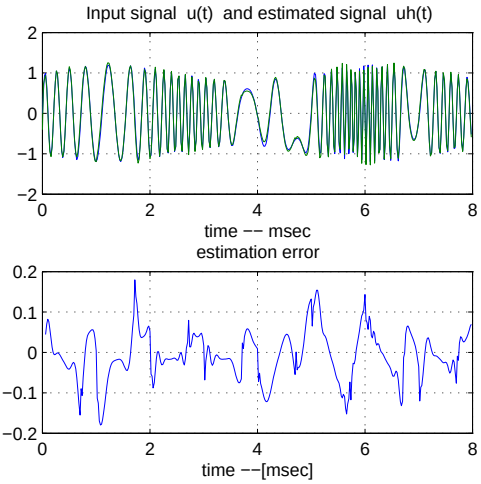


Figure 3–5: Input signal, $u(t)$, the estimated input signal, $\hat{u}(t)$, and the estimation error, $u(t) - \hat{u}(t)$,

- It can be noticed that the estimation error at each step is small, less that 4% of the signal amplitude. The resulting weight vector

```
w = 0.7933 -0.0646 -0.7231 0.0714
```

is similar to the parameters of the noise colouring filter, $\mathbf{b}$.

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:

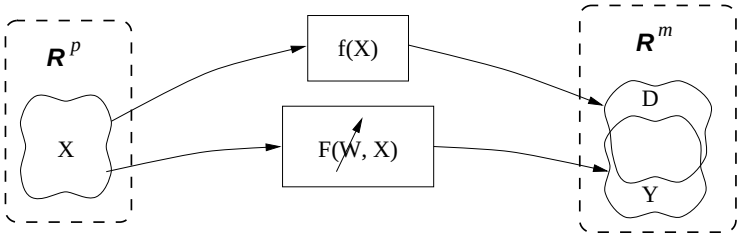


Figure 4–1: Mapping from a p-dimensional domain into an m-dimensional output space

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

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

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

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

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).

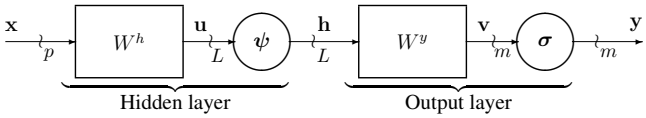


Figure 4–2: A block-diagram of a single-hidden-layer feedforward neural network

- 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:

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

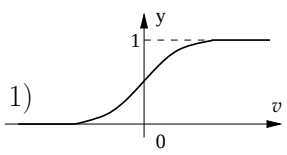
or simply as

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

- 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:

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

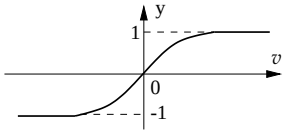


The derivative of the unipolar sigmoidal function:

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

Sigmoidal bipolar:

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



The derivative of the bipolar sigmoidal function:

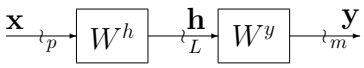
$$y' = \frac{d\sigma}{dv} = \frac{4\beta e^{2\beta v}}{(e^{2\beta v}+1)^2} = \beta(1-y^2)$$

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:

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

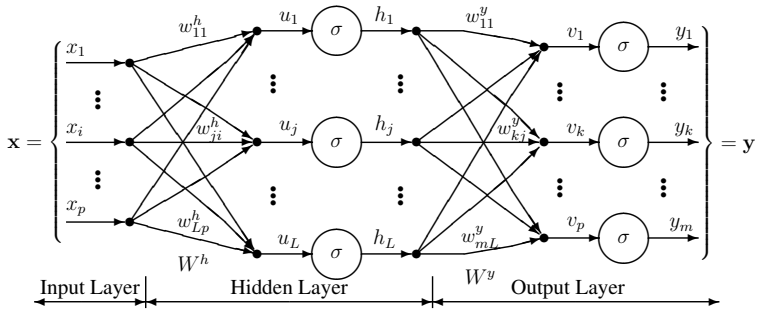
where

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

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

$$h_L = 1$$

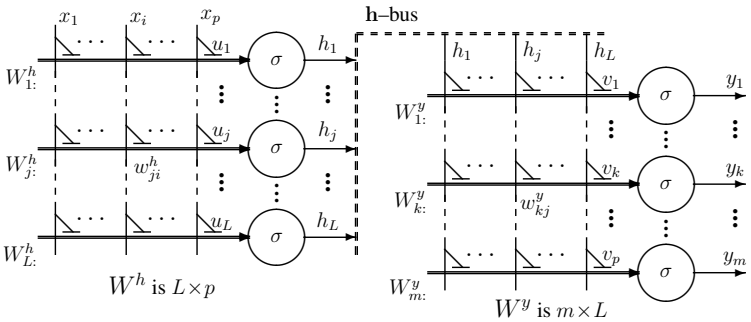
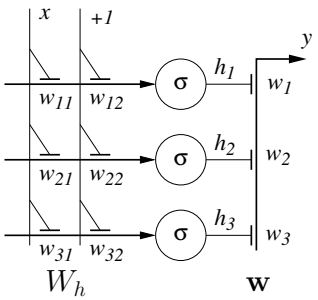
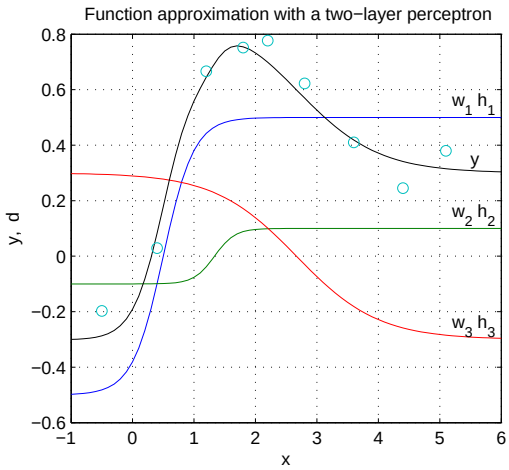


Figure 4–3: Various representations of a Two-Layer Perceptron

4.3 Example of function approximation with a two-layer perceptron

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



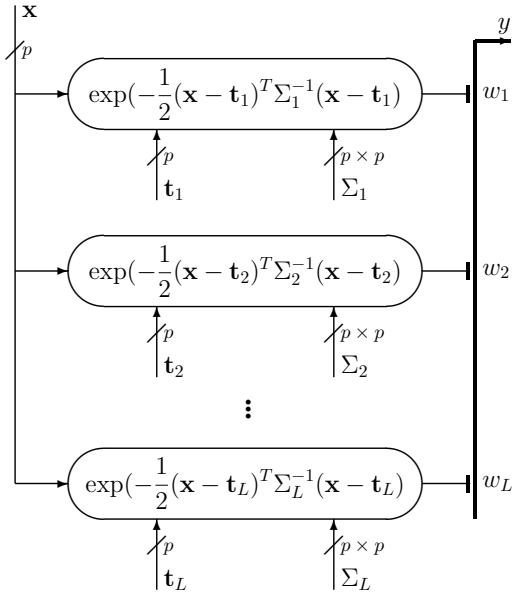


The neural net generates the following function:

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

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:

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

where

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

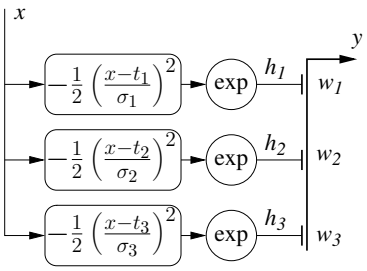
Σ_l, $p \times p$ covariance matrices, specified a shape of the Gaussian functions

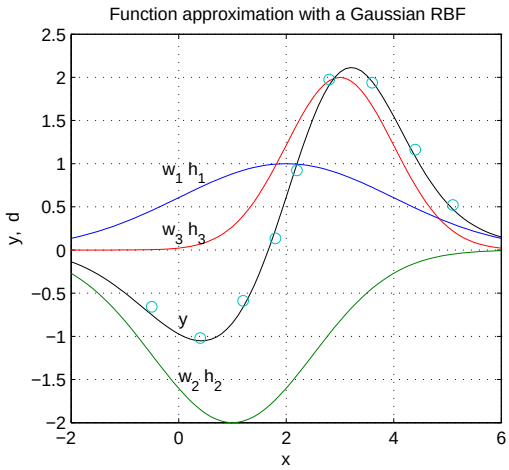
All parameters: $(\mathbf{t}_1, \ldots, \mathbf{t}_L, \Sigma_1, \ldots, \Sigma_L, \mathbf{w})$ are adjusted during learning procedure.

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:

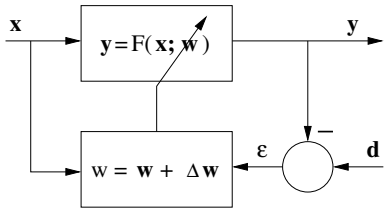




The neural net generates the following function:

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

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

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

- 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:

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

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

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

  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:

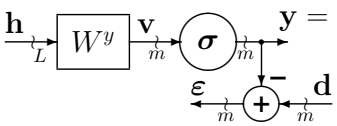
$$\nabla E(\mathbf{w}) = \left[\frac{\partial E}{\partial \mathbf{w}^h} \quad \frac{\partial E}{\partial \mathbf{w}^y} \right] = \left[\frac{\partial E}{\partial w^h_{11}} \cdots \frac{\partial E}{\partial w^h_{ji}} \cdots \frac{\partial E}{\partial w^h_{Lp}} \; \frac{\partial E}{\partial w^y_{11}} \cdots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

- 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:



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

$$\frac{\partial \mathbf{y}}{\partial \mathbf{v}} = \text{diag}(\boldsymbol{\sigma}') = \text{diag}(\left[\begin{array}{ccc} \frac{\partial y_1}{\partial v_1} & \cdots & \frac{\partial y_m}{\partial v_m} \end{array} \right]) \tag{4.12}$$

  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**:

$$\boldsymbol{\delta}^T = \boldsymbol{\varepsilon}^T \cdot \text{diag}(\boldsymbol{\sigma}') = \left[\begin{array}{ccc} \varepsilon_1 \cdot \sigma'_1 & \cdots & \varepsilon_m \cdot \sigma'_m \end{array} \right] \tag{4.13}$$

  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

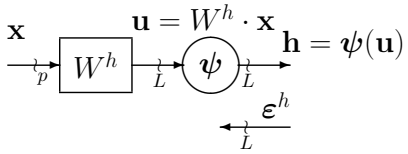
- As for the output layer we start with eqn (4.14) which for the hidden weight can be re-written in a form analogous to eqn (4.15) and then, using a chain rule of differentiation, we can write:

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

- Since $\mathbf{v} = W^y \cdot \mathbf{h}$ we have $\dfrac{\partial \mathbf{v}}{\partial \mathbf{h}} = W^y$ 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**:

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

- 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:

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

  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: $$S^h = \left[\boldsymbol{\delta}^h(1)\ \ldots\ \boldsymbol{\delta}^h(N)\right] \tag{4.32}$$

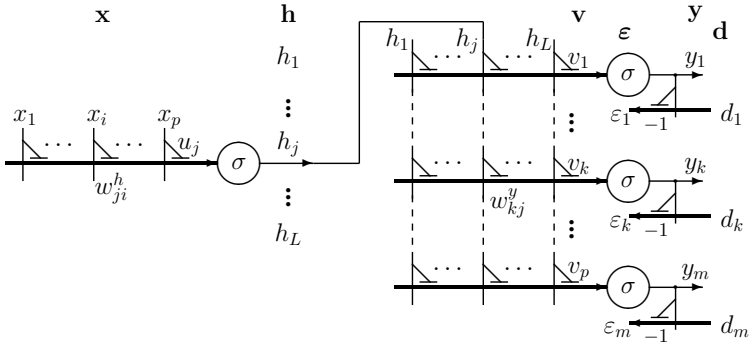
and X is the $p\times N$ matrix of the input signals.

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

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

4.7.3 Alternative derivation

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



- Good to remember

$$E = \frac{1}{2}\sum_{k=1}^{m}\varepsilon_k^2\ ,\ \ \varepsilon_k = d_k - y_k$$

$$y_k = \sigma(\mathbf{w}_k^y\cdot\mathbf{h}) = \sigma(\sum_{j=1}^{L} w_{kj}^y\cdot h_j)$$

$$h_j = \sigma(\mathbf{w}_j^h\cdot\mathbf{x}) = \sigma(\sum_{i=1}^{p} w_{ji}^y\cdot x_i)$$

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

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

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, $W_{k:}^y$ of the kth output neuron can be calculated as:

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

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:

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

where

$$\boldsymbol{\delta} = \begin{bmatrix} \delta_1 \\ \vdots \\ \delta_m \end{bmatrix} = \begin{bmatrix} \varepsilon_1 \cdot \sigma_1' \\ \vdots \\ \varepsilon_m \cdot \sigma_m' \end{bmatrix} = \boldsymbol{\varepsilon} \odot \boldsymbol{\sigma'}\ , \tag{4.35}$$

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:

$$\begin{aligned}
\frac{\partial E(n)}{\partial w_{ji}^h} &= -\sum_{k=1}^{m} \varepsilon_k \frac{\partial y_k}{\partial w_{ji}^h} && \left\| \; y_k = \sigma(v_k) \right. \\
&= -\sum_{k=1}^{m} \varepsilon_k \frac{\partial y_k}{\partial v_k}\frac{\partial v_k}{\partial w_{ji}^h} && \left\| \; v_k = \ldots + w_{kj}^y h_j + \ldots \right. \\
&= -(\sum_{k=1}^{m} \delta_k\, w_{kj}^y)\frac{\partial h_j}{\partial w_{ji}^h} && \left\| \; \delta_k = \varepsilon_k \cdot \sigma_k' \right. \\
&= -\, W_{:j}^{yT}\, \boldsymbol{\delta}\, \frac{\partial h_j}{\partial w_{ji}^h} && \left\| \; W_{:j}^{yT}\boldsymbol{\delta} = \boldsymbol{\delta^T} W_{:j}^y = \sum_{k=1}^{m} \delta_k\, w_{kj}^y = \varepsilon_j^h \right.
\end{aligned}$$

- 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.

$$\begin{aligned} \frac{\partial E(n)}{\partial w^h_{ji}} &= -\varepsilon^h_j \frac{\partial h_j}{\partial w^h_{ji}} && \Big\| \ h_j = \psi(u_j) \ , \ u_j = W^h_{j:} \cdot \mathbf{x} \\ &= -\varepsilon^h_j \cdot \psi'_j \cdot x_i && \Big\| \ \psi'_j = \frac{\partial \psi_j}{\partial u_j} \\ &= -\delta^h_j \cdot x_i && \Big\| \ \delta^h_j = \varepsilon^h_j \cdot \psi'_j \end{aligned}$$

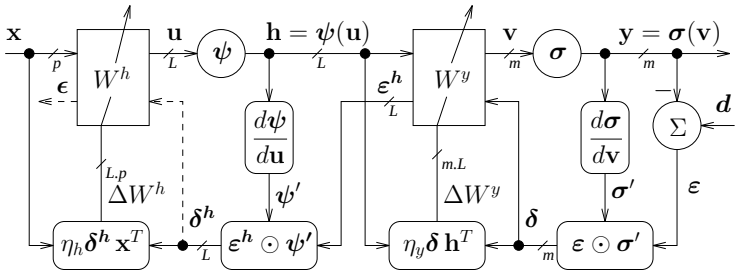
  where the back-propagated error has been used to generate the delta-error for the hidden layer, δ^h_j.

- 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):

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

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,

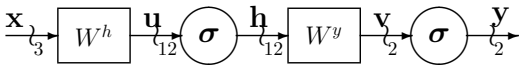
$$\mathbf{y} = \mathbf{f}(\mathbf{x}) \ , \text{ or } \ y_1 = f_1(x_1, x_2) \ , \ \ y_2 = f_2(x_1, x_2)$$

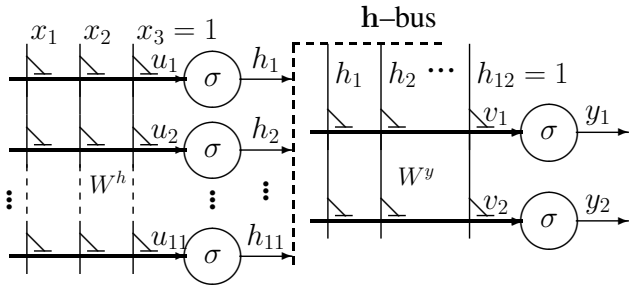
using a two-layer perceptron,

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

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 $p \times N$:

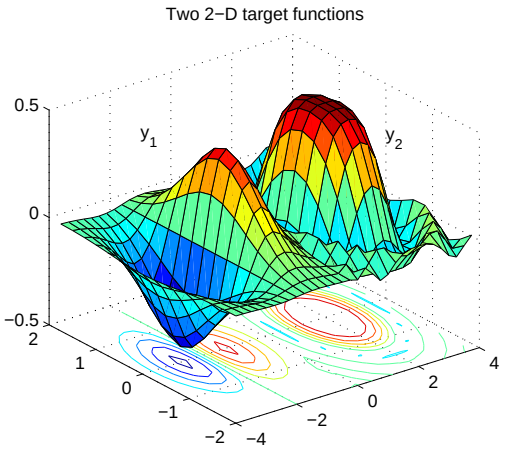
```
X = [X1(:)'; X2(:)';ones(1,N)];
```

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
```

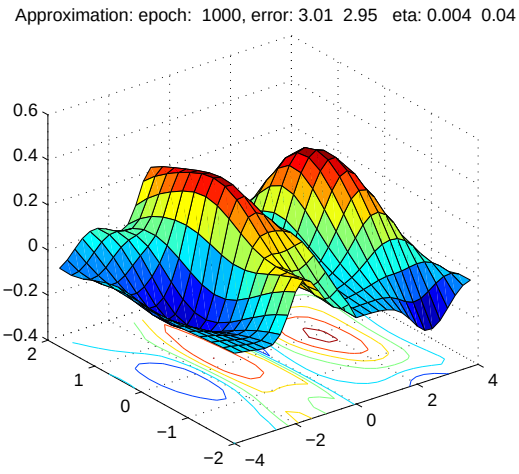
The backward pass:

```
   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
```

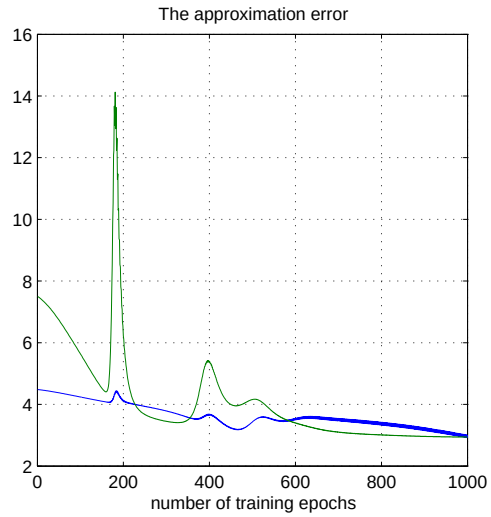
Two 2-D approximated functions are plotted after each epoch.

```
  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.

5 Feedforward Multilayer Neural Networks — part II

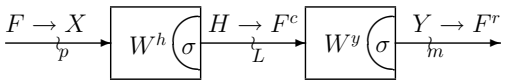
In this section we first consider selected applications of the multi-layer perceptrons.

5.1 Image Coding using Multi-layer Perceptrons

- In this example we study an application of a two-layer feed-forward neural network (perceptron) in image coding.

The general concept is as follows:

- A two-layer perceptron is trained using a representative set of images, F.

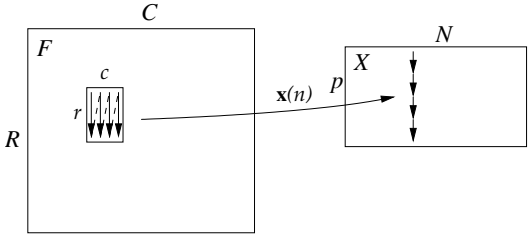


- Once the training is completed, appropriate hidden, W^h and output, W^y weights are available.
- An image, F, can now be encoded into an image F^c, represented by the hidden signals.
- If $L < p$ an image compression occurs.
- The encoded (compressed) image, F^c, represented by the hidden signals can now be reconstructed using the output layer of the perceptron as F^r.

