# Image Processing Using Neural Networks

A Dissertation Submitted in Partial fulfillment for

the requirement of the award of Degree of

**MASTER OF ENGINEERING**

**in**

**COMPUTER TECHNOLOGY AND APPLICATIONS**

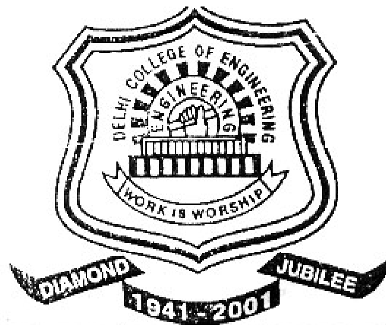
By:


**YITAGESSU BIRHANU**

**31/CTA/O3**

**Under the guidance of**

**Dr. S.K. Saxena**



**DEPARTMENT OF COMPUTER ENGINEERING**

**DELHI COLLEGE OF ENGINEERING**

**UNIVERSITY OF DELHI-110042**


**2003-2005**

# *DECLARATION BY THE CANDIDATE*

I hereby certify that the work which is being presented in the dissertation entitled "**Image Processing Using Neural Networks**", in the partial fulfillment of the requirements for the award of the degree of Master of Engineering in Computer Technology and Applications, is an authentic record of my work carried out under the supervision of **Dr. S.K. Saxena**. I have not submitted this work anywhere else for the award of any other degree.


(Yitagessu Birhanu)

M.E. (CTA)

Department of Computer Engineering

DCE, Delhi-1100042

# CERTIFICATE

This is to certify that declaration made by the candidate is correct to the best of my knowledge and belief. This is to certify that the project entitled"**Image Processing Using Neural Networks"** is an authentic record of candidate's own work carried out by him under my guidance and supervision .He has not submitted this work for the award of any other degree.


Dr. S.K. Saxena
Project Guide
Department of Computer Engineering
DCE, Delhi-110042

Dr. D. Roy Choudhury
Head
Department of Computer Engineering
DCE, Delhi-110042

# ACKNOWLEDGEMENTS

I am delighted to express my heartily and sincere gratitude and indebtedness to Dr. S.K. Saxena, Lecturer, Computer Engineering Department, Delhi College of Engineering, Delhi, for his invaluable guidance and wholehearted cooperation. His continuous inspiration only has made me complete this project.

I am greatly thankful to Prof. D. Roy Choudhary, Head of Department, and Dr. Goldie Gabrani, Project Coordinator, for their support in providing resource. My heartily thanks to all professors for their expertise and all rounded personality they have imparted to me.

(YITAGESSU BIRHANU)

Table of Contents


1 Introduction. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

2 Artificial neural networks. . . . . . . . . . . . . . . . . . . . . . . . . . 2

2.1 What is a neural network? . . . . . . . . . . . . . . . . . . . . . . . . 2

2.2 Historical background. . . . . . . . . . . . . . . . . . . . . . . . . . . 2

2.3 Why use neural networks? . . . . . . . . . . . . . . . . . . . . . . . . 3

2.4 Neural networks versus conventional computers. . . . . . . . . . . . 5

3 Artificial neurons. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

3.1 Models of a neuron. . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

3.2 Types of activation function. . . . . . . . . . . . . . . . . . . . . . . 8

3.2.1 Threshold function. . . . . . . . . . . . . . . . . . . . . . . . . 9

3.2.2 Piecewise-linear function. . . . . . . . . . . . . . . . . . . . . 9

3.2.3 Sigmoid function. . . . . . . . . . . . . . . . . . . . . . . . . . 11

3.3 Stochastic model of a neuron. . . . . . . . . . . . . . . . . . . . . . . 12

4 Architecture of neural networks. . . . . . . . . . . . . . . . . . . . . . 12

4.1 Single-layer feedforward networks. . . . . . . . . . . . . . . . . . . 13

4.2 Multilayer feedforward networks. . . . . . . . . . . . . . . . . . . . 13

4.3 Recurrent networks. . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

5 The learning processes. . . . . . . . . . . . . . . . . . . . . . . . . . . 17

5.1 Error-correction learning. . . . . . . . . . . . . . . . . . . . . . . . . 18

5.2 Memory-based learning. . . . . . . . . . . . . . . . . . . . . . . . . . 21

5.3 Hebbian learning. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

5.3.1 Synaptic enhancement and depression. . . . . . . . . . . . . . . 24

5.3.2 Mathematical models of Hebbian functions. . . . . . . . . . . . 25

5.4 Competitive learning. . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

5.5 Bolzmann learning. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

6 Training algorithms. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

6.1 The error back-propagation algorithm. . . . . . . . . . . . . . . . . . . 30

6.1.1 The two passes of computation. . . . . . . . . . . . . . . . . . 38

6.1.2 Rate of learning. . . . . . . . . . . . . . . . . . . . . . . . . 39

6.2 Method of conjugate gradients. . . . . . . . . . . . . . . . . . . . . 41

7 Digital image processing. . . . . . . . . . . . . . . . . . . . . . . . . . 46

7.1 Taxonomy for image processing algorithms. . . . . . . . . . . . . 46

7.2 Exploring the image processing chain. . . . . . . . . . . . . . . . . 48

7.2.1 Preprocessing. . . . . . . . . . . . . . . . . . . . . . . . . . 49

7.2.1.1 Image reconstruction. . . . . . . . . . . . . . . . . . . 49

7.2.1.2 Image restoration. . . . . . . . . . . . . . . . . . . . . 49

7.2.1.3 3 ANNs in preprocessing. . . . . . . . . . . . . . . . . 50

7.2.2 Data reduction and feature extraction. . . . . . . . . . . . . . . 51

7.2.2.1 Image compression applications. . . . . . . . . . . . . . 52

7.2.2.2 Feature extraction applications. . . . . . . . . . . . . . 52

7.2.3 Image segmentation. . . . . . . . . . . . . . . . . . . . . . . . 54

7.2.3.1 Segmentation based on pixel data. . . . . . . . . . . 54

7.2.3.2 Segmentation based on features. . . . . . . . . . . . 55

7.2.3.3 Open issues in segmentation by ANNs. . . . . . . . . 56

7.2.4 Object recognition. . . . . . . . . . . . . . . . . . . . . . . . 57

7.2.4.1 Based on pixel data. . . . . . . . . . . . . . . . . . . . 57

7.2.4.2 Based on features. . . . . . . . . . . . . . . . . . . . . 59

7.2.4.3 Using pixels or features as input? . . . . . . . . . . . . 60

7.2.4.4 Issues in pattern recognition. . . . . . . . . . . . . . . 62

7.2.4.5 Obstacles for PR in DIP. . . . . . . . . . . . . . . . . . . . 63

7.2.5 Image understanding. . . . . . . . . . . . . . . . . . . . . . . . . 64

7.2.6 Optimization. . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

8 Edge recognition in shared weight networks. . . . . . . . . . . . . . . . . 66

8.1 Network architecture. . . . . . . . . . . . . . . . . . . . . . . . . . 67

8.2 Training. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

8.3 Results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

8.4 Conclusion. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

9 Appendix. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

10 Bibliography. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109

## Abstract

Artificial neural networks (ANNs) are very general function approximators, which can be trained based on a set of examples. Given their general nature, ANNs would seem useful tools for nonlinear image processing. This paper explores the application of neural networks in digital image processing. Particularly, it details the design and implementation of a neural network based edge recognizer, and studies the effects of incorporating prior knowledge in its design. After a brief introduction to ANNs, ANN training algorithms, and digital image processing, the paper explains the design of a neural network based edge-recognizer and its training using the conjugate gradient descent (CGD) training algorithm. When the network is trained using the aforementioned training algorithm, it was observed that ANNs can be used as edge detectors. However, the presence of receptive fields in the architecture in itself does not guarantee that shift-invariant feature detectors will be found. The appendix at the end of the report contains the implementation source code.

# 1. Introduction

Image processing is the field of research concerned with the development of computer algorithms working on digitized images (e.g. Pratt, 1991; Gonzalez and Woods, 1992). The range of problems studied in image processing is large, encompassing everything from low-level signal enhancement to high-level image understanding. In general, image processing problems are solved by a chain of tasks. This chain outlines the possible processing needed from the initial sensor data to the outcome (e.g. a classification or a scene description). The pipeline consists of the steps of pre-processing, data reduction, segmentation, object recognition and image understanding. In each step, the input and output data can either be images (pixels), measurements in images (features), and decisions made in previous stages of the chain (labels) or even object relation information (graphs).

There are many problems in image processing for which good, theoretically justifiable solutions exists, especially for problems for which linear solutions suffice. For example, for pre-processing operations such as image restoration, methods from signal processing such as the Wiener filter can be shown to be the optimal linear approach. However, these solutions often only work under ideal circumstances; they may be highly computationally intensive (e.g. when large numbers of linear models have to be applied to approximate a nonlinear model); or they may require careful tuning of parameters. Where linear models are no longer sufficient, nonlinear models will have to be used. This is still an area of active research, as each problem will require specific nonlinearities to be introduced. That is, a designer of an algorithm will have to weigh the different criteria and come to a good choice, based partly on experience. Furthermore, many algorithms quickly become intractable when nonlinearities are introduced. Problems further in the image processing chain, such object recognition and image understanding, cannot even (yet) be solved using standard techniques. For example, the task of recognizing any of a number of objects against an arbitrary background calls for human capabilities such as the ability to generalize, associate etc. All this leads to the idea that nonlinear algorithms that can be trained, rather than designed, might be valuable tools for image processing. To explain why, a brief introduction into artificial neural networks will be given first.

# 2. Artificial neural networks

## 2.1 What is a neural network?

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurons) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurons. This is true of ANNs as well.

## 2.2 Historical background

Neural network simulations appear to be a recent development. However, this field was established before the advent of computers, and has survived at least one major setback and several eras. Many important advances have been boosted by the use of inexpensive computer emulations. Following an initial period of enthusiasm, the field survived a period of frustration and disrepute. During this period when funding and professional support was minimal, important advances were made by relatively few researchers. These pioneers were able to develop convincing technology which surpassed the limitations identified by Minsky and Papert. Minsky and Papert, published a book (in 1969) in which they summed up a general feeling of frustration (against neural networks) among researchers, and was thus accepted by most without further analysis. Currently, the neural network field enjoys a resurgence of interest and a corresponding increase in funding.

The first artificial neuron was produced in 1943 by the neurophysiologist Warren McCulloch and the logician Walter Pitts. But the technology available at that time did not allow them to do too much.

## 2.3 Why use neural networks?

Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. A trained neural network can be thought of as an "expert" in the category of information it has been given to analyze. This expert can then be used to provide projections given new situations of interest and answer “what if” questions. Other advantages include:

1. Adaptive learning: Neural networks have a built-in capability to adapt their synaptic weights to changes in the surrounding environment. In particular, a neural network trained to operate in a specific environment can be easily retrained to deal with minor changes in the operating environmental conditions. Moreover, when it is operating in a *nonstationary* environment (i.e., one where statistics change with time) a neural network can be designed to change its synaptic weights in real time. The natural architecture of a neural network for pattern classification, signal processing, and control applications, coupled with the adaptive capability of the network, make it a useful tool in adaptive pattern classification, adaptive signal processing, and adaptive control. As a general rule, it may be said that the more adaptive we make a system, all the time ensuring that the system remains stable, the more robust its performance will likely be when the system is required to operate in a nonstationary environment. It should be emphasized, however, that adaptability does not always lead to robustness; indeed, it may do the very opposite.
2. Self-Organization: An ANN can create its own organization or representation of the information it receives during learning time.
3. Real Time Operation: ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which take advantage of this capability.

4. Fault Tolerance via Redundant Information Coding: Partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage.
5. Nonlinearity: An artificial neuron can be linear or nonlinear. A neural network, made up of an interconnection of nonlinear neurons, is itself nonlinear. Moreover, the nonlinearity is of a special kind in the sense that it is distributed throughout the network. Nonlinearity is a highly important property, particularly if the underlying physical mechanism responsible for generation of the input signal is inherently nonlinear.
6. Input-Output mapping: A popular paradigm of learning called *learning with a teacher or supervised learning* involves modification of the synaptic weights of a neural network by applying a set of labeled *training samples* or *task examples*. Each example consists of a unique *input signal* and a corresponding *desired response*. The network is presented with an example picked at random from the set, and the synaptic weights (free parameters) of the network are modified to minimize the difference between the desired response and the actual response of the network produced by the input signal in accordance with an appropriate statistical criterion. The training of the network is repeated for many examples in the set until the network reaches a steady state where there are no further significant changes in the synaptic weights. The previously applied training examples may be reapplied during the training session but in different order. Thus the network learns form the examples by constructing an *input-output mapping* for the problem at hand.
7. VLSI implementability: the massively parallel nature of a neural network makes it potentially fast for the computation of certain tasks. This same feature makes a neural network well suited for implementation using very-large-scale-integrated (VLSI) technology. One particular beneficial virtue of VLSI is that it provides a means of capturing truly complex behavior in a highly hierarchical fashion.
8. Uniformity of analysis and design: basically, neural networks enjoy universality as information processors. We say this in the sense that the same

notation is used in all domains involving the application of neural networks. This feature manifests itself in different ways:

- Neurons, in one form or another, represent an ingredient *common* to all neural networks;
- This commonality makes it possible to *share* theories and learning algorithms in different applications of neural networks;
- Modular networks can be built through seamless *integration of modules.*

9. Evidential response: in the context of pattern classification, a neural network can be designed to provide information not only about which particular pattern to *select,* but also about the confidence in the decision made. This later information may be used to reject ambiguous patterns, should they arise, and thereby improve the classification performance of the network.

10. Contextual information: knowledge is represented by the very structure activation state of a neural network. Every neuron in the network is potentially affected by the global activity of all other neurons in the network. Consequently, contextual information is dealt with naturally by a neural network.

## 2.4 Neural networks versus conventional computers

Neural networks take a different approach to problem solving than that of conventional computers. Conventional computers use an algorithmic approach i.e. the computer follows a set of instructions in order to solve a problem. Unless the specific steps that the computer needs to follow are known the computer cannot solve the problem. That restricts the problem solving capability of conventional computers to problems that we already understand and know how to solve. But computers would be so much more useful if they could things that we don't exactly know how to do. .

Neural networks process information in a similar way the human brain does. The network is composed of a large number of highly interconnected processing elements (neurons) working in parallel to solve a specific problem. Neural networks learn by

example. They cannot be programmed to perform a specific task. The examples must be selected carefully otherwise useful time is wasted or even worse the network might be functioning incorrectly. The disadvantage is that because the network finds out how to solve the problems by itself, its operation can be unpredictable. .

On the other hand, conventional computers use a cognitive approach to problem solving; the way the problem is to be solved must be known and stated in small unambiguous instructions. These instructions are then converted to a high level language program and then into machine code that the computer can understand. These machines are totally predictable; if anything goes wrong is due to a software or hardware fault. Neural networks and conventional algorithmic computers are not in competition but complement each other. There are tasks are more suited to an algorithmic approach like arithmetic operations and tasks that are more suited to neural networks. Even more, a large number of tasks, require systems that use a combination of the two approaches (normally a conventional computer is used to supervise the neural network) in order to perform at maximum efficiency. *Neural networks do not perform miracles. But if used sensibly they can produce some amazing results.*

## 3. Artificial neurons

### 3.1 Models of a neuron

A neuron is an information processing unit that is fundamental to the operation of a neural network. The block diagram of figure 4 shows the *model* of a neuron, which forms the basis for designing (artificial) neural networks. Here we identify three basic elements of the neuronal model:

1. A set of *synapses* or *connecting links*, each of which is characterized by a *weight* or s*trength* of its own. Specifically, a signal $x_j$ at the input of synapse j connected to neuron k is multiplied by the synaptic weight $w_{kj}$. Unlike a synapse in the brain, the synaptic weight of an artificial neuron may lie in a range that includes negative as well as positive values.
2. An *adder* for summing the input signal, weighted by the respective synapses of the neuron; the operations described here constitutes a *linear combiner*.

3. An *activation function* for limiting the amplitude of the output of a neuron. The activation function is also referred to as a *squashing function* in that it squashes (limits) the permissible amplitude range of the output signal to some finite value.

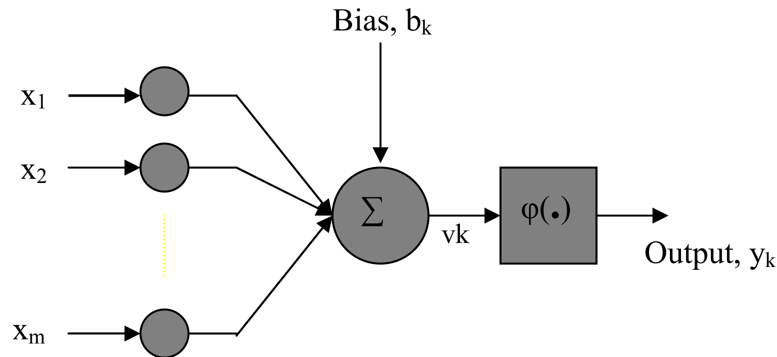



Figure 3.1: Nonlinear model of a neuron

The neuronal model of figure 3.1 also includes an externally applied *bias*, denoted by $b_k$. The bias $b_k$ has the effect of increasing or lowering the net input of the activation function, depending on whether it is positive or negative, respectively. In mathematical terms, we

$$u_k = \sum_{j=1}^{m} w_{kj}\, x_j \qquad 3.1$$

and

$$y_k = \varphi(u_k + b_k) \qquad 3.2$$

where $x_1$, $x_2$, …, $x_m$ are the input signals; $w_{k1}$, $w_{k2}$, …, $w_{km}$ are the synaptic weights of neuron k; $u_k$ is the *linear combiner output* due to the input signals; $b_k$ is the bias; $\varphi(.)$ is the *activation function*, and $y_k$ is the output signal of the neuron. The use of bias $b_k$ has an affine transformation to the output $u_k$ of the linear combiner in the model of figure 4, as shown

$$v_k = u_k + b_k \qquad 3.3$$

In particular, depending on the whether the bias $b_k$ is positive or negative, the relationship between the induced local field or activation function $v_k$ of neuron k and the linear

combiner output $u_k$ is modified in the manner illustrated in the figure 3.2. Note that as a result of this affine transformation, the graph $v_k$ versus $u_k$ no longer passes the origin.

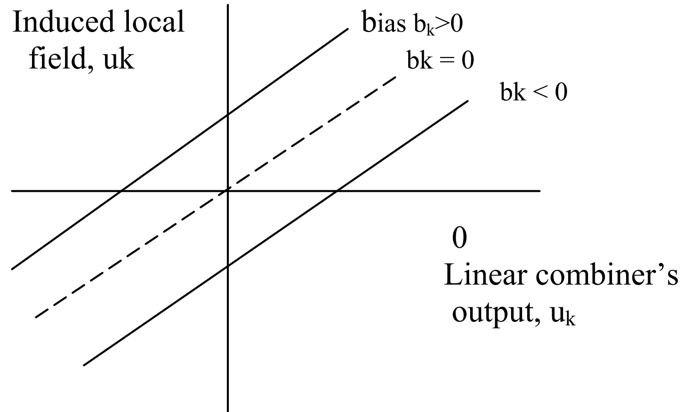



Figure 3.2: affine transformation produced by the presence of a bias; note that $v_k = b_k$ at $u_k = 0$.

The bias $b_k$ is an external parameter of artificial neuron k. We may account for its presence as in Eq. (3.2). Equivalently, we may formulate the combination of Eqs. (3.1) to (3.3) as follows:

$$v_k = \sum_{j=0}^{m} w_{kj} x_j \qquad 3.4$$

and

$$y_k = \varphi(v_k) \qquad 3.5$$

In Eq. (3.4) we have added a new synapse. Its input is

$$x_0 = +1 \qquad 3.6$$

and its weight is

$$w_{k0} = b_k \qquad 3.7$$

## 3.2 Types of activation function

The activation function, denoted by $\varphi(v)$, defines the output of a neuron in terms of the induced local field v. Here we identify three basic types of activation function:

### 3.2.1 Threshold function

For this type of activation function, described in figure 3.3a, we have

$$\varphi(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{if } v < 0 \end{cases} \qquad 3.8$$

In engineering literature this form of a threshold function is commonly referred to as a *Heaviside* function. Correspondingly, the output of neuron k employing such a threshold function is expressed as

$$y_k = \begin{cases} 1 & \text{if } v_k \geq 0 \\ 0 & \text{if } v_k < 0 \end{cases} \qquad 3.9$$

where $v_k$ is the induced local field of the neuron; that is ,

$$v_k = \sum_{j=1}^{m} w_{kj}\, x_j + b_k \qquad 3.10$$

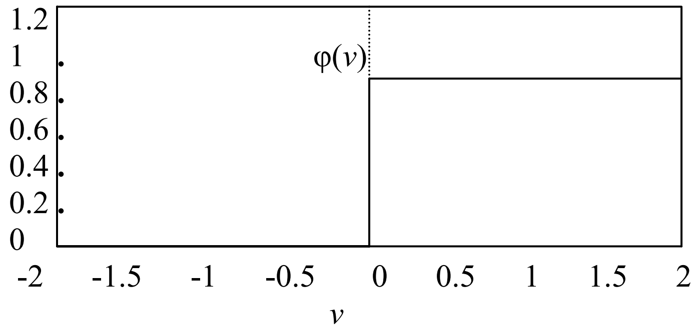
Such a neuron is referred to in literatures as the *McCulloch-Pitts model*, in recognition of the pioneering work done by McCulloch and Pitts (1943). In this model, the output of a neuron takes on the value of 1 if the induced local field of that neuron is nonnegative and 0 otherwise. This statement describes the *all-or-none property* of the McCulloch-Pitts model.

### 3.2.2 Piecewise-linear function

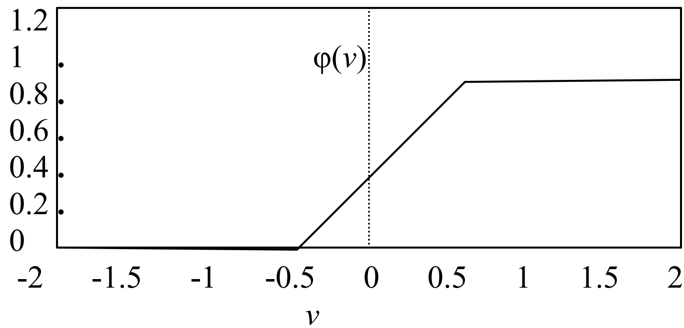
For this piecewise-linear function described in fig. 3.3b, we have

$$\varphi(v) = \begin{cases} 1, & v \geq +\tfrac{1}{2} \\ v, & +\tfrac{1}{2} > v > -\tfrac{1}{2} \\ 0, & v \leq -\tfrac{1}{2} \end{cases} \qquad 3.11$$

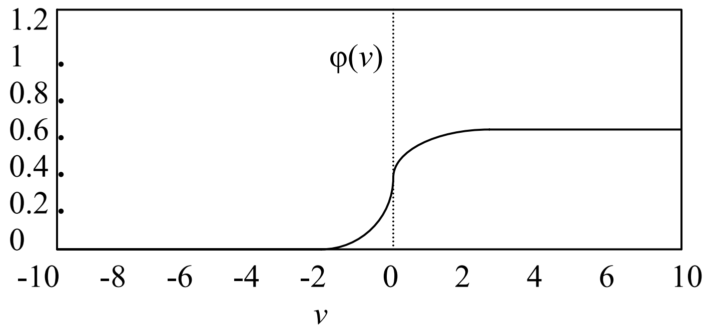
- where the amplification factor inside the linear region of operation is assumed to be unity. This form of an activation function may be viewed as an *approximation* to a non- linear amplifier.