Training procedure:

- Training is conducted for a representative class of images using the back-propagation algorithm.
- Assume that an image, F, used in training is of size $R \times C$ and consists of $r \times c$ blocks.
- Convert a block matrix F into a matrix X of size $p \times N$ containing training vectors, $\mathbf{x}(n)$, formed from image blocks. Note that

$$p = r \cdot c\,, \quad \text{and} \quad p \cdot N = R \cdot C$$



Use the following MATLAB function to perform this conversion: `X = blkM2vc(F, [r c]);`

```
function vc = blkM2vc(M, blkS)
[rr cc] = size(M) ;
r = blkS(1) ; c = blkS(2) ;
if  (rem(rr, r) ~= 0) | (rem(cc, c) ~= 0)
 error('blocks do not fit into matrix')
end
nr = rr/r ; nc = cc/c ; rc = r*c ;
vc = zeros(rc, nr*nc);
for ii = 0:nr-1
 vc(:,(1:nc)+ii*nc)=reshape(M((1:r)+ii*r,:),rc,nc)
end
```

- As a **target data** use the **input data**, that is:

$$D = X$$

- Train the network until the mean squared error, J, is sufficiently small. The matrices W^h and W^y will be subsequently used in the image encoding and decoding steps.

Image encoding:

- An image, F, is divided into $r \times c$ blocks of pixels. Each block is then scanned to form an input vector $\mathbf{x}(n)$ of size $p = r \times c$.

  The image to be encoded is now represented by an $N \times p$ matrix X, each column storing a block of pixels.

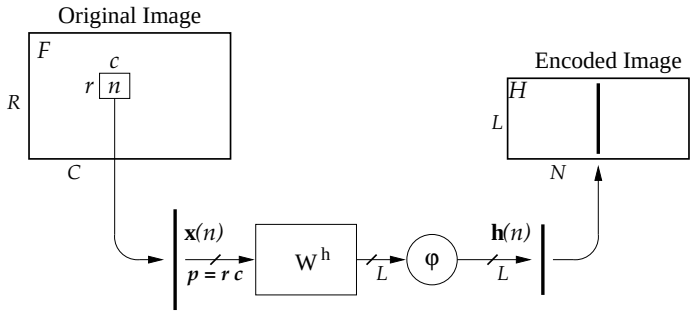


Figure 5–1: A hidden layer in image encoding

- Assume that the hidden layer of the neural network consists of L neurons each with p synapses, and that it is characterised by the appropriately selected weight matrix W^h.
- The encoding procedure can be described as follows:

$$F \longrightarrow X\ , \quad H = \sigma(W^h \cdot X) \longrightarrow F^c$$

  where F^c represented by H is an encoded image.

Image reconstruction:

- Assume that the output layer consists of $m = p = r \times c$ neurons, each with L synapses.

  Let W^y be an appropriately selected output weight matrix.
- The decoding procedure can be described as follows:

$$Y = \sigma(W^y \cdot H)\ , \quad Y \longrightarrow F^r$$

- Re-assemble the output signals into $p = r \times c$ image blocks to obtain a re-constructed image, F^r.

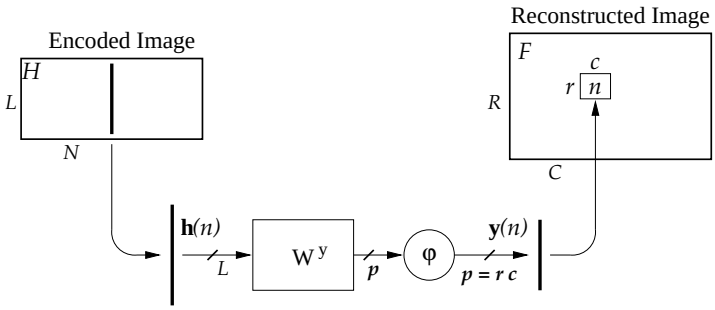


Figure 5–2: the outpur layer in image reconstruction

- The quality of image coding is typically assessed by the Signal-to-Noise Ratio (SNR) defined as

$$\mathrm{SNR} = 10 \log \frac{\Sigma_{i,j}(F_{i,j})^2}{\Sigma_{i,j}(F^r_{i,j} - F_{i,j})^2}$$

- The conversion of the vectors of the reconstructed image stored in the $p \times N$ matrix Y into blocks of the reconstructed image, F^r, can be performed using the following MATLAB function:

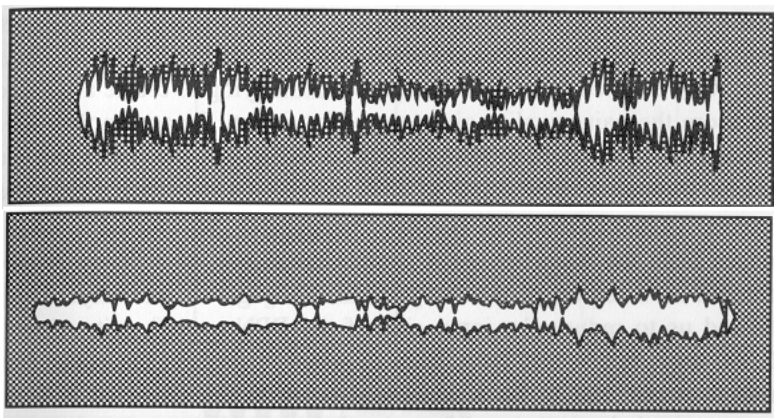
Fr = vc2blkM(Y, r, R);

```
function M = vc2blkM(vc, r, rM)
%vc2blkM  Reshaping a matrix vc of rc by 1 vectors into a
%         block-matrix   M  of  rM  by  cM  size
% Each rc-element column of vc is converted into a  r by c
% block of a matrix  M  and placed as a block-row element
[rc nb] = size(vc) ; pxls = rc*nb ;
if ( (rem(pxls, rM) ~= 0) | (rem(rM, r) ~= 0) )
  error('incorrect number of rows of the matrix')
end
cM = pxls/rM ;
if ( (rem(rc, r) ~= 0) | (rem(nb*r, rM) ~= 0) )
  error('incorrect block size')
end
c = rc/r ;
xM = zeros(r, nb*c);
xM(:) = vc ;
nrb = rM/r ;
M = zeros(rM, cM);
for ii = 0:nrb-1
  M((1:r)+ii*r, :) = xM(:, (1:cM)+ii*cM) ;
end
```


5.2 Paint-Quality Inspection

Adapted from (Freeman and Skapura, 1991)

- Visual inspection of painted surfaces, such as automobile body panels, is a very time-consuming and labor-intensive process.
- To reduce the amount of time required to perform this inspection, one of the major U.S. automobile manufacturers reflects a laser beam off the painted panel and on to a projection screen.
- Since the light source is a coherent beam, the amount of scatter observed in the reflected image of the laser provides an indication of the quality of the paint finish on the car.

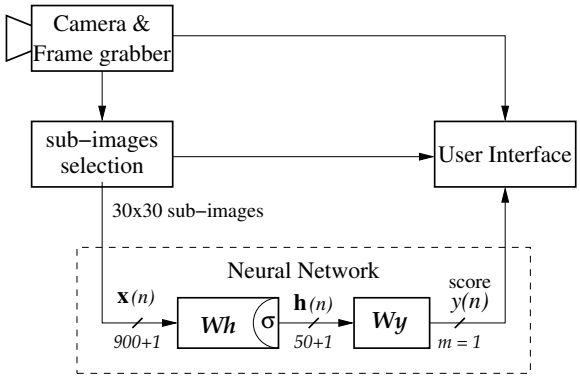


Reflection of a laser beam off painted sheet-metal surfaces:

top — a poor-quality paint finish: reflection is relatively difused.

bottom — a better-quality paint finish: reflection is very close to uniform throughout its image.

- A neural network, a two-layer perceptron in this case, is used to capture the expertise of the human inspectors scoring the paint quality from observation of the reflected laser images.
- The block-diagram of the Automatic Paint QA System:



- Images of the reflected laser beam are recorded by a camera and an associated frame grabber. Each image contains 400-by-75 8-bit pixels.
- To keep the size of the network needed to solve the problem manageable, we elected to take 10 sub-images from the snapshot, each sub-image consisting of a 30-by-30-pixel square centered on a region of the image with the brightest intensity.
- These 8-bit pixels are input to the neural network. In addition there is one biasing input, therefore, $p = 901$.
- The hidden layer consists of $L = 50$ neurons, hence the hidden-matrix, W^h is 900×50, and there are $900 \times 50 = 45000$ synapses in the hidden layer. A unipolar sigmoidal function is used.
- A single output signal from the network represents a **numerical score** in the range of 1 through 20 (a 1 represented the best possible paint finish; a 20 represented the worst).
- The output layer is **linear** and the output matrix W^y has a size 51×1 (there is a biasing input to the output layer).

- Once the network was constructed (and trained), 10 sub-images were taken from the snapshot using two different sampling techniques.
- In the first test, the samples were selected randomly from the image (in the sense that their position on the beam image was random).
- In the second test, 10 sequential samples were taken, so as to ensure that the entire beam was examined.
- In both cases, the input sample was propagated through the trained MLP, and the score produced as output by the network was averaged across the 10 trials.
- The average score, as well as the range of scores produced, were then provided to the user for comparison and interpretation.

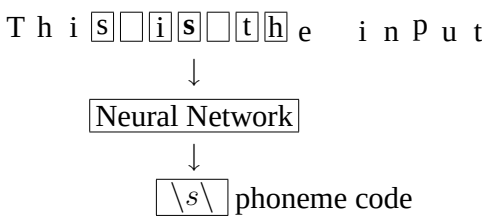
Training the Paint QA Network

- At the time of the development of this application, (1988) this network was significantly larger than any other network we had yet trained.
- The network is relatively big: 901 inputs, 51 hidden neurons, 1 output. Total number of trainable weights (synapses) is 45 101.
- The number of training patterns with which we had to work was a function of the number of control paint panels to which we had access (18), as well as of the number of sample images we needed from each panel to acquire a relatively complete training set (approximately 6600 images per panel).
- During training, the samples were presented to the network randomly to ensure that no single paint panel dominated the training.
- From these numbers, we can see that there was a great deal of computer time consumed during the training process.
- For example, one training epoch required the computer to perform approximately 13.5 million weight updates, which translates into roughly 360,000 floating-point operations (FLOPS) per pattern (2 FLOPS per connection during forward propagation, 6 FLOPS during error propagation), or 108 million FLOPS per epoch.
- However, once the network was trained, decoding is very fast and can be efficiently used in manufacturing.

5.3 NETtalk

Sejnowski and Rosenberg, 1987

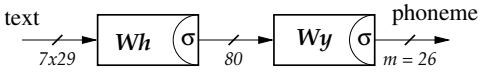
- The NETtalk project aimed at training a network to pronounce English text.
- The conceptual structure of the network is as follows:



- A character from a text and its three proceeding and three following characters are entered into a neural network which generates a phoneme code for the central character.
- The phoneme code can be sent to a speech generator giving the pronunciation of the central letter from the input window.

Network structure

- There are 29 English letters (including punctuation) and each letter is coded in a 1-of-29 code. Therefore, there are $p = 7 \times 29 = 203$ binary inputs to the network.
- Similarly, there are 26 different phonemes, hence, the network has 26 binary outputs.
- In addition, 80 hidden neurons are employed.



Training

- During training, the desired data were supplied by a commercially available DEC-talk, which is based on hand-coded linguistic rules.
- The network was trained on 1024 words, obtaining intelligible speech after 10 training epochs and 95% accuracy after 50 epochs.

5.4 Efficient initialization of the learning algorithms

- The simplest initialization of the weights is based on assigning them a "small" random values.
- This is not always a good solution because the activation potentials can be big enough to drive the activation functions into saturation.
- In saturation, the derivatives of the activation functions are zero, hence no weight update will take place.
- Efficient initialization can speed up the convergence process of the learning algorithms significantly, even by the order of magnitude.
- A popular initialization algorithm developed by Nguyen and Widrow and used in the MATLAB Neural Network Toolbox is presented below.
- Let us consider for simplicity a **single layer** of m neurons with p synapses, each including the bias. Then for jth neuron we have

$$y_j = \sigma(v_j) \ , \quad \text{where} \quad v_j = \mathbf{w}_j \cdot \mathbf{x} \ , \quad x_p = 1$$

- For an activation function $\sigma(v)$ we can specify its **active region** $\bar{v} = [v_{min} \ v_{max}]$ outside which the function is consider to be in saturation.
- For example, for the hyperbolic tangent we can assume the active region as:

$$\bar{v} = [-2 \ +2] \ , \quad \text{then} \quad \tanh(v) \in [-0.96 \ 0.96]$$

- In addition we need to specify the range of input signals,

$$\bar{\mathbf{x}}_i = [x_{i,min}\ x_{i,max}] \quad \text{for} \quad i = 1 \ldots p-1$$

Unified range:

Assume first that the range of input signals and non-saturating activation potential is $[-1\ +1]$.

- The initial weight vectors will now have evenly distributed magnitudes and random directions:

For $p = 2$ (**single input plus bias**) the weights are initialised in the following way:

- generate m random numbers $a_j \in (-1,\ +1)$ for $j = 1, \ldots, m$
- Set up weights as follow

$$W(j,1) = 0.7\frac{a_j}{|a_j|}\ , \quad W(:,2) = 0$$

For $p > 2$ the weight initialisation is as follows

- Specify the magnitude of the weight vectors as

$$\bar{W} = 0.7\, m^{\frac{1}{p-1}}$$

- generate m random unity vectors, $\mathbf{a}_j$, that is, generate an $m \times (p-1)$ array A of random numbers, $a_{ji} \in (-1,\ +1)$ and normalise it in rows:

$$\mathbf{a}_j = \frac{A(j,:)}{\|A(j,:)\|}$$

- Set up weights as follow

$$W(j, 1:p-1) = \bar{W} \cdot \mathbf{a}_j \quad \text{for} \quad j = 1, \ldots, m$$

and the bias weights

$$W(j,p) = \text{sgn}(W(j,1)) \cdot \bar{W} \cdot \beta_j \quad \text{for} \quad \beta_j = -1 : \frac{2}{m-1} : 1$$

- Finally, the weights are linearly rescaled to account for different range of activation potentials an input signals.
- Details can be found in the MATLAB script, nwini.m.

```
function w = nwini(xr, m, vr)
%
% nwini Calculates Nugyen-Widrow initial conditions.
%  adapted from NNet toolbox             8 Axril 1999
%  xr - p-1 by 2 matrix of  [xmin xmax]
%  assumes that the bias is added
%  m - Number of neurons.
%  vr - Active region of the transfer function
%  vr = [Vmin Vmax].
%  e.g. vr = [-2 -2] for tansig , [-4 4] for logsig
%  w  is  m by p

r = size(xr,1);  p = r+1 ;

% Null case
if (r == 0) | (m == 0)
  w = zeros(s,p) ;
  return
end

% Remove constant inputs that provide no useful info
R = r;
ind = find(xr(:,1) ~= xr(:,2));
r = length(ind);
xr = xr(ind,:);

% Nguyen-Widrow Method
% Assume inputs and activation potentials range in [-1 1].
%          Weights
wMag = 0.7*m^(1/r); % weight vectors magnitude
% weight vectors directions: wDir are row unity vectors
a = 2*rand(m,r)-1 ;
if r == 1
```



```
  b = ones./abs(a);
else
  b=sqrt(ones./(sum((a.*a)')))';
end
wDir=b(:,ones(1,r)).*a;
w = wMag*wDir;
%         Biases
if (m==1)
  wb = 0;
else
  wb = wMag*[2*(0:m-2)/(m-1)-1 1]'.*sign(w(:,1));
end

% Conversion of activation potentials of [-1 1] to [Nmin Nmax]
a1 = 0.5*(vr(2)-vr(1));
a2 = 0.5*(vr(2)+vr(1));
w = a1*w;
wb = a1*wb+a2;

% Conversion of inputs of xr to [-1 1]
a1 = 2./(xr(:,2)-xr(:,1));
a2 = 1-xr(:,2).*a1;

ap = a1';
wb = w*a2+wb;
w = w.*ap(ones(1,m),:);

% Replace constant inputs
ww = w; w = zeros(m,R);
w(:,ind) = ww;
%  combine with biasing weights
w = [w wb] ;
```

5.5 Why backpropagation is slow

- The basic pattern-based back-propagation learning law is a gradient-descent algorithm based on the estimation of the gradient of the instantaneous sum-squared error for each layer:

$$\Delta W(n) = -\eta \cdot \nabla_W E(n) = \eta \cdot \boldsymbol{\delta}(n) \cdot \mathbf{x}^T(n) \tag{5.1}$$

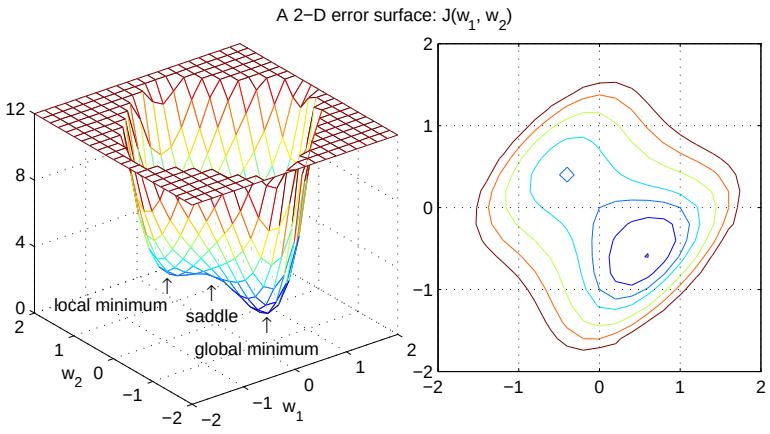
  Such an algorithm is slow for a few reasons:

- It uses an instantaneous sum-squared error $E(W, n)$ to minimise the mean squared error, $J(W)$, over the training epoch.
- The gradient of the instantaneous sum-squared error is not a good estimate of the gradient of the mean squared error.
- Therefore, satisfactory minimisation of this error typically requires many repetitions of the training epochs.
- It is a first-order minimisation algorithm which is based on the first-order derivatives (a gradient). Faster algorithms utilise also the second derivatives (the Hessian matrix)
- The error back propagation, which is conceptually very interesting, serialises computations on the layer by layer basis.

A general problem is that the mean squared error, $J(W)$, is a relatively complex surface in the weight space, possibly with many local minima, flat sections, narrow irregular valleys, and saddle points, therefore, it is difficult to navigate directly to its minimum.

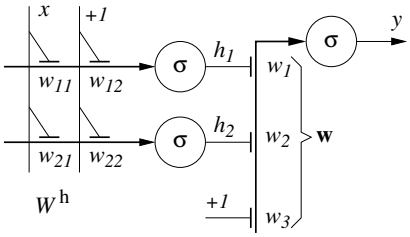
5.6 Examples of error surfaces

- Consider the following function of two weights, $J(w_1, w_2)$ representing a possible mean-squared error together with its contour map.
- Note the local and global minima, and a saddle point.
- Consider the importance of the proper initialisation to be able to reach the global minimum.



- Complexity of the error surface is the main reason that behaviour of a simple steepest descend minimisation algorithm can be very complex often with oscillations around a local minimum.