(a)




(b)




(c)

Figure 3.3: (a) Threshold function. (b) piecewise-linear function. (c) Sigmoid function.

The following two situations may be viewed as special forms of the piecewise-linear function:

- A linear combiner arises if the linear region of operation is maintained without running into saturation;

- The piecewise linear function reduces to a *threshold function* if the amplification of the linear region is made infinitely large.

### 3.2.3 Sigmoid function

The Sigmoid function, whose graph is s-shaped, is by far the most common form of activation function used in the construction of artificial neural networks. It is defined as a strictly increasing function that exhibits a graceful balance between linear and nonlinear behavior. An example of the sigmoid function is the logistic function, defined

$$\varphi(v) = 1 / (1 + \exp(-av)) \qquad 3.12$$

where $a$ is the *slope parameter* of the sigmoid function. By varying the parameter a, we obtain sigmoid functions of different slopes. In the limit, as the slope parameter approaches infinity, the sigmoid function becomes simply a threshold function. Whereas a threshold function assumes the value of 0 or 1, a sigmoid function assumes a continuous range of values from 0 to 1. Note also that the sigmoid function is differentiable, whereas the threshold function is not. Differentiability is an important feature of neural network theory. The activation functions defined in Eqs. (3.8), (3.11), and (3.12) range from 0 to +1. It is sometimes desirable to have the activation function range from -1 to +1, in which case the activation function assumes an antisymmetric form with respect to the origin; that is, the activation function is an odd function of the induced local field. Specifically, the threshold function of Eq. (3.8) is now defined as

$$\varphi(v) = \begin{cases} 1 & \text{if } v > 0 \\ 0 & \text{if } v = 0 \\ -1 & \text{if } v < 0 \end{cases} \qquad 3.13$$

which is commonly referred to as the *signum function.* For the corresponding form of a sigmoid function we may use the *hyperbolic tangent function,* defined by

$$\varphi(v) = \tanh(v) \qquad 3.14$$

Allowing an activation function of the sigmoid type to assume negative values as prescribed by Eq. (3.14) has analytic benefits.

## 3.3 Stochastic model of a neuron

The neuronal model of Fig 3.1 is deterministic in that its input-output behavior is precisely defined for all inputs. For some applications of neural network, it desirable to base the analysis on a stochastic neuronal model. In an analytically tractable approach, the activation function of the McCulloch-Pitts model is given a probabilistic interpretation. Specifically, a neuron is permitted to reside in only one of two states: +1 or -1, say. The decision for a neuron to *fire* (i.e., switch its state from "off" to "on") is probabilistic. Let x denote the state of a neuron, and $P(v)$ denote the *probability* of firing, where v is the induced local field of the neuron.

We may then write

$$x = \begin{cases} +1 & \text{with probability } P(v) \\ -1 & \text{with probability } 1 - P(v) \end{cases}$$

A standard choice for $P(v)$ is the sigmoid shaped function:

$$P(v) = 1/ ( 1 + \exp(-v/T) ) \qquad 3.15$$

where $T$ is a *pseudotemperature* that is used to control the noise level and therefore the uncertainty in firing. It is important to realize, however, that $T$ is not the physical temperature of s neural network, but is a biological or an artificial neural network. Rather, as already stated, we should think of $T$ merely as a parameter that controls the thermal fluctuations representing the effects of synaptic noise. Note that when $T \rightarrow 0$, the stochastic neuron described by Eq. (3.15) reduces to a noiseless (i.e., deterministic) form, namely the McCulloch-Pitts model.

# 4. Architecture of neural networks

The manner in which the neurons of a neural network are structured is intimately linked with the learning algorithm used to train the network. We may therefore speak of learning

algorithms (rules) used in the design of neural networks as being *structured*. In general, we may identify three fundamentally different classes of network architectures:

## 4.1 Single-layer feedforward networks

In a layered neural network the neurons are organized in the form of layers. In the simplest form of a layered network, we have an *input-layer* of source nodes that projects onto an *output layer* of neurons (computation nodes), but not vice versa. In other words, this network is strictly a *feedforward* or *acyclic* type. It is illustrated in Fig. 4.1 for the case of four nodes in both the input and output layers. Such a network is called is called a *single-layer-network,* with the designation "single-layer" referring to the output layer of computation nodes (neurons). We do not count the input layer of source nodes because no computation is performed there.

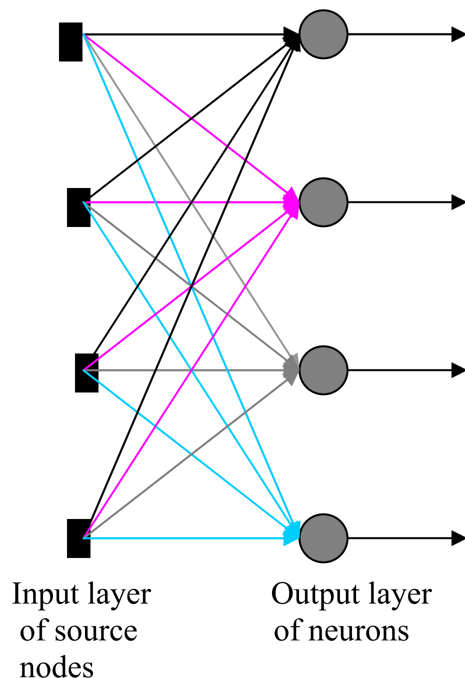



Fig. 4.1 Feedforward or acyclic network with a single layer of neurons

## 4.2 Multilayer feedforward networks

The second class of a feedforward neural network distinguishes itself by the presence of one or more *hidden layers*, whose computation nodes are correspondingly called *hidden*

*neurons* or *hidden units*. The function of hidden neurons is to intervene between the external input and the network output in some useful manner. By adding one or more hidden layers, the network is enabled to extract higher-order statistics. In a rather loose sense the network acquires a *global* perspective despite its local connectivity due to the extra set of synaptic connections and the extra dimension of neural interaction. The ability of hidden neurons to extract higher order statistics is particularly valuable when the size of the input layer is large.

The source nodes in the input layer of the network supply respective elements of the activation pattern (input vector), which constitute the input signals applied to the neurons (computation nodes) in the second layer (i.e., the first hidden layer). The output signals of the second layer are used as inputs to the third layer, and so on for the rest of the network. Typically the neurons in each layer of the network have as their inputs the output signals of the preceding layer only. The set of output signals of the neurons in the output (final) layer of the network constitute the overall response of the network to the activation pattern supplied by the source nodes in the input (first) layer. The architectural graph in Fig. 3.2 illustrates the layout of a multilayer feedforward neural network for the case of a single hidden layer. For brevity the network in Fig. 3.2 is referred to as a 10-4-2 network because it has 10 source nodes, 4 hidden neurons, and 2 output neurons. As another example, a feedforward network with $m$ source nodes, $h_1$ neurons in the first hidden layer, $h_2$ neurons in the second hidden layer, q neurons in the output layer is referred to as an $m$-$h_1$-$h_2$-$q$ network. The neural network in Fig. 4.2 is said to be *fully connected* in the sense that every node in each layer of the network is connected to every other node in the adjacent forward layer. If, however, some of the communication links (synaptic connections) are missing from the network, we say that the network is *partially connected*.

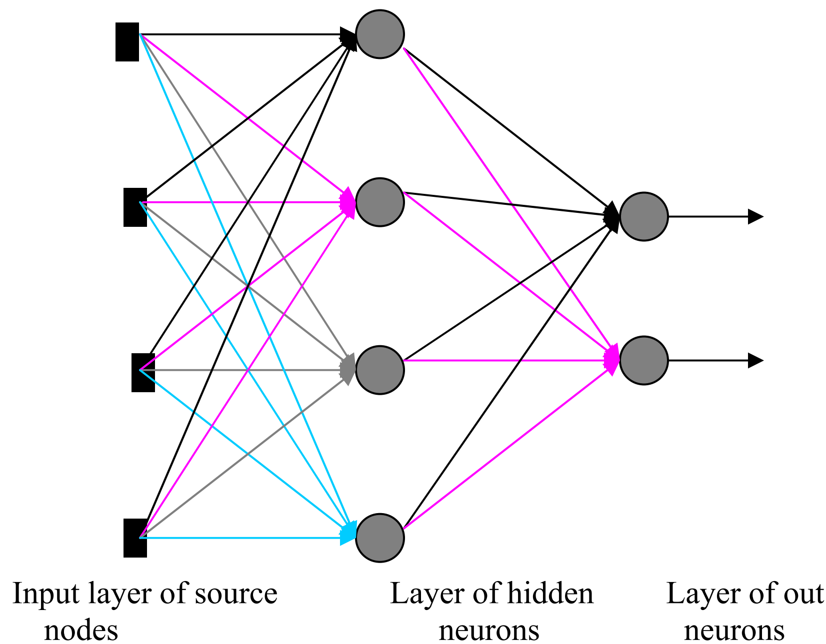



Fig. 4.2 fully connected feedforward network with one hidden layer and one output layer.

## 4.3 Recurrent networks

A *recurrent neural network* distinguishes itself from a feedforward neural network in that it has at least one feedback loop. For example, a recurrent network may consist of a single layer of neurons with each neuron feeding its output signal back to the inputs of all the other neurons, as illustrated in the architectural graph in Fig. 4.3. In the structure depicted in this figure there are no self-feedback loops in the network; self-feedback refers to a situation where the output of a neuron is fed back into its own input. The recurrent network illustrated in Fig. 4.3 also has no hidden neurons. In Fig. 4.4 we illustrate another class of recurrent networks with hidden neurons. The feedback connections shown in Fig. 4.4 originate from the hidden neurons as well as from the output neurons. The presence of feedback loops, whether in the recurrent structure of Fig. 4.3 or that of Fig. 4.4, has a profound impact on the learning capability of the network and on its performance. Moreover, the feedback loops involve the use of particular branches composed of *unit-delay element* (denoted by $z^{-1}$), which result in a nonlinear dynamical behavior, assuming that the neural network contains nonlinear units.

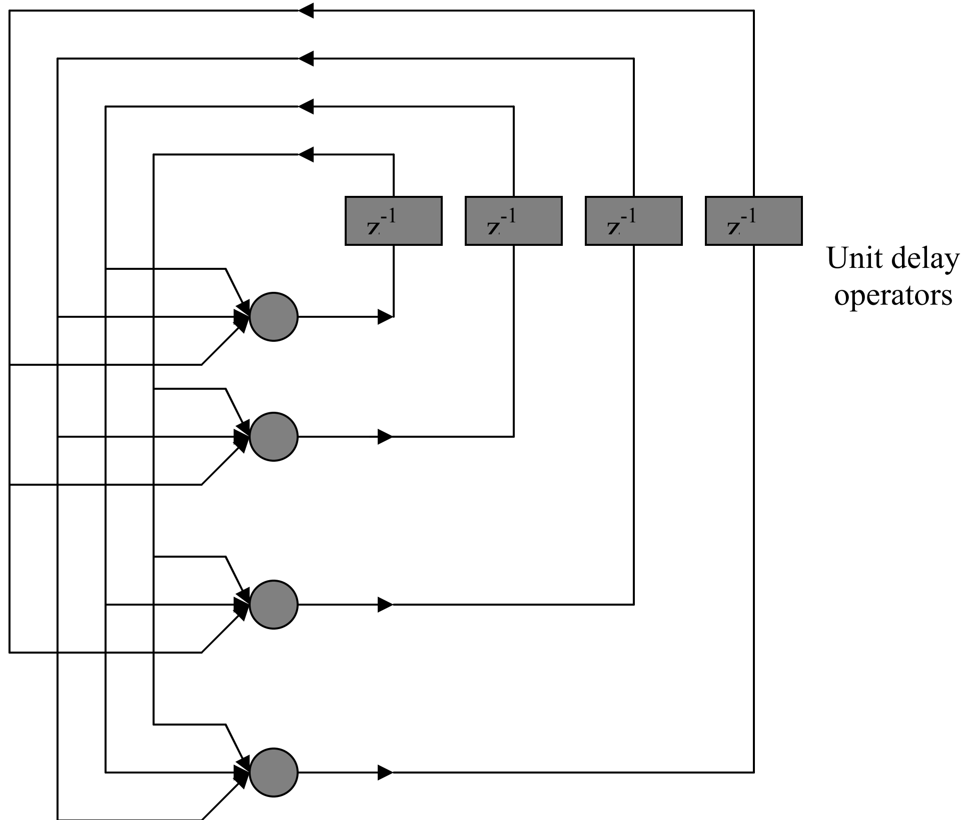



Figure 4.3: Recurrent network with no self-feedback loops and no hidden neurons

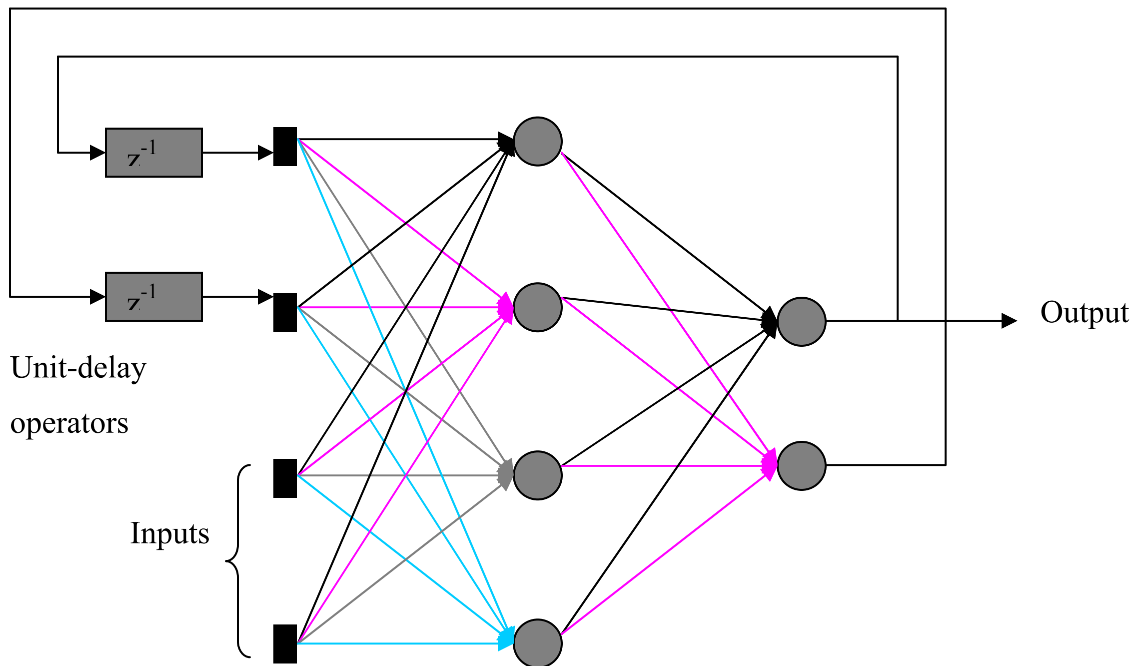



Figure 4.4: Recurrent network with hidden neurons

# 5. The learning processes

The property that is of primary significance for a neural network is the ability of the network to *learn* from its environment, and to *improve* its performance through learning. The improvement in performance takes place over time in accordance with some prescribed measure. A neural network learns about its environment through an interactive process of adjustments applied to its synaptic weights and bias levels. Ideally, the network becomes more knowledgeable about its environment after each iteration of the learning process.

There are too many activities associated with the notion of “learning” to justify defining it in precise manner. Moreover, the process of learning is a matter of viewpoint, which makes it all the more difficult to agree on a precise definition of the term. For example, learning as viewed by a psychologist is quite different from learning in a classroom sense. Recognizing that our particular interest is in neural networks, we use a definition of learning that is adapted from Mendel and McClaren. We define learning in the context of neural networks as:

> *Learning is a process by which the free parameters of a neural network are adapted through a process of simulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameter changes take place.*

This definition of the learning process implies the following sequence of events:

1. The neural network is *stimulated* by an environment.
2. The neural network *undergoes changes* in its free parameters as a result of this stimulation.
3. The neural network *responds in a new way* to the environment because of the changes that have occurred in its internal structure.

A prescribed set of well-defined rules for the solution of a learning problem is called a *learning algorithm*. As one would expect, there is no unique learning algorithm for the design of neural networks. Rather, we have a “kit of tools” represented by a diverse variety of learning algorithms, each of which offers advantages of its own. Basically, learning algorithms differ from each other in the way in which the adjustment to a synaptic weight of a neuron is formulated. Another factor to be considered is the manner

in which a neural network (learning machine), made up of a set of interconnected neurons, relates to its environment. In this latter context we speak of a *learning paradigm* that refers to a *model* of the environment in which the neural network operates. There are five basic learning rules: error-correction learning, memory-based learning, Hebbian learning, competitive learning, and Boltzmann learning, each of which is discussed below.

## 5.1 Error-correction learning

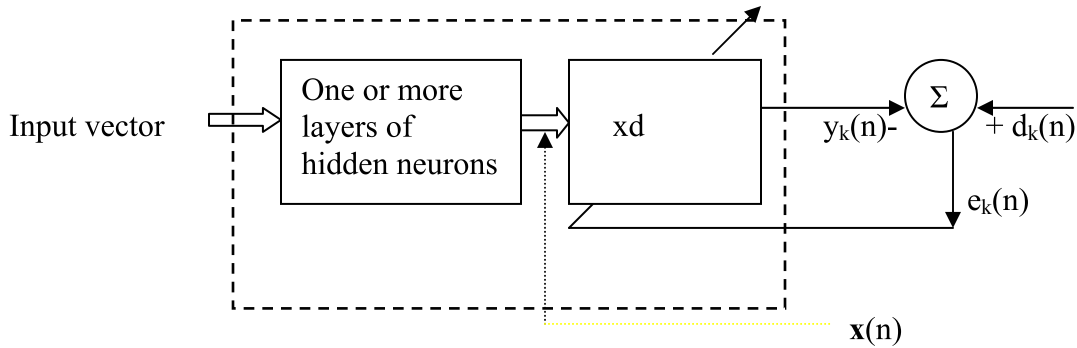
To illustrate our first learning rule, consider case of a neuron k constituting the only computational node in the output layer of a feed forward neural network, as shown in figure 5.1a. Neuron k is driven by a signal vector x(n) produced by one or more layers of hidden neurons, which are themselves driven by an input vector applied to the source nodes of the neural network. The argument n denotes discrete time, more precisely, the time step of an iterative process involved in adjusting the synaptic weights of neuron k. The *output signal* of neuron k is denoted by $y_k(n)$. This output signal, representing the only output of the neural network, is compared to a desired response denoted by $d_k(n)$. Consequently an error signal, denoted by $e_k(n)$, is produced. By definition, we thus have

$$e_k(n) = d_k(n) - y_k(n) \qquad 5.1$$

The error signal actuates a control mechanism, the purpose of which is to apply a sequence of corrective adjustments to the synaptic weights of the neuron k. The corrective adjustments are designed to the output signal $y_k(n)$ come closer to the desired response $d_k(n)$ in a step-by-step manner. This objective is achieved by minimizing a cost function or index of performance, $\varepsilon(n)$, defined in terms of the error signal $e_k(n)$ is:

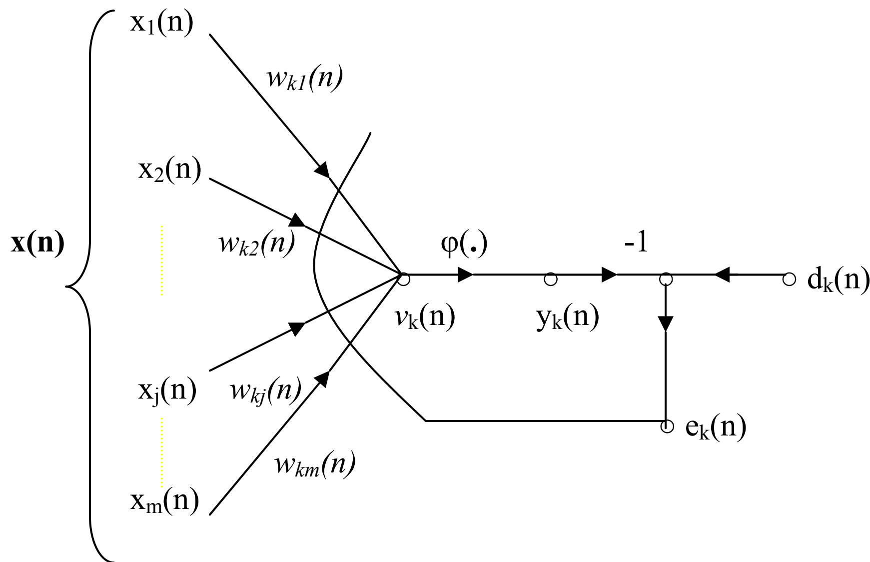
$$\varepsilon(n) = \tfrac{1}{2} e^2_k(n) \qquad 5.2$$

That is $\varepsilon(n)$ is the instantaneous value of the error energy. The step-by-step adjustments to the synaptic weights of neuron k are continued until the system reaches a steady state (i.e., the synaptic weights are essentially stabilized). At that point the learning point is terminated.




Multilayer feedforward network

(a) Block diagram of a neural network, highlighting the only neuron in the output layer.




(b) Signal flow diagram of output neuron

Fig. 5.1 illustrating error-correction learning

The learning process described herein is obviously referred to as error-correction learning. In particular, minimization of the cost function ε(n) leads to a learning rule commonly referred to as the *delta rule* or *Widrow-Hoff rule,* named in honor of its originators. Let $w_{kj}(n)$ denote the value of synaptic weight $w_{kj}$ of neuron k excited by element $x_j(n)$ of the signal vector **x(n)** at time step n. According to the delta rule, the adjustment $\Delta w_{kj}(n)$ applied to the synaptic weight $w_{kj}(n)$ at time step n is defined by

$$\Delta w_{kj}(n) = \eta \, e_k(n) \, x_j(n) \qquad 5.3$$

where $\eta$ is a positive constant that determines the rate of learning as we proceed from one step in the learning process to another. It is therefore natural to refer to $\eta$ the *learning-rate parameter*. In other words, the delta rule may be stated as:

*The adjustment made to a synaptic weight of a neuron is proportional to the product of error signal and the input signal of the synapse in question.*

Keep in mind that the delta rule, as stated herein, presumes that the error signal is *directly measurable.* For this measurement to be feasible we clearly need a supply of desired response from some external source, which is directly accessible to neuron k. In other words, neuron k is visible to the outside world. Having computed the synaptic adjustment $\Delta w_{kj}(n)$, the update value of synaptic weight $w_{kj}$ is determined by

$$w_{kj}(n + 1) = w_{kj}(n) + \Delta w_{kj}(n) \qquad 5.4$$

In effect, $w_{kj}(n)$ and $w_{kj}(n+1)$ may be viewed as the *old* and *new* values of synaptic weight $w_{kj}$, respectively. In computational terms we may also write

$$w_{kj}(n) = z^{-1}[w_{kj}(n+1)] \qquad 5.5$$

where $z^{-1}$ is the *unit-delay operator*. That is, $z^{-1}$ represents a *storage element.*

Figure 5.1b shows a signal-flow graph representation of the error-correction learning process, focusing on activity surrounding neuron k. The input signal $x_j$ and induced local $v_k$ of neuron k are referred to as the *presynaptic* and *postsynaptic signals* of the $j^{th}$ synapse of neuron k, respectively. From Fig 5.1b we see that error-correction learning is an example of an example of a *closed-loop feedback system*. From control theory we know that the stability of such a system is determined by those parameters that constitute the feedbacks of the system. In our case we only have a single feedback loop, and one of those parameters of particular interest is the learning-rate parameter $\eta$. It is therefore important that $\eta$ is carefully selected to ensure that the stability or convergence of the iterative learning process is achieved. The choice of $\eta$ also has a profound influence on the accuracy and other aspects of the learning process. In short, the learning-rate parameter $\eta$ plays a key role in determining the performance of error-correction learning in practice.