- In order to gain more insight into the shape of the error surfaces let us consider a simple two-layer network approximating a single-variable function similar to that considered in sec. 4.3

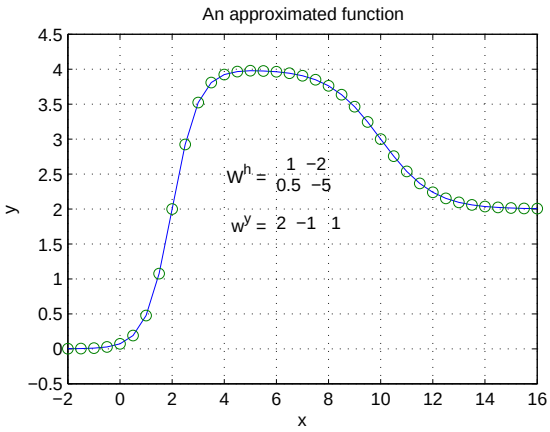


$$y(n) = \boldsymbol{\sigma}(\mathbf{w}^y \cdot \begin{bmatrix} \boldsymbol{\sigma}(W^h \cdot \begin{bmatrix} x(n) \\ 1 \end{bmatrix}) \\ 1 \end{bmatrix})$$

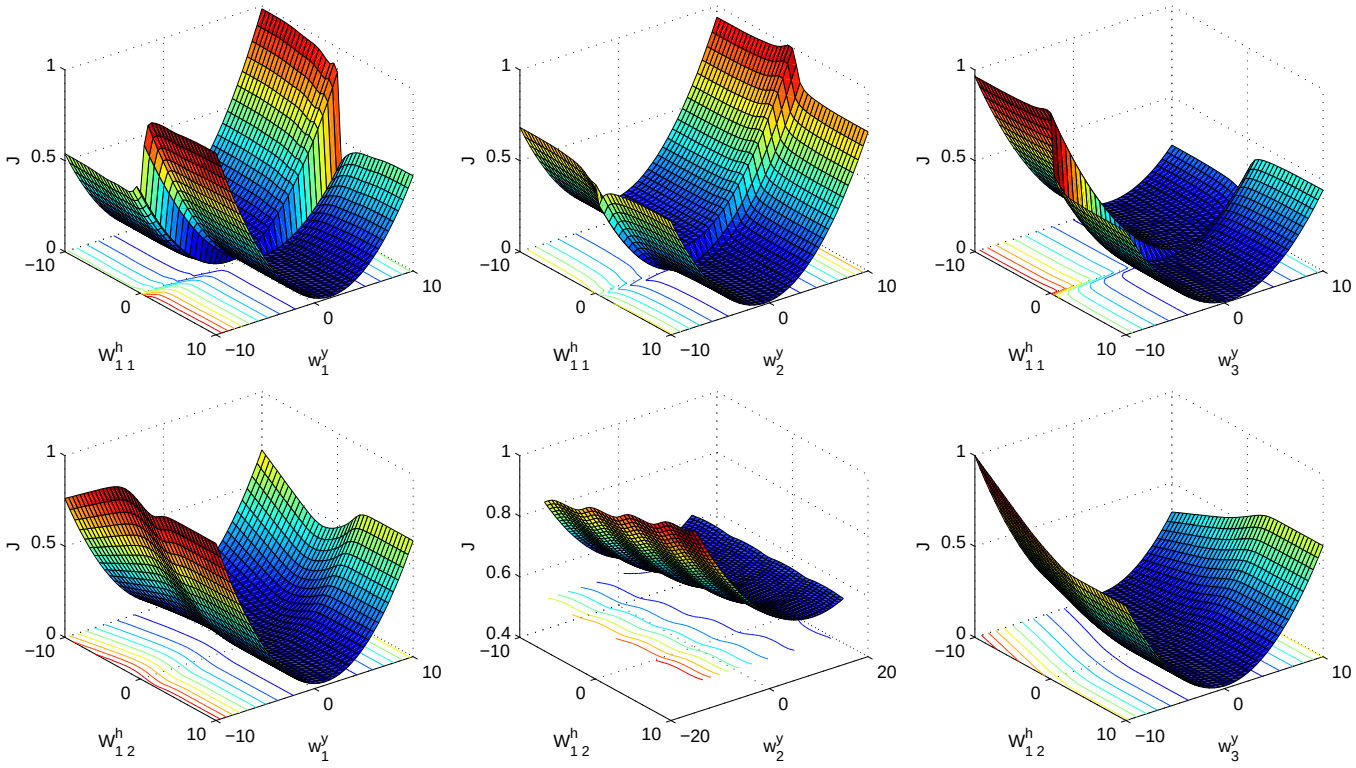
- For a specific set of parameters:

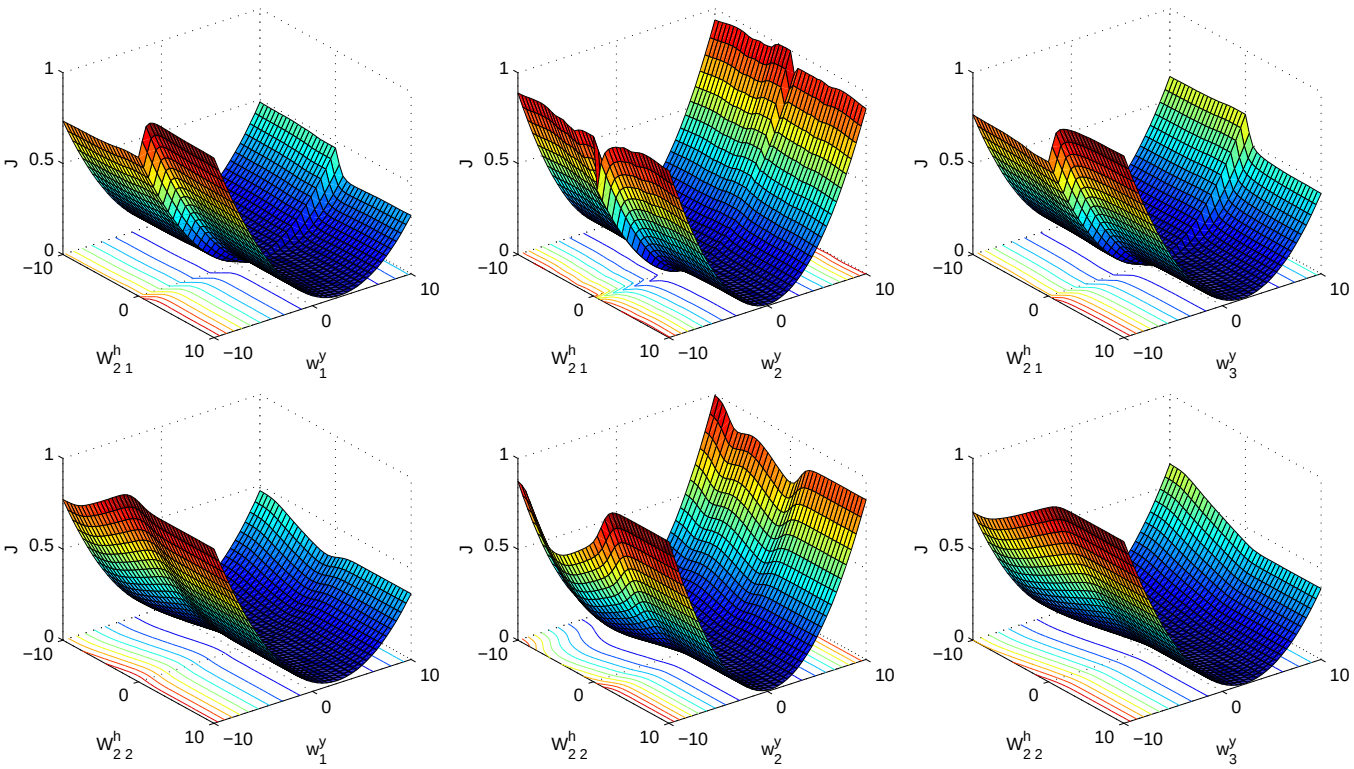
$$W^h = \begin{bmatrix} 1 & -2 \\ 0.5 & -5 \end{bmatrix} \quad \text{and} \quad \mathbf{w}^y = \begin{bmatrix} 2 & -1 & 1 \end{bmatrix}$$

the network approximate the following function:



- In order to obtain the error surface, we will vary $\mathbf{w} = [W^h \;\; \mathbf{w}^y]$ and calculate $J(\mathbf{w})$ for the selected inputs X.
- The error function $J(\mathbf{w})$ is an 8-dimensional object, hence difficult to visualise.
  Therefore we will vary only a pair of selected weights at a time.



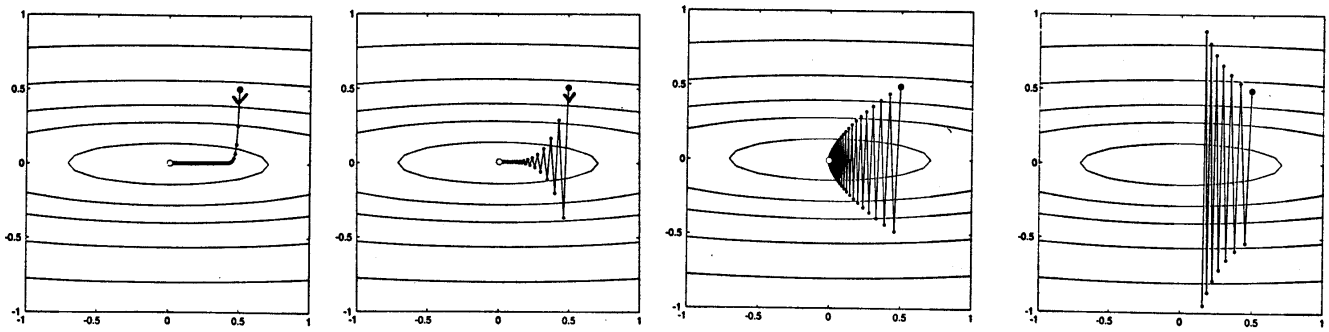


Note that the surfaces are very far away from and ideal second order paraboloidal shapes.
Finding the minimum is very sensitive to the initial position, learning gain and the direction of movement.

5.7 Illustration of sensitivity to a learning rate

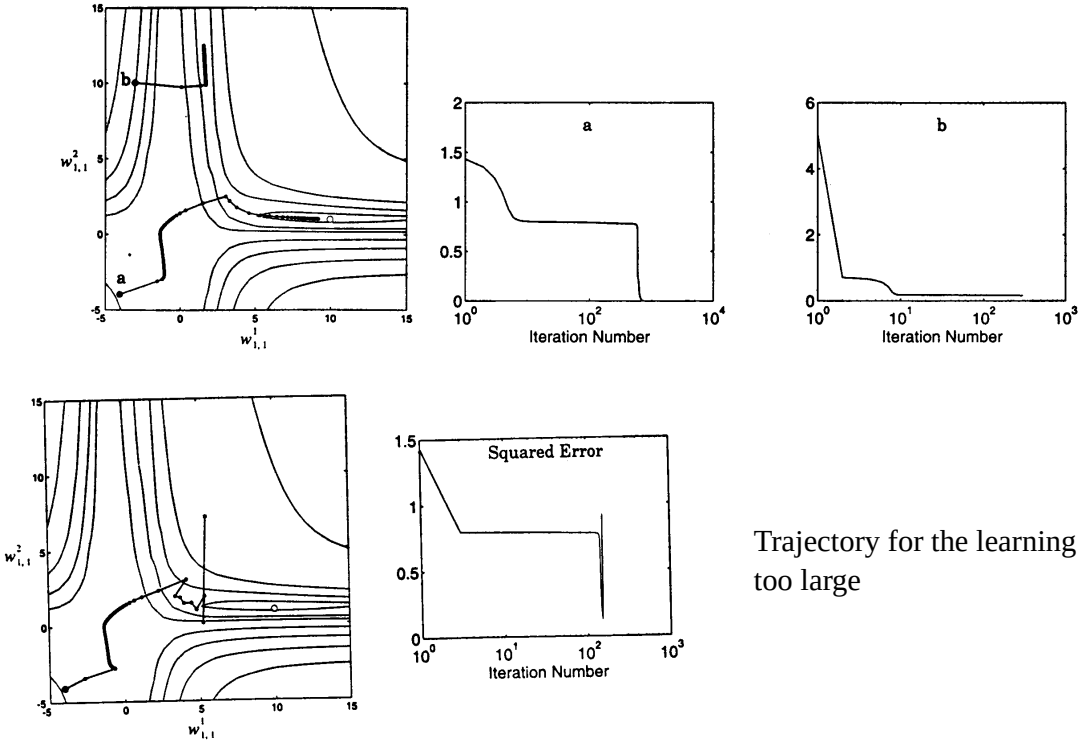
Figures 9.1, 9.2, 9.3, 12.6, 12.7 12.8 from: M.T. Hagan, H. Demuth, M. Beale, *Neural Network Design*, PWS Publishing, 1996

- For an Adaline, when the error surface is paraboloidal, the maximum stable learning gain can be evaluated from eqn (3.24) and is inversely proportional to the largest eigenvalue of the input correlation matrix, R.
- As an illustration we consider the case when $\eta_{mx} = 0.04$ and observe the learning trajectory on the error surface for a linear case:



$\eta = 0.01$ $\eta = 0.035$ $\eta = 0.039$ $\eta = 0.041$

- Examples of learning trajectories for the steepest descent backpropagation algorithm in the batch mode. Plots on the right shows the error versus the iteration number.



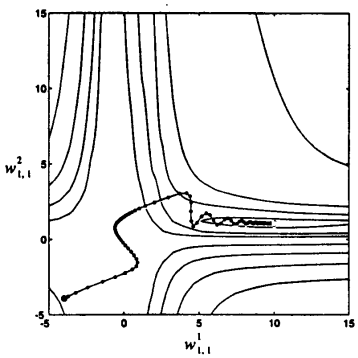
Trajectory for the learning gain too large

5.8 Heuristic Improvements to the Back-Propagation Algorithm

- The first group of consider to the basic back-propagation algorithms based on heuristic methods.
- These methods do not directly address the inherent weaknesses of the back-propagation algorithm, but aim at improvement of the behaviour of the algorithm by making modifications to its form or parameters.

5.8.1 The momentum term

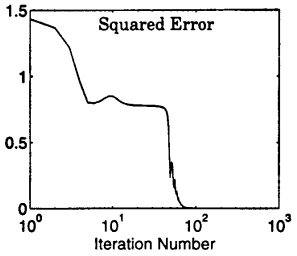
- One of the simple method to avoid an error trajectory in the weight space being oscillatory is to add to the weight update a momentum term.
- Such a term is proportional to the weight update at the previous step.



$$\Delta W(n) = \eta \cdot \boldsymbol{\delta}(n) \cdot \mathbf{x}^T(n) + \alpha \cdot \Delta W(n-1) \ , \quad 0 < \alpha < 1 \quad (5.2)$$

where α is a momentum term parameter.

- Such modification to the steepest descend learning law acts as a low-pass filter smoothing the error trajectory.
- As a result it is possible to apply higher learning rate, η.



5.8.2 Adaptive learning rate

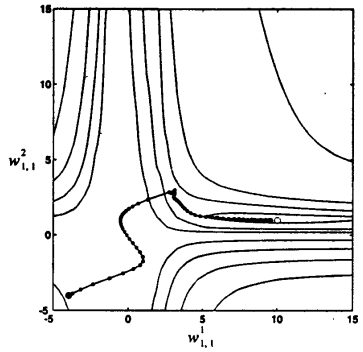
- One of the ways of increasing the convergence speed, that is, to move faster downhill to the minimum of the mean-squared error, $J(W)$, is to vary adaptively the learning rate parameter, η.
- A typical strategy is based on monitoring the rate of change of the mean-squared error and can be described as follows:
- If J is decreasing consistently, that is, ∇J is negative for a prescribed number of steps, then the learning rate is increased linearly:

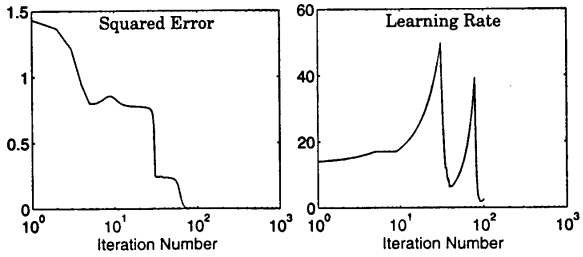
$$\eta(n+1) = \eta(n) + a\ , \quad a > 0 \tag{5.3}$$

- If the error has increased, $(\nabla J > 0)$, the learning rate is exponentially reduced:

$$\eta(n+1) = b \cdot \eta(n)\ , \quad 0 < b < 1 \tag{5.4}$$

- In general, increasing the value of the learning rate the learning tends to become unstable which is indicated be an increase in the value of the error function.
- Therefore it is important to quickly reduce η.





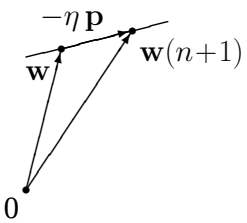
Advanced methods of optimisation

- Optimization or minimisation of a function of many variables (multi-variable function), $J(\mathbf{w})$, has been researched since the XVII century and its principles were formulated by such mathematicians as Kepler, Fermat, Newton, Leibnitz, Gauss.
- In general the problem is to find an optimal learning gain and the optimal search direction that takes into account the shape of the error function, that is its curvature.

5.9 Line search minimisation procedures

- Gradient descent minimization procedures are based on updating the weight vector

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta\,\mathbf{p}(n) \tag{5.5}$$



  where η is the learning gain and the vector $\mathbf{p}(n)$ describes the direction of modification of the weight vector.
- The vector $\mathbf{p}$ is typically equal to the negative gradient of the error function

$$\mathbf{p}(n) = -\mathbf{g}(n)\ , \quad \text{where} \quad \mathbf{g}(n) = \nabla J(\mathbf{w}(n))$$

- Note that the next value of weight vector, $\mathbf{w}(n+1)$, is obtained from the current value of weight vector, $\mathbf{w}$, by moving it along the direction of a vector, $\mathbf{p}$.

- We can now find an optimal value of η for which the performance index

$$J(\mathbf{w}(n+1)) = J(\mathbf{w} + \eta\,\mathbf{p}) \tag{5.6}$$

  is minimised.

- The optimal η is typically found through a search procedure along the direction $\mathbf{p}$ (line optimization)
- In order to find some properties of the line optimization we calculate the partial derivative of J with respect to η:

$$\frac{\partial J(\mathbf{w}(n+1))}{\partial \eta} = \frac{\partial J(\mathbf{w}(n+1))}{\partial \mathbf{w}(n+1)} \frac{\partial \mathbf{w}(n+1)}{\partial \eta} = \frac{\partial J(\mathbf{w}(n+1))}{\partial \mathbf{w}(n+1)}\,\eta\,\mathbf{p}^T = \eta\,\nabla J(\mathbf{w}(n+1)) \cdot \mathbf{p}^T \tag{5.7}$$

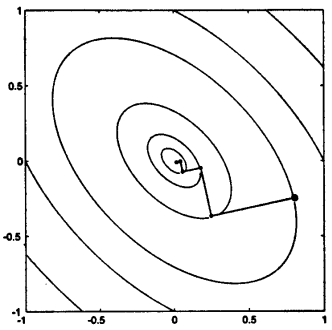
- For the optimal value of η, this derivative needs to be zero and we have the following relationship

$$\nabla J(\mathbf{w}(n+1)) \cdot \mathbf{p}^T = \mathbf{g}(n+1) \cdot \mathbf{p}^T = 0 \tag{5.8}$$

- It states that the next estimate of the gradient, $\mathbf{g}(n+1) = \nabla J(\mathbf{w}(n+1))$, is to be orthogonal to the current search direction, $\mathbf{p}$ for the optimal value of η

This means, in particular, that

- if we combine the **line minimisation** technique with the **steepest descent** algorithm when we move in the direction opposite to the gradient, $\mathbf{p} = -\mathbf{g}$,
- then we will be descending along the zig-zag line, each segment being orthogonal to the next one.



- In order to smooth the descend direction, the steepest-descent technique is replaced with the conjugate gradient algorithm.

5.10 Conjugate Gradient Algorithm

- The conjugate gradient algorithms also involved the line optimisation with respect to η, but
- in order to avoid the zig-zag movement through the error surface, the next search direction, $\mathbf{p}(n+1)$, instead of being exactly orthogonal to the gradient, tries to maintain the current search direction, $\mathbf{p}(n)$, namely

$$\mathbf{p}(n+1) = -\mathbf{g}(n) + \beta(n)\mathbf{p}(n) \tag{5.9}$$

  where scalar $\beta(n)$ is selected in such a way that

- the directions $\mathbf{p}(n+1)$ and $\mathbf{p}(n)$ are **conjugate with respect to the Hessian matrix**, $\nabla^2 J(\mathbf{w}) = H$ (the matrix of all second derivatives of J), that is,

$$\mathbf{p}(n+1) \cdot H \cdot \mathbf{p}^T(n) = 0 \tag{5.10}$$

- In practice, the Hessian matrix is not being calculated and the following three approximate choices of $\beta(n)$ are the most commonly used

- Hestenes-Steifel formula

$$\beta(n) = \frac{(\mathbf{g}(n) - \mathbf{g}(n-1)) \cdot \mathbf{g}^T(n)}{(\mathbf{g}(n) - \mathbf{g}(n-1)) \cdot \mathbf{p}^T(n-1)} \tag{5.11}$$

- Fletcher-Reeves formula

$$\beta(n) = \frac{\mathbf{g}(n) \cdot \mathbf{g}^T(n)}{\mathbf{g}(n-1) \cdot \mathbf{g}^T(n-1)} \tag{5.12}$$

- Polak-Ribiére formula

$$\beta(n) = \frac{(\mathbf{g}(n) - \mathbf{g}(n-1)) \cdot \mathbf{g}^T(n)}{\mathbf{g}(n-1) \cdot \mathbf{g}^T(n-1)} \tag{5.13}$$

In summary, the conjugate gradient involves:

- initial search direction, $\mathbf{p}(0) = -\mathbf{g}(0)$,
- line minimisation with respect of η,
- calculation of the next search direction as in eqn (5.9), and
- β from one of the above formulae.