## 5.2 Memory-based learning

In memory-based learning, all (or most) of the past experiences are explicitly stored in a large memory of correctly classified input-output examples: $\{(x_i, d_i)^N_{i=1}$ , where $x_i$ denotes an input vector and $d_i$ denotes the corresponding desired response. Without loss of generality, we have restricted the desired response to be a scalar. For example, in a binary pattern classification problem there are two classes/hypotheses, denoted by $l_1$ and $l_2$, to be considered. In this example, the desired response $d_i$ takes the value 0 (or -1) for class $l_1$ and the value 1 for class $l_2$. When classification of a test vector $x_{test}$ (not seen before) is required, the algorithm responds by retrieving and analyzing the training data in a "local neighborhood" of $x_{test}$.

All memory based learning algorithms involve two essential ingredients:

- Criterion used for defining the local neighborhood of the test vector $x_{test.}$
- Learning rule applied to the training examples in the local neighborhood of $x_{test.}$

The algorithms differ from each other in the way in which these two ingredients are defined. In a simple yet effective type of memory-based learning known as the nearest neighbor rule, the local neighborhood is defined as the training that lies in the immediate neighborhood of the test vector $x_{test}$. In particular, the vector

$$\mathbf{X}'_N \in \{\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_N}\} \qquad 5.6$$

is said to be the nearest neighborhood of $x_{test}$ if

$$\underset{i}{\text{mid}}\ d(\mathbf{x}_i, \mathbf{x}_{test}) = d(\mathbf{x}'_N, \mathbf{x}_{test}) \qquad 5.7$$

where $d(\mathbf{x}'_N, \mathbf{x}_{test})$ is the Euclidean distance between the vectors $\mathbf{x}_i$ and $\mathbf{x}_{test}$. The class associated with the minimum distance, that is, vector $\mathbf{x}'_N$, is reported as the classification of $\mathbf{x}_{test}$. This rule is independent of the underlying distribution responsible for generating the training examples.

Cover and Hat(1967) have formally studied the nearest neighbor rule as a tool for pattern classification. The analysis presented therein is based on two assumptions:

- The classified examples ($\mathbf{x}_j$, $d_i$) are *independently and identically distributed (iid),* according to the joint probability distribution of the example ($\mathbf{x}$, d).

- The sample size $N$ is infinitely large.

Under these two assumptions, it is shown that the probability of classification error incurred by the nearest neighbor rule bounded above by twice the *Bayes probability of error*, that is, the minimum probability of error over all decision rules. In this sense, it may be said that half the classification information in a training set of infinite size is contained in the nearest neighbor, which is a surprising result.

A variant of the nearest neighbor classifier is the *k- nearest neighbor classifier*, which proceeds as follows:

- Identify the k classified patterns that lie nearest to the test vector $\mathbf{x}_{test}$ for some integer k.
- Assign $\mathbf{x}_{test}$ to the class (hypothesis) that is most frequently represented in the k nearest neighbors to $\mathbf{x}_{test}$ (i.e., use a majority vote to make the classification).

Thus the k-nearest neighbor classifier acts like an averaging device. In particular, it discriminates against a single outlier, as illustrated in Fig. 5.2 for k = 3. An outlier is an observation that is improbably large for a nominal model of interest.

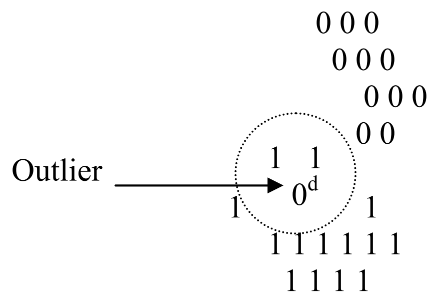



Figure 5.2: The area lying inside the dashed circle includes two points pertaining to class 1 and an outlier from class 0. The point d corresponds to the test vector $\mathbf{x}_{test}$. With k = 3, the *k-nearest neighbor classifier* assigns class 1 to point d even though it lies closest to the *outlier.*

## 5.3 Hebbian learning

*Hebb's postulate of learning* is the oldest and most famous of all learning rules; it is named in honor of the neuropsychologist Hebb (1949). Quoting from Hebb's book, *The Organization of Behavior* (1949, p.62):

> When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic changes take place in one or both cells such that A's efficiency as one of the cells firing B, is increased.

Hebb proposed this change as a basis of associative learning (at the cellular level), which would result in an enduring modification in the activity pattern of a spatially distributed "assembly of nerve cells."

This statement is made in neurobiological context. We may expand and rephrase it as a two-part rule:

1. *If two neurons on either side of a synapse (connection) are activated simultaneously (i.e., synchronously), then the strength of that synapse is selectively increased.*
2. *If two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened or eliminated.*

Such a synapse is called a *Hebbian synapse*. More precisely, we define a Hebbian synapse as a synapse that uses a *time-dependent, highly local, and strongly interactive mechanism to increase synaptic efficiency as a function of the correction between the presynaptic and post synaptic activities.* From this definition we may deduce the following four key mechanisms (properties) that characterize a Hebbian synapse:

***1. Time-dependent mechanism****:* This mechanism refers to the fact that the modifications in Hebbian synapse depend on the exact time of occurrence of the presynaptic and postsynaptic signals.

**2. *Local mechanism****:* By its very nature, a synapse is the transmission site where information-bearing signals (representing ongoing activity in the presynaptic and postsynaptic units) are in *spatiotemporal* contiguity. This locally available information is used by a Hebbian synapse to produce a local synaptic modification that is input specific.

**3. *Interactive mechanism****:* The occurrence of a change in a Hebbian synapse depends on signals on both sides of the synapse. That is, a Hebbian form of learning depends on a "true interaction" between presynaptic and postsynaptic signals in the sense that we cannot make a prediction from either on these tow activities by itself. Note also that this dependence or interaction may be deterministic or statistical in nature.

**4.** ***Conjunctional or correlational mechanism****:* One interpretation of Hebb's postulate of learning is that the condition for a change in synaptic efficiency is the conjunction of presynaptic and postsynaptic signals. Thus, according to this interpretation, the co-occurrence of presynaptic and postsynaptic signals (within a short interval of time) is sufficient to produce the synaptic modification. It is for this reason that a Hebbian synapse is sometimes referred to as a *conjunctional synapse.* For another interpretation of Hebb's postulate of learning, we may think of the interactive mechanism characterizing a Hebbian synapse in statistical terms. In particular, the correlation over time between presynaptic and postsynaptic signals is viewed as being responsible for a synaptic change. Accordingly, a Hebbian synapse is also referred to as a *correlational synapse.* Correlation is indeed the basis of learning.

### 5.3.1 Synaptic enhancement and depression

The definition of a Hebbian synapse represented here does not include additional processes that may result in weakening of a synapse connecting a pair of neurons. Indeed, we may generalize the concept of a Hebbian modification by recognizing that positively correlated activity produces synaptic strengthening, and that either uncorrelated or negatively correlated activity produces synaptic weakening. Synaptic depression may also be of a noninteractive type. Specifically, the interactive condition for synaptic weakening may simply be noncoincident presynaptic or postsynaptic activity.

We may go one step further by classifying synaptic modification as *Hebbian, anti-Hebbian, and non-Hebbian.* According to this scheme, a Hebbian synapse increases its strength with positively correlated presynaptic and postsynaptic signals, and decreases its strength when these signals are either uncorrelated or negatively correlated. Conversely, in anti-Hebbian synapse weakens positively correlated presynaptic and postsynaptic signals, and strengthens negatively correlated signals. In both Hebbian and anti-Hebbian synapses, however, the modification of synaptic efficiency relies on a mechanism that is time-dependent, highly local, and strongly interactive in nature. In that sense, an anti-Hebbian synapse is still Hebbian in nature, though not in function. A non-Hebbian synapse, on the other hand, does not involve a Hebbian mechanism of either kind.

### 5.3.2 Mathematical models of Hebbian functions

To formulate Hebbian learning in mathematical terms, consider a synaptic weight $w_{kj}$ of neuron *k* with presynaptic and postsynaptic signals denoted by $x_j$ and $y_k$, respectively. The adjustment applied to the synaptic weight $w_{kj}$ at time step n is expressed in the general form

$$\Delta w_{kj}(n) = F(y_k(n), x_j(n)) \qquad 5.8$$

where F(•, •) is a function of both presynaptic and postsynaptic signals. The signals $x_j(n)$ and $y_k(n)$ are often treated as dimensionless. The formula of Eq. 5.8 admits many forms, all of which qualify as Hebbian. In what follows, we consider two such forms:

**Hebb's hypothesis:** The simplest form of Hebbian learning is described by:

$$\Delta w_{kj}(n) = \eta\, y_k(n)\, x_j(n) \qquad 5.9$$

where η is a positive constant that determines the *rate of learning*. Eq. 5.9 clearly emphasizes the correlational nature of a Hebbian synapse. It is sometimes referred to as the *activity product rule*. The top curve of figure 5.3 shows a graphical representation of Eq. (5.9) with the change $\Delta w_{kj}$ plotted versus the output signal (postsynaptic activity) $y_k$. From this representation we see that the repeated application of the input signal (presynaptic activity) $x_j$ leads an increase in $y_k$ and therefore *exponential growth* that finally drives the synaptic connection into saturation. At that point no information will be stored in the synapse and selectivity is lost. .

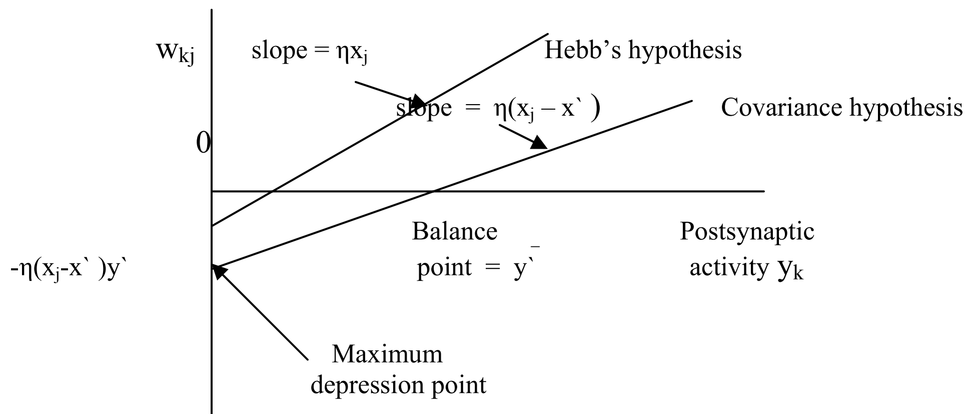



Fig 5.3: Illustration of Hebb's hypothesis and the covariance hypothesis.

**Covariance hypothesis**

One way of overcoming the limitation of Hebb's hypothesis is to use the *covariance hypothesis*. In this hypothesis, the presynaptic and postsynaptic signals in Eq. 5.9 are replaced by the departure of presynaptic and postsynaptic signals from their respective average values over a certain time interval. Let x` and y` denote the *time-averaged values* of the presynaptic signal $x_j$ and postsynaptic signal $y_k$, respectively. According o the covariance hypothesis, the adjustment applied to the synaptic weight $w_{kj}$ is defined by

$$\Delta w_{kj} = \eta(x_j - x`)(y_k - y`) \qquad 5.10$$

where $\eta$ is the learning-rate parameter. The average values x` and y` constitute presynaptic and postsynaptic thresholds, which determine the sign of synaptic modification. In particular, the covariance hypothesis allows for the following:

- Convergence to a nontrivial, which is reached when $x_k = x`$ or $y_j = y'$.
- Prediction of both synaptic *potentiation* (i.e., increase in synaptic strength) and synaptic *depression* (i.e., decrease in synaptic strength).

Fig. 5.3 illustrates the difference between Hebb's hypothesis and the covariance hypothesis. In both cases the dependence of $\Delta w_{kj}$ on $y_k$ is linear; however, the intercept with the $y_k$-axis in Hebb's hypothesis is at the origin, whereas in the covariance hypothesis it is at $y_k = y`$.

We make the following important observations from Eq. 5.10:

1. Synaptic weight $w_{kj}$ is enhanced if there are sufficient levels of presynaptic and postsynaptic activities, that is, the conditions $x_j > x`$ and $y_k > y`$ are both satisfied.
2. Synaptic weight $w_{kj}$ is depressed if there is either
    - a presynaptic activation (i.e., $x_j > x'$) in the absence of sufficient postsynaptic activation (i.e., $y_k < y`$);
    - postsynaptic activation (i.e., $y_k > y`$) in the absence of sufficient presynaptic activation (i.e., $x_j < x`$);

This behavior may be regarded as a form of temporal competition between the incoming patterns. There is strong physiological evidence for Hebbian learning in the area of the

brain called the *hippocampus*. The hippocampus plays an important role in certain aspects of learning or memory. This physiological evidence makes Hebbian learning all the more appealing.

## 5.4 Competitive learning

In competitive learning, as the name implies, the output neurons of a neural network compete among themselves to become active. Whereas in a neural network based on Hebbian learning several output neurons may be active simultaneously, in competitive learning only a single output neuron is active at a time. It is this feature that makes competitive learning highly suited to discover statistically salient features that may be used to classify a set of input patterns. There are three basic elements to a competitive learning rule:

- A set of neurons that are all the same except for some randomly distributed synaptic weights, and which therefore *respond differently* to a given set of input patterns.
- A *limit* imposed on the "strength" of each neuron.
- A mechanism that permits the neurons to *compete* for the right to respond to a given subset of inputs, such that only *one* output neuron, or only one neuron per group, is active at a time. The neuron that wins the competition is called a winner-*takes-all neuron*.

Accordingly, the individual neurons of the network learn to specialize on ensembles of similar patterns; in so doing they become feature detectors for different classes of input patterns.

In the simplest form of competitive learning, the neural network has a single layer of output neurons, each of which is fully connected to the input nodes. The network may include feedback connections among the neurons, as indicated in the fig 5.4. In the network architecture described herein, the feedback connections perform *lateral inhibition*, with each neuron tending to inhibit the neuron to which it is laterally connected. In contrast, the feedforward synaptic connections in the network of fig 5.4 are all *excitatory*. For a neuron k to be the winning neuron, its induced local field $v_k$ for a

specified input pattern **x** must be the largest among all the neurons in the network. The output signal $y_k$ of winning neuron k is set equal to one; the output signals of all the neurons that lose the competition are set equal to zero. We thus write

$$y_k = \begin{cases} 1 & \text{if } v_k > v_j \text{ for all } j, j \neq k \\ 0 & \text{otherwise} \end{cases} \qquad 5.11$$

where the induced local field $v_k$ represents the combined action of all the forward and feedback inputs to neuron k.

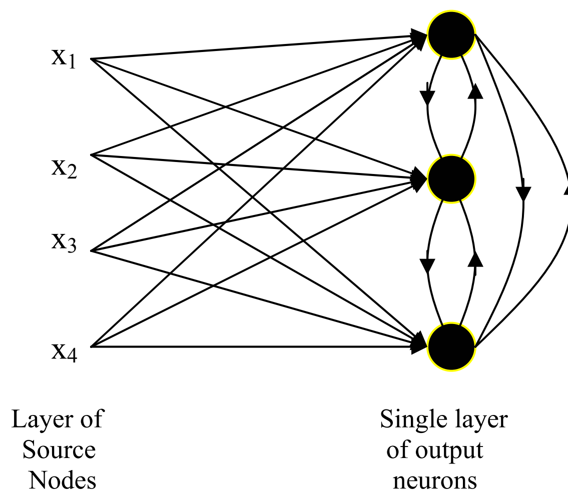



Figure 5.4 Architectural graph of a simple competitive learning network with feedforward (excitatory) connection from the source nodes to the neurons, and lateral (inhibitory) connections among neurons; the lateral connections are signified by open arrows.

Let $w_{kj}$ denote the synaptic weight connecting input node j to neuron k. Suppose that each neuron is allotted a *fixed* amount of synaptic weight (i.e., all synaptic weights are positive), which distributed among its input nodes; that is,

$$\sum_j w_{kj} = 1 \qquad \text{for all } k \qquad 5.12$$

A neuron then learns by shifting synaptic weights from its inactive to active nodes. If a neuron does respond to a particular input pattern, no learning takes place in that neuron. If a particular neuron wins the competition, each input node of that neuron relinquishes

some proportion of its synaptic weight, and the weight relinquished is then distributed equally among the active input nodes. According to the standard *competitive learning rule*, the change $\Delta w_{kj}$ applied to synaptic weight $w_{kj}$ is defined by

$$\Delta w_{kj} = \begin{cases} \eta(xj - wkj) & \text{if neuron } k \text{ wins competition} \\ 0 & \text{if neuron } k \text{ loses the competition} \end{cases} \quad 5.13$$

where $\eta$ is the learning-rate parameter. This rule has the overall effect of moving the synaptic weight vector $\mathbf{w}_k$ of winning neuron *k* toward the input pattern $\mathbf{x}$.

## 5.5 Bolzmann learning

The Bolzmann learning rule, named in honor of Ludwig Boltzmann, is a stochastic learning algorithms derived from ideas rooted in statistical mechanics. A neural network designed on the basis of the Boltzmann learning rule is called a *Boltzmann machine*.

In a Boltzmann machine, the neurons constitute a recurrent structure, and they operate in a binary manner since, for example, they are either in an “on” state denoted by +1 or in an “off” state denoted by -1. The machine is characterized by an *energy function, E,* the value of which is determined by the particular states occupied by the individual neurons of the machine, as shown by

$$E = -\tfrac{1}{2} \sum_{j} \sum_{k} w_{kj}\, x_k x_j \qquad j \neq k \quad 5.15$$

where $x_j$ is the state of neuron *j*, and $w_{kj}$ is the synaptic weight connecting neuron *j* to neuron *k*. The fact that $j \neq k$ means simply that none of the neurons in the machine has self-feedback. The machine operates by choosing a neuron at random- for example, neuron k- at some step of the learning process, then flipping the state of neuron *k* from state $x_k$ to state $-x_k$ at some temperature *T* with probability

$$P(x_k \rightarrow -x_k) = 1/(1 + \exp(-\Delta E_k / T)) \quad 5.16$$

where $\Delta E_k$ is the *energy change* (i.e., the change in the energy function of the machine) resulting from such a flip. Notice that *T* is not a physical temperature, but rather a

*pseudotemprature*. If this rule is applied repeatedly, the machine will reach *thermal equilibrium*.

The neurons of a Boltzmann machine partition into two functional groups: *visible* and *hidden*. The visible neurons provide an interface between the network and the environment in which it operates, whereas the hidden neurons always operate freely.
There are two modes of operation to be considered:

- *Clamped condition*, in which the visible neurons are all clamped onto specific states determined by the environment.
- *Free-running condition*, in which all neurons (visible and hidden) are allowed to operate freely.

Let $\rho^{+}_{kj}$ denote the *correlation* between the states of neurons *j* and *k*, with the network in its clamped condition. Let $\rho^{-}_{kj}$ denote the *correlation* between the states of neurons *j* and *k*, with the network in its free-running condition. Both correlations are averaged over all possible states the machine when it is in thermal equilibrium. Then, according to the *Bolzmann learning rule*, the change $\Delta w_{kj}$ applied to the synaptic weight $w_{kj}$ from neuron *j* to neuron *k* is defined by

$$\Delta w_{kj} = \eta(\rho^{+}_{kj} - \rho^{-}_{kj}), \quad j \neq k \qquad 5.17$$

Note that both $\rho^{+}_{kj}$ - $\rho^{-}_{kj}$ range in value from -1 to +1.

# 6. Training algorithms

## 6.1 The error back-propagation algorithm

Error Backpropagation was created by generalizing the Widrow-Hoff learning rule to multiple-layer networks and nonlinear differentiable transfer functions. Input vectors and the corresponding target vectors are used to train a network until it can approximate a function, associate input vectors with specific output vectors, or classify input vectors in an appropriate way as defined by you. Networks with biases, a sigmoid layer, and a linear output layer are capable of approximating any function with a finite number of discontinuities.

Standard error backpropagation is a gradient descent algorithm, as is the Widrow-Hoff learning rule, in which the network weights are moved along the negative of the gradient of the performance function. The term *error backpropagation* refers to the manner in which the gradient is computed for nonlinear multilayer networks. There are a number of variations on the basic algorithm that are based on other standard optimization techniques, such as conjugate gradient and Newton methods.

Basically, error back-propagation learning consists of two passes through the different layers of the network: a forward pass and a backward pass. In the *forward pass*, an activity pattern (input vector) is applied to the sensory nodes of the network, and its effect propagates through the network layer by layer. Finally, a set of outputs is produced as the actual response of the network. During the forward pass the synaptic weights of the network are all fixed. During the *backward pass*, on the other hand, the synaptic weights are all adjusted in accordance with an error-correction rule. Specifically, the actual response of the network is subtracted from a desired (target) response to produce an *error signal*. This error signal is then propagated backward through the network against the direction of synaptic connections- hence the name "error back-propagation." The synaptic weights are adjusted to make the actual response of the network move closer to the desired response in a statistical sense. The error back-propagation algorithm is also referred to in the literature as the *back-propagation algorithm*, simply *back-prop*.

The error signal at the output of neuron *j* at iteration *n* (i.e., presentation of the *n*th training example) is defined by

$$e_j(n) = d_j(n) - y_j(n), \qquad \text{neuron } j \text{ is an output node} \qquad 6.1$$

We define the instantaneous value of the error energy for neuron *j* as $\frac{1}{2}e^2_j(n)$. Correspondingly, the instantaneous value $\varepsilon(n)$ of the total error energy is obtained by summing $\frac{1}{2}e^2_j(n)$ over *all neurons in the output layer*; these are the only "visible" neurons for which error signals can be calculated directly. We may thus write

$$\varepsilon(n) = \frac{1}{2} \sum_{j \in C} e^2_j(n) \qquad 6.2$$

where the set $C$ includes all the neurons in the output layer of the network. Let $N$ denote the total number of patterns (examples) contained in the training set. The *average squared error energy* is obtained by summing $\varepsilon(n)$ over all n and then normalizing with respect to the size of $N$, as shown by

$$\varepsilon_{av} = 1/N \sum_{n=1}^{N} \boldsymbol{\varepsilon(n)} \qquad 6.3$$

The instantaneous error energy $\boldsymbol{\varepsilon(n)}$, and therefore the average error energy $\varepsilon_{av}$, is a function of all the free parameters (i.e., synaptic weights and bias levels) of the network. For a given training set, $\varepsilon_{av}$ represents the *cost function* as a measure of learning performance. The objective of the learning process is to adjust the free parameters of the network to minimize $\varepsilon_{av}$. To do this minimization, we use an approximation similar in rationale to that used for the derivation of the LMS algorithm. Specifically, we consider a simple method of training in which the weights are updated on a *pattern-by-pattern* basis until an *epoch*, that is, one complete presentation of the entire training set has been dealt with. The adjustments to the weights are made in accordance with the respective errors computed for *each* pattern presented to the network.

The Arithmetic average of these individual weight changes over the training set is therefore an *estimate* of the true change would result from modifying the weights based on minimizing the cost function $\varepsilon_{av}$ the entire training set. Consider Fig. 6.1, which depicts neuron $j$ being fed by a set of function signals produced by a layer of neurons to its left.

The induced local field $v_j(n)$ produced at the input of the activation function associated with neuron $j$ is therefore

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \qquad 6.4$$

where $m$ is the total number inputs (excluding the bias ) applied to neuron $j$. The synaptic weight $w_{j0}$ (corresponding to the fixed input $y_0 = +1$) equals the bias $b_j$ applied to neuron $j$. hence the function signal $y_j(n)$ appearing at the output of neuron $j$ at iteration $n$ is

$$y_j(n) = \varphi_j(v_j(n)) \qquad 6.5$$

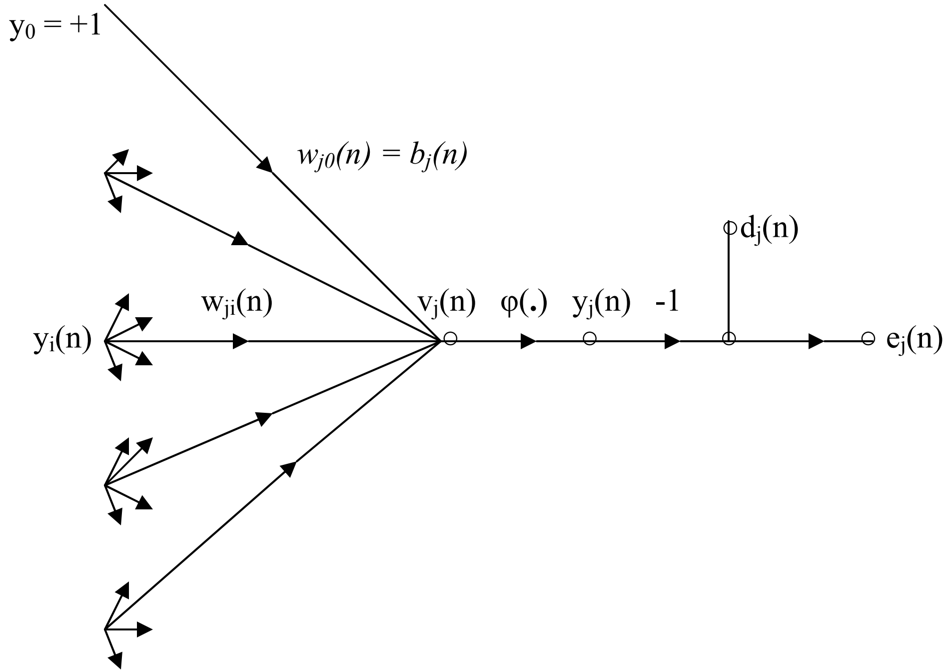



Figure 6.1 a signal-flow graph highlighting the details of output neuron *j*.

In a manner similar to the LMS algorithm, the back-propagation algorithm applies a correction $\Delta w_{ji}(n)$ to the synaptic weight $w_{ji}(n)$, which is proportional to the partial derivative $\partial \boldsymbol{\varepsilon}(\mathbf{n}) / \partial w_{ji}(n)$. According to the *chain rule* of calculus, we may express this gradient as:

$$\partial \boldsymbol{\varepsilon}(\mathbf{n}) / \partial w_{ji}(n) = (\partial \boldsymbol{\varepsilon}(\mathbf{n}) / \partial e_j(n)) \mathrm{x} (\partial e_j(n) / \partial y_j(n)) \mathrm{x} (\partial y_j(n) / \partial v_j(n)) \mathrm{x} (\partial v_j(n) / \partial w_{ji}(n)) \qquad 6.6$$

The partial derivative $\partial \boldsymbol{\varepsilon}(\mathbf{n}) / \partial w_{ji}(n)$ represents a *sensitivity factor*, determining the direction of search in weight space for the synaptic weight $w_{ji}$.
Differentiating both sides of Eq. 6.2 with respect to $e_j(n)$, we get

$$\partial \boldsymbol{\varepsilon}(\mathbf{n}) / \partial e_j(n) = e_j(n) \qquad 6.7$$

Differentiating both sides of Eq. 6.1 with respect to $y_j(n)$, we get

$$\partial e_j(n) / \partial y_j(n) = -1 \qquad 6.8$$

Next, differentiating Eq. 6.5 with respect to $v_j(n)$, we get

$$\partial y_j(n)/ \partial v_j(n) = \varphi'_j(v_j(n)) \qquad 6.9$$

where the use of prime (on the right-and side) signifies differentiation with respect to the argument. Finally, differentiating Eq. (6.4) with respect to $w_{ji}$(n) yields

$$\partial v_j(n)/ \partial w_{ji}(n) = y_j(n) \qquad 6.10$$

The use of Eqs. (6.7) to (6.10) in (6.6) yields

$$\partial \boldsymbol{\varepsilon}(\mathbf{n})/ \partial e_j(n) = -e_j(n)\varphi'_j(v_j(n))y_i(n) \qquad 6.11$$

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

$$\Delta w_{ji}(n) = -\eta\ (\partial \boldsymbol{\varepsilon}(\mathbf{n})/ \partial\ w_{ji}(n)) \qquad 6.12$$

where η is the *learning-rate parameter* of the backpropagation algorithm. The use of the minus sign in Eq. (6.12) accounts for *gradient descent* in weight space (i.e., seeking a direction for weight change that reduces the value of **ε(n)).** Accordingly, the use of Eq. (6.11) in (6.12) yields

$$\Delta w_{ji}(n) = \eta\delta_j(n)y_i(n) \qquad 6.13$$

where the *local gradient* $\eta\delta_j(n)$ is defined by

$$\eta\delta_j(n) = -\partial \boldsymbol{\varepsilon}(\mathbf{n})/ \partial v_j(n) \qquad 6.14$$
$$= (\partial \boldsymbol{\varepsilon}(\mathbf{n}) /\partial e_j(n))\mathrm{x}(\partial e_j(n)/ \partial y_j(n))\mathrm{x}(\partial y_j(n)/ \partial v_j(n)$$

$$= e_j(n)\varphi'(v_j(n)) \qquad 6.15$$

The local gradient points to required changes in synaptic weights. According to Eq. (6.15), the local gradient $\delta_j(n)$ for output neuron *j* is equal to the product of the corresponding error signal $e_j(n)$ for that neuron and the derivative $\varphi'_j(v_j(n))$ of the associated activation function. From Eqs. (6.13) and (6.14) we note that a key factor involved in the calculation of the weight adjustment $\Delta w_{ji}(n)$ is the error signal $e_j(n)$ at the output of neuron *j*. In this context we may identify two distinct cases, depending on

where in the network neuron *j* is located. In case 1, neuron *j* is an output node. This case is simple to handle because each output node of the network is supplied with a desired response of its own, making it a straight forward matter to calculate the associated error signal. In case 2, neuron *j* is a hidden node. Even though hidden neurons are not directly accessible, they share responsibility for any error made at the output of the network. The question, however, is to know how to penalize or reward hidden neurons for their share of the responsibility. This problem is the called the *credit-assignment problem*. It is solved in an elegant fashion by back-propagating the error signals through the network.

**Case 1 Neuron *j* is an output node**

When neuron *j* is located in the output layer of the network, it is supplied with a desired response of its own. We may Eq. (6.1) to compute the error signal $e_j(n)$ associated with this neuron; Having determined $e_j(n)$ it is a straight forward matter to compute the local gradient $\delta_j(n)$ using Eq. (6.14).

**Case 2 Neuron *j* is a hidden node**

When neuron *j* is located in a hidden layer of the network, there is no specified desired response for that neuron. Accordingly, the error signal for a hidden neuron would have to be determined recursively in terms of the error signals of all the neurons to which that hidden neuron is directly connected; this is where the development of the back-propagation algorithm gets complicated. Consider the situation depicted in Fig. (6.4), which depicts neuron *j* as a hidden node of the network. According to Eq. (6.14), we may redefine the local gradient $\delta_j(n)$ for hidden neuron *j* as

$$\delta_j(n) = -(\partial \boldsymbol{\varepsilon}(\mathbf{n}) / \partial y_j(n)) \text{x} (\partial y_j(n) / \partial v_j(n))$$

$$= (\partial \boldsymbol{\varepsilon}(\mathbf{n}) / \partial y_j(n)) \varphi'(v_j(n)), \qquad 6.16$$

where in the second line we have used Eq. (6.9). To calculate the partial derivative $\partial \boldsymbol{\varepsilon}(\mathbf{n}) / \partial y_j(n)$, we may proceed as follows. From Fig. 6.4 we see that

$$\boldsymbol{\varepsilon}(\mathbf{n}) = \tfrac{1}{2} \sum_{k \in C} e^2_k(n), \qquad \text{neuron } k \text{ is an output node} \qquad 6.17$$

Differentiating equation Eq. (6.17) with respect to the function signal $y_j(n)$, we get

$$\partial \boldsymbol{\varepsilon}(\mathbf{n}) / \partial y_j(n) = \sum_k \partial e_k(n) / \partial y_j(n) \qquad 6.18$$

Next we use the chain rule for the partial derivative $\partial e_k(n) / \partial y_j(n)$, and rewrite Eq. (6.18) in the equivalent form

$$\partial e_k(n) / \partial y_j(n) = \sum_k e_k(n) \; (\partial e_k(n) / \partial v_k(n)) \mathrm{x} (\partial v_k(n) / \partial y_j(n)) \qquad 6.19$$

However, from Fig. 6.4, we note that

$$\begin{aligned} e_k(n) &= d_k(n) - y_k(n) \\ &= d_k(n) - \varphi_k(v_k(n)), \qquad \text{neuron } k \text{ is an output node} \end{aligned} \qquad 6.20$$

Hence

$$\partial e_k(n) / \partial v_k(n) = - \varphi'_k(v_k(n)) \qquad 6.21$$

We also note that from Fig. 6.4 that for neuron $k$ the induced local field is

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \qquad 6.22$$

where $m$ is the total number of inputs (excluding the bias) applied to neuron $k$. Here again, the synaptic weight $w_{k0}(n)$ is equal to the bias $b_k(n)$ applied to neuron $k$, and the corresponding input is fixed at the value +1. Differentiating Eq. (6.22) with respect to $y_j(n)$ yields

$$\partial v_k(n) / \partial y_j(n) = w_{kj}(n) \qquad 6.23$$

By using Eqs. (6.21) and (6.23) we get the desired partial derivative:

$$\begin{aligned} \partial \boldsymbol{\varepsilon}(\mathbf{n}) / \partial y_j(n) &= -\sum_k e_k(n) \varphi'_k(v_k(n)) \, w_{kj}(n) \\ &= \sum_k \delta_k(n) \, w_{kj}(n) \end{aligned} \qquad 6.24$$

where in the second line we have used the definition of the local gradient $\delta_k(n)$ given in Eq. (6.14) with the index $k$ substituted for $j$.

Finally, using Eq. (6.24) in (6.16), we get the *back-propagation formula* for the local gradient $\delta_k(n)$ as described by:

$$\delta_k(n) = \varphi'_k(v_k(n)) \sum_k \delta_k(n)\, w_{kj}(n), \text{ neuron } j \text{ is hidden} \qquad 6.25$$

The factor $\varphi'_k(v_k(n))$ involved in the computation of the local gradient $\delta_j(n)$ in Eq. (6.25) depends solely on the activation function associated with hidden neurons $j$. The remaining factor involved in this computation, namely the summation over $k$, depends on two sets of terms. The first set of terms, $\delta_k(n)$, requires knowledge of the error signal $e_k(n)$, for all neurons that lie in the layer to the immediate right of hidden neuron $j$, and that are directly connected to neuron $j$. The second set of terms, the $w_{kj}(n)$, consists of the synaptic weights associated with these connections.

We now summarize the relation that we have derived for the back-propagation algorithm. First, the correction $\Delta w_{ji}(n)$ applied to the synaptic weight connecting neuron $i$ to neuron $j$ is defined by the delta rule:

$$\begin{pmatrix} \text{Weight} \\ \text{Correction} \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} \textit{learning} \\ \textit{rate parameter} \\ \eta \end{pmatrix} \cdot \begin{pmatrix} \textit{local -} \\ \textit{gradient} \\ \delta_j(n) \end{pmatrix} \cdot \begin{pmatrix} \textit{input signal} \\ \textit{of neuron } j \\ y_i(n) \end{pmatrix} \qquad 6.26$$

Second, the local gradient $\delta_j(n)$ depends on whether neuron $j$ is an output node or a hidden node:

1 If neuron $j$ is an output node, $\delta_j(n)$ equals the product of the derivative $\varphi'_k(v_k(n))$ and the error signal $e_j(n)$, both of which are associated with neuron $j$.

2 If neuron $j$ is a hidden node, $\delta_j(n)$ equals the product of the associated derivative $\varphi'_k(v_k(n))$ and the weighted sum of the $\delta$s computed for the neuron in the next hidden or output layer that are connected to neuron $j$.

### 6.1.1 The two passes of computation

In the application of the back-propagation algorithm, two distinct passes of computation are distinguished. The first pass is referred to as the forward pass, and the second is referred to as the backward pass. In the *forward pass* the synaptic weights remain unaltered throughout the network, and the function signals of the network are computed on a neuron-by-neuron basis. The function signal appearing at the output of neuron $j$ is computed as

$$y_j(n) = \varphi(v_j(n)) \qquad 6.27$$

where $v_j(n)$ is the induced local field of neuron $j$, defined by

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n)\, y_i(n) \qquad 6.28$$

where m is the total number of inputs (including the bias) applied to neuron $j$, and $w_{ji}(n)$ is synaptic weight connecting neuron $i$ to neuron $j$, and $y_i(n)$ is the input signal of neuron $j$ or equivalently, the function signal appearing at the output of neuron $i$. If neuron $j$ is in the first hidden layer of the network, $m = m_0$ and the index $i$ refers to the $i^{th}$ input terminal of the network, for which we write

$$y_i(n) = x_i(n) \qquad 6.29$$

where $x_i(n)$ is the $i^{th}$ element of the input vector (pattern). If, on the other hand, neuron $j$ is in the output layer of the network, $m = m_L$ and the index $i$ refers to the $i^{th}$ output terminal of the network, for which we write

$$y_j(n) = o_j(n) \qquad 6.30$$

where $o_j(n)$ is the $j^{th}$ element of the output vector (pattern). This output is compared with the desired response $d_j(n)$, obtaining the error signal $e_j(n)$ for the $j^{th}$ output neuron. Thus the forward phase of computation begins at the first hidden layer by presenting it with the input vector, and terminates at output layer by computing the error signal for each neuron of this layer.

The backward phase, on the other hand, starts at the output layer by passing the error signal leftward through the network, layer by layer, and recursively computing the δ (i.e., the local gradient) for each neuron. This recursive process permits the synaptic

weights of the network to undergo changes in accordance with the delta rule of Eq. (6.26). For a neuron located in the output layer, the $\delta$ is simply equal to the error signal of that neuron multiplied by the first derivative of its nonlinearity. Hence we use Eq. (6.26) to compute the changes to the weights of all the connections feeding into the output layer. Given the $\delta$s for all neurons in the output layer, we next use Eq. (6.25) to compute the $\delta$s for all neurons the penultimate layer and therefore the changes to the weights of all connections feeding into it. The recursive computation is continued, layer by layer, by propagating the changes to all synaptic weights in the network.

Note that for the presentation of each training example; the input pattern is fixed ("clamped") throughout the round-trip process, encompassing the forward pass followed by the backward pass.

### 6.1.2 Rate of learning

The propagation algorithm provides an "approximation" to the trajectory in the weight space computed by the method of steepest descent. The smaller we make the learning-rate parameter $\eta$, the smaller the changes to the synaptic weights in the network will be from one iteration to the next, and the smoother will be the trajectory in weight space. This improvement, however, is attained at the cost of a slower rate of learning. If, on the other hand, we make the learning-rate parameter $\eta$ too large in order to speed up the rate of learning, the resulting large changes in the synaptic weights assume such a form that the network may become unstable (i.e., oscillatory). A simple method of increasing the rate of learning yet avoiding the danger of instability is to modify the delta rule of Eq. (6.13) by including a momentum term, as shown by

$$\Delta w_{ji}(n) = \alpha \, \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n) \qquad 6.31$$

where $\alpha$ is usually a positive number called the *momentum constant*. It controls the feedback loop acting around $\Delta w_{ji}(n)$. Equation (6.31) is the *generalized delta rule*.

In order to see the effect of the sequence of pattern presentations on the synaptic weights due to the momentum constant $\alpha$ constant, we rewrite Eq. (6.31) as a time series with index t. The index t goes from the initial time 0 to the current time *n*. equation (6.31)

may be viewed as a first-order difference equation in the weight correction $\Delta w_{ji}(n)$. Solving this equation for $\Delta w_{ji}(n)$ we have

$$\Delta w_{ji}(n) = \eta \sum_{t=0}^{n} \alpha^{n-t} \delta_j(t) y_i(t) \qquad 6.32$$

which represents a time series of length $n+1$. From equations (6.11) and (6.14) we note the product $\delta_j(t)y_i(t)$ is equal to $\partial \boldsymbol{\varepsilon}(\mathbf{n}) / \partial w_{ji}(n)$. Accordingly, we may rewrite Eq. (6.32) in the equivalent form

$$\Delta w_{ji}(n) = - \eta \sum_{t=0}^{n} \alpha^{n-t} \partial \boldsymbol{\varepsilon}(\mathbf{n}) / \partial w_{ji}(n) \qquad 6.33$$

Based on this relation, we may make the following insightful observations:

1. The current adjustment $\Delta w_{ji}(n)$ represents the sum of an exponentially weighted time series. For the time series to be *convergent*, the momentum constant must be restricted to the range $0 \le | \alpha | < 1$. When $\alpha$ is zero, the back-propagation algorithm operates without momentum. Also the momentum constant $\alpha$ can be positive or negative, although it is unlikely that a negative $\alpha$ would be used in practice.
2. When the partial derivative $\partial \boldsymbol{\varepsilon}(\mathbf{n})/\partial w_{ji}(n)$ has the same algebraic sign on consecutive iterations, the exponentially weighted sum $\Delta w_{ji}(n)$ grows in magnitude, and so the weight $w_{ji}(n)$ is adjusted by a large amount. The inclusion of momentum in the back-propagation algorithm tends to *accelerate descent* in steady downhill directions.
3. When the partial derivative $\partial \boldsymbol{\varepsilon}(\mathbf{n})/\partial w_{ji}(n)$ has opposite signs on consecutive iterations, the exponentially weighted sum $\Delta w_{ji}(n)$ shrinks in magnitude, so the weight $w_{ji}(n)$ is adjusted by a small amount. The inclusion of momentum in the back-propagation algorithm has a *stabilizing effect* in directions that oscillate in sign.

The incorporation of momentum in the back-propagation algorithm represents a minor modification to the weight update, yet it may have some beneficial effects on the learning behavior of the algorithm. The momentum term may also have the benefit of preventing the learning process from terminating in a shallow local minimum on the error surface.

## 6.2 Method of conjugate gradients

The reason why the method of Steepest Descent converges slowly is that it has to take a right angle turn after each step, and consequently search in the same direction as earlier steps. The method of *Conjugate Gradients* is an attempt to mend this problem by *learning* from experience. *Conjugancy* means two unequal vectors, $d_i$ and $d_j$, are orthogonal with respect to any symmetric positive definite matrix.

$$d^T_i \cdot Q \cdot d_j = 0 \qquad 6.34$$

This can be looked upon as a generalization of orthogonality, for which Q is the unity matrix. The idea is to let each search direction $d_i$ be dependent on all the other directions searched to locate the minimum of *f(x)*. A set of such search directions is referred to as a Q-orthogonal, or conjugate, set, and it will take a positive definite *n*-dimensional quadratic function to its minimum point in, at most, *n* exact linear searches. This method is often referrred to as *conjugate directions*, and a short description follows:

The best way to visualize the working of Conjugate Directions is by comparing the space we are working in with a ``stretched'' space. An example of this ``stretching'' of space is illustrated in Figure 52: (a) demonstrates the shape of the contours of a quadratic function in real space, which are elliptical (for $b \neq 0$). Any pair of vectors that appear perpendicular in this space, would be orthogonal. (b) show the same drawing in a space that are stretched along the eigenvector axes so that the elliptical contours from (a) become circular. Any pair of vectors that appear to be perpendicular in this space, is in fact Q-orthogonal. The search for a minimum of the quadratic functions starts at $x_0$ in Figure 6.2(a), and takes a step in the direction $d_0$ and stops at the point $x_1$. This is a minimum point along that direction, determined the same way as for the Steepest Decent method in the previous section: the minimum along a line is where the directional derivative is zero .The essential difference between the Steepest Descent and the

Conjugate Directions lies in the choice of the next search direction from this minimum point. While the Steepest Descent method would search in the direction $r_1$ in Figure 6.2(a), the Conjugate Direction method would chose $d_1$. How come Conjugate Directions manage to search in the direction that leads us straight to the solution x? The answer is found in Figure 6.2(b): In this stretched space, the direction $d_0$ appears to be a tangent to the now circular contours at the point $x_1$. Since the next search direction $d_1$ is constrained to be Q-orthogonal to the previous, they will appear perpendicular in this modified space. Hence, $d_1$will take us directly to the minimum point of the quadratic function f(x).