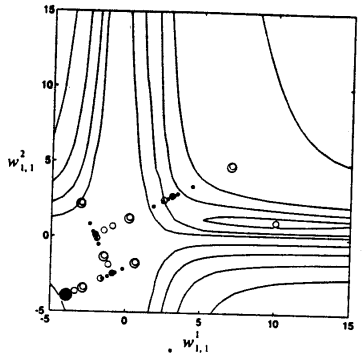


Figure 12.15 Intermediate Steps of CGBP

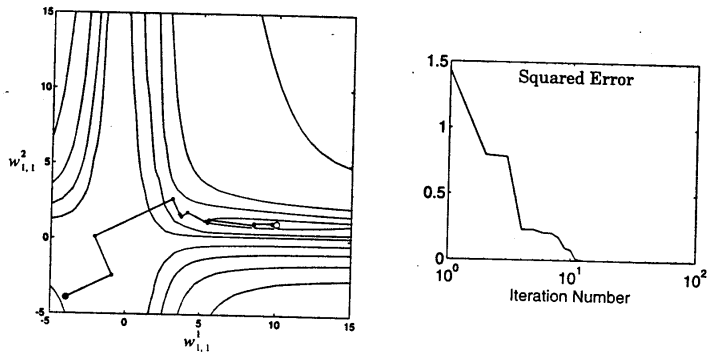


Figure 12.16 Conjugate Gradient Trajectory

5.11 Newton's Methods

- In Newton's methods minimisation is based on utilisation of not only the **first derivatives** (the gradient) of the error function, but also its **second derivatives** (Hessian matrix).
- Consider the Taylor series expansion of the performance index as in eqn (4.5) which can be re-written as

$$J(\mathbf{w}(n+1)) = J(\mathbf{w}) + \Delta\mathbf{w} \cdot \nabla J + \frac{1}{2}\Delta\mathbf{w} \cdot \mathbf{H} \cdot \Delta\mathbf{w}^T + \cdots$$

- To minimise $J(\mathbf{w}(n+1))$ we calculate the gradient and equate it to zero

$$\nabla J(\mathbf{w}(n+1)) = \nabla J + \Delta\mathbf{w} \cdot \mathbf{H} + \cdots = 0$$

- Neglecting the higher order expansion terms, we have the following fundamental for Newton's methods equation:

$$\Delta\mathbf{w} = -\nabla J \cdot \mathbf{H}^{-1} \tag{5.14}$$

- This equation says that a more accurate weight update is the direction opposite to the gradient vector modified (rotated) by the inverse of the Hessian matrix of the performance index J.
- The Hessian matrix provides additional information about the shape of the performance index surface in the neighbourhood of $\mathbf{w}(n)$.

- The Newton's methods are typically faster than conjugate gradient algorithms.
- However, they require computations of the inverse of the Hessian matrix which are relatively complex.
- Many specific algorithms originate from the Newton's method, the fastest and most popular being the **Levenberg-Marquardt algorithm**, which originate from the Gauss-Newton method.
- The Newton's methods use the batch training mode, rather then the pattern mode which is based on derivatives of instantaneous errors.

5.12 Gauss-Newton method

In the Gauss-Newton method the **Hessian matrix** is approximated by a product of the **Jacobian matrix**.

In order to explain details of the method let us repeat the standard assumption:

- all weights have been arranged in one row vector

$$\mathbf{w} = [w_1 \dots w_j \dots w_K]$$

- all (instantaneous) errors form a column vector

$$\boldsymbol{\varepsilon}(\mathbf{w}(n)) = \mathbf{d}(n) - \mathbf{y}(n) = [\varepsilon_1 \dots \varepsilon_k \dots \varepsilon_m]^T$$

- the instantaneous performance index $E(\mathbf{w}(n))$ is a sum of squares of errors

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

  (for brevity, arguments like $\mathbf{w}$ and n are often omitted)

- The total performance index (mean squared error)

$$F(\mathbf{w}) = \frac{1}{M} \sum_{n=1}^{N} E(\mathbf{w}(n))$$

  where $M = m\,N$. The symbol F is used in place of J to avoid confusion with the Jacobian matrix.

- We consider first derivatives of instantaneous errors. The jth element of the instantaneous gradient vector can now be expressed as

$$[\nabla E(\mathbf{w}(n))]_j = \frac{\partial E(\mathbf{w}(n))}{\partial w_j} = \sum_{i=1}^{m} \varepsilon_k(\mathbf{w}) \frac{\partial \varepsilon_i(\mathbf{w})}{\partial w_j} = \boldsymbol{\varepsilon}^T(\mathbf{w}) \left[\frac{\partial \varepsilon_1(\mathbf{w})}{\partial w_j} \dots \frac{\partial \varepsilon_m(\mathbf{w})}{\partial w_j} \right]^T$$

- This expression can be generalised into a matrix form for the gradient:

$$\nabla E(\mathbf{w}(n)) = \boldsymbol{\varepsilon}^T(\mathbf{w}(n)) \mathcal{J}(\mathbf{w}(n)) \tag{5.15}$$

  where

$$\mathcal{J}(\mathbf{w}(n)) = \begin{bmatrix} \dfrac{\partial \varepsilon_1(\mathbf{w})}{\partial w_1} & \cdots & \dfrac{\partial \varepsilon_1(\mathbf{w})}{\partial w_K} \\ \vdots & & \vdots \\ \dfrac{\partial \varepsilon_m(\mathbf{w})}{\partial w_1} & \cdots & \dfrac{\partial \varepsilon_m(\mathbf{w})}{\partial w_K} \end{bmatrix} \tag{5.16}$$

  is the $m \times K$ matrix of first derivatives known as the **Jacobian** matrix.

- In order to find the Hessian matrix of the instantenous performance index we differentiate eqn (5.15):

$$\nabla^2 E(\mathbf{w}(n)) = \frac{\partial (\boldsymbol{\varepsilon}^T(\mathbf{w}) \mathcal{J}(\mathbf{w}))}{\partial \mathbf{w}}$$

- Let us calculate first, for simplicity, the k, j element of the Hessian matrix

$$\begin{aligned}[\nabla^2 E(\mathbf{w}(n))]_{k,j} &= \frac{\partial^2 E(\mathbf{w})}{\partial w_k \partial w_j} = \sum_{i=1}^{m} \left(\frac{\partial \varepsilon_i}{\partial w_k} \frac{\partial \varepsilon_i}{\partial w_j} + \varepsilon_i \frac{\partial^2 \varepsilon_i}{\partial w_k \partial w_j} \right) \\ &= \left[\frac{\partial \varepsilon_1}{\partial w_k} \cdots \frac{\partial \varepsilon_m}{\partial w_k} \right] \begin{bmatrix} \frac{\partial \varepsilon_1}{\partial w_j} \\ \vdots \\ \frac{\partial \varepsilon_m}{\partial w_j} \end{bmatrix} + \boldsymbol{\varepsilon}^T \frac{\partial^2 \varepsilon_i}{\partial w_k \partial w_j}\end{aligned}$$

- Generalizing the above expression into a matrix form, we obtain:

$$\nabla^2 E(\mathbf{w}) = \mathcal{J}^T(\mathbf{w}) \mathcal{J}(\mathbf{w}) + \boldsymbol{\varepsilon}^T(\mathbf{w}) R(\mathbf{w}) \tag{5.17}$$

  where

$$R(\mathbf{w}) = \left\{ \frac{\partial^2 \varepsilon_i}{\partial w_k \partial w_j} \right\}$$

  it the matrix of all second derivatives of errors.

- If we neglect the term

$$\boldsymbol{\varepsilon}^T(\mathbf{w}) R(\mathbf{w})$$

  due to the fact that errors are small, then we obtain the Gauss-Newton method.

- In this method the Hessian matrix is approximated as:

$$H(\mathbf{w}(n)) = \nabla^2 E(\mathbf{w}(n)) \approx \mathcal{J}^T(\mathbf{w}(n)) \mathcal{J}(\mathbf{w}(n))$$

  and the weight update equation (5.14) becomes:

$$\Delta \mathbf{w}(n) = -\nabla E(n) H^{-1}(n) = -\boldsymbol{\varepsilon}^T(n) \mathcal{J}(n) \left(\mathcal{J}^T(n) \mathcal{J}(n) \right)^{-1} \tag{5.18}$$

- For simplicity, we have considered the pattern update, however, in the following section we consider a modification of the Gauss-Newton algorithm in which the batch update of weights is employed.

5.13 Levenberg-Marquardt algorithm

5.13.1 The algorithm

- One problem with the Gauss-Newton method is that the approximated Hessian matrix may not be invertible.
- To overcome this problem in the **Levenberg-Marquardt** algorithm a small constant μ is added such that

$$\mathbf{H}(\mathbf{w}) \approx \mathbf{J}^T(\mathbf{w})\mathbf{J}(\mathbf{w}) + \mu I$$

  where $\mathbf{H}(\mathbf{w})$ and $\mathbf{J}(\mathbf{w})$ are the **batch Hessian and Jacobian matrices**, respectively, I is the identity matrix and μ is a small constant.
- The gradient of the batch performance index, $F(\mathbf{w})$, can be calculated as

$$\nabla F(\mathbf{w}) = \sum_{n=1}^{N} \nabla E(\mathbf{w}(n)) = \sum_{n=1}^{N} \boldsymbol{\varepsilon}^T(\mathbf{w}(n)) \cdot \mathcal{J}(\mathbf{w}(n)) = \mathbf{e}^T(\mathbf{w})\mathbf{J}(\mathbf{w})$$

  where

$$\mathbf{e}^T = \text{scan}(D - Y) = [\boldsymbol{\varepsilon}^T(1) \dots \boldsymbol{\varepsilon}^T(N)]$$

  is the vector of all instantaneous errors.

- The batch Jacobian matrix, $\mathbf{J}(\mathbf{w})$, is a block-column matrix consisting of the instantaneous Jacobian matrices, $\mathcal{J}(n)$

$$\mathbf{J}(\mathbf{w}) = \begin{bmatrix} \mathcal{J}(1) \\ \vdots \\ \mathcal{J}(N) \end{bmatrix}$$

- As a result, the batch weight update is as in eqn (5.19):

$$\Delta\mathbf{w} = -\nabla F \cdot \mathbf{H}^{-1} = -\mathbf{e}^T(\mathbf{w})\mathbf{J}(\mathbf{w}) \left(\mathbf{J}^T(\mathbf{w})\mathbf{J}(\mathbf{w}) + \mu I\right)^{-1} \tag{5.19}$$

5.13.2 Calculation of the Jacobian matrix

- Calculation of the Jacobian matrix is similar to calculation of the gradient of the performance index.
- The main difference is that in the case of the gradient we differentiate the sum of squared errors, whereas in the case of the Jacobian we differentiate errors themselves, see eqn (5.16).
- Following the derivation of the basic backpropagation algorithm we consider a **two-layer perceptron** with two weight matrices, W^h, W^y. The weight vector $\mathbf{w}$ is formed by scanning these matrices in rows, so that we have:

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

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

- The instantaneous Jacobian matrix, $\mathcal{J}(n)$, is $m \times K$, one column per weight, and can be partitioned into two blocks related to the hidden and output weights, respectively:

$$\mathcal{J}(n) = [\mathcal{J}^h(n) \;\; \mathcal{J}^y(n)]$$

5.13.3 Output layer

- The output Jacobian matrix $\mathcal{J}^y(n)$ is $m \times mL$, where m is the number of output neurons and L is the number of hidden signals, h_j, as in Figure 4–3.
- The elements of $\mathcal{J}^y(n)$ are the first derivatives $\dfrac{\partial \varepsilon_i}{\partial w_{kj}^y}$ of errors $\varepsilon_i(n)$ with respect to weights w_{kj}^y. We have

$$\varepsilon_i(n) = d_i(n) - y_i(n) \;, \quad y_i(n) = \sigma(v_i(n)) \;, \quad v_i(n) = W_{i:}^y \cdot \mathbf{h}(n)$$

- Now, an element of the output Jacobian matrix can be calculate in the following way

$$\frac{\partial \varepsilon_i}{\partial w_{kj}^y} = \begin{cases} 0 & \text{if} \quad i \neq k \quad \text{(error is local to the } k\text{th neuron)} \\ -\dfrac{\partial y_k}{\partial w_{kj}^y} & \text{if} \quad i = k \end{cases}$$

- Hence, the output Jacobian matrix has a block-diagonal structure.

- Subsequently, we have

$$\frac{\partial y_k}{\partial w_{kj}^y} = \frac{\partial y_k}{\partial v_k} \frac{\partial v_k}{\partial w_{kj}^y} = \sigma_k' \cdot h_j \;, \quad \text{where} \quad \sigma_k' = \frac{\partial y_k}{\partial v_k}$$

  which can be generalised to the matrix of non-zero blocks of $\mathcal{J}^y$ as

$$P_{k:} = \frac{\partial \varepsilon_k}{\partial W_{k:}^y} = -\sigma_k' \cdot \mathbf{h}^T \;, \quad P = \frac{\partial \varepsilon_k}{\partial W^y} = -\boldsymbol{\sigma}' \cdot \mathbf{h}^T$$

- Rows of the matrix P form the diagonal blocks of the Jacobian $\mathcal{J}^y$.
- More formally, we can write

$$\mathcal{J}^y = -\mathrm{diag}(\boldsymbol{\sigma}') \otimes \mathbf{h}^T \tag{5.20}$$

  where $\otimes$ denotes the Kronecker product.

5.13.4 Hidden layer

- The hidden Jacobian matrix $\mathcal{J}^h(n)$ is $m \times mp$, where m is the number of output neurons and p is the number of input signals, $x_i(n)$.
- An element of the hidden Jacobian matrix can be calculate in the following way

$$\frac{\partial \varepsilon_k}{\partial w_{ji}^h} = -\frac{\partial y_k}{\partial w_{ji}^y} = -\frac{\partial y_k}{\partial v_k} \frac{\partial v_k}{\partial w_{ji}^h} = -\sigma_k' \frac{\partial v_k}{\partial w_{ji}^h}$$

- If we take into account that

$$v_k = W^y_{k:} \cdot \mathbf{h} = \cdots + w^y_{kj} \cdot h_j + \cdots$$

and

$$h_j = \psi(W^h_{j:} \cdot \mathbf{x}) = \psi(\cdots + w^h_{ji} \cdot x_i + \cdots)$$

- then we can arrive at the final form for a single element of the hidden Jacobian matrix:

$$[\mathcal{J}^h]_{k,ji} = \frac{\partial \varepsilon_k}{\partial w^h_{ji}} = -\sigma'_k \cdot w^y_{kj} \cdot \frac{\partial h_j}{\partial w^h_{ji}} = -\sigma'_k \cdot w^y_{kj} \cdot \psi'_j \cdot x_i \tag{5.21}$$

- A $1 \times p$ block of the hidden Jacobian matrix can be expressed as follows

$$[\mathcal{J}^h]_{k,j:} = -s_{kj} \cdot \mathbf{x}^T \quad , \quad \text{where} \quad s_{kj} = \sigma'_k \cdot w^y_{kj} \cdot \psi'_j$$

and finally, we have

$$\mathcal{J}^h = -S^h \otimes \mathbf{x}^T \quad , \quad \text{where} \quad S^h = \text{diag}(\boldsymbol{\sigma}') \cdot W^y \cdot \text{diag}(\boldsymbol{\psi}') \tag{5.22}$$

The complete algorithm — general description

The complete Levenberg-Marquardt algorithm can be described as follows:

1. For the input matrix, X, calculate the matrices of:
   - hidden signals, H,
   - output signals, Y,
   - related derivatives, Ψ', and Φ',
   - errors, $D - Y$, and $\mathbf{e}$.
2. Calculate instantaneous Jacobian matrices, $\mathcal{J}^h$ and $\mathcal{J}^y$ as in eqns (5.22) and (5.20), and arrange them in the batch Jacobian $\mathbf{J}$.
3. Calculate the weight update, $\Delta\mathbf{w}$, according to eqn (5.22) for a selected value of μ.
4. Calculate the batch performance index, $F(\mathbf{w} + \Delta\mathbf{w})$, and compare it with the previous value, $F(\mathbf{w})$.

   If $F(\mathbf{w} + \Delta\mathbf{w}) > F(\mathbf{w})$, reduce the value of the parameter μ and recalculate $\Delta\mathbf{w}$ (step 3), until $F(\mathbf{w} + \Delta\mathbf{w}) > F(\mathbf{w})$ is reduced.
5. Repeat calculations from step 1 for the updated weights, $\mathbf{w} = \mathbf{w} + \Delta\mathbf{w}$.

5.13.5 Some computational details

$\mathbf{J}^T\mathbf{J}$:

- The size of $\mathbf{J}$ is $M \times K = mN \times L(p+m)$, that is, (size of the data set)×(set of the weight set) which might be prohibitively large for a big data set.
- In order to reduce the size of the intermediate data, we can proceed in the following way:

$$\mathbf{J}^T\mathbf{J} = [\mathcal{J}^T(1) \dots \mathcal{J}^T(N)] \begin{bmatrix} \mathcal{J}(1) \\ \vdots \\ \mathcal{J}(N) \end{bmatrix} = \sum_{n=1}^{N} \mathcal{J}^T(n)\mathcal{J}(n)$$

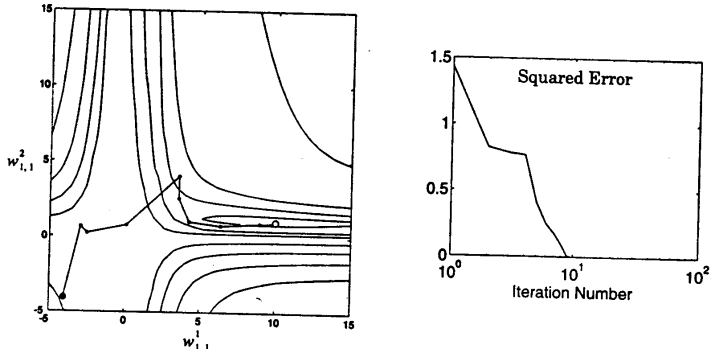
- The size of the matrix to be inverted, $(\mathbf{J}^T\mathbf{J} + \mu I)$ is $K \times K$.

$\mathbf{e}^T\mathbf{J}$:

- Similarly, we can calculate the above gradient as

$$\nabla F = \mathbf{e}^T\mathbf{J} = \sum_{n=1}^{N} \boldsymbol{\varepsilon}^T(n)\mathcal{J}(n)$$

- In this way, we need not store the complete batch Jacobian, $\mathbf{J}$.



LMbp trajectory

5.14 Speed comparison

Some of the functions available for the batch training in **Neural Network Toolbox** are listed in the following table together with a relative time to reach convergence.

Function	Algorithm	Relative time
LM trainlm	Levenberg-Marquardt	1.00
BFG trainbfg	BFGS Quasi-Newton	4.58
RP trainrp	Resilient Backpropagation	4.97
SCG trainscg	Scaled Conjugate Gradient	5.34
CGB traincgb	Conjugate Gradient with Powell/Beale Restarts	5.80
CGF traincgf	Fletcher-Powell Conjugate Gradient	6.89
CGP traincgp	Polak-Ribiére Conjugate Gradient	7.23
OSS trainoss	One-Step Secant	8.46
GDX traingdx	Variable Learning Rate Backpropagation	24.29

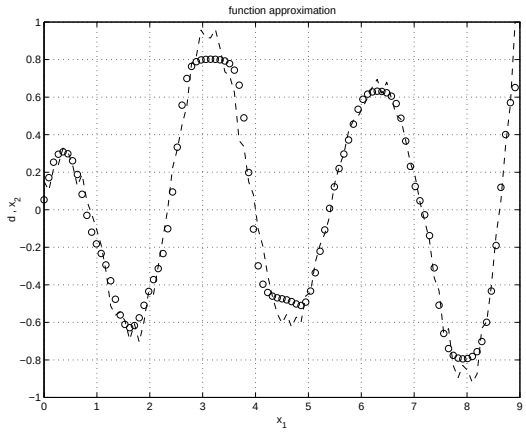
6 Self-Organizing Neural Networks

6.1 Supervised and Unsupervised Learning

- Learning algorithms which were considered for a single perceptron, linear adaline, and multilayer perceptron belong to the class of **supervised learning** algorithms.
- In this case the training data is divided into input signals, $\mathbf{x}(n)$, and target signals, $\mathbf{d}(n)$.
- A typical learning algorithm is driven by error signals $\boldsymbol{\varepsilon}(n)$ which are the differences between the actual network output, $\mathbf{y}(n)$, and the desire (or target) output for a given input.
- For a pattern learning, we can express the weight update in the following general form

$$\Delta\mathbf{w}(n) = \mathcal{L}(\mathbf{w}(n), \mathbf{x}(n), \boldsymbol{\varepsilon}(n))$$

where $\mathcal{L}$ represents a learning algorithm.



- If we say that a neural network can describe a model of data, then a multilayer perceptron describes the data in a form of a curve, or surface or hypersurface which approximates a functional relationship between $\mathbf{x}(n)$, and $\mathbf{d}(n)$.

Self-organizing neural networks — Unsupervised Learning

- Self-organising neural networks employ unsupervised learning laws and discover **characteristic features** in input data without using a target or desired output.
- Information about the characteristic features of input data is created during the learning process and stored in the synaptic weights.
- Output signals describe relationship between the current input signals and the weight vectors.
- Two basic groups of unsupervised learning algorithms and related self-organizing neural networks, namely:
  - (Generalised) Hebbian Learning
  - Competitive Learning

  can be distinguished by the type of characteristic features that they "discover" from the input data, namely, "shape" of data and constellation of clusters of points.

Generalised Hebbian Learning

- Generalised Hebbian Learning extracts from data a set of **principal directions** along which data is organised in a p-dimensional space.
- Each direction is represent by a relevant weight vector. The number of those principal directions is, at most, equal to the dimensionality of the input space p.
- In an illustrative example presented in Figure 6–1 the two-dimensional data is organised along two principal directions, $\mathbf{w}_1$ and $\mathbf{w}_2$.

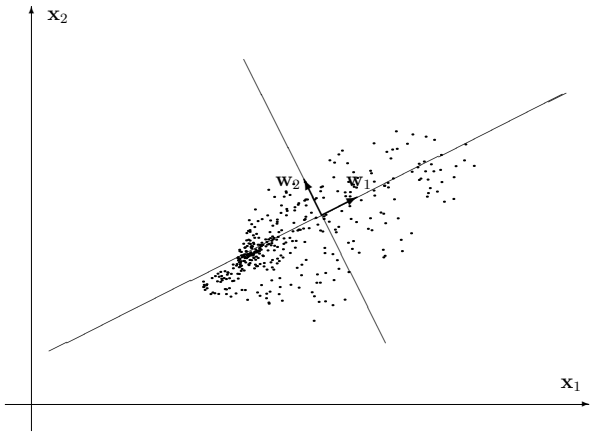


Figure 6–1: A 2-D pattern with principal directions

Competitive Learning

- Competitive Learning extracts from data a set of **centers of data clusters**.
- Each center point is stored as a weight vector. It is obvious that the number of clusters is independent of dimensionality of the input space.

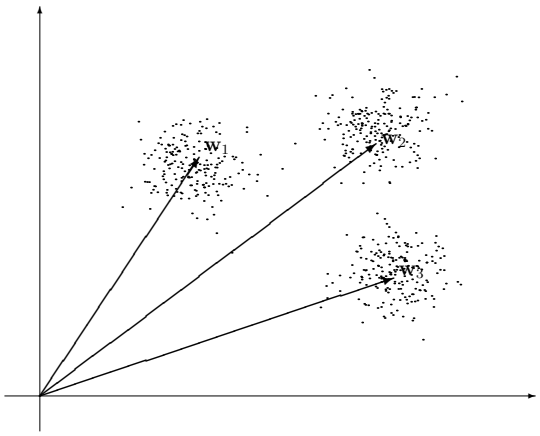


Figure 6–2: Example of two-dimensional data organised in three clusters. Claster centres are represented by three weight vectors.

- An important extension of a basic competitive learning is known as **feature maps**. A feature map is obtained by adding some form of topological organization to neurons.

6.2 Hebbian learning

6.2.1 Basic structure of Hebbian learning neural networks

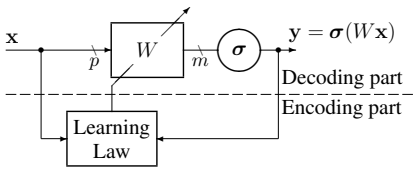


Figure 6–3: A block-diagram of a basic Hebbian learning neural network consisting of a single layer decoding part and a learning (encoding) part

- The decoding layer contains a single weight matrix W which is $m \times p$.
  Each row weight vector is associated with one neuron.
- The decoding layer is often linear, which further simplifies the network structure.
  The complexity of the network comes from the structure of the learning law employed.
- Note the absence of the target value in the encoding/learning part (un-supervised learning).
- The basic idea behind a Hebbian learning law is to make the update of a synaptic weight proportional to both input and output signals:

$$\begin{aligned} w_{ji}(n+1) &= f(w_{ji}(n), y_j(n), x_i(n)) \\ &= w_{ji}(n) + \eta y_j(n) x_i(n) \end{aligned} \qquad (6.1)$$

- These two signals, y_j and x_i are locally available at the ji synapse, therefore, this type of a learning law is termed as a **local learning law**.

The concept of the **local learning** law is illustrated in Figure 6–4.

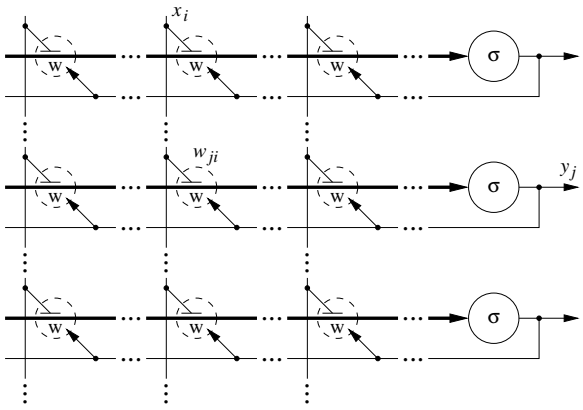


Figure 6–4: A neural network with a local learning law circuitry.

It may be observed that the neuron output signal, y_j, is available locally at the synapse through the additional feedback connection. This feedback is essential to the learning process.

- The **basic Hebbian learning** law in the form as in eqn (6.1) cannot be used because it is fundamentally **unstable**, that is, weights reveal an unlimited grow during the learning process.
- It is relatively simple to show that if a stable solution to the learning law (6.1) exists it must be zero.
- Assuming for simplicity linear neurons and re-writing eqn (6.1) in a matrix form gives:

$$\mathbf{W}(n+1) = \mathbf{W}(n) + \eta \mathbf{y}(n)\mathbf{x}^T(n) = \mathbf{W}(n) + \eta \mathbf{W}(n)\mathbf{x}(n)\mathbf{x}^T(n)$$

- Application of the expectation operator, $E[\cdot]$, to both sides yields

$$E[\mathbf{W}] = E[\mathbf{W}] + \eta E[\mathbf{W}]E[\mathbf{x}\mathbf{x}^T]$$

- Hence, the steady-state value of the weight matrix, $\bar{\mathbf{W}} = E[\mathbf{W}]$, must satisfy the following equation

$$\bar{\mathbf{W}} R = 0$$

  where

$$R = E[\mathbf{x}\mathbf{x}^T] \approx \frac{1}{N} X X^T$$

  is the input **correlation matrix**.
- The input correlation matrix is non-singular, therefore, the only possible steady-state value of the weight matrix is $\bar{\mathbf{W}} = 0$.

6.2.2 Stable Hebbian learning

In order to stabilize a Hebbian learning law two basic steps are required

- Assuming that $\mathbf{x}$ is a p-dimensional random vector representing the input data, it is required that its **mean** to be zero:

$$E[\mathbf{x}] = 0$$

  In practical calculations with MATLAB, when input vectors are collected in the $p \times N$ input matrix X, the non-zero mean is removed from the input data in the following way:

```
mX=mean(X,2) ; X=X-mX(:,ones(1,N));
```

  Note that if the mean is zero **correlation** matrix is identical to the **covariance** matrix.
- The basic Hebbian learning law is to be modified in such a way that the magnitude (lenght) of weight vectors should tend to unity.

  Assuming for simplicity a single neuron network, it can be achieve by the following normalization:

$$\mathbf{w} \Leftarrow \mathbf{w} + \eta y \mathbf{x}^T \; ; \quad \mathbf{w} \Leftarrow \mathbf{w}/||\mathbf{w}|| \tag{6.2}$$

  where the vector magnitude is calculated in the usual way as:

$$||\mathbf{w}|| = \sqrt{\sum_{i=1}^{p} w_i^2}\,, \quad \text{also} \quad ||\mathbf{w}||^2 = \mathbf{w}\mathbf{w}^T$$

- Normalization as in eqn (6.2) is computationally relatively complex, therefore, we can use the following simplification based on the Taylor series expansion.

$$\frac{1}{||\mathbf{w}+\eta y\mathbf{x}^T||} = \frac{1}{\sqrt{(\mathbf{w}+\eta y\mathbf{x}^T)(\mathbf{w}+\eta y\mathbf{x}^T)^T}} = \frac{1}{\sqrt{\mathbf{w}\mathbf{w}^T + 2\eta y\mathbf{w}\mathbf{x} + \eta^2 y^2\mathbf{x}^T\mathbf{x}}}$$

- If we assume that the previous weight vector was normalised, that is,

$$\mathbf{w}\mathbf{w}^T = ||\mathbf{w}||^2 = 1$$

  and that $\eta \ll 1$ is small so that η^2 is negligible, then we can further write:

$$\begin{aligned}\frac{\mathbf{w}+\eta y\mathbf{x}^T}{||\mathbf{w}+\eta y\mathbf{x}^T||} &\approx \frac{\mathbf{w}+\eta y\mathbf{x}^T}{\sqrt{1+2\eta y\mathbf{w}\mathbf{x}}} \approx (\mathbf{w}+\eta y\mathbf{x}^T)(1-\eta y\mathbf{w}\mathbf{x}) \\ &\approx \mathbf{w}+\eta y\mathbf{x}^T - \eta y^2\mathbf{w} - \eta^2 y^2\mathbf{x}^T \approx \mathbf{w}+\eta y(\mathbf{x}^T - y\mathbf{w})\end{aligned}$$

- It can be proved that if we update weights according to the last equation, that is,

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta y(n)(\mathbf{x}^T(n) - y(n)\mathbf{w}(n)) \tag{6.3}$$

  then the magnitude of the weight vector will be close to unity, $||\mathbf{w}|| \to 1$.

- This is the form of the weight update used in the **Generalised Hebbian Learning** (GHL).

6.2.3 A single neuron case — the Oja's rule

The unsupervised learning algorithm described in eqn (6.3) for a single neuron case is known as the Oja's rule, and can be written in the following form:

$$\Delta\mathbf{w} = \eta y(\mathbf{x}^T - y\mathbf{w}) = \eta y\tilde{\mathbf{x}}^T\ , \quad y = \mathbf{w}\mathbf{x} = \mathbf{x}^T\mathbf{w}^T \tag{6.4}$$

where the augmented input vector is:

$$\tilde{\mathbf{x}} = \mathbf{x} - y\mathbf{w}^T \tag{6.5}$$

The negative term brings in the required stabilization of the learning law. To show this we calculate the projection of the update vector, $\Delta\mathbf{w}$ onto the current weight vector, $\mathbf{w}$:

$$\Delta\mathbf{w}\,\mathbf{w}^T = \mathbf{x}^T\mathbf{w}^T - y\mathbf{w}\mathbf{w}^T = y(1-||\mathbf{w}||^2)$$

Therefore, if the current weight vector is not on the unit circle, the update vector will bring the next weight vector closer to the the unit circle.

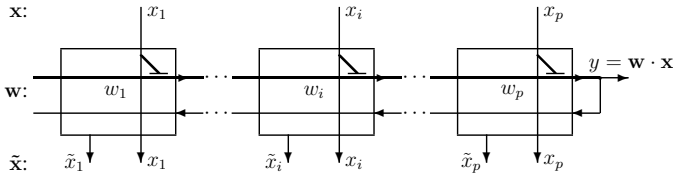


Figure 6–5: Internal structure of a neural network implementing the Oja's rule (6.4)

Each synapse aggregates the dendritic signal, and, during learning, generates the augmented input signal, and updates its weight.

Extraction of the first principal direction

- It is now possible to show that applying the learning law of eqn (6.4), the weight vector $\mathbf{w}$ converges to the **eigenvector** $\mathbf{q}_1$ of the input **correlation matrix** R associated with the largest **eigenvalue** λ_1 of R.
- The direction of the eigenvector $\mathbf{q}_1$ is referred to as the first **principal direction**.
- The sketch of the proof of the necessary convergence condition is as follows. Substitution of eqn (6.5) in eqn (6.4) yields:
$$\Delta\mathbf{w} = \eta(\mathbf{w}\mathbf{x}\mathbf{x}^T - \mathbf{w}\mathbf{x}\mathbf{x}^T\mathbf{w}^T\mathbf{w})$$
- Applying the statistical expectation operator to both sides gives:
$$E[\Delta\mathbf{w}] = \eta(\mathbf{w}E[\mathbf{x}\mathbf{x}^T] - \mathbf{w}E[\mathbf{x}\mathbf{x}^T]\mathbf{w}^T\mathbf{w}) \tag{6.6}$$
- The terms in eqn (6.6) can be estimated as follows. If we assume that the weight vector converges to a steady-state value, then the expectation of the weight update vectors is zero, that is, $E[\Delta\mathbf{w}] = 0$.
- The expectation of outer products of input vectors is the covariance (correlation) matrix:
$$R = E[\mathbf{x}\mathbf{x}^T] \approx \frac{1}{N}XX^T$$
- Now, eqn (6.6) can be written as
$$\mathbf{w}R = (\mathbf{w}R\mathbf{w}^T)\mathbf{w} \tag{6.7}$$
- Denote the scalar:
$$\lambda_1 = \mathbf{w}R\mathbf{w}^T \tag{6.8}$$

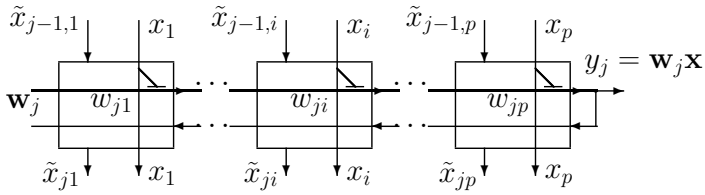
- Using definition (6.8) we can finally re-write eqn (6.7) in the following form
$$\mathbf{w}R = \lambda_1\mathbf{w} \tag{6.9}$$
- Alternatively, if we assume that
$$\mathbf{w}(n) \to \pm\mathbf{q}_1^T \;\; \text{as} \;\; n \to \infty \tag{6.10}$$
we have from eqn (6.9):
$$R\mathbf{q}_1 = \lambda_1\mathbf{q}_1 \quad \text{where} \quad \lambda_1 = \mathbf{q}_1^T R\mathbf{q}_1 \tag{6.11}$$
- Eqn (6.11) specifies a pair: an eigenvector $\mathbf{q}_1$ and related eigenvalue λ_1 which are characteristic values of a matrix (R in this case) such that, if R acts on $\mathbf{q}_1$ it modifies only the magnitude of the eigenvector.
- It can be shown that a "well behaving" $p \times p$ matrix has exactly p eigenvector-eigenvalue pairs.
- It can also be shown that λ_1 defined in eqn (6.8) or (6.11) and obtained using the Oja's rule is the largest eigenvalue of the input correlation matrix, R.
- As it is stated in eqn (6.10) the weight vector converges to the eigenvector $\mathbf{q}_1$, but the orientation of these two vectors does not have to be the same.
- Therefore, comparing the weight vector with the eigenvector when monitoring the convergence process, it is better to use the projection rather than the difference, that is:
$$|\mathbf{w} \cdot \mathbf{q}_1| \to 1 \quad \text{whereas} \quad ||\mathbf{w} - \mathbf{q}_1|| \to 0 \text{ or } +2 \tag{6.12}$$
- Condition (6.12) can be used to conveniently monitor the convergence process.

6.3 Self-Organizing Principal Component Analysis

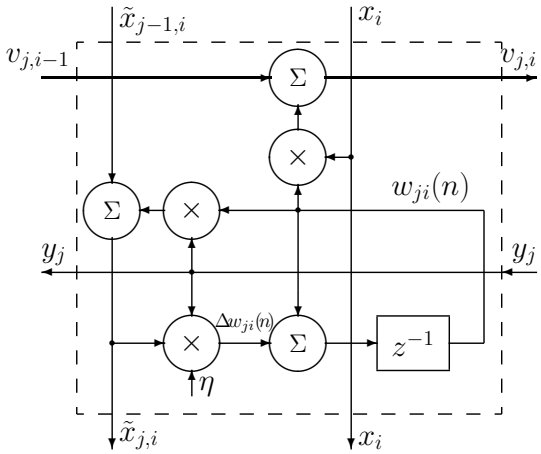
- The single neuron structure can be extended into a p-neuron network, the weight vector associated with subsequent neurons extracting the subsequent eigenvectors of the input correlation matrix.
- Such a network performs the **Principal Component Analysis** also known as the Karhunen-Loève transform.
- The objective of this analysis (transform) is to extract all principal directions characterising input data.
- The learning law involved is known as the Sanger's rule or Generalized Hebbian Algorithm (GHA).
- The idea behind the generalization of the Oja's rule neural network is to use in the learning part of neurons the **augmented input vectors** as specified by eqn (6.5).
- The weight update in the Sanger's rule, that is, the **Generalized Hebbian Algorithm** (GHA) can be described in the following way:

$$\begin{array}{lll} y_1 = \mathbf{w}_1\mathbf{x}\ , & \tilde{\mathbf{x}}_1 = \mathbf{x} - y_1\mathbf{w}_1^T\ , & \Delta\mathbf{w}_1 = \eta y_1\tilde{\mathbf{x}}_1^T \\ y_2 = \mathbf{w}_2\mathbf{x}\ , & \tilde{\mathbf{x}}_2 = \tilde{\mathbf{x}}_1 - y_2\mathbf{w}_2^T\ , & \Delta\mathbf{w}_2 = \eta y_2\tilde{\mathbf{x}}_2^T \\ & \cdots & \\ y_j = \mathbf{w}_j\mathbf{x}\ , & \tilde{\mathbf{x}}_j = \tilde{\mathbf{x}}_{j-1} - y_j\mathbf{w}_j^T\ , & \Delta\mathbf{w}_j = \eta y_j\tilde{\mathbf{x}}_j^T \\ & \cdots & \end{array} \tag{6.13}$$

- The jth neuron of the GHA has the following structure:



6.3.1 Structure of a single synapse implementing the Generalised Hebbian Learning



Identify:

- the dendritic activation signal

$$v_{j,i} = v_{j,i-1} + w_{ji}\cdot x_i\ , \quad (v_{j,0} = 0\ ,\ \ y_j = v_{j,p})$$

- the augmented input signal

$$\tilde{x}_{ji} = \tilde{x}_{j-1,i} - w_{ji}\cdot y_j$$

- the synaptic weight update (learning law)

$$\Delta w_{ji}(n) = \eta\cdot y_j(n)\cdot\tilde{x}_{ji}(n)$$

$$w_{ji}(n+1) = w_{ji}(n) + \Delta w_{ji}(n)$$

6.3.2 A matrix form of the Generalised Hebbian Learning

- In order to re-write the **Generalised Hebbian Learning** algorithm in a matrix for, let us first note that from eqn (6.13), we can write the augmented input signal vectors in the following form:

$$\tilde{\mathbf{x}}_j^T = \mathbf{x}^T - [y_1 \dots y_j] \begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_j \end{bmatrix} = \mathbf{x}^T - [y_1 \dots y_j \, 0 \dots 0] \, W = \mathbf{x}^T - \tilde{\mathbf{y}}_j^T W \tag{6.14}$$

  where

$$\tilde{\mathbf{y}}_j^T = [y_1 \dots y_j \, 0 \dots 0]$$

  is the output vector $\mathbf{y}$ in which the last $m - j$ components are set to zero.