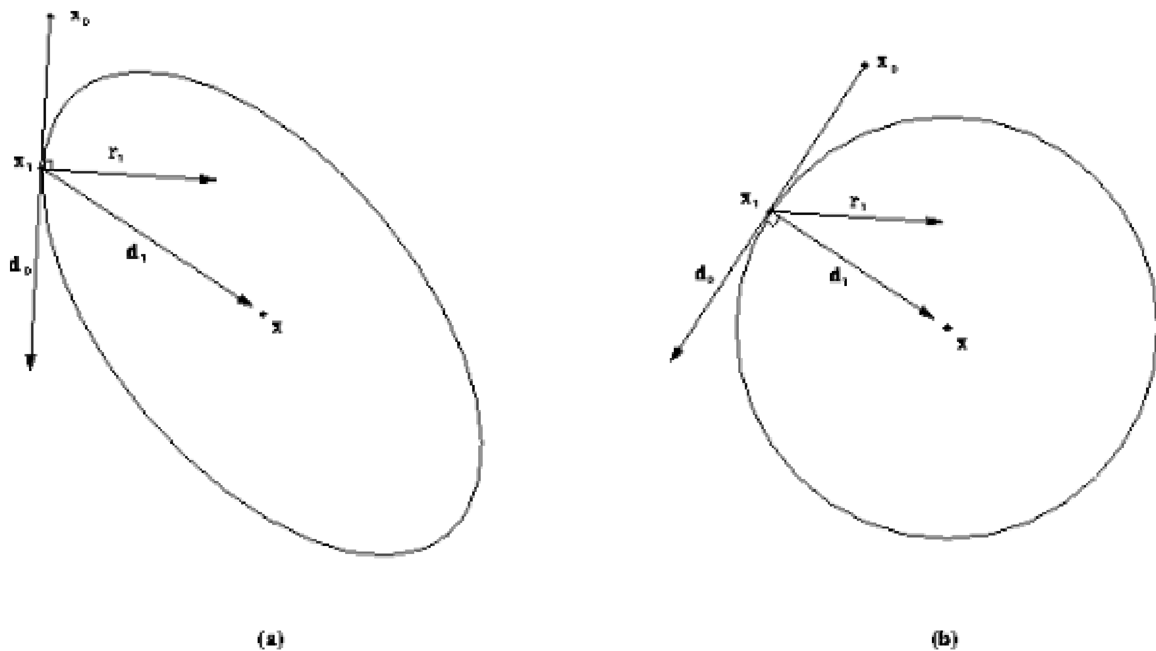



**Figure 6.2:** Optimality of the method of Conjugate Directions. (a) Lines that appear perpendicular are orthogonal. (b) The same problem in a ``stretched" space. Lines that appear perpendicular are Q-orthogonal.

To avoid searching in directions that have been searched before, the Conjugate Direction guarantees that the minimization of $f(x_k)$ along one direction does not ``spoil" the minimization along another; i.e. that after *i* steps, $f(x_i)$ will be minimized over all searched directions. This is essentially what is stated in equation (6.34): A search along $d_i$ has revealed where the gradient is perpendicular to $d_i$, $\nabla f_i \cdot d_i = 0$, and as we now move along some new direction $d_j$, the gradient changes by $\delta(\nabla f) = Q \cdot d_j$. In order to not

interfere with the minimization along $d_i$, we require that the gradient remain perpendicular to $d_i$; i.e. that the change in gradient itself be perpendicular to $d_i$. Hence, we have equation (6.34). We see from Figure 6.2(b), where $d_0$ and $d_1$appear perpendicular because they are Q-orthogonal, that it is clear that $d_1$ must point to the solution x.

The Conjugate Gradients method is a special case of the method of Conjugate Directions, where the conjugate set is generated by the gradient vectors. This seems to be a sensible choice since the gradient vectors have proved their applicability in the Steepest Descent method, and they are orthogonal to the previous search direction. For a quadratic function the procedure is as follows:

Steepest descent:

$$d_0 = - g(x_0) = -g_0 \qquad 6.35$$

Subsequently, the mutually conjugate directions are chosen so that

$$d_{k+1} = -g_{k+1} + \beta_k + d_k \qquad k = 0, 1, \ldots, \qquad 6.36$$

where the coefficient $\beta_k$ is given by, for example, the so called Fletcher-Reeves formula:

$$\beta_k = (g^T_{k+1} \cdot g_{k+1}) / ( g^T_k \cdot g_k ) \qquad 6.37$$

The step length along each direction is given by

$$\lambda_k = (d^T_k \cdot g_k) / (d^T_k \cdot (Q \cdot d_k)) \qquad 6.38$$

When the matrix Q in (6.38) is not known, or is too computationally costly to determine, the stepsize can be found by linear searches. For the Conjugate Gradient method to converge in *n* iterations, these linear searches need to be accurate. Even small deviations can cause the search vectors to lose Q-orthogonality, resulting in the method spending more than *n* iterations in locating the minimum point. In practice, accurate linear searches are impossible, both due to numerical accuracy and limited computing time. The direct use of equation (6.38) will most likely not bring us to the solution in *n* iterations either, the reason being the limited numerical accuracy in the computations which gradually will make the search vectors lose their conjugacy. It should also be mentioned that if the

matrix Qis badly scaled, the convergence will be slowed down considerably, as it was for the steepest descent method.

Even though the Conjugate Gradients method is designed to find the minimum point for simple quadratic functions, it also does the job well for any continuous function f(x)for which the gradient $\nabla f(x)$ can be computed. Equation (6.38) can not be used for non-quadratic functions, so the step length has to be determined by linear searches. The conjugacy of the generated directions may then progressively be lost, not only due to the inaccurate linear searches, but also due to the non-quadratic terms of the functions. An alternative to the Fletcher-Reeves formula that, to a certain extent, deals with this problem, is the Polak-Ribie`re fromula press:

$$\beta_k = (g^T_{k+1} \cdot (g_{k+1} - g_k)) / (g^T_k \cdot g_k) \qquad 6.39$$

The difference in performance between these two formulas is not big though, but the Polak-Ribie`re fromula is known to perfom better for non-quadratic functions. The iterative algorithm for the nonlinear conjugate gradients method for non-quadratic functions using the Polak-Ribie`re fromula is presented in the following table: Regardless of the direction-update formula used, one must deal with the loss of conjugacy that results from the the non-quadratic terms. The Conjugate Gradients method is often employed to problems where the number of variables *n* is large, and it is not unusual for the method to start generating nonsensical and inefficient directions of search after a few iterations. For this reason it is important to operate the method in cycles, with the first step being the Steepest Descent step. One example of a restarting policy is to restart with the steepest descent step after *n* iterations after the preceding restart.

Another practical issue relates to the accuracy of the linear search that is necessary for efficient computation. On one hand, an accurate linear search is needed to limit the loss of direction conjugacy. On the other hand, insisting on very accurate line search can be computationally expensive. To find the best possible middle path, trial and error is needed. The Conjugate Gradients method is apart from being an optimization method, also one of the most prominent iterative method for solving sparse systems of linear equations. It is fast and uses small amounts of storage since it only needs the calculation and storage of the second derivative at each iteration. The latter becomes

significant when $n$ is so large that problems of computer storage arises. But, everything is not shiny; the less similar $f$ is to a quadratic function, the more quickly the search directions lose conjugacy, and the method may in the worst case not converge at all. Although, it is generally to be preferred over the speepesr descent method.

**TABLE 1** Summary of the nonlinear conjugate gradient algorithm for the supervised training of a multilayer perceptron

*Initialization*
Unless prior knowledge on the weight vector **w** is available, choose the initial value **w(0)**

*Computation*

1. For **w(0)**, use back-propagation to compute the gradient vector **g(0)**.
2. Set **d(0)** = **r(0)** = -**g(0)**.
3. At time step $n$, use a line search to find $\eta(n)$ that minimizes $\varepsilon_{av}(\eta)$ sufficiently, representing the cost function $\varepsilon_{av}$ expressed as a function of $\eta$ for fixed values of **w** and **d**.
4. Test to determine if the Euclidean norm of the residual $\mathbf{r}(n)$ has fallen below a specified value, that is, a small fraction of the initial value $\| \mathbf{r}(0) \|$.
5. Update the weight vector:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta(n)\, \mathbf{d}(n)$$

6. Set $\mathbf{r}(n+1) = -\mathbf{g}(n+1)$
7. Use the Polak-Ribie`re method to calculate $\beta_k(n+1)$:

$$\beta_k(n+1) = \max \left\{ \mathbf{r}^T(n+1) \bullet ((\mathbf{r}(n+1) - \mathbf{r}(n)) / ( \mathbf{r}^T(n) \bullet \mathbf{r}(n)), 0 \right\}$$

8. Update the direcion vector:

$$\mathbf{d}(n+1) = \mathbf{r}(n+1) + \beta_k(n+1)\, \mathbf{d}(n)$$

9. Set $n = n+1$, and go back to step 3.

*Stopping criterion*. Terminate the algorithm when the following condition is satisfied:

$$\| \mathbf{r}(n) \| \leq \epsilon \| \mathbf{r}(0) \|$$

where $\epsilon$ is a prescribed small number.

# 7. Digital Image processing

Techniques from statistical pattern recognition have, since the revival of neural networks, obtained a widespread use in digital image processing. Initially, pattern recognition problems were often solved by linear and quadratic discriminants or the (non-parametric) k-nearest neighbor classifier and the Parzen density estimator. In the mid-eighties, the PDP group, together with others, introduced the back-propagation learning algorithm for neural networks. This algorithm for the first time made it feasible to train a non-linear neural network equipped with layers of the so-called hidden nodes. Since then, neural networks with one or more hidden layers can, in theory, be trained to perform virtually any regression or discrimination task. Moreover, no assumptions are made as with respect to the type of underlying (parametric) distribution of the input variables, which may be nominal, ordinal, real or any combination hereof.

Vis-à-vis ANN based image processing; there are two central questions that need to be answered:

1. What are major applications of neural networks in image processing now and in the nearby future?
2. Which are the major strengths and weaknesses of neural networks for solving image processing tasks?

There is a two-dimensional taxonomy for image processing techniques. This taxonomy establishes a framework in which the advantages and unresolved problems can be structured in relation to the application of neural networks in image processing.

## 7.1 Taxonomy for image processing algorithms

Traditional techniques from statistical pattern recognition like the Bayesian discriminant and the Parzen windows were popular until the beginning of the 1990s. Since then, neural networks (ANNs) have increasingly been used as an alternative to classic pattern classifiers and clustering techniques. Non-parametric feed-forward ANNs quickly turned out to be attractive trainable machines for feature-based segmentation and object recognition. When no gold standard is available, the self-organizing feature map (SOM) is an interesting alternative to supervised techniques. It may learn to discriminate, e.g.,

different textures when provided with powerful features. The current use of ANNs in image processing exceeds the aforementioned traditional applications. The role of feed-forward ANNs and SOMs has been extended to encompass also low-level image processing tasks such as noise suppression and image enhancement. Hopfield ANNs were introduced as a tool for finding satisfactory solutions to complex (NP-complete) optimization problems. This makes them an interesting alternative to traditional optimization algorithms for image processing tasks that can be formulated as optimization problems.

The following distinction is made between steps in the image processing chain (see Fig. 1):

1. Preprocessing/filtering: Operations that give as a result a modified image with the same dimensions as the original image (e.g., contrast enhancement and noise reduction).
2. Data reduction/feature extraction: Any operation that extracts significant components from an image (window). The number of extracted features is generally smaller than the number of pixels in the input window.
3. Segmentation: Any operation that partitions an image into regions, which are coherent with respect to some criterion. One example is the segregation of different textures.
4. Object detection and recognition: Determining the position and, possibly, also the orientation and scale of specific objects in an image, and classifying these objects
5. Image understanding: Obtaining high level (semantic) knowledge of what an image shows.
6. Optimization: Minimization of a criterion function which may be used for, e.g., graph matching or object delineation.

Optimization techniques are not seen as a separate step in the image processing chain but as a set of auxiliary techniques, which support the other steps.

Besides the actual task performed by an algorithm, its processing capabilities are partly determined by the abstraction level of the input data. We distinguish between the following abstraction levels:

1. **Pixel level**: The intensities of individual pixels are provided as input to the algorithm.
2. **Local feature** level: A set of derived, pixel-based features constitute the input.
3. **Structure (edge) level**: The relative location of one or more perceptual features (e.g., edges, corners, junctions, surfaces, etc.).
4. **Object level**: Properties of individual objects.
5. **Object set level**: The mutual order and relative location of detected objects.
6. **Scene characterization**: A complete description of the scene possibly including lighting conditions, context, etc.

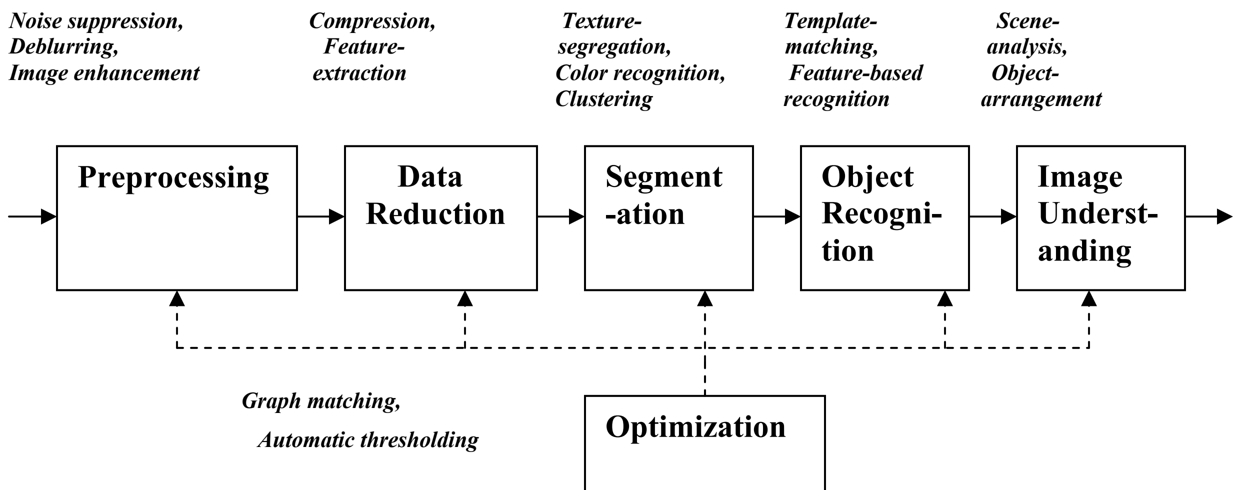



Fig. 7.1; the image processing chain containing the five different tasks: preprocessing, data reduction, segmentation, object recognition and image understanding. Optimization techniques are used as a set of auxiliary tools that are available in all steps of the image processing chain.

## 7.2 Exploring the image processing chain

In this section, the six tasks in the image processing chain will be discussed in detail.

### 7.2.1 Preprocessing

The first step in the image processing chain consists of preprocessing. Loosely defined, by preprocessing we mean any operation of which the input consists of sensor data, and of which the output is a full image. Preprocessing operations generally fall into one of three categories: image reconstruction (to reconstruct an image from a number of sensor measurements), image restoration (to remove any aberrations introduced by the sensor, including noise) and image enhancement (accentuation of certain desired features, which may facilitate later processing steps such as segmentation or object recognition).

#### 7.2.1.1 Image reconstruction

Image reconstruction problems often require quite complex computations and a unique approach is needed. For example, an ADALINE network can be to perform an electrical impedance tomography (EIT) reconstruction, i.e., a reconstruction of a 2D image based on 1D measurements on the circumference of the image, or a modified Hopfield network can be trained to perform the inverse Radon transform (e.g., for reconstruction of computerized tomography images).

#### 7.2.1.2 Image restoration

The majority of applications of ANNs in preprocessing can be found in image restoration. In general, one wants to restore an image that is distorted by the (physical) measurement system. The system might introduce noise, motion blur, out-of-focus blur, distortion caused by low resolution, etc. Restoration can employ all information about the nature of the distortions introduced by the system, e.g., the point spread function. The restoration problem is ill-posed because conflicting criteria need to be fulfilled: resolution versus smoothness. In the most basic image restoration approach, noise is removed from an image by simple filtering. For instance, a regression feed-forward network in a convolution-like way can be used to suppress noise (with a 5x5 pixel window as input and one output node). Another interesting ANN architecture is the generalized adaptive neural filter (GANF) which has been used for noise suppression. A GANF consists of a

set of neural operators, based on stack filters that use binary decompositions of grey-value data.

Traditional methods for more complex restoration problems such as deblurring and diminishing out-of-focus defects are maximum a posteriori estimation (MAP) and regularization. Applying these techniques entails solving high-dimensional convex optimization tasks. The objective functions of MAP estimation or the regularization problem can both be mapped onto the energy function of the Hopfield network. Often, mapping the problem turned out to be difficult, so in some cases the network architecture had to be modified as well.

#### 7.2.1.3 ANNs in preprocessing

There seem to be three types of problems in preprocessing (unrelated to the three possible operation types) to which ANNs can be applied:

- optimization of an objective function defined by a traditional preprocessing problem;
- approximation of a mathematical transformation used for image reconstruction , e.g., by regression;
- mapping by an ANN trained to perform a certain task, usually based directly on pixel data (neighborhood input, pixel output).

To solve the first type of problems, traditional methods for optimization of some objective function may be replaced by a Hopfield network. For the approximation task, regression (feed-forward) ANNs could be applied. Although for some applications such ANNs were indeed successful, it would seem that these applications call for more traditional mathematical techniques, because a guaranteed (worst-case) performance is crucial in preprocessing.

In several other applications, regression or classification (mapping) networks were trained to perform image restoration or enhancement directly from pixel data. A remarkable finding was that non-adaptive ANNs (e.g., CNNs) were often used for preprocessing. Secondly, when networks were adaptive, their architectures usually

differed much from those of the standard ANNs: prior knowledge about the problem was used to design the networks that were applied for image restoration or enhancement (e.g., by using shunting mechanisms to force a feed-forward ANN to make binary decisions). The interest in non-adaptive ANNs indicates that the fast, parallel operation and the ease with which ANNs can be embedded in hardware may be important criteria when choosing for a neural implementation of a specific preprocessing operation. However, the ability to learn from data is apparently of less importance in preprocessing. While it is relatively easy to construct a linear filter with a certain, desired behavior, e.g., by specifying its frequency profile, it is much harder to obtain a large enough data set to learn the optimal function as a high-dimensional regression problem. This holds especially when the desired network behavior is only critical for a small subset of all possible input patterns (e.g., in edge detection). Moreover, it is not at all trivial to choose a suitable error measure for supervised training, as simply minimizing the mean squared error might give undesirable results in an image processing setting.

An important caveat is that the network parameters are likely to become tuned to one type of image (e.g., a specific sensor, scene setting, scale, etc.), which limits the applicability of the trained ANN. When the underlying conditional probability distributions, $p(\mathbf{x}|\omega_j)$ or $p(\mathbf{y}|\mathbf{x})$, change, the classification or regression network-like all statistical models-needs to be retrained.

### 7.2.2 Data reduction and feature extraction

Two of the most important applications of data reduction are image compression and feature extraction. In general, an image compression algorithm, used for storing and transmitting images, contains two steps: encoding and decoding. For both these steps, ANNs have been used. Feature extraction is used for subsequent segmentation or object recognition. The kind of features one wants to extract often correspond to particular geometric or perceptual characteristics in an image (edges, corners and junctions), or application dependent ones, e.g., facial features.

#### 7.2.2.1 Image compression applications

Two different types of image compression approaches can be identified: direct pixel-based encoding/decoding by one ANN and pixel-based encoding/decoding based on a modular approach. Different types of ANNs have been trained to perform image compression: feed-forward networks, SOMs, adaptive fuzzy leader clustering (a fuzzy ART-like approach), a learning vector quantifier and a radial basis function network. Auto-associator networks have been applied to image compression where the input signal was obtained from a convolution window. These networks contain at least one hidden layer, with fewer units than the input and output layers. The network is then trained to recreate the input data. Its bottle-neck architecture forces the network to project the original data onto a lower dimensional (possibly non-linear) manifold from which the original data should be predicted.

Other approaches rely on a SOM, which after training acts as a code book. The most advanced approaches are based on specialized compression modules. These approaches either combine different ANNs to obtain the best possible image compression rate or they combine more traditional statistical methods with one or more ANNs. ANN approaches have to compete with well-established compression techniques such as JPEG, which should serve as a reference. The major advantage of ANNs is that their parameters are adaptable, which may give better compression rates when trained on specific image material. However, such a specialization becomes a drawback when novel types of images have to be compressed.

#### 7.2.2.2 Feature extraction applications

Feature extraction can be seen as a special kind of data reduction of which the goal is to first a subset of informative variables based on image data. Since image data are by nature very high dimensional, feature extraction is often a necessary step for segmentation or object recognition to be successful. Besides lowering the computational cost, feature extraction is also a means for controlling the so-called curse of dimensionality. When used as input for a subsequent segmentation algorithm, one wants to extract those features that preserve the class separability well.

There is a wide class of ANNs that can be trained to perform mappings to a lower-dimensional space. A well-known feature-extraction ANN is a neural implementation of a one-dimensional principal component analysis. It can be proven that training three-layer auto-associator networks corresponds to applying PCA to the input data. Auto-associator networks with five layers can be shown to be able to perform non-linear dimensionality reduction (i.e., finding principal surfaces). It is also possible to use a mixture of linear subspaces to approximate a non-linear subspace.

Another approach to feature extraction is first to cluster the high-dimensional data, e.g., by a SOM, and then use the cluster centers as prototypes for the entire cluster. Usually, neural-network feature extraction is performed for:

- subsequent automatic target recognition in remote sensing (accounting for orientation) and character recognition;
- subsequent segmentation of food images and of magnetic resonance (MR) images;
- finding the orientation of objects (coping with rotation);
- finding control points of deformable models;
- clustering low-level features found by the Gabor filters in face recognition and wood defect detection;
- subsequent stereo matching;
- clustering the local content of an image before it is encoded.

In most applications, the extracted features are used for segmentation, image matching or object recognition. For (anisotropic) objects occurring at the same scale, rotation causes the largest amount of intra-class variation. Some feature extraction approaches were designed to cope explicitly with (changes in) orientation of objects.

It is important to make a distinction between application of supervised and unsupervised ANNs for feature extraction. For a supervised auto-associator ANN, the information loss implied by the data reduction can be measured directly on the predicted output variables, which is not the case for unsupervised feature extraction by the SOM. Both supervised and unsupervised ANN feature extraction methods have advantages compared to traditional techniques such as PCA. Feed-forward ANNs with several

hidden layers can be trained to perform non-linear feature extraction, but lack a formal, statistical basis.

### 7.2.3 Image segmentation

Segmentation is the partitioning of an image into parts that are coherent according to some criterion. When considered as a classification task, the purpose of segmentation is to assign labels to individual pixels or voxels. Some neural-based approaches perform segmentation directly on the pixel data, obtained either from a convolution window (occasionally from more bands as present in, e.g., remote sensing and MR images), or the information is provided to a neural classi1er in the form of local features.

#### 7.2.3.1 Segmentation based on pixel data

Many ANN approaches have been presented that segment images directly from pixel or voxel data. Several different types of ANNs have been trained to perform pixel-based segmentation: feed-forward ANNs, SOMs, Hopfield networks, probabilistic ANNs, radial basis function networks, CNNs, constraint satisfaction ANNs and RAM-networks. A self-organizing architecture with fuzziness measures can also be used.

Hierarchical segmentation approaches have been designed to combine ANNs on different abstraction levels. The guiding principles behind hierarchical approaches are specialization and bottom–up processing: one or more ANNs are dedicated to low level feature extraction/segmentation, and their results are combined at a higher abstraction level where another (neural) classifier performs the final image segmentation. Reddick et al. developed a pixel-based two-stage approach where a SOM is trained to segment multispectral MR images. The segments are subsequently classified into white matter, grey matter, etc., by a feed-forward ANN. Non-hierarchical, modular approaches have also been developed ?connecting edges and lines.

In general, pixel-based (often supervised) ANNs have been trained to classify the image content based on:

- texture
- a combination of texture and local shape

ANNs have also been developed for pre- and post processing steps in relation to segmentation, e.g., for:

- delineation of contours
- connecting edge pixels
- identification of surfaces
- deciding whether a pixel occurs inside or outside a segment
- defuzzifying the segmented image;

and for:

- clustering of pixels
- motion segmentation

In most applications, ANNs were trained as supervised classifiers to perform the desired segmentation. One feature that most pixel-based segmentation approaches lack is a structured way of coping with variations in rotation and scale. This shortcoming may deteriorate the segmentation result.