- Subsequently, the weight update for the j neuron specified in eqn (6.13) can be re-written in the following form

$$\Delta \mathbf{w}_j = \eta(y_j \mathbf{x}^T - y_j \tilde{\mathbf{y}}_j^T W)$$

- Finally, the **pattern update** of the whole weight matrix in the GHA algorithm can be expressed in the following compact form:

$$\mathbf{y} = W\mathbf{x} \; , \quad \Delta W = \eta(\mathbf{y}\mathbf{x}^T - \text{tril}(\mathbf{y}\mathbf{y}^T)W) \tag{6.15}$$

  where $\text{tril}(\cdot)$ denotes the lower-triangular matrix with elements above the main diagonal set to zero.

- The sketch of the prove that the network weight vectors converge to the eigenvalues of the input correlation matrix is as follows.
- Taking the statistical expectation operator on both sides of the weight update equation (6.15), and assuming that in the steady-state the updates are zeros, we have

$$0 = E[W\mathbf{x}\mathbf{x}^T] - E[\text{tril}(W\mathbf{x}\mathbf{x}^T W^T)W]$$

- This can be re-written as

$$WR = \text{tril}(WRW^T)W$$

  or

$$\begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_p \end{bmatrix} R = \text{tril}(\begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_p \end{bmatrix} R \begin{bmatrix} \mathbf{w}_1^T & \dots & \mathbf{w}_p^T \end{bmatrix}) \begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_p \end{bmatrix}$$

- The lower-triangular matrix in the right-hand side is of the form:

$$\Lambda = \text{tril}(WRW^T) = \begin{bmatrix} \lambda_1 & & \\ \{\lambda_{ji}\} & \ddots & \mathbf{0} \\ & & \lambda_p \end{bmatrix} , \quad \text{where} \quad \lambda_{ji} = \mathbf{w}_j R \mathbf{w}_i^T$$

- Therefore we finally have

$$\begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_p \end{bmatrix} R = \begin{bmatrix} \lambda_1 & & \\ \{\lambda_{ji}\} & \ddots & \mathbf{0} \\ & & \lambda_p \end{bmatrix} \begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_p \end{bmatrix} \tag{6.16}$$

or

$$WR = \Lambda W \tag{6.17}$$

- The first row of eqn (6.16), namely,

$$\mathbf{w}_1 R = \lambda_1 \mathbf{w}_1 \,, \qquad \text{where} \qquad \lambda_1 = \mathbf{w}_1 R \mathbf{w}_1^T$$

  is exactly the same as eqn (6.11) from the Oja's network, the weight vector of the first neuron, $\mathbf{w}_1$, converging to the eigenvector of the input correlation matrix associated with the largest eigenvalue, λ_1.

- In order to show that the other weight vectors converge to subsequent eigenvectors it is enough to show that the off-diagonal coefficients $\lambda_{ji} = \mathbf{w}_j R \mathbf{w}_i^T$ converge to zero due to orthogonality of the eigenvectors.

6.4 Example of image compression using GHA

An image to be compressed is a `rr×cc` sub-image from `'gatlin'`:

```
load gatlin
rr = 120 ;  cc = 180 ;  % numbers of rows and columns of Img
Img = X((1:rr)+20, (1:cc)+20) ;
figure(1), image(Img), colormap(map)
```

The image is divided into $r \times c$ blocks, each nth block being converted into a $p = r \times c$ component vector $\mathbf{x}(n)$. These vectors are stored in a $p \times N$ matrix X:

```
r = 4 ; c = 4 ; p = r*c ;
X = blkM2vc(Img, [r c]) ;
[p N] = size(X) ;        % X  is  16 by 1350  = 120 by 180
```

The next step is to remove the mean from X, that is, to normalize it. The pattern matrix X is now ready to be used in the learning algorithm:

```
Xm = mean(X')' ;
X = X - Xm(:, ones(1, N)) ;
X = X/max(max(abs(X))) ;
```

The internal learning loop goes through all patterns from X (one epoch) updating weights in a 'pattern mode'. In order to monitor the convergence process the length of the weight vectors, `lw`, is calculated. It is expected that when the weights converge to respective eigenvectors their lengths will be unity. This condition is checked in the outer loop.

```
m = 4 ;                      %  number of neurons
W = 0.6*(rand(m, p)-0.5) ;  % weight initialisation
lw = sum((W.^2)') ;          % length of weight vectors
W2 = zeros(N, m) ;  % changes to the length of weight vectors
figure(2)
eta = 2e-2 ;         % learning gain
er = .05 ;           % the length convergence error
ep = 0 ;             % number of training epochs
while (sum( abs(1-lw) < er ) < m) & (ep < 16)
  [rs rn] = sort(rand(1, N)) ;
  for n = 1:N
    x  = X(:, rn(n)) ;   % randomised selection of patterns
    y  = W*x ;
    dW = eta*(y*x' - tril(y*y')*W) ;
    W  = W + dW ;
    lw = sum((W.^2)') ;    % length of weight vectors
    W2(n, :) = lw ;
  end
  plot(W2),
  ep = ep+1 ;
  if  ep == 1
    plot(W2),  axis([0 1400 0 1.1])
    title(['lengths of weight vectors during training'])
    xlabel('pattern number')
  end
  grid on, drawnow
end
plot(W2),  axis([0 1400 0.5 1.1])
title(['lengths of weight vectors after ',num2str(ep),'epochs'])
xlabel('pattern number (the last epoch)'),  grid on, drawnow
```


It the above example, there are $m = 4$ neurons, that is, the neural network is trained for only $m = 4$ out of $p = 16$ 'principal directions', specified by the eigenvectors of the input correlation matrix.

After one pass through the training data only the first weight vector representing the most significant eigenvector has converged to achieve the unity length as demonstrated in Figure 6–6.

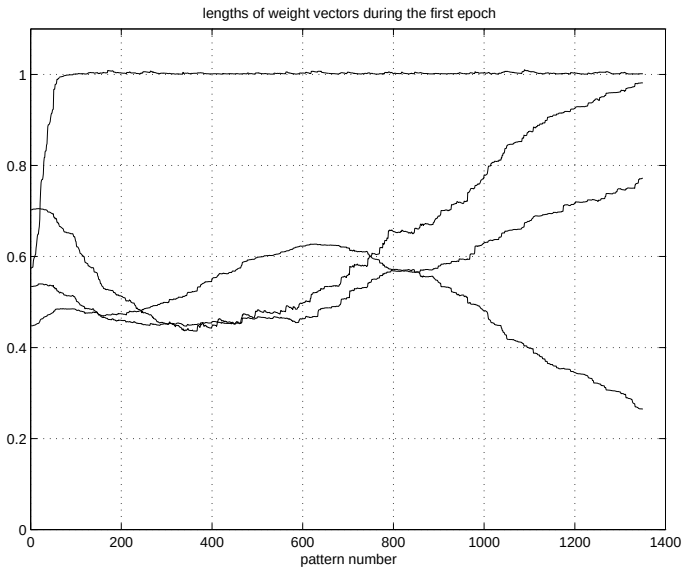


Figure 6–6: The length of the weight vectors representing the m most significant principal directions.

After `ep = 7` epochs, all $m = 4$ weight vectors have attained the unity length within the error specified

by `er` as presented in Figure 6–7.

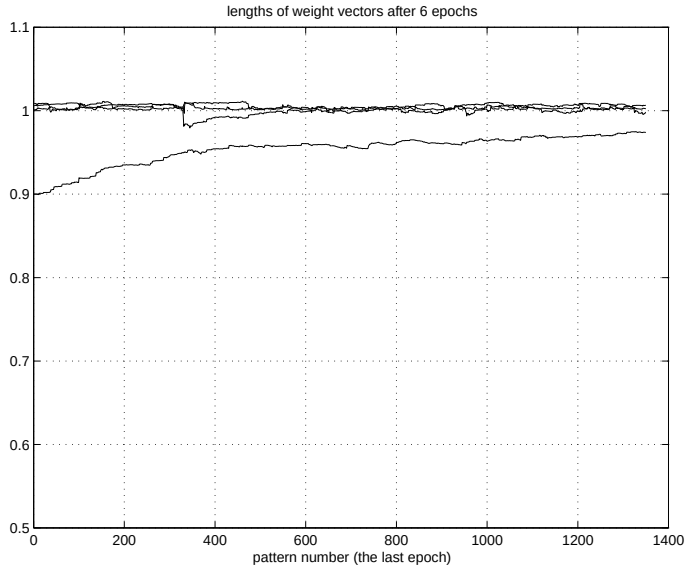


Figure 6–7: The length of the weight vectors representing the m most significant principal directions.

It is possible to observe that during training the weight vectors converge in a serial way, one by one, starting from the weight vector representing the most significant principal direction.

Next we will check that the weight vectors are really equal to the eigenvectors of the input correlation matrix, $R = X \cdot X'$. A number of aspects need to be taken into account in this comparison.

First, the `eig` function arranges the eigenvectors as the column vectors whereas the weights are row vectors.
Secondly, the eigenvalues must be sorted in the descending order and the respective eigenvectors must be re-arranged accordingly.
Thirdly, the orientation of the eigenvectors and the weight vectors might be opposite. This can be done as follows:

```
R = (X*X')/N ;  % the autocorrelation matrix
[V D] = eig(R) ;
dd = diag(D)
d = flipud(dd(p-m+1:p, :)) ;
VV = fliplr( V(:, p-m+1:p)) ;
WV = W*VV
```

Now, `d` contains m largest eigenvalues, and `VV` is a $p \times m$ matrix of respective eigenvectors of the input correlation matrix.
In order to verify that the weight matrix has converged to m principal directions we calculate all possible inner products between all m weight vectors and m eigenvectors. The $m \times m$ matrix `WV` stores these products. The diagonal terms of this matrix should ideally be equal to ± 1, whereas the off-diagonal terms should be zero. In our example, we have;

```
WV = W*VV = 1.0004   0.0308  -0.0029   0.0164
            0.0059   0.9972  -0.0465  -0.0316
            0.0039  -0.0199  -1.0029  -0.0001
           -0.0022   0.0353   0.0100  -0.9860
```

which looks as a sensible approximation.

The next test is to check that the variance of the output signals is equal to the eigenvalues of the input correlation matrix.

```
yy = W*X ;  % the output signals  m by N
vy = var(yy')' ;
  [   d        vy    ]
    2.2875    2.2913
    0.1575    0.1571
    0.1138    0.1147
    0.0424    0.0415
```

Indeed, the approximation seems to be satisfactory.

In order to obtain the compressed image, the matrix of output vectors Y= `yy` is transformed first into a reconstructed input matrix $\hat{X}$= `Xr`

$$\hat{X} = W' \cdot Y \tag{6.18}$$

The reconstructed input matrix can now be re-arranged into $r \times c$ image blocks. This can be done in the following way:

```
Xr = W'*yy ;
Imr = vc2blkM(Xr, r, rr) ;
Imr = round(mc*Imr/max(max(Imr))) ;
figure(3), image(Imr), colormap(gray(mc))
```

The original and compressed images are shown in Figure 6–8.

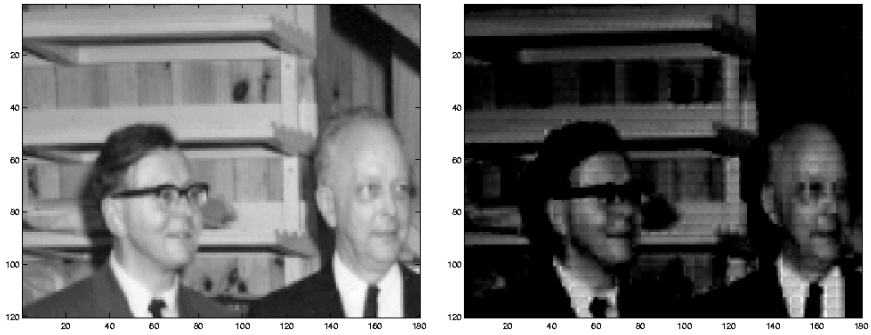


Figure 6–8: The original and compressed images.

7 Competitive Neural Networks

The basic competitive neural network consists of two layers of neurons:

- The similarity-measure layer,
- The competitive layer, also known as a "Winner-Takes-All" (WTA) layer

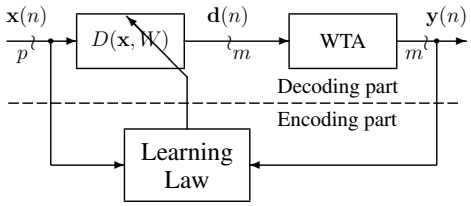


Figure 7–1: The structure of the competitive neural network

The **similarity-measure** layer contains an $m \times p$ weight matrix, W, each row associated with one neuron.

This layer generates signals $\mathbf{d}(n)$ which indicate the similarity between the current input vector $\mathbf{x}(n)$ and each synaptic vector $\mathbf{w}_j(n)$.

The **competitive layer** generates m binary signals y_j. This signal is asserted "1" for the neuron j-th winning the competition, which is the one for which the distance signal d_j attains minimum.

In other words, $y_j = 1$ indicates that the j-th weight vector, $\mathbf{w}_j(n)$, is most similar to the current input vector, $\mathbf{x}(n)$.

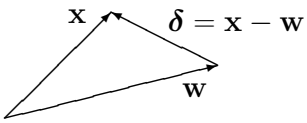
7.1 The similarity-Measure Layer

The detailed structure of the similarity-measure layer depends on the specific measure employed.
Let

$$d = D(\mathbf{x}, \mathbf{w})$$

denote a distance, or similarity measure between two vectors, $\mathbf{x}$ and $\mathbf{w}$. The following measures can be taken into considerations:

- The most obvious similarity measure is the **Euclidean norm**, that is, the magnitude of the difference vector, $\boldsymbol{\delta}$,



$$d = ||\mathbf{x} - \mathbf{w}|| = ||\boldsymbol{\delta}|| = \sqrt{\delta_1^2 + \ldots + \delta_p^2} = \sqrt{\boldsymbol{\delta}^T \cdot \boldsymbol{\delta}}$$

  Such a measure is relatively complex to calculate.

- The square of the Euclidean norm:

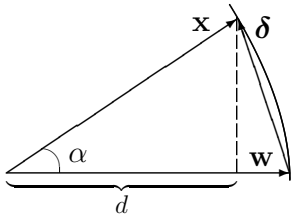
$$d = ||\mathbf{x} - \mathbf{w}||^2 = ||\boldsymbol{\delta}||^2 = \sum_{j=1}^{p} \delta_j^2 = \boldsymbol{\delta}^T \cdot \boldsymbol{\delta}$$

  The square root has been eliminated, hence calculations of the similarity measure have been simplified.

- The Manhattan distance, that is, the sum of absolute values of the coordinates of the difference vector

$$d = \sum_{j=1}^{p} |\delta_j| = \mathrm{sum}(\mathrm{abs}(\boldsymbol{\delta}))$$

- The **projection** of $\mathbf{x}$ on $\mathbf{w}$. This is the simplest measure of similarity of the **normalised** vectors:



$$d = \frac{\mathbf{w}^T}{||\mathbf{w}||} \cdot \mathbf{x} = ||\mathbf{x}|| \cdot \cos\alpha$$

For normalised vectors, when $||\mathbf{w}|| = ||\mathbf{x}|| = 1$

we have

$$d = \cos\alpha \in [-1, +1]$$

and also

if	$d = +1$	then	$\|\|\boldsymbol{\delta}\|\| = 0$	vectors are identical
if	$d = 0$	then	$\|\|\boldsymbol{\delta}\|\| = \sqrt{2}$	vectors are orthogonal
if	$d = -1$	then	$\|\|\boldsymbol{\delta}\|\| = 2$	vectors are opposite

In general, we have

$$||\boldsymbol{\delta}||^2 = (\mathbf{x} - \mathbf{w})^T(\mathbf{x} - \mathbf{w}) = \mathbf{x}^T\mathbf{x} - 2\mathbf{x}^T\mathbf{w} + \mathbf{w}^T\mathbf{w} = ||\mathbf{x}||^2 + ||\mathbf{w}||^2 - 2\mathbf{w}^T\mathbf{x}$$

Hence, for normalised vectors, the projection similarity measure, d, can be expressed as follows

$$d = \mathbf{w}^T\mathbf{x} = 1 - \frac{1}{2}||\boldsymbol{\delta}||^2$$

Normalisation of the input vectors can be achieved without any loss of information by adding another dimension to the input space during preprocessing of the input data.

Assume that $(p-1)$-dimensional input vectors, $\hat{\mathbf{x}}(n)$, have been already scaled into the $[-1, +1]$ range, that is,

$$||\hat{\mathbf{x}}(n)|| \leq 1 \quad \forall (n = 1, \ldots, N)$$

If we add the pth component, x_p to $\hat{\mathbf{x}}$, we obtain a p-dimensional vector, $\mathbf{x}$ and we can write the following relationship

$$||\mathbf{x}(n)||^2 = ||\hat{\mathbf{x}}(n)||^2 + x_p^2(n)$$

Now, in order to normalise all p-dimensional vectors, $\mathbf{x}$, the pth components must be calculated as follows

$$x_p^2(n) = 1 - ||\hat{\mathbf{x}}(n)||^2 \quad \forall (n = 1, \ldots, N)$$

This operation is equivalent to the projection of the input data from the $(p-1)$-dimensional hyper-plane up onto the p-dimensional unity hyper-sphere, as illustrated in Figures 7–2 and 7–3.

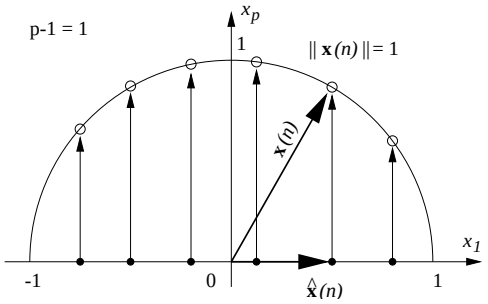


Figure 7–2: Normalisation of 1-D input data by projection onto a unity circle

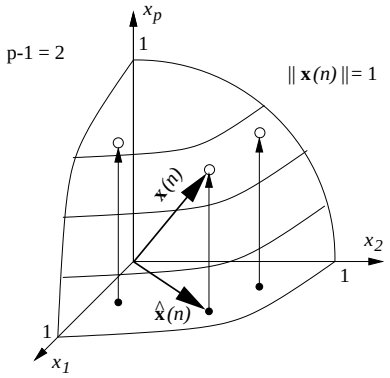


Figure 7–3: Normalisation of 2-D input data by projection onto a unity sphere

For the **normalised vectors** the similarity-measure layer is linear, that is,

$$\mathbf{d} = W \cdot \mathbf{x}$$

The structure of the competitive neural network which employs projections as the similarity measures is illustrated in Figure 7–4 in the form of a signal-flow block-diagram and the dendritic diagram.

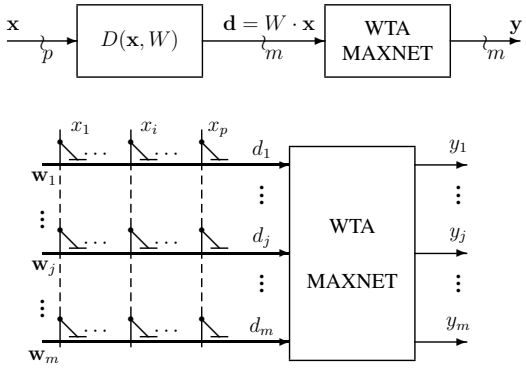


Figure 7–4: The structure of the similarity-measure layer for the normalised vectors

The greater the signal $d_j(n)$ is, the more similar is the jth weight vector, $\mathbf{w}_j$ to the current input signal $\mathbf{x}(n)$.