### 7.2.3.2 Segmentation based on features

Several feature-based approaches apply ANNs for segmentation of images. Different types of ANNs have been trained to perform feature-based image segmentation: feed-forward ANNs, recursive networks, SOMs, variants of radial basis function networks and CNNs, Hopfield ANNs, principal component networks and a dynamic ANN.
Hierarchical network architectures have been developed for optical character recognition and for segmentation of range images. Feature-based ANNs have been trained to segment images based on the differences in:

- texture;
- a combination of texture and local shape.

Besides direct classification, ANNs have also been used for:

- estimation of ranges;
- automatic image thresholding by annealing or by mapping the histogram;
- estimation of the optical flow;

- connecting edges and lines;
- region growing.

A segmentation task that is most frequently performed by feature-based ANNs is texture segregation, which is typically based on:

- co-occurrence matrices;
- wavelet features;
- multiresolution features extracted from the Gabor wavelets;
- spatial derivatives computed in the linear scale-space.

The Gabor and wavelet-based features, and features extracted from the linear scale-space provide information at several scales to the classifier, which, however, needs to cope explicitly with variations in scale. As with respect to orientation, the Gabor and wavelet-based approaches are, in general, sensitive to horizontal, vertical and diagonal features. These three directions can be combined into a local orientation measure such that rotation invariance is obtained. The scale-space features can be reduced to a few invariants that are indeed rotation invariant. The generalized co-occurrence matrices cope with variations in orientation by averaging over four orthogonal orientations. Scale can also be taken into account by varying the distance parameter used to compute the co-occurrence matrix.

#### 7.2.3.3 Open issues in segmentation by ANNs

Three central problems in image segmentation by ANNs are: how to incorporate context information, the inclusion of (global) prior knowledge, and the evaluation of segmentation approaches. Context information can be obtained from, for instance, multiscale wavelet features or from features derived from the linear scale space (computed at a coarse scale). How context information can best be incorporated, is an interesting issue for further research.

A caveat is how to obtain a gold standard for the (in most cases supervised) segmentation algorithms. In general, the true class membership of the pixels/voxels in the

training set is known with varying degrees of confidence. This problem can be addressed by letting an expert demarcate the inner parts of areas with a similar (coherent) texture but leaving the transition areas unclassified. Certainly, intra- and inter-observer variability needs to be assessed thoroughly (e.g., by the kappa statistic) before suitable training and test images can be compiled. Even when a reliable gold standard is available, objective performance assessment entails more than simply computing error rates on novel test images. There is not yet a single measure capable of unequivocally quantifying segmentation quality. Besides statistical performance aspects such as coverage, bias and dispersion, desirable properties such as within-region homogeneity and between-region heterogeneity are also important.

### 7.2.4 Object recognition

Object recognition consists of locating the positions and possibly orientations and scales of instances of objects in an image. The purpose may also be to assign a class label to a detected object. In most applications, ANNs have been trained to locate individual objects based directly on pixel data. Another less frequently used approach is to map the contents of a window onto a feature space that is provided as input to a neural classifier.

#### 7.2.4.1 Based on pixel data

Among the ANN approaches developed for pixel-based object recognition, several types of ANNs can be distinguished: feed-forward-like ANNs, variants using weight sharing, recurrent networks, the ART networks, (evolutionary) fuzzy ANNs, bi-directional auto-associative memories, the Neocognitron and variants, piecewise-linear neural classifiers, higher-order ANNs, and Hopfield ANNs. Besides, interesting hardware ANNs have been built for object recognition: the RAM network and optical implementations. Finally, SOMs are occasionally used for feature extraction from pixel data; the output of the map is then propagated to a (neural) classifier.

Several novel network architectures have been developed specifically to cope with concomitant object variations in position, (in-plane or out-of-plane) rotation and scale (in one case, an approach has been developed that is invariant to changes in

illumination). It is clear that a distinction needs to be made between invariant recognition in 2D (projection or perspective) images and in 3D volume images. An interesting approach that performs object recognition, which is invariant to 2D translations in-plane rotation and scale, is the neurally inspired what-and-where filter. It combines a multiscale oriented filter bank (what) with an invariant matching module (where). Other approaches rely on learning the variations explicitly by training. Egmont-Petersen and Arts built a statistical intensity model of the object that should be detected. The convolution ANN was trained using synthetic images of the (modeled) object with randomly chosen orientations.

Clearly, when object recognition is performed by teaching a classifier to recognize the whole object from a spatial pattern of pixel intensities, the complexity of the classifier grows exponentially with the size of the object and with the number of dimensions (2D versus 3D). An interesting approach that circumvents this problem is iterative search through the image for the object centre. The output of the ANN is the estimated displacement vector to the object centre. Depending on the contents of the scene, even context information may be required before the objects of interest can be recognized with confidence. The incorporation of context information may again lead to a large number of extra parameters and thereby a more complex classifier. To cope with this problem the so-called multiresolution approaches have been developed, which combine the intensities from pixels located on different levels of a pyramid but centered on around the same location. This provides the classifier with context information, but a combinatorial explosion in the number of parameters is circumvented. Still, variations in scale have to be learned explicitly by the classifier. A disadvantage of ANN pyramid approaches is that they sample the scale space coarsely as the resolution is reduced with a factor two at each level in the pyramid (in, e.g., the linear scale space , scale is a continuous variable). A special type of ANN that incorporates the scale information directly in a pyramidal form is the so-called higher-order ANN. This network builds up an internal scale-space-like representation by what is called coarse coding. However, higher-order ANNs need to learn variations in scale explicitly too. They should be used with caution because the coarse coding scheme may lead to aliasing, as the high-resolution images are not blurred before computing the coarser image at the next level.

Recurrent ANNs (with feed-back loops ) can be used to develop special approaches for object recognition [179]. The added value of recurrent network architecture lies in its memory: the current state contains information about the past, which may constitute valuable context information. For instance, a recurrent network can be developed to perform a convolution with an image in order to detect oil spills. The recurrence principle introduces averaging, which can give a more robust performance. Several of the approaches for object detection and classification operate on binary images. Although binarisation simplifies the recognition problem considerably, it generally decreases the recognition performance of an ANN.

#### 7.2.4.2 Based on features

Several neural-network approaches have been developed for feature-based object recognition including: feed-forward ANNs, Hopfield ANNs, fuzzy ANN and RAM ANNs. SOMs are occasionally used to perform feature extraction prior to object recognition, although SOMs have also been trained to perform object classification. The smaller variety of neural architectures developed for feature-based object recognition compared to the pixel-based approaches discussed in the previous section, reflects the fact that most effort is focused on developing and choosing the best features for the recognition task. Common for many feature-based approaches is that variations in rotation and scale are coped with by the features, e.g., statistical moments. A certain amount of noise will influence the computed features and deteriorate the recognition performance. So the major task of the subsequent classifier is to filter out noise and distortions propagated by the features. Moreover, when the object to be detected is large and needs to be sampled densely, feature extraction is inevitable. Otherwise, a neural classifier will contain so many parameters that a good generalization will be impeded.

In general, the types of features that are used for object recognition differ from the features used by the neural-based segmentation approaches already discussed. For object recognition, the features typically capture local geometric properties:

- points with a high curvature on the detected object contours;
- (Gabor) filter banks including wavelets;

- dedicated features: stellate features and OCR features;
- projection of the (sub)image onto the x- and y-axis;
- principal components obtained from the image;
- (distances to) feature space trajectories, which describe objects in all rotations, translations or scales;

Multiresolution approaches have also been developed for object recognition based on features from:

- the linear scale-space;
- the Gauss pyramid;
- the Laplace pyramid.

Which set of features is best suited for a particular recognition task, depends on the variations among the objects (and of the background) with respect to position, (in-plane) orientation and scale. Knowledge of the degrees of freedom the approach has to cope with is needed for choosing a suited set of features.

#### 7.2.4.3 Using pixels or features as input?

Most ANNs that have been trained to perform image segmentation or object recognition obtain as input either pixel/voxel data (input level A) or a vector consisting of local, derived features (input level B). For pixel- and voxel-based approaches, all information (within a window) is provided directly to the classifier. The perfect (minimal error-rate) classifier should, when based directly on pixel data, be able to produce the best result if the size of the window is comparable to that of the texture elements (texels) or the window encompasses the object and the (discriminative) surrounding background. When, on the other hand, the input to the classifier consists of a feature vector, the image content is always compressed. Whether sufficient discriminative information is retained in the feature vector, can only be resolved experimentally.

Two-dimensional image modalities such as radiography, 2D ultrasound and remote sensing often exhibit concomitant variations in rotation and scale. If such invariances are not built into a pixel-based ANN, careful calibration (estimation of the

physical size of a pixel) and subsequent rescaling of the image to a standard resolution are required steps to ensure a confident result. When only rotations occur, features obtained from a polar mapping of the window may ensure a good segmentation or detection result. In many applications, however, calibration is unfeasible and 2D/3D rotation and scale invariance needs to be incorporated into the ANN. For pixel-based approaches, invariance can be either built directly into the neural classifier (e.g., using weight sharing or by taking symmetries into account), or the classifier has to be trained explicitly to cope with the variation by including training images in all relevant orientations and scales. A major disadvantage of these approaches is that object variations in rotation and scale have to be learned explicitly by the classifier (translation can usually be coped with by convolution). This again calls for a very large, complete training set and a classifier that can generalize well. Model-based approaches have been presented that can generate such a complete training set, see the discussion above. How to design robust pixel-based algorithms for segmentation and object recognition that can cope with the three basic affine transforms is a challenging subject for future research.

In situations where many concomitant degrees of freedom occur (2D or 3D rotation, scale, affine greylevel transformations, changes in color, etc.), only feature-based approaches may guarantee that the required invariance is fully obtained. It is clear that when variations in orientation and scale occur and reliable calibration is unfeasible, an ANN based on invariant features should be preferred above a pixel-based approach. Another advantage of feature-based approaches is that variations in rotation and scale may remain unnoticed by the user, who may then end up with a poor result. When there is no limited set of images on which an algorithm has to work (e.g., image database retrieval), the more flexible pixel-based methods can prove useful.

The recommendation to prefer feature-based over pixel/voxel-based image processing (when significant variations in rotation and scale actually occur in the image material), puts emphasis on the art of developing and choosing features which, in concert, contain much discriminative power in relation to the particular image processing task. Prior knowledge regarding the image processing task (e.g., invariance) should guide the development and selection of discriminative features. Feature-based classifiers will, in general, be easier to train when the chosen features cope adequately with the degrees of

freedom intrinsic to the image material at hand. The removal of superfluous features is often necessary to avoid the peaking phenomenon and guarantee a good generalization ability of the classifier.

#### 7.2.4.4 Issues in pattern recognition

When trying to solve a recognition problem, one may be faced with several problems that are fundamental to applied statistical pattern recognition: avoiding the curse of dimensionality, selecting the best features and achieving a good transferability. The first problem, the curse of dimensionality, occurs when too many input variables are provided to a classifier or regression function. The risk of ending up with a classifier or regressor that generalizes poorly on novel data increases with the number of dimensions of the input space. The problem is caused by the inability of existing classifiers to cope adequately with a large number of (possibly irrelevant) parameters, a deficiency that makes feature extraction and/or feature selection necessary steps in classifier development. Feature extraction has been discussed in detail in Section 7.2.2. Feature selection is, by virtue of its dependence on a trained classifier, an ill-posed problem. Besides offering a way to control the curse of dimensionality, feature selection also provides insight into the properties of a classifier and the underlying classification problem.

A problem that is especially important in applications such as medical image processing is how to ensure the transferability of a classifier. When trained to classify patterns obtained from one setting with a specific class distribution, $\mathbf{P(\omega j)}$, a classifier will have a poorer and possibly unacceptably low performance when transferred to a novel setting with another class distribution $\mathbf{P'(\omega j)}$. How to cope with varying prior class distributions is a subject for future research. Another problem related to transferability is how to account for changing underlying feature distributions, $\mathbf{p(x|\omega j)}$ or $\mathbf{p(y|x)}$. In general, the parameters of the classifier or regression function need to be re-estimated from a data set that is representative for the novel distribution. This problem is intrinsic to all statistical models as they are based on inductive inference. Note that for a classifier that has been trained, e.g., to recognize objects appearing at a certain scale directly from pixel data, recognition of similar objects at a different scale is equivalent to classifying

patterns from a novel distribution $\mathbf{p'(x|\omega j)}$. Classifiers or regression models that have not been retrained, should catch patterns occurring outside the space spanned by the training cases and leave these patterns unprocessed, thereby avoiding the assignment of "wild-guess" class labels or unreliable prediction of the conditional mean (in regression). Moreover, the question of how to incorporate costs of different misclassifications (again, an important topic in, e.g., medical image processing) or the computational costs of features, is not yet fully answered.

### 7.2.4.5 Obstacles for pattern recognition in image processing

Besides fundamental problems within the field of pattern recognition, other problems arise because statistical techniques are used on image data. First, most pixel-based techniques consider each pixel as a separate random variable. A related problem is how one should incorporate prior knowledge into pattern recognition techniques. Also, the evaluation of image processing approaches is not always straightforward. A challenging problem in the application of pattern recognition techniques on images is how to incorporate context information and prior knowledge about the expected image content. This can be knowledge about the typical shape of objects one wants to detect, knowledge of the spatial arrangement of textures or objects, or prior knowledge of a good approximate solution to an optimization problem. The key to restraining the highly flexible learning algorithms for ANNs lies in the very combination with prior (geometric) knowledge. However, most pattern recognition methods do not even use the prior information that neighboring pixel/voxel values are highly correlated.

This problem can be circumvented by extracting features from images first, by using distance or error measures on pixel data which do take spatial coherency into account, or by designing ANN relations between objects in mind. Context information can also be obtained from the pyramid and scale space approaches discussed in Section 7.4.1. In most applications, prior knowledge is mainly used to identify local features (input level B) that can be used as input to neural classifiers. Fuzzy ANNs may play a special role because they can be initialized with (fuzzy) rules elicited from domain experts. Using prior knowledge to constrain the highly parameterized (neural) classifiers is a scientific challenge. There is a clear need for a thorough validation of image

processing algorithms. Validation and comparison of different algorithms are only possible when a reliable gold standard exists and meaningful (objective) quality measures are available. For example, for object recognition, a gold standard is in most cases easy to obtain. In other applications, different (human) observers may not fully agree about the gold standard (e.g., segmentation of medical images). Even with a reliable gold standard being available, it is clear that performance assessment entails much more than simply computing error rates on novel test images.

Finally, in image processing, classification and regression problems quickly involve a very large number of input dimensions, especially when the algorithms are applied directly to pixel data. This is problematic, due to the curse of dimensionality already discussed. However, the most interesting future applications promise to deliver even more input. Whereas, in applications, ANNs are applied to two-dimensional images, e.g., (confocal) microscopy and CT/MR (medical) imaging are three-dimensional modalities. One way to cope with this increased dimensionality is by feature-based pattern recognition, another way would be to develop architecture that inherently down samples the original image. As it is already mentioned, the search for the optimal set of features that in concert gives the best class separability is a never-ending quest. To avoid such a quest for all kinds of features that capture certain specific aspects in a (sub) image, a general mapping (invariant to changes in position, rotation and scale) of a (sub) image to a manifold subspace should be developed. This will change the focus from selection of individual features to optimization of the sampling density in the invariant space.

### 7.2.5 Image understanding

Image understanding is a complicated area in image processing. It couples techniques from segmentation or object recognition with knowledge of the expected image content. For example, ANNs can be used in combination with background knowledge to classify objects such as chromosomes from extracted structures (input level C) and to classify ships, which were recognized from pixel data (input level A) by an advanced modular approach. In another application, ANNs can be used to analyze camera images for robot control from local features (input level B). Neural (decision) trees, semantic models based on extracted structures (input level C) or neural belief networks can be used to

represent knowledge about the expected image content. This knowledge is then used to restrict the number of possible interpretations of single objects as well as to recognize different configurations of image objects.

A major problem when applying ANNs for high level image understanding is their black-box character. It is virtually impossible to explain why a particular image interpretation is the most likely one. An approach to coping with the black-box problem is to use the generic explanation facility developed for ANNs or to use rule extraction. Another problem in image understanding relates to the level of the input data. When, for instance, seldom occurring features (input level C) or object positions (input level E) are provided as input to a neural classifier, a large number of images are required to establish statistically representative training and test sets. In general, it can be said that image understanding is the most dubious application of ANNs in the image processing chain.

### 7.2.6 Optimization

Some image processing (sub) tasks such as graph and stereo-matching can best be formulated as optimization problems, which may be solved by Hopfield ANNs. In some applications, the Hopfield network obtains pixel-based input (input level A); in other applications the input consists of local features (input level B) or detected structures (typically edges, input level C). Hopfield ANNs have been applied to the following optimization problems:

- segmentation of an image with an intensity gradient by connecting edge pixels (input level A);
- thresholding images by relaxation (input level A);
- two-dimensional and three-dimensional object recognition by (partial) graph matching (input level C);
- establishing correspondence between stereo images based on features (landmarks) and stereo correspondence between line cameras from detected edges;
- approximation of a polygon from detected edge points;
- controlling Voronoi pyramids.

Hopfield ANNs have mainly been applied to segmentation and recognition tasks that are too difficult to realize with conventional neural classifiers because the solutions entail partial graph matching or recognition of three-dimensional objects. Matching and recognition are both solved by letting the network converge to a stable state while minimizing the energy function. It was also shown that iterating the Hopfield network can be interpreted as a form of probabilistic relaxation.

## 8. Edge recognition in shared weight networks

This paper particularly discusses, in detail, the design and implementation of shared weight ANNs for the task of edge recognition. The problem of edge recognition is treated here as a classification problem: the goal is to train an ANN to give high output for image samples containing edges and low output for samples containing uniform regions. This makes it different from edge detection, in which localization of the edge in the sample is important as well. A data set is constructed by drawing edges at $0^o$, $15^o$,. . . ,$345^o$ angles in a 16×16 pixel binary image. In total, 24 edge images are created. An equal number of images just containing uniform regions of background or foreground pixels are then added, giving a total of 48 samples.

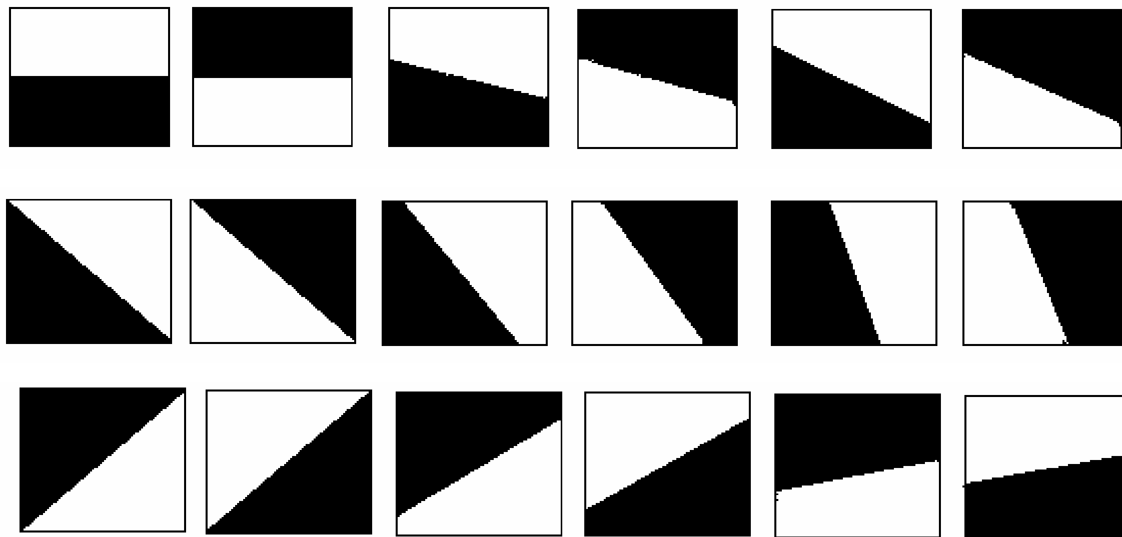


Figure 7.2 the edge samples in the edge data set.

## 8.1 Network architecture

To implement edge recognition in a shared weight ANN, it should consist of at least 3 layers (including the input layer). The input layer contains 16×16 units. The 14×14 unit hidden layer will be connected to the input layer through a 3×3 weight receptive field, which should function as an edge recognition template. The hidden layer should then, using bias, shift the high output of a detected edge into the nonlinear part of the transfer function, as a means of thresholding. Finally, a single output unit is needed to sum all outputs of the hidden layer and rescale to the desired training targets. The architecture described here is depicted in figure 8.1.

This approach consists of two different subtasks. First, the image is convolved with a template (filter) which should give some high output values when an edge is present and low output values overall for uniform regions. Second, the output of this operation is (soft-) thresholded and summed, which is a nonlinear neighborhood operation. A simple summation of the convolved image (which can easily be implemented in a feed-forward ANN) will not do. Since convolution is a linear operation, for any template the sum of a convolved image will be equal to the sum of the input image multiplied by the sum of the template. This means that classification would be based on just the sum of the inputs, which (given the presence of both uniform background and uniform foreground samples, with sums smaller and larger than the sum of an edge image) is not possible. The data set was constructed like this on purpose, to prevent the ANN from finding trivial solutions.

As the goal is to detect edges irrespective of their orientation, a rotation-invariant edge detector template is needed. The first order edge detectors known from image processing literature (Pratt, 1991; Young et al., 1998) cannot be combined into one linear rotation-invariant detector. However, the second order Laplacian edge detector can be. The resulting image processing operation is shown below the ANN in figure 8.1. We have used the architecture just described with double sigmoid transfer functions in order to implement a rotation invariant edge recognizer. The training targets are set to $t = 0.5$ for samples containing an edge and $t = -0.5$ for samples containing uniform regions. The first order edge detectors known from image processing literature (Pratt, 1991; Young et

al., 1998) cannot be combined into one linear rotation-invariant detector. However, the second order Laplacian edge detector can be.

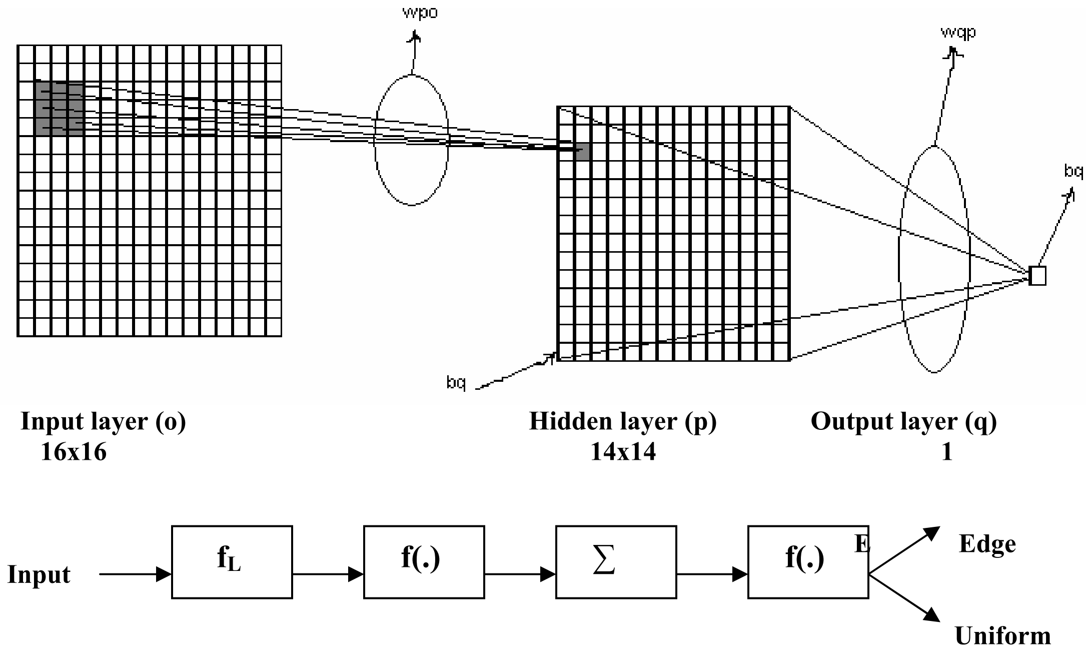



Figure 8.1: A sufficient ANN architecture for edge recognition. Weights and biases for hidden units are indicated by $w^{po}$ and $b^{p}$ respectively. After the ANN architecture, the image processing operation is shown: convolution with the Laplacian template fL, pixel-wise application of the sigmoid f(.), (weighted) summation and another application of the sigmoid.

## 8.2 Training

The network is trained using the conjugate gradient descent (CGD) training algorithm because this algorithm is less prone to finding local minima or diverging than back-propagation, as it uses a line minimization technique to find the optimal step size in each iteration. The method has only one parameter, the number of iterations for which the directions should be kept conjugate to the previous ones and it is set to 10. In general, when an ANN is trained, the training algorithm should always take care of the danger of overtraining: instead of finding templates or feature detectors that are generally applicable, the weights are adapted too much to the training set at hand. However, here the artificial edge data set is constructed specifically to contain all possible edge orientations, so overtraining cannot occur. Therefore, no validation set is used.

## 8.3 Results

When the network was trained, the MSE dropped below to $1 \times 10^{-7}$ after 200 training cycles. After completing training, the network performance was tested and it was capable of recognizing any edge in an image regardless of its orientation. It is also observed that the sequence of presenting the sample data plays a crucial role in network convergence. That means we should not train the network completely with input vectors of one class, and then switch to another class: The network will forget the original training.

## 8.4 Conclusion

One of the major advantages of ANNs is that they are applicable to a wide variety of problems. There are, however, still caveats and fundamental problems that require attention. Some problems are caused by using a statistical, data-oriented technique to solve image processing problems; other problems are fundamental to the way ANNs work.

***Problems with data-oriented approaches*:** A problem in the application of data-oriented techniques to images is how to incorporate context information and prior knowledge about the expected image content. Prior knowledge could be knowledge about the typical shape of objects one wants to detect, knowledge of the spatial arrangement of textures or objects or of a good approximate solution to an optimization problem. According to (Perlovsky, 1998), the key to restraining the highly flexible learning algorithms ANNs are, lies in the very combination with prior knowledge. However, most ANN approaches do not even use the prior information that neighboring pixel values are highly correlated. The latter problem can be circumvented by extracting features from images first, by using distance or error measures on pixel data which do take spatial coherency into account (e.g. Hinton et al., 1997; Simard et al., 1993), or by designing an ANN with spatial coherency (e.g. Le Cun et al., 1989a; Fukushima and Miyake, 1982) or contextual relations between objects in mind. On a higher level, some methods, such as hierarchical object recognition ANNs can provide context information.

In image processing, classification and regression problems quickly involve a very large number of input dimensions, especially when the algorithms are applied directly on pixel data. This is problematic, as ANNs to solve these problems will also grow, which makes them harder to train. However, the most interesting future applications (e.g. volume imaging) promise to deliver even more input. One way to cope with this problem is to develop feature-based pattern recognition approaches; another way would be to design architecture that quickly adaptively down samples the original image. Finally, there is a clear need for thorough validation of the developed image processing algorithms (Haralick, 1994; De Boer and Smeulders, 1996). Unfortunately, only few of the publications about ANN applications ask the question whether an ANN really is the best way of solving the problem. Often, comparison with traditional methods is neglected.

***Problems with ANNs*:** Several theoretical results regarding the approximation capabilities of ANNs have been proven. Although feed-forward ANNs with two hidden layers can approximate any (even discontinuous) function to an arbitrary precision, theoretical results on, e.g., convergence, are lacking. The combination of initial parameters, topology and learning algorithm determines the performance of an ANN after its training has been completed. Furthermore, there is always a danger of overtraining an ANN, as minimizing the error measure occasionally does not correspond to finding a well-generalizing ANN. Another problem is how to choose the best ANN architecture. Although there is some work on model selection (Fogel, 1991; Murata et al., 1994), no general guidelines exist, which guarantee the best trade-off between model bias and variance for a particular size of the training set. Training unconstrained ANNs using standard performance measures such as the mean squared error might even give very unsatisfying results. This, we assume, is the reason why in a number of applications, ANNs were not adaptive at all or heavily constrained by their architecture. ANNs suffer from what is known as the black-box problem: the ANN, once trained, might perform well but offers no explanation on how it works. That is, given any input a corresponding output is produced, but it cannot be easily explained why this decision was reached, how reliable it is, etc. In some image processing applications, e.g., monitoring of (industrial) processes, electronic surveillance, biometrics, etc. a measure of the reliability is highly

necessary to prevent costly false alarms. In such areas, it might be preferable to use other, less well performing methods that do give a statistically profound measure of reliability.

As it was mentioned, this paper specifically details the implementation of ANN based edge recognition, and the results show that ANNs can indeed be used as edge detectors. However, the presence of receptive fields in the architecture in itself does not guarantee that shift-invariant feature detectors will be found. Also, the mere fact that performance is good (i.e., the MSE is low) does not imply that such a feature extraction process is used. When the ANN was further restricted by sharing biases and other weights convergence became a problem. The explanation for this is that the optimal weight set is rather special in ANN terms, as the template has to have a zero DC component (i.e., its weights have to add up to zero). Although this seems to be a trivial demand, it has quite large consequences for ANN training. Optimal solutions correspond to a range of interdependent weights, which will result in long, narrow valleys in the MSE “landscape”. A small perturbation in one of the template weights will have large consequences for the MSE. Simple gradient descent algorithms such as back-propagation will fail to find these valleys, so the line-optimization step used by CGD becomes crucial.

Another important observation was that there is a trade-off between complexity and the extent to which experiments are true-to-life on the one hand, and the possibility of interpretation on the other. This effect might be referred to as a kind of ANN interpretability trade-off. If an unrestricted ANN is trained on a real-world data set, the setup most closely resembles the application of ANNs in everyday practice. However, the subtleties of the data set and the many degrees of freedom in the ANN prevent gaining a deeper insight into the operation of the ANN. On the other side, once an ANN is restrained, e.g. by sharing or removing weights, lowering the number of degrees of freedom or constructing architectures only specifically applicable to the problem at hand, the situation is no longer a typical one. The ANN may even become too constrained to learn the task at hand. The same holds for editing a data set to influence its statistics or to enhance more preferable features with regard to ANN.