7.2 The Competitive Layer

The competitive layer, also known as the MinNet (MaxNet), or the "Winner-Takes-All" (WTA) network, generates binary output signals, y_j, which, if asserted, point to the winning neuron, that is, the one with the weight vector being closest to the current input vector:

$$y_j = \begin{cases} 1 & \text{if} \quad j = \arg\min\limits_k D(\mathbf{x}, \mathbf{w}_k) \\ 0 & \text{otherwise} \end{cases}$$

In other words, the MaxNet (MinNet) determines the largest (smallest) input signal, $d_j = D(\mathbf{x}, \mathbf{w}_j)$.

The competitive layer is, in itself, a **recurrent** neural network with the predetermined and fixed feedback connection matrix, M. The matrix M has the following structure:

$$M = \begin{bmatrix} 1 & & \\ & \ddots & -\alpha \\ & -\alpha & \ddots \\ & & & 1 \end{bmatrix}$$

where $\alpha < 1$ is a small positive constant. Such a matrix describes a network with a local unity feedback, and a feedback to other neurons with the connection strength $-\alpha$.

The MaxNet network

The MaxNet network is an implementation of the competitive layer described by the following block diagram

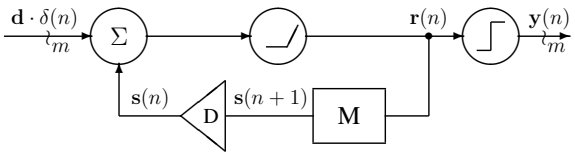


Figure 7–5: The block-diagram of the MaxNet network (competitive layer)

The input vector, $\mathbf{d}$, is active only at the initial time, $n = 0$, which is accomplished by means of the delta function

$$\delta(n) = \begin{cases} 1 & \text{for} \quad n = 0 \quad \text{(initial condition)} \\ 0 & \text{for} \quad n \neq 0 \end{cases}$$

The state equation, which describes how the state vector, $\mathbf{s}(n)$, evolves with time, can be written in the following form

$$\mathbf{s}(n+1) = M \cdot \mathbf{r}(n) + \mathbf{d} \cdot \delta(n)$$

where, the feedback signals $\mathbf{r}(n)$, are formed by clipping out the negative part of the state signals, that is,

$$r_j(n) = \max(0, s_j(n)) = \begin{cases} s_j(n) & \text{for} \quad s_j(n) \geq 0 \\ 0 & \text{for} \quad s_j(n) < 0 \end{cases} \quad \text{for} \quad j = 1, \ldots, m$$

Finally, the binary output signals $\mathbf{y}(n)$, are formed as follows

$$y_j(n) = \begin{cases} 1 & \text{for } r_j(n) > 0 \\ 0 & \text{for } r_j(n) = 0 \end{cases} \quad \text{for } j = 1, \ldots, m$$

Alternatively, we can evaluate the state signals as:

$$\begin{aligned} \mathbf{s}(1) &= \mathbf{d} \\ \mathbf{r}(n) &= \max(0, \mathbf{s}(n)) \\ s_j(n+1) &= r_j(n) - \alpha \sum_{k \neq j} r_k \text{ , for } n > 1 \text{ and } j = 1, \ldots, m \end{aligned}$$

At each step n, signals $s_j(n+1)$ consists of the self-excitatory contribution, $r_j(n)$, and a total lateral inhibitory contribution, $\alpha \sum_{k \neq j} r_k$.

After a certain number of iterations all r_k signals but the one associated with the largest input signal, d_j, are zeros.

Example

Let $\alpha = 0.2$ and $m = 8$. The feedback signals $\mathbf{r}(n)$ can be calculated as follows:

$\mathbf{r}(1)$	$-$	$\mathbf{r}(2)$	$-$	$\mathbf{r}(3)$	$-$	$\mathbf{r}(4)$	$-$	$\mathbf{r}(5)$	$-$	$\mathbf{r}(6)$
7.3	6.98	.32	.99	0	$*$	0	$*$	0	$*$	0
4.2	7.60	0	$*$	0	$*$	0	$*$	0	$*$	0
9.6	6.52	3.08	.44	2.64	.24	2.4	.13	2.27	.04	2.23
.7	8.30	0	$*$	0	$*$	0	$*$	0	$*$	0
5.5	7.34	0	$*$	0	$*$	0	$*$	0	$*$	0
2.9	7.86	0	$*$	0	$*$	0	$*$	0	$*$	0
8.6	6.72	1.88	.68	1.20	.53	.67	.48	.19	.45	0
3.4	7.76	0	$*$	0	$*$	0	$*$	0	$*$	0

The columns marked '$-$' represent the total inhibitory contribution.

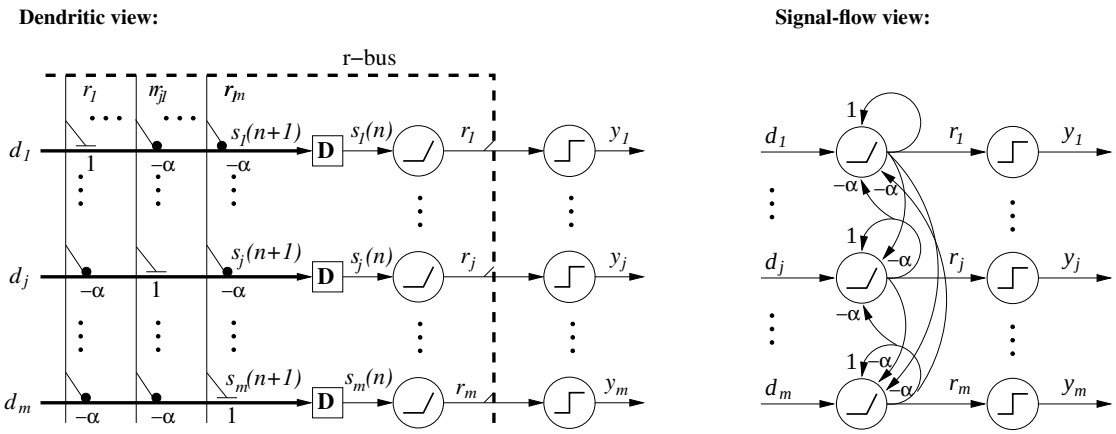


Figure 7–6: The dendritic and the signal-flow views of the competitive layer

Note the unity self-excitatory connections, and the lateral inhibitory connections.

7.3 Unsupervised Competitive Learning

The objective of the competitive learning is to adaptively quantize the input space, that is, to perform **vector quantization** of the input space

It is assumed that the input data is organised in, possibly overlapping, **clusters**. Each synaptic vector, $\mathbf{w}_j$, should converge to a **centroid** of a cluster of the input data.

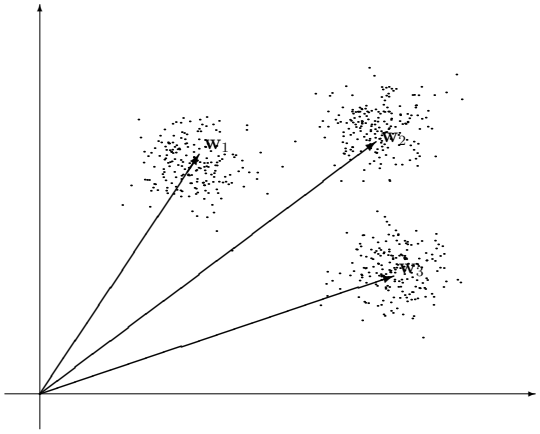


Figure 7–7: An example of a 2-D pattern with three clusters of data

In other words, the input vectors are categorized into m classes (clusters), each weight vector representing the center of a cluster.

It is said that such a set of weight vectors describes **vector quantization** also known as **Voronoi** (or **Dirichlet**) **tessellation** of the input space.

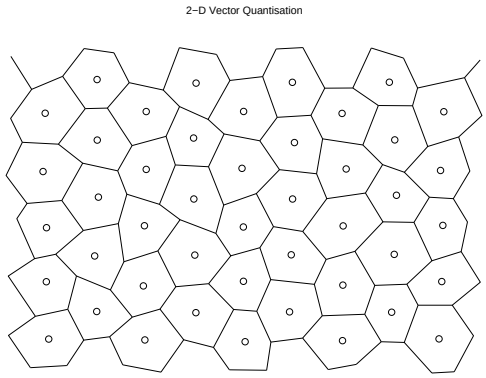


Figure 7–8: Example of Voronoi tessellation (vector quantization) of a 2-D space.

The space is partitioned into polyhedral regions with centres represented by weight vectors (dots in Figure 7–8).
The boundaries of the regions are planes perpendicularly bisecting lines joining pairs of centres (prototype vectors) of the neighbouring regions.

A very important application of the vector quantization is in data coding/compression. In this context the set of weights (prototype vectors) is referred to as a **codebook**.

We can find a set of prototype vectors with competitive learning algorithms.

A simple competitive learning

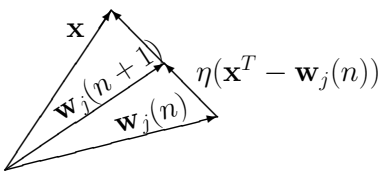
A simple competitive learning can be describe as follow:

- Weight vectors are usually initialise with n randomly selected input vectors:

$$\mathbf{w}_j(0) = \mathbf{x}^T(\mathrm{rand}(j))$$

- For each input vector, $\mathbf{x}(n)$, determine the winning neuron, j for which its weight vector, $\mathbf{w}_j(n)$, is closest to the input vector. For this neuron, $y_j(n) = 1$.
- Adjust the weight vector of the winning neuron, $\mathbf{w}_j(n)$, in the direction of the input vector, $\mathbf{x}(n)$; do not modify weight vectors of the loosing neurons, that is,

$$\Delta\mathbf{w}_j(n) = \eta(n)y_j(n)(\mathbf{x}^T(n) - \mathbf{w}_j(n))$$



- In order to arrive at a static solution, the learning rate is gradually linearly reduced, for example

$$\eta(n) = 0.1(1 - \frac{n}{N})$$

Example — scripts Cmpti.m, Cmpts.m

In this example we consider vector quantization of the 2-D input space into $m = 5$ regions specified by m weight vectors arranged in a 5×2 weight matrix W.

We start with generation of a 2-D pattern consisting of m clusters of normally distributed points. Weights are initialised with points from the data matrix X.

```
%  Cmpti.m
% Initialisation of competitive learning
p = 2; m = 5 ;      % p inputs, m  outputs
clst = randn(p, m); %  cluster centroids
Nk = 100;       % points per cluster
N = m*Nk ;      % total number of points
sprd = 0.2 ;    % a relative spread of the Gaussian "blobs"
X = zeros(p,N+m); % X is  p by m+N input data matrix
wNk = ones(1, Nk);
for k = 1:m  % generation of m Gaussian "blobs"
  xc = clst(:,k) ;
  X(:,(1+(k-1)*Nk):(k*Nk))=sprd*randn(p,Nk)+xc(:,wNk) ;
end
[xc k] = sort(rand(1,N+m));
X = X(:, k) ;      % input data is shuffled randomly
winit = X(:,1:m)'; % Initial values of weights
X = X(:,m+1:N+m);  % Data matrix  is  p by N
```

In the script implementing a simple competitive learning algorithm we pass once over the data matrix, aiming at weights to converge to the centres of Gaussian "blobs".


```
%   Cmpts.m
W = winit ;
V = zeros(N, m, p);  % to store all weights
V(1,:,:) = W ;
wnm = ones(m,1) ;
eta = 0.08 ;    % learning gain
deta = 1-1/N ;  % learning gain decaying factor
for  k = 1:N    % main training loop
  xn = X(:,k)' ;
% the current vector is compared to all weight vectors
  xmw = xn(wnm,:)-W ;
  [win jwin] = min(sum((xmw.^2),2));
% the weights of the winning neurons are update
  W(jwin,:) = W(jwin,:) + eta*xmw(jwin,:);
  V(k,:,:) = W;
% eta = eta*deta ;
end

plot(X(1,:),X(2,:),'g.',clst(1,:),clst(2,:),'ro', ...
     winit(:,1),winit(:,2),'bx' , ...
     V(:,:,1), V(:,:,2), 'b', W(:,1),W(:,2) , 'r*'), grid
```

In Figure 7–9 there are four examples of data organised in five overlapping clusters. After one training epoch, the weights converged to centroids of the clusters with varying degree of success.

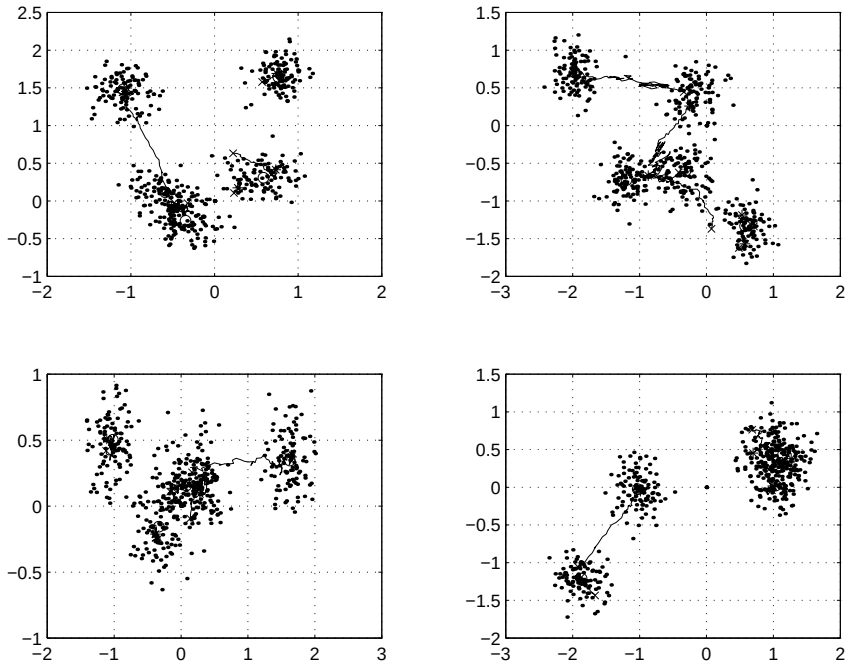


Figure 7–9: Simple competitive learning: ‘o’ – centroids of generated clusters, ‘$\times$’ – initial weights, ‘*’ – Final weights

7.4 Competitive Learning and Vector Quantization

- Competitive learning is used to create a **codebook**, that is a weight matrix, W, which stores the centers of data clusters.

  For a p-dimensional input space and m-neurons (size of the codebook) the structure of the codebook (weight matrix, W) is as follows

cluster #	W	1	$\cdots$	p
1	$\mathbf{w}_1$	w_{11}	$\ldots$	w_{1p}
2	$\mathbf{w}_2$	w_{21}	$\ldots$	w_{2p}
$\vdots$	$\vdots$	$\vdots$	W	$\vdots$
m	$\mathbf{w}_m$	w_{m1}	$\ldots$	w_{mp}

- The codebook (weight matrix) describes the tessellation of the input space known as **vector quantization** (VQ). For a 2-D input space the above concepts can be illustrated as in Figure 7–10.
- The codebook is created from a representative set of data using a competitive learning.

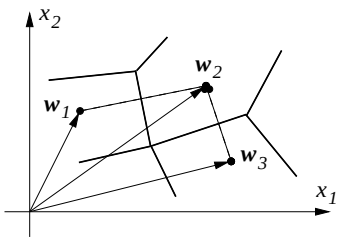


Figure 7–10: Voronoi tessellation (vector quantization) of a 2-D space.

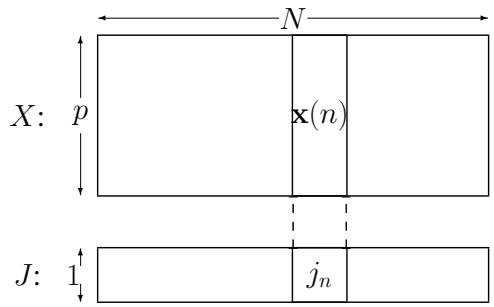
Vector quantization is used in **data compression**.

Given a codebook, $W = [\mathbf{w}_1; \ldots; \mathbf{w}_1]$ the process of data compression can be described as follows:

1. For every input vector, $\mathbf{x}(n)$ find the closest codebook entry, that is the weight vector $\mathbf{w}_{j_n}$, for which the distance

   $$|\mathbf{w}_{j_n} - \mathbf{x}(n)|$$

   attains minimum. Then we identify that the input vector, $\mathbf{x}(n)$ belongs to the j-th cluster.
2. Replace all input data $X = \{\mathbf{x}(n)\}_{1:N}$ by their corresponding indices $J = \{j_n\}_{1:N}$.



3. Transmit J instead of X. The codebook, W must be made known/transmitted to the receiver.
4. At the receiver, using a codebook, replace every j_n with the corresponding $\mathbf{w}_{j_n}$ which will now represent $\mathbf{x}(n)$. The representation error is:

   $$\varepsilon_n = |\mathbf{w}_{j_n} - \mathbf{x}(n)|$$

   The total squared error is

   $$F = \sum_{n=1}^{N} \varepsilon_n^2$$

In order to calculate the **compression ratio**, C, let us assume that

- Weights and input data are represented by B-bit numbers,
- The length of the codebook

$$m \leq 2^b$$

that is, all cluster indices are b-bit numbers

Then,

- the total number of bits to represent the input data, that is, the size of X is

$$B \times p \times N$$

- The size of the compressed data, J, is

$$b \times N$$

- The size of the codebook, W, is

$$B \times p \times m$$

- The compression ratio is:

$$C = \frac{BpN}{bN + Bpm} \approx \frac{Bp}{b}$$

For example, for $B = 16$, $p = 12$, $b = 8$

$$C \approx 24$$

8 Self-Organizing Feature Maps

Self-Organizing Feature Maps (SOFM or SOM) also known as Kohonen maps or topographic maps were first introduced by von der Malsburg (1973) and in its present form by Kohonen (1982).

According to Kohonen the idea of feature map formation can be stated as follows:

> The **spatial location** of an output neuron in the topographic map corresponds to a particular domain, or feature of the input data.

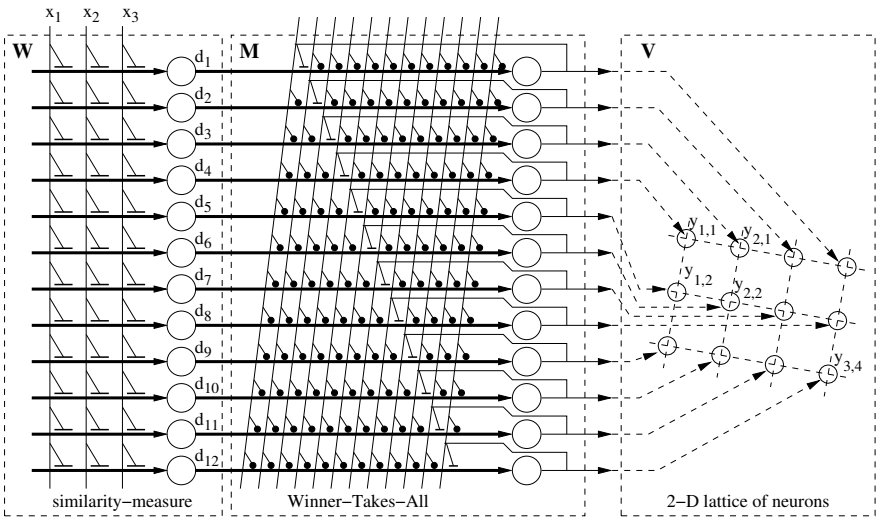
More specifically:

> Self-Organizing Feature maps are competitive neural networks in which neurons are organized in an l-dimensional lattice (grid) representing the **feature space**

The output lattice characterizes a relative position of neurons with regards to its neighbours, that is their topological properties rather than exact geometric locations.

In practice, dimensionality of the feature space is often restricted by its its visualisation aspect and typically is $l = 1, 2$ or 3.

Example of a self-organizing feature map in which the **input space** is 3-dimensional $(p = 3)$ and **feature space** is 2-dimensional $(l = 2)$. There are 12 neurons organized on a 3×4 grid, $m = [3\ \ 4]$.

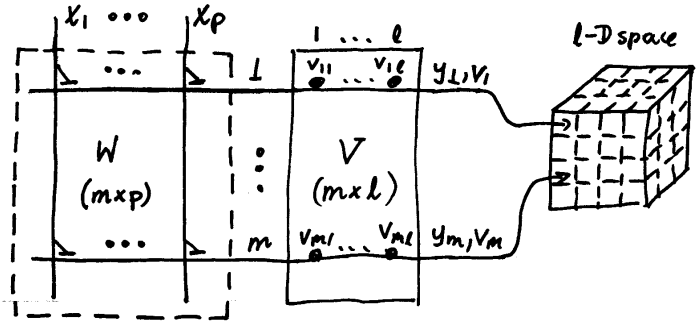


Each neuron, $y_{\mathbf{v}}$ in the above SOFM is characterized by its position in the lattice specified by a 2-D vector $\mathbf{v} = [v_1\ v_2]$, and by a 3-D weight vector $\mathbf{w_v} = [w_{1\mathbf{v}}\ w_{2\mathbf{v}}\ w_{3\mathbf{v}}]$.

SOFM, as a competitive neural network, consists of a distance-measure layer and a competitive layer which implements the MinNet algorithm through the lateral inhibitive and local self-excitatory connections.

During the competition phase (the MinNet), the winner is selected from all neurons in the lattice.

A general structure of a Self-Organizing Feature Map can be presented in the following way:



and can be characterized by the following parameters:

- p — dimensionality of the **input space**
- l — dimensionality of the **neuronal space**
- m — the total number of neurons
- W — $m \times p$ matrix of synaptic weights
- V — $m \times l$ matrix of topological positions of neurons

In subsequent considerations neurons will be identified either by their index $k = 1, \ldots, m$, or by their position vector $\mathbf{v}_k = V(k, :)$ on the neuronal grid, that is, in the feature space.

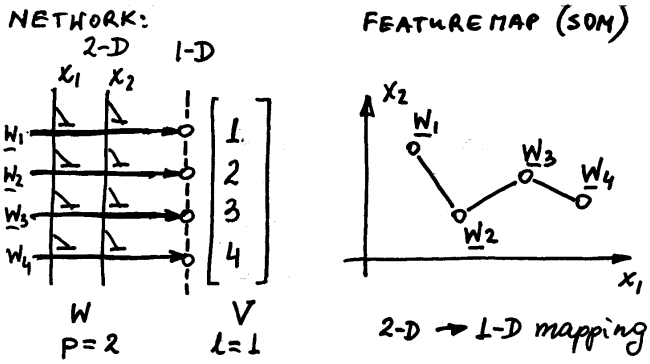
It can be observed that a SOM performs **mapping** from a p-dimensional **input space** to an l-dimensional **neuronal space**.

8.1 Feature Maps

A **Feature Map** aka **Self-Organizing Map** is a plot of synaptic **weights** in the **input space** in which weights of the neighbouring neurons are joined by lines or plane segments (patches).

Example: 2-D input space, 1-D feature space

Consider a SOM neural network with two inputs ($p = 2$) and m outputs organized in a 1-D feature space:



Neurons are organized along an elastic string, and

a feature map describes the mapping from a 2-D input space into a 1-D neuronal space.

Figure 8–1: A general structure of a (2-D,1-D) SOM and a feature map for a (2-D,1-D) SOM

Note that in the feature map the point representing the weight vector, $\mathbf{w}_k$, is joined by line segments with points representing weights $\mathbf{w}_{k-1}$ and $\mathbf{w}_{k+1}$ an so no because neurons $k-1$, k, and $k+1$ are located in the adjacent positions of the 1-D neurona lattice.

Example: 2-D input space, 2-D feature space

Let us consider a SOFM with two inputs $(p = 2)$ and m neurons arranged in a 2-D lattice as in Figure 8–2.

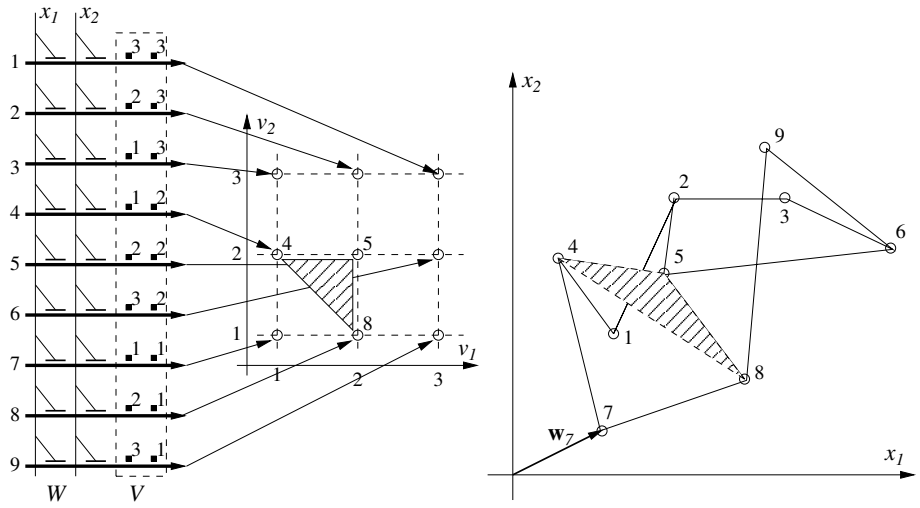
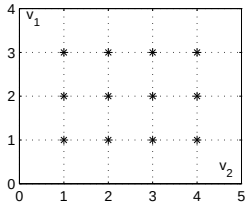


Figure 8–2: A general structure of a (2-D,2-D) SOM and an example of a feature map describing mapping from a 2-D input space into a 2-D neuronal space

Consider a neuron #5 located at the central vertex of the 3×3 neuronal lattice. The neuron has four neighbours: #4, #6, and #2, #8. Therefore, in the feature maps the nodes $\mathbf{w}_4, \mathbf{w}_6, \mathbf{w}_2, \mathbf{w}_8$ will all be joint with a line to the node $\mathbf{w}_5$. In addition, we can map triangular patches as shown in the Figure 8–2.

Plotting (2-D,2-D) feature maps with MATLAB

Consider a (2-D,2-D) SOM with $p = 2$ inputs and $m = 12$ organized on a 3×4 neuronal lattice:



The following MATLAB code can be used to generate an example of the weight and position matrices:

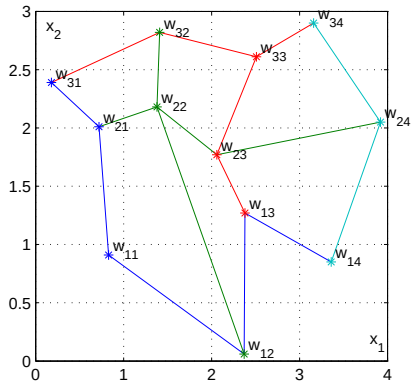
```
% SOM22.m
%           6 May 2005
% Plotting a 2-D Feature map in a 2-D input space

clear, close all
m = [3 4]; mm = prod(m) ; % p = 2 ;
%  formation of the neuronal position matrix
[V2, V1] = meshgrid(1:m(2), 1:m(1)) ;
V = [V1(:), V2(:)] ;
% Example of a weight matrix
% W = V-1.4*rand(mm, 2) ;
W =[0.83  0.91; 0.72  2.01;  0.18  2.39; 2.37  0.06;
    1.38  2.18; 1.41  2.82;  2.38  1.27; 2.06  1.77;
    2.51   .61; 3.36  0.85;  3.92  2.05; 3.16  2.90  ] ;
[W V]
```

The resulting W and V matrices can be as follows:

k	W		V	
1	0.83	0.91	1	1
2	0.72	2.01	2	1
3	0.18	2.39	3	1
4	2.37	0.06	1	2
5	1.38	2.18	2	2
6	1.41	2.82	3	2
7	2.38	1.27	1	3
8	2.06	1.77	2	3
9	2.51	2.61	3	3
10	3.36	0.85	1	4
11	3.92	2.05	2	4
12	3.16	2.90	3	4

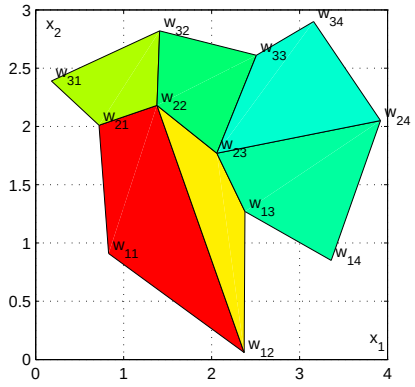
```
figure(1)
% Plotting a feature map: grid method
FM1 = full(sparse(V(:,1), V(:,2), W(:,1))) ;
FM2 = full(sparse(V(:,1), V(:,2), W(:,2))) ;
FM = FM1+j*FM2;
plot(FM), hold on, plot(FM.'), plot(FM, '*'), hold off
grid on
% the following section marks coordinates of each neuron
tt = 'w_{11}' ; tt = tt(ones(mm,1),:) ;
tt(:,4:5) = [num2str(V(:,1),1) num2str(V(:,2),1)] ;
text(W(:,1)+0.05, W(:,2)+0.05, tt) ;
axs = axis ;
text(0.95*axs(2), 0.04*axs(2), 'x_1');
text(0.04*axs(4), 0.95*axs(4), 'x_2') ;
% print -f1 -depsc2 hSOM22g
```

The grid line version:




```
figure(2)
% Plotting a feature map: colur patch method
pcolor(FM1, FM2, (FM1+FM2)) ;
axis([0 4 0 3])
grid on, colormap(hsv)
% the following section marks coordinates of each neuron
tt = 'w_{11}' ; tt = tt(ones(mm,1),:) ;
tt(:,4:5) = [num2str(V(:,1),1) num2str(V(:,2),1)] ;
text(W(:,1)+0.05, W(:,2)+0.05, tt) ;
axs = axis ;
text(0.95*axs(2), 0.04*axs(2), 'x_1');
text(0.04*axs(4), 0.95*axs(4), 'x_2') ;
% print -f2 -depsc2 hSOM22p
```

The colour patches version:



8.2 Learning Algorithm for Self-Organizing Feature Maps

The objective of the learning algorithm for the SOFM neural networks is formation of the feature map which captures of the essential characteristics of the p-dimensional input data and maps them on the typically 1-D or 2-D feature space.
The learning algorithm captures two essential aspects of the map formation, namely, **competition** and **cooperation** between neurons of the output lattice.

Competition is implemented as in competitive learning: each input vector $\mathbf{x}(n)$ is compared with each weight vector from the weight matrix W and the position $V(k(n),:)$ of the winning neuron $k(n)$ is established. For the winning neuron the distance

$$|\mathbf{x}^T(n) - W(k(n),:)|$$

attains minimum.

Cooperation All neurons located in a topological neighbourhood of the winning neurons $k(n)$ will have their weights updated usually with a strength $\Lambda(j)$ related to their distance $\rho(j)$ from the winning neuron,

$$\rho(j) = |V(j,:) - V(k(n),:)| \quad \text{for} \quad j = 1,\ldots,m.$$

The **neighbourhood function**, $\Lambda(j)$, is usually an l-dimensional Gausssian function:

$$\Lambda(j) = \exp(-\frac{\rho^2(j)}{2\sigma^2})$$

where σ^2 is the variance parameter specifying the spread of the Gaussian function.

Example of a 2-D Gaussian neighbourhood function for a 40×30 neuronal lattice is given in Figure 8–3.

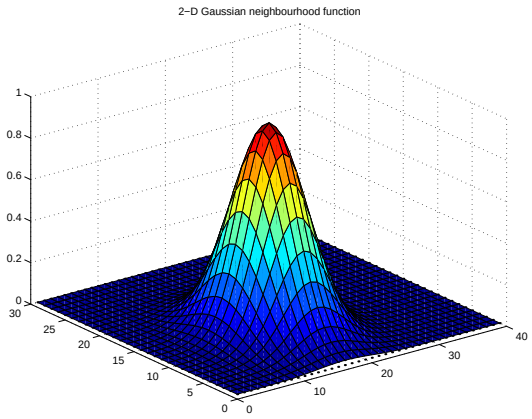


Figure 8–3: 2-D Gaussian neighbourhood function

Feature map formation is critically dependent on the learning parameters, namely, the learning gain, η, and the spread of the neighbourhood function specified for the Gaussian case by the variance, σ^2. In general, both parameters should be time-varying, but their values are selected experimentally.

Usually, the **learning gain** should stay close to unity during the **ordering phase** of the algorithm which can last for, say, 1000 iteration. After that, during the **convergence phase**, should be reduced to reach the value of, say, 0.1.

The **spread** of the neighbourhood function should initially include all neurons for any winning neuron and during the ordering phase should be slowly reduced to eventually include only a few neurons in the winner's neighbourhood. During the convergence phase, the neighbourhood function should include only the winning neuron.

The complete SOFM learning algorithm

The complete algorithm can be described as consisting of the following steps

1. Initialise:
   (a) the weight matrix W with a random sample of m input vectors.
   (b) the learning gain and the spread of the neighbourhood function.
2. for every input vector, $\mathbf{x}(n), \quad n = 1, \ldots, N$:
   (a) Determine the winning neuron, $k(n)$, and its position $V(k,:)$ as

$$k(n) = \arg\min_j |\mathbf{x}^T(n) - W(j,:)|$$

   (b) Calculate the neighbourhood function

$$\Lambda(n,j) = \exp(-\frac{\rho^2(j)}{2\sigma^2})$$

   where

$$\rho(j) = |V(j,:) - V(k(n),:)| \quad \text{for} \quad j = 1, \ldots, m.$$

(c) Update the weight matrix as

$$\Delta W = \eta(n) \cdot \Lambda(n) \cdot (\mathbf{x}^T(n) - W(j,:))$$

All neurons (unlike in the simple competitive learning) have their weights modified with a strength proportional to the neighbourhood function and to the distance of their weight vector from the current input vector (as in competitive learning).

(d) During the ordering phase, shrink the neighbourhood until it includes only one neuron:

$$\sigma(n+1) = \sigma(n) \cdot \delta\sigma$$

(e) During the convergence phase, "cool down" the learning process by reducing the learning gain:

$$\eta(n+1) = \eta(n) \cdot \delta\eta$$

8.3 A demo script sofm.m

A MATLAB script, sofm.m, can be used to study the behaviour of the Kohonen learning algorithm which creates self-organizing feature maps. A process of generation an example of 1-D and 2-D feature maps using the sofm.m script is illustrated in Figures 8–4 and 8–5, respectively.

The first plot in Figure 8–4 represents a 2-D input space in which a uniformly distributed points form a letter 'A'. Subsequent plots illustrate the feature space from its initial to final form which is attained after one pass through the training data. Neurons are organized in a 1-D lattice, their 2-D weight vectors forming an elastic string which approximates two dimensional object 'A'.

Similarly, the plots in Figure 8–5 represent formation of a 2-D feature map approximating a 2-D triangle from the input space.

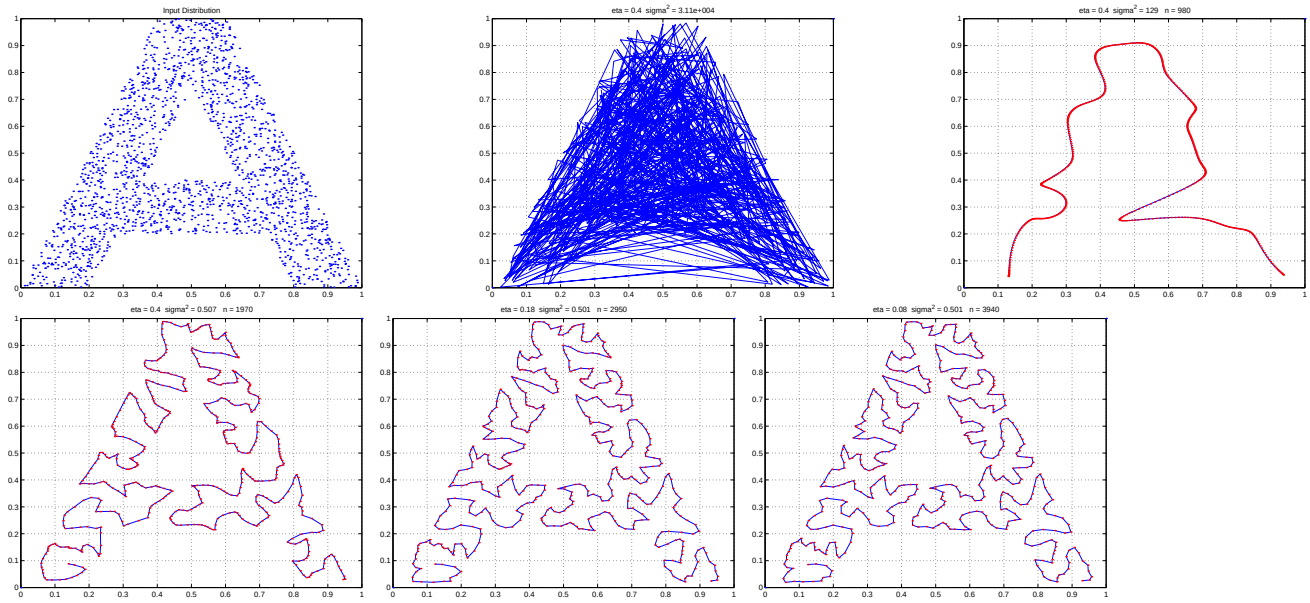


Figure 8–4: A 1-D Self-Organizing Feature Map

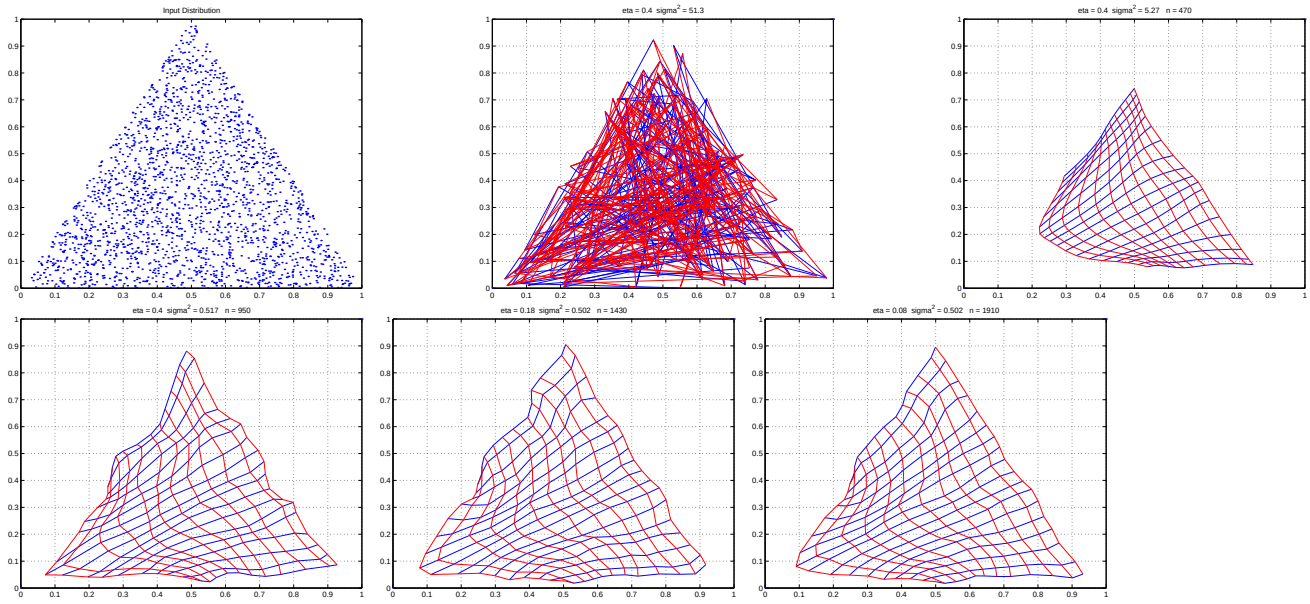


Figure 8–5: A 2-D Self-Organizing Feature Map