# Appendix

```
//*********************************************************************
//                       THE HEADER FILE                              *
//*********************************************************************


#define NEURONHIGH   1.0          //neuron's high output value

#define NEURONLOW    0.0          //neuron's low output value

#define TRUE         1

#define FALSE        0

#define MAXLAYERS        3

#define ALPHA            0.7

#define EPOCH_SIZE       48

#define MAX_ITERATION    200

#define ERROR_THRESHOLD  0.001

#define EPSILON          1e-4

struct WEIGHTIMAGE
{
      double data;          //weight value

      int sneuron;          //source neuron for this weight

      int dneuron;          //destination neuron for this weight

      WEIGHTIMAGE *next;
};

struct NETRESULTS
{
      int index;            //neurons identification number

      double value;         //neurons output value

      char character;       //char representation of digit

};

class CNetFileData
{
      private:

            double temperature;      //neurons temperature

            double threshold;        //neurons firing threshold

            int Nlayers;             //number of layers in the net

            int neurons[MAXLAYERS];  //stores the number of neurons for

                                     //each layer
```

```
        int status;                         //error status (0 = OK)
        WEIGHTIMAGE* weights[MAXLAYERS-1];  //temp weight storage
                                            //area
        void ADDweights(int,int,int,double );
        double GETweights(int,int,int);

    public:
        CNetFileData();
        int setupNet(char*, int);

        double GetTemp()
        {
            return temprature;
        }

        double GetThresh()
        {
            return threshold;
        }

        int GetNlayers()
        {
            return Nlayers;
        }

        int GetLayerSize(int layer)
        {
            return neurons[layer];
        }

        double GetWeight(int l, int d, int s)
        {
            return GETweights(l-1,d,s);
        }
        int GetStatus()
        {
            return status;
        }

};


//*****************************************************************
//                  THE CWeight CLASS                            *
//*****************************************************************


class CNeuron;                      //forward reference

class CWeight
{
private:
    CNeuron* SCRneuron;              //source neuron for this weight
    double WtVal;                    //magnitude of weight
```

```
	CWeight* next;				//hook so weights can be list
						//members
public:
	CWeight(double w, CNeuron* SN)
	{
		next = (CWeight*)NULL;
		SCRneuron = SN;
		WtVal = w;
	}

	CNeuron* GetSCRNeuron()
	{
		return SCRneuron;
	}
	double getWeight()
	{
		return WtVal;
	}

	void SetNext(CWeight* W)
	{
		next = W;
	}
	CWeight* GetNext()
	{
		return next;
	}

	void UpdateWeight(double wgt)
	{
		WtVal = wgt;
	}

};


//*********************************************************************
//                   THE CLeftWeight CLASS                            *
//*********************************************************************


class CLeftWeight
{
private:
	CNeuron* lSCRneuron;			//source neuron for this weight
	double lWtVal;				//magnitude of weight
	CLeftWeight *lnext;			//hook so weights can be list
						//members
```

```
public:
      CLeftWeight(double w, CNeuron *SN)
      {
            lnext = (CLeftWeight*)NULL;
            lSCRneuron = SN;
            lWtVal = w;
      }

      CNeuron* GetlSCRNeuron()
      {
            return lSCRneuron;
      }

      double getlWeight()
      {
            return lWtVal;
      }

      void SetLeftNext(CLeftWeight* W)
      {
            lnext = W;
      }
      CLeftWeight* GetLeftNext()
      {
            return lnext;
      }

      void lUpdateWeight(double wgt)
      {
            lWtVal = wgt;
      }

};


//*********************************************************************
//                    THE CRightWeight CLASS
//*********************************************************************


class CRightWeight
{
private:
      CNeuron* rSCRneuron;              //source neuron for this weight
      double rWtVal;                    //magnitude of weight
      CRightWeight *rnext;              //hook so weights can be list
                                        //members

public:
      CRightWeight(double w, CNeuron *SN)
      {
            rnext = (CRightWeight*)NULL;
            rSCRneuron = SN;
            rWtVal = w;
      }
```

```
      CNeuron* GetrSCRNeuron()
      {
            return rSCRneuron;
      }

      double getrWeight()
      {
            return rWtVal;
      }

      void SetRightNext(CRightWeight* W)
      {
            rnext = W;
      }

      CRightWeight* GetRightNext()
      {
            return rnext;
      }

      void rUpdateWeight(double wgt)
      {
            rWtVal = wgt;
      }

};


//*********************************************************************
//                         THE CNeuron CLASS                         *
//*********************************************************************


class CNeuron
{
private:
      static double temprature; //holds a single copy for all neurons
      static double threshold;  //holds a single copy for all neurons
      double NET;               //holds sum of products
      double bsweight;          //holds the bias weight
      double localgrad;         //holds value of the local gradient
      double error;             //holds error value
      int id;                   //holds a neuron identification No.
      double out;               //holds  a neuron output value
      double outprime;          //holds f'(x)
      CWeight* weight1;         //pointer to list of weight(head)
      CWeight* weightL;         //pointer to list of weight(tail)
      CLeftWeight* lweight1;    //pointer to list of weight(head)
      CLeftWeight* lweightL;    //pointer to list of weight(tail)
      CRightWeight *rweight1;   //pointer to list of weight(head)
```

```
        CRightWeight *rweightL;    //pointer to list of weight(tail)
        int BiasFlg;               //1 = Bias neuron, 0 otherwise
        CNeuron *next;             //hook to allow neurons to be a list
                                   //members
        double r;
        double s;
        double rprev;
        double eta;
public:

CNeuron()
        {
                id = 0;
                out = 0;
                outprime = 0;
                NET = 0.0;
                bsweight = 0.01;
                localgrad = 0;
                eta = 0;
                error = 0;
                weight1 = (CWeight*)NULL;
                lweight1 = (CLeftWeight*)NULL;
                rweight1 = (CRightWeight*)NULL;
                next = (CNeuron*)NULL;
        }

        CNeuron(int ident, int bias=0)
        {
                id = ident;
                out = 0;
                outprime = 0;
                NET = 0.0;
                bsweight = 0.01;
                localgrad = 0;
                eta = 0;
                error = 0;
                BiasFlg = bias;
                weight1 = (CWeight*)NULL;
```

```
        lweight1 = (CLeftWeight*)NULL;
        rweight1 = (CRightWeight*)NULL;
        next = (CNeuron*)NULL;
}

void calc(int);
double GetWeight(int);
double GetLeftWeight(int);
double GetRightWeight(int);
CNeuron* GetSRCNeuron(int);
CNeuron* GetlSRCNeuron(int);
CNeuron* GetrSRCNeuron(int);

void SetNext(CNeuron *N)
{
        next = N;
}

void SetLocalgrad1(double err)
{
        localgrad = outprime*err;
}

void SetLocalgrad(double grad)
{
        localgrad = grad;
}

double GetOutPrime()
{
        return outprime;
}

double GetLocalGrad()
{
        return localgrad;
}

CNeuron* GetNext()
{
        return next;
}

void SetWeight(double, CNeuron*);
void SetLeftWeight(double, CNeuron*);
void SetRightWeight(double, CNeuron*);
void CalcLocalgrad();
void UpdateWeight(double, int);
void lUpdateWeight(double, int);
```

```
void rUpdateWeight(double, int);

void ModifyWeights(int);

int GetId()
{
      return id;
}

double GetOut()
{
      return out;
}

double GetNET()
{
      return NET;
}

void SetTemperature(double tmpr)
{
      temprature = tmpr;
}

void SetThreshold(double thrsh)
{
      threshold = thrsh;
}

void SetOut(double val)
{
      out = val;
}

int IsBias()
{
      return BiasFlg;
}

void InitializeRS();

double GetR()
{
      return r;
}

double GetRprev()
{
      return rprev;
}

double GetS()
{
      return s;
}
```

```
	void UpdateR();

	void UpdateS();
};

double CNeuron::temperature = 0.0;

double CNeuron::threshold = 0.0;


//*******************************************************************
//                    THE CLayer CLASS                             *
//*******************************************************************


class CLayer
{
private:
	int LayerID;           //0 for input layer,1 for 1st hidden, ...
	unsigned int Ncount;   //number fo neurons in layer
	CNeuron* Neuron1;      //pointer to first neuron in layer
	CNeuron* NeuronL;      //pointer to last neuron in layer
	CLayer* next;          //hook so layers can be list members
	CLayer* left;          //hook so layers can be list members
	CLayer* right;         //hook so layers can be list members

public:
	CLayer(int, CNetFileData*);
	int SetWeights(CNeuron*, CNetFileData*);
	int SetSharedWeights(CNeuron*, CNetFileData*);
	int SetSharedLeftWeights(CNeuron*, CNetFileData*);
	int SetSharedRightWeights(CNeuron*, CNetFileData*);
	void CalcLocalgrad();
	void ModifyWeights(int);
	void SetNext(CLayer* Nlayer)
	{
		next = Nlayer;
	}

	void SetLeftLayer(CLayer* Nlayer)
	{
		left = Nlayer;
	}

	void SetRightLayer(CLayer* Nlayer)
	{
		right = Nlayer;
	}
```

```
    int GetLayerID()
    {
          return LayerID;
    }

    CNeuron* GetFirstNeuron()
    {
          return Neuron1;
    }

    CLayer* GetNext()
    {
          return next;
    }

    CLayer* GetLeftLayer()
    {
          return left;
    }

    CLayer* GetRightLayer()
    {
          return right;
    }

    unsigned int getCount()
    {
          return Ncount;
    }

    void calc(int);

    void InitializeRS();

};


//*********************************************************************
//               THE CNetwork CLASS
//*********************************************************************


class CNetwork
{
private:
    int Alive;                      //true when weights are valid

    int netID;                      //network number

    CNetFileData  netdata;          //class to load saved weights

    CNetFileData  leftnetdata;      //class to load saved weights

    CNetFileData  rightnetdata;     //class to load saved weights

    int Nlayers;                    //number of layers in the network

    CLayer* INlayer;                //pointer to the input layer

    CLayer* OUTlayer;               //pointer to the output layer
```

```
	CNetwork* left;			//pointer to the next network
	CNetwork* right;		//pointer to the next network

public:

	CNetwork()
	{
		Alive = 0;
		netID = 0;
		left = (CNetwork*)NULL;
		right = (CNetwork*)NULL;
	}

	int Setup(int);
	void ApplyVector();
	void RunNetwork();
	int RequestNthOutNeuron(int, NETRESULTS*);
	double RequestTemp();
	double RequestThresh();
	int RequestLayerSize(int);
	int GetAlive()
	{
		return Alive;
	}

	CLayer* GetINlayer()
	{
		return INlayer;
	}

	CLayer* GetOUTlayer()
	{
		return OUTlayer;
	}

	int SetSharedWeights();
	int SetSharedLeftWeights();
	int SetSharedRightWeights();
	void CalcLocalgrad();
	void ModifyWeights();
	void SetLeftNet();
	void SetRightNet();
```

```
	CNetwork* GetLeftNet()
	{
		return left;
	}

	CNetwork* GetRightNet()
	{
		return right;
	}

	void InitializeRS();

};


//*********************************************************************
//               THE CEdgeDetectingNetworkView CLASS
//*********************************************************************


class CEdgeDetectingNetworkView : public CView
{
protected:
	int m_Row;                     //holds no. of rows of the image
	int m_Column;                  //holds no. of columns of the image
	double m_Bias;                 //holds the bias weight of the output
	double m_Out;                  //holds the network response
	double m_Outprime;             //holds f'(m_Out)
	double m_Error;                //holds the error value
	double m_Avg_Error;            //holds the average error
	double m_Localgrad;            //holds local gradient
	                               //of the output neuron
	BITMAP m_BM;
	CBitmap m_Bitmap;
	byte* image;                   //an array used to store pixel
	                               //values of the image
	double avg[16][16];            //holds average of the sample images
	UINT imageIndex[50];           //holds identifiers of sample images
	double outputIndex[50];        //holds expected result
	int sequence[50];              //holds sequence of input
	double r0;
	double r;
	double rprev;
	double s;
	double eta;
```

```
public:
      void SetupNet();
      void ForwardPass();
      void BackwardPass();
      void ModifyWeights();
      void CalcOut();
      void CalcLocalgrad();
      void DrawNeurons(CDC*);
      void DrawLinks(CDC*);
      void ReadImage(int);
      char TrainNetwork();
      void RandomizeSamples();
      void CalcSampleAvg(UINT);
      void BackProp(int);
      void LineSearch(int);
      void InitializeRS();
      double CalcResidue();
      void CalcBeta();
      void UpdateR();
      void UpdateS();

};

//*******************************************************************
//                  GLOBAL VARIABLES                                *
//*******************************************************************

double beta;
double desired_out;
double currentImage[16][16];
CNetwork* netptr[16];
CNetwork  net[16];

**************************************************************
**                    THE SOURCE FILE                       **
**************************************************************
```

```
//***************************************************************
//        METHODS FOR THE CNetFileData CLASS                    *
//***************************************************************

CNetFileData::CNetFileData()
{
      for(int i=0;i<MAXLAYERS-1;i++)
            weights[i] = (WEIGHTIMAGE*)NULL;
}

int CNetFileData::setupNet(char* wgt_file_name, int ntkID)
{
      FILE *wgt_file_ptr;
      double AWeight;

      if((wgt_file_ptr = fopen(wgt_file_name , "r")) == NULL)
      {
            AfxMessageBox(wgt_file_name);
            return 1;
      }

      threshold = 0;
      temprature = 1.0;
      Nlayers = 3;

      for(int i=0;i<MAXLAYERS;i++)
            neurons[i] = 0;

      neurons[0] = 16;
      neurons[1] = 14;
      neurons[2] = 1;
      long offset = 56*ntkID*(sizeof(float)+sizeof(char));

      if(ntkID>12)
            offset+= 2*sizeof(char);

      fseek(wgt_file_ptr,offset,0);

      for(int lyr=1;lyr<Nlayers;lyr++)
      {
            if(lyr == 1)
            {
                  int x = 0;
                  for(int dn=0; dn<neurons[lyr];dn++)
                  {
                         for(int sn=x; sn<x+3;sn++)
                         {
                              fscanf( wgt_file_ptr, "%lg", &AWeight );
                              ADDweights(lyr-1,dn,sn,AWeight);
                         }
                         x++;
                  }
            }
```

```
            else
            {
                  for(int dn=0; dn<neurons[lyr];dn++)
                  {
                        for(int sn=0; sn<neurons[lyr-1];sn++)
                        {
                              fscanf( wgt_file_ptr, "%lg", &AWeight );
                              ADDweights(lyr-1,dn,sn,AWeight);
                        }
                  }
            }
      }

      fclose(wgt_file_ptr);
      return 0;
}

void CNetFileData::ADDweights(int l,int d, int s, double w)
{
      WEIGHTIMAGE *Wl = weights[l],
            *Wnew = new WEIGHTIMAGE,
            *cursor,
            *trailer;
      Wnew->data = w;
      Wnew->dneuron = d;
      Wnew->sneuron = s;
      Wnew->next = (WEIGHTIMAGE*)NULL;
      if(Wl)
      {
            cursor = Wl;
            trailer = (WEIGHTIMAGE*)NULL;

      while(cursor)
            {
                  trailer = cursor;
                  cursor = cursor->next;
            }

            trailer->next = Wnew;
      }

      else
            weights[l] = Wnew;
}

double CNetFileData::GETweights(int l, int d, int s)
{
      WEIGHTIMAGE *Wl = weights[l];
      while(Wl)
      {
            if((Wl->sneuron == s) && (Wl->dneuron == d))
            {
                  status = 0;
                  return Wl->data;
            }
```

```
        Wl = Wl->next;
        }
        status = 1;
        return 0.0;
}


//*****************************************************************
//    METHODS FOR THE CNeuron CLASS                               *
//*****************************************************************


void CNeuron::SetWeight(double Wght, CNeuron* ScrPtr)
{
        CWeight* W = new CWeight(Wght,ScrPtr);
        if(weight1 == NULL)
              weight1 = weightL = W;
        else
        {
              weightL->SetNext(W);
              weightL = W;
        }
}

void CNeuron::SetLeftWeight(double lWght, CNeuron* lScrPtr)
{

        CLeftWeight *lW = new CLeftWeight(lWght, lScrPtr);

        if(lweight1 == NULL)
              lweight1 = lweightL = lW;

        else
        {
              lweightL->SetLeftNext(lW);
              lweightL = lW;

        }
}

void CNeuron::SetRightWeight(double rWght, CNeuron* rScrPtr)
{
        CRightWeight *rW = new CRightWeight(rWght, rScrPtr);
        if(rweight1 == NULL)
              rweight1 = rweightL = rW;
        else
        {
              rweightL->SetRightNext(rW);
              rweightL = rW;
        }
}
```

```
double CNeuron::GetWeight(int pos)
{
	CWeight *W = weight1;
	for(int i=0;i<pos;i++)
		W = W->GetNext();

	return W->getWeight();
}

double CNeuron::GetLeftWeight(int pos)
{
	CLeftWeight *W = lweight1;
	for(int i=0;i<pos;i++)
		W = W->GetLeftNext();

	return W->getlWeight();
}

double CNeuron::GetRightWeight(int pos)
{
	CRightWeight *W = rweight1;
	for(int i=0;i<pos;i++)
		W = W->GetRightNext();

	return W->getrWeight();
}

CNeuron* CNeuron::GetSRCNeuron(int pos)
{
	CWeight *W = weight1;
	for(int i=0;i<pos;i++)
		W = W->GetNext();

	return W->GetSCRNeuron();
}

CNeuron* CNeuron::GetlSRCNeuron(int pos)
{
	CLeftWeight *W = lweight1;
	for(int i=0;i<pos;i++)
		W = W->GetLeftNext();

	return W->GetlSCRNeuron();
}

CNeuron* CNeuron::GetrSRCNeuron(int pos)
{
	CRightWeight *W = rweight1;
	for(int i=0;i<pos;i++)
		W = W->GetRightNext();

	return W->GetrSCRNeuron();
}
```

```
void CNeuron::calc(int outlayer)
{
	int counter;
	if(outlayer)
		counter = 14;
	else
		counter = 3;

	for(int src=0; src < counter; src++)
	{
		NET+= GetWeight(src)*GetSRCNeuron(src)->GetOut();

		if(!outlayer)
		{
			NET+= GetLeftWeight(src)*GetlSRCNeuron(src)-
				>GetOut()+
				  GetRightWeight(src)*GetrSRCNeuron(src)-
				>GetOut();
		}

	}

	//add bias
	if(!outlayer)
		NET+= bsweight;

	out = 2/(1+exp(-(NET + threshold)/temprature));

	outprime = 2*out*(1-out)/temprature;
}

void CNeuron::CalcLocalgrad()
{
	//calc localgrads of neurons of the middle layer
	for(int src=0;src<14;src++)
		GetSRCNeuron(src)->SetLocalgrad1(localgrad*GetWeight(src));
}

void CNeuron::UpdateWeight(double wnew, int pos)
{
	CWeight* W = weight1;
	for(int i=0;i<pos;i++)
		W = W->GetNext();
	W->UpdateWeight(wnew);
}

void CNeuron::lUpdateWeight(double wnew, int pos)
{
	CLeftWeight* W = lweight1;
	for(int i=0;i<pos;i++)
		W = W->GetLeftNext();
	W->lUpdateWeight(wnew);
}

void CNeuron::rUpdateWeight(double wnew, int pos)
{
	CRightWeight* W = rweight1;
```

```
	for(int i=0;i<pos;i++)
		W = W->GetRightNext();
	W->rUpdateWeight(wnew);
}

void CNeuron::ModifyWeights(int outlayer)
{
	int counter;

	if(outlayer)
	{
		eta = 0.07;
		counter = 14;
	}
	else
	{
		counter = 3;
		eta = 0.58;
	}

	double wnew, deltaweight;
	double lwnew, ldeltaweight;
	double rwnew, rdeltaweight, bdeltaweight;
	static double deltaprev, ldeltaprev, rdeltaprev, bdeltaprev;
	deltaprev = ldeltaprev = rdeltaprev = bdeltaprev = 0.0;
	for(int src=0;src<counter;src++)
	{
		//update weight
		deltaweight = ALPHA*deltaprev +
		eta*localgrad*GetSRCNeuron(src)->GetOut();
		wnew = GetWeight(src) + deltaweight;
		//wnew = GetWeight(src) + s*eta;
		UpdateWeight(wnew, src);
		deltaprev = deltaweight;

		if(!outlayer)
		{
			//update left weight
			ldeltaweight = ALPHA*ldeltaprev +
			eta*localgrad*GetlSRCNeuron(src)->GetOut();
			lwnew = GetLeftWeight(src) + ldeltaweight;
			//lwnew = GetLeftWeight(src) + s*eta;
			lUpdateWeight(lwnew, src);
			ldeltaprev = ldeltaweight;
			//update right weight
```

```
                        rdeltaweight = ALPHA*rdeltaprev +
                        eta*localgrad*GetrSRCNeuron(src)->GetOut();
                        rwnew = GetRightWeight(src) + rdeltaweight;
                        //rwnew = GetRightWeight(src) + s*eta;
                        rUpdateWeight(rwnew, src);
                        rdeltaprev = rdeltaweight;
                        //update the bias weight
                        bdeltaweight = ALPHA*bdeltaprev + eta*localgrad;
                        bsweight += eta*localgrad;
                        bdeltaprev = bdeltaweight;
                        //bsweight+= eta*s;
                  }

            }

}

void CNeuron::InitializeRS()
{
      r = rprev = -localgrad;
      s = -localgrad;
}

void CNeuron::UpdateS()
{
      s = r + beta*s;

      rprev = r;

}

void CNeuron::UpdateR()
{
      r = -localgrad;
}


//*****************************************************************
//                   METHODS FOR THE CLayer CLASS                 *
//*****************************************************************


CLayer::CLayer(int layer_id, CNetFileData* netdata)
{
      CNeuron* Nptr;
      LayerID = layer_id;
      Neuron1 = (CNeuron*)NULL;
      NeuronL = (CNeuron*)NULL;
      next = (CLayer*)NULL;
```

```
	left = (CLayer*)NULL;
	right = (CLayer*)NULL;
	//Get # of neurons in #layer_id
	Ncount = netdata->GetLayerSize(layer_id);

	for(unsigned int i=0; i<Ncount;i++)
	{
		if(i == Ncount)
		{
			Nptr = new CNeuron(i, TRUE);
		}
		else
		{
			Nptr = new CNeuron(i);
		}

		//attach neuron to the list
		if(Neuron1 == NULL)
		{
			Neuron1 = NeuronL = Nptr;
		}

		else
		{
			NeuronL->SetNext(Nptr);
			NeuronL = Nptr;
		}

	}

}

void CLayer::calc(int outlayer)
{
	CNeuron* Nptr = Neuron1;
	while(Nptr)
	{
		Nptr->calc(outlayer);
		Nptr = Nptr->GetNext();

	}
}

int CLayer::SetSharedWeights(CNeuron* PrevNeuron, CNetFileData*
netdata)
{
	CNeuron* CurNeuron = Neuron1, *PrevPtr;
	double ZWeight = 0.0;
	int curx = 0,
		prevx =0,
```

```
            status = 0,
            counter;
      while(CurNeuron != NULL)
      {
            if(!CurNeuron->IsBias())
            {
                  PrevPtr = PrevNeuron;
                  counter = 0;
                  while(PrevPtr && (counter < 3))
                  {
                        ZWeight = netdata->GetWeight(LayerID, curx,
                                  prevx++);
                        status = netdata->GetStatus();
                        if(status > 0)
                              return status;
                        CurNeuron->SetWeight(ZWeight, PrevPtr);
                        PrevPtr = PrevPtr->GetNext();
                        counter++;
                  }
            }

            curx++;
            prevx = prevx-3;
            prevx++;
            CurNeuron = CurNeuron->GetNext();
            PrevNeuron = PrevNeuron->GetNext();
      }
      return status;
}

int CLayer::SetWeights(CNeuron* PrevNeuron, CNetFileData* netdata)
{
      CNeuron* CurNeuron = Neuron1, *PrevPtr;
      double ZWeight = 0.0;
      int curx = 0,
            prevx,
            status = 0;

      while(CurNeuron != NULL)
      {
            if(!CurNeuron->IsBias())
            {
                  PrevPtr = PrevNeuron;
                  prevx = 0;
```

```
                        while(PrevPtr)
                        {
                              ZWeight = netdata->GetWeight(LayerID, curx,
                                          prevx++);
                              status = netdata->GetStatus();
                              if(status > 0)
                                    return status;
                              CurNeuron->SetWeight(ZWeight, PrevPtr);
                              PrevPtr = PrevPtr->GetNext();
                        }
                  }
                  curx++;
                  CurNeuron = CurNeuron->GetNext();
            }
            return status;
}

int CLayer::SetSharedLeftWeights(CNeuron* lPrevNeuron, CNetFileData*
leftnetdata)
{
      CNeuron* CurNeuron = Neuron1, *lPrevPtr;
      double ZWeight = 0.0;
      int curx = 0,
      lprevx =0,
      status = 0,
      counter;
      while(CurNeuron != NULL)
      {
            if(!CurNeuron->IsBias())
            {
                  lPrevPtr = lPrevNeuron;
                  counter = 0;
                  while(lPrevPtr && (counter < 3))
                  {
                        ZWeight = leftnetdata->GetWeight(LayerID, curx,
                                    lprevx++);
                        status = leftnetdata->GetStatus();
                        if(status > 0)
                              return status;
                        CurNeuron->SetLeftWeight(ZWeight, lPrevPtr);
                        lPrevPtr = lPrevPtr->GetNext();
                        counter++;

                  }
            }
```

```
			curx++;
			lprevx = lprevx-3;
			lprevx++;
			CurNeuron = CurNeuron->GetNext();
			lPrevNeuron = lPrevNeuron->GetNext();
	}
	return status;
}

int CLayer::SetSharedRightWeights(CNeuron* rPrevNeuron, CNetFileData*
rightnetdata)
{
	CNeuron* CurNeuron = Neuron1, *rPrevPtr;
	double ZWeight = 0.0;
	int curx = 0,
		rprevx =0,
		status = 0,
		counter;
	while(CurNeuron != NULL)
	{
		if(!CurNeuron->IsBias())
		{
			rPrevPtr = rPrevNeuron;
			counter = 0;
			while(rPrevPtr && (counter < 3))
			{
				ZWeight = rightnetdata->GetWeight(LayerID,
						curx, rprevx++);
				status = rightnetdata->GetStatus();
				if(status > 0)
					return status;
				CurNeuron->SetRightWeight(ZWeight, rPrevPtr);
				rPrevPtr = rPrevPtr->GetNext();
				counter++;
			}
		}
		curx++;
		rprevx = rprevx-3;
		rprevx++;
		CurNeuron = CurNeuron->GetNext();
		rPrevNeuron = rPrevNeuron->GetNext();
	}
```

```
	return status;
}

void CLayer::CalcLocalgrad()
{
	Neuron1->CalcLocalgrad();
}

void CLayer::ModifyWeights(int outlayer)
{
	CNeuron* Nptr = Neuron1;

	while(Nptr)
	{
		Nptr->ModifyWeights(outlayer);
		Nptr = Nptr->GetNext();
	}
}

void CLayer::InitializeRS()
{
	CNeuron* Nptr = GetFirstNeuron();
	while(Nptr)
	{
		Nptr->InitializeRS();
		Nptr = Nptr->GetNext();
	}
}


//*****************************************************************
//                 METHODS FOR THE CNetwork CLASS                 *
//*****************************************************************


int CNetwork::Setup(int n)
{
	char* wgt_file_name = "weights.wgt";
	char* lwgt_file_name = "lweights.wgt";
	char* rwgt_file_name = "rweights.wgt";
	netID = n;
	int status = 0;
	int lstatus = 0;
	int rstatus = 0;
	status = netdata.setupNet(wgt_file_name, n);
	lstatus = leftnetdata.setupNet(lwgt_file_name, n);
	rstatus = rightnetdata.setupNet(rwgt_file_name, n);
```

```
	if((status > 0)||(lstatus > 0)||(rstatus > 0))
		return 1;

	CLayer *Lptr;

	CNeuron N;

	Nlayers = netdata.GetNlayers();

	N.SetTemperature(netdata.GetTemp());

	N.SetThreshold(netdata.GetThresh());

	for(int i=0; i<Nlayers; i++)
	{
		Lptr = new CLayer(i,&netdata);
		if(i == 0)
		{
			INlayer = OUTlayer = Lptr;

		}

		else
		{
			OUTlayer->SetNext(Lptr);

			//assuming there are only three layers

			OUTlayer->SetRightLayer(Lptr);

			if(i == 1)
				Lptr->SetLeftLayer(INlayer);
			else if(i == 2)
				Lptr->SetLeftLayer(INlayer->GetRightLayer());
			OUTlayer = Lptr;
		}
	}

	status = SetSharedWeights();

	Alive = 1;

	return 0;

}

int CNetwork::SetSharedWeights()
{
	CLayer *L1ptr = INlayer,
	        *L2ptr = INlayer->GetNext();

	int status =0;

	status = L2ptr->SetSharedWeights(L1ptr->GetFirstNeuron(),
	         &netdata);
```

```
        L1ptr = L1ptr->GetNext();
        L2ptr = L2ptr->GetNext();
        status = L2ptr->SetWeights(L1ptr->GetFirstNeuron(), &netdata);
        return status;
}

int CNetwork::SetSharedLeftWeights()
{
        int status = 0;
        if(netID!=0)
        {
                CLayer* L2ptr = INlayer->GetRightLayer();
                CNetwork* netptr = GetLeftNet();
                CLayer* L1ptr = netptr->GetINlayer();
                status = L2ptr->SetSharedLeftWeights(L1ptr-
                         >GetFirstNeuron(), &leftnetdata);
        }

        return status;
}

int CNetwork::SetSharedRightWeights()
{
        int status = 0;
        if(netID!=15)
        {
                CLayer *L2ptr = INlayer->GetRightLayer();
                CNetwork* netptr = GetRightNet();
                CLayer *L1ptr = netptr->GetINlayer();
                status = L2ptr->SetSharedRightWeights(L1ptr-
                        >GetFirstNeuron(), &rightnetdata);
        }
        return status;
}

void CNetwork::CalcLocalgrad()
{
        OUTlayer->CalcLocalgrad();
}

void CNetwork::ModifyWeights()
{
        CLayer* Lptr = OUTlayer;
        int outlayer = 1;
```

```
	while(Lptr && (Lptr!=INlayer))
	{
		if(Lptr->GetLeftLayer() == INlayer)
			outlayer = 0;
		Lptr->ModifyWeights(outlayer);
		Lptr = Lptr->GetLeftLayer();
	}
}

void CNetwork::ApplyVector()
{
	double InpVect[16];
	for(int i=0;i<16;i++)
		InpVect[i] = currentImage[netID][i];
	CLayer* Lptr =  INlayer;
	CNeuron* Nptr = Lptr->GetFirstNeuron();
	i = 0;
	while(Nptr&&!(Nptr->IsBias()))
	{
		Nptr->SetOut(InpVect[i]);
		Nptr = Nptr->GetNext();
		i++;
	}
}

void CNetwork::RunNetwork()
{
	CLayer *Lptr = INlayer->GetNext();
	int out_layer = 0;
	while(Lptr)
	{
		if(Lptr->GetNext() == NULL)
			out_layer = 1;
		Lptr->calc(out_layer);
		Lptr = Lptr->GetNext();
	}
}
```

```
void CNetwork::SetLeftNet()
{
      if(netID == 0)
            left = (CNetwork*)NULL;
      else
            left = netptr[netID-1];

}

void CNetwork::SetRightNet()
{
      if(netID == 15)
            right = (CNetwork*)NULL;
      else
            right = netptr[netID+1];

}

void CNetwork::InitializeRS()
{
      CLayer* Lptr = INlayer->GetNext();
      while(Lptr)
      {
            Lptr->InitializeRS();
            Lptr = Lptr->GetRightLayer();
      }

}


//*****************************************************************
//      METHODS FOR THE CEdgeDetectingNetworkView CLASS           *
//*****************************************************************


void CEdgeDetectingNetworkView::OnNetworkOutput()
{
      CNetResponse responsedlg;
      responsedlg.m_Desired_Out = desired_out;
      responsedlg.m_Output = m_Out;
      responsedlg.m_Error = m_Error;
      responsedlg.m_AvgError = m_Avg_Error;
      responsedlg.DoModal();

}
```

```
void CEdgeDetectingNetworkView::OnNetworkTrain()
{
	char ch = TrainNetwork();
	if(ch == 's')
	AfxMessageBox("Successful Training : The Error did Converge ");
	else
		AfxMessageBox("UnSuccessful Training  : The Error did not
				Converge");
}

void CEdgeDetectingNetworkView::OnNetworkTest()
{
	ReadImage(49);
	ForwardPass();
	CTestNetwork testdlg;
	testdlg.m_ExpectedOutPut = desired_out;
	testdlg.m_NetworkOutPut = m_Out;
	testdlg.DoModal();
}

void CEdgeDetectingNetworkView::SetupNet()
{
	//create 16 independent nets
	for(int i=0;i<16;i++)
	{
		net[i].Setup(i);
		netptr[i] = &net[i];
	}

	//hook up nets to their left and right neighbors
	for(i=0;i<16;i++)
	{
		netptr[i]->SetLeftNet();
		netptr[i]->SetRightNet();
	}

	for(i=1;i<15;i++)
	{
		netptr[i]->SetSharedLeftWeights();
		netptr[i]->SetSharedRightWeights();
	}
}
```

```
void CEdgeDetectingNetworkView::ForwardPass()
{
	//apply input vector
	for(int i=0;i<16;i++)
		netptr[i]->ApplyVector();
	//calculate output of each network
	for(i=1;i<15;i++)
		netptr[i]->RunNetwork();
	CalcOut();
}

void CEdgeDetectingNetworkView::CalcOut()
{
	double sum_of_prod = 0.0;
	CNeuron* Nptr;
	for(int i=1;i<15;i++)
	{
		Nptr = netptr[i]->GetOUTlayer()->GetFirstNeuron();
		sum_of_prod += Nptr->GetNET();
	}
	//add bias weight
	sum_of_prod+= m_Bias;
	m_Out = 2/(1+exp(-(sum_of_prod)));
	m_Outprime = 2*m_Out*( 1 - m_Out);
	m_Error = desired_out - m_Out;
	//update the bias weight
	//m_Bias += eta*s;
}

void CEdgeDetectingNetworkView::BackProp(int n)
{
	ReadImage(sequence[n]);
	ForwardPass();
	CalcLocalgrad();
}

void CEdgeDetectingNetworkView::ModifyWeights()
{
	static double deltaprev = 0.0;
	double deltaweight = ALPHA*deltaprev + eta*m_Bias;
	m_Bias+= deltaweight;
```

```
	deltaprev = deltaweight;
	for(int i=1;i<15;i++)
	netptr[i]->ModifyWeights();
}

void CEdgeDetectingNetworkView::InitializeRS()
{
	s = -m_Localgrad;
	r = rprev = -m_Localgrad;
	for(int i=1;i<15;i++)
		netptr[i]->InitializeRS();
}

double CEdgeDetectingNetworkView::CalcResidue()
{
	double residue = 0.0;
	for(int i=1;i<15;i++)
	{
		CLayer* Lptr = netptr[i]->GetINlayer()->GetNext();
		while(Lptr)
		{
			CNeuron* Nptr = Lptr->GetFirstNeuron();
			while(Nptr)
			{
				residue+= (Nptr->GetR())*(Nptr->GetR());
				Nptr = Nptr->GetNext();
			}
			Lptr = Lptr->GetNext();

		}
	}

	return sqrt(residue);
}

void CEdgeDetectingNetworkView::CalcBeta()
{
	double r = 0, rsqr = 0;
	for(int i=1;i<15;i++)
	{
		CLayer* Lptr = netptr[i]->GetINlayer()->GetNext();
		while(Lptr)
		{
			CNeuron* Nptr = Lptr->GetFirstNeuron();
```

```
                        while(Nptr)
                        {
                                double tmp1 = Nptr->GetR();
                                double tmp2 = Nptr->GetRprev();
                                r+= tmp1*(tmp1 - tmp2);
                                rsqr+= tmp2*tmp2;
                                //r+= Nptr->GetR()*(Nptr->GetR() - Nptr-
                                        >GetRprev());
                                //rsqr+= Nptr->GetR()*Nptr->GetR();
                                Nptr = Nptr->GetNext();
                        }
                        Lptr = Lptr->GetNext();
                }
        }
        beta = r/rsqr;
        if(beta<0)
                beta = 0;
}

void CEdgeDetectingNetworkView::LineSearch(int n)
{
        double sm, psp, psm;
        double g, g_1stderiv, g_2nderiv, h, h_1stderiv, h_2nderiv;
        double G,G_1stderiv, H, H_1stderiv, firstderiv, seconderiv;
        eta = -10;
        for(int i=0;i<n;i++)
        {
                for(int j=0;j<EPOCH_SIZE; j++)
                {
                        ForwardPass();


                        for(int k=0;k<15;k++)
                        {
                                CNeuron* Nptr = netptr[i]->GetOUTlayer()-
                                                >GetFirstNeuron();
                                psp+= Nptr->GetNET();
                                Nptr = netptr[i]->GetINlayer()->GetNext()-
                                        >GetFirstNeuron();
```

```
					while(Nptr)
					{
						sm+= Nptr->GetOut()*Nptr->GetS();
						Nptr = Nptr->GetNext();
					}
				}

				psp+= m_Bias;
				psm = psp/sm;
				double tmp1 = exp(-sm*(eta + psm));
				g = 4-desired_out*(1 + tmp1);
				g_1stderiv = 4*sm*desired_out*tmp1;
				g_2nderiv = -4*sm*sm*desired_out*tmp1;
				h = pow(1+tmp1, 2);
				h_1stderiv = -2*sm*(1 + tmp1)*tmp1;
				double tmp2 = exp(-2*sm*(eta + psm));
				h_2nderiv = 2*sm*sm*(2 + tmp2 + tmp1);
				G = g_1stderiv*h - g*h_1stderiv;
				G_1stderiv = g_2nderiv*h - g*h_2nderiv;
				H = h*h;
				H_1stderiv = 2*h*h*h_1stderiv;
				firstderiv += (G/H);
				seconderiv += (G_1stderiv*H - G*H_1stderiv)/(H*H);
			}

			firstderiv/= 96;
			seconderiv/= 96;
			eta = eta - (firstderiv/seconderiv);

		}
}

void CEdgeDetectingNetworkView::UpdateR()
{
		r = -m_Localgrad;
		for(int i=1;i<15;i++)
		{
			CLayer* Lptr = netptr[i]->GetINlayer()->GetNext();
			while(Lptr)
			{
				CNeuron* Nptr = Lptr->GetFirstNeuron();
```

```
                    while(Nptr)
                    {
                          Nptr->UpdateR();

                          Nptr = Nptr->GetNext();
                    }
                    Lptr = Lptr->GetNext();
              }
        }

}

void CEdgeDetectingNetworkView::UpdateS()
{
        s = r + beta*s;
        rprev = r;
        for(int i=1;i<15;i++)
        {
              CLayer* Lptr = netptr[i]->GetINlayer()->GetNext();
              while(Lptr)
              {
                    CNeuron* Nptr = Lptr->GetFirstNeuron();

                    while(Nptr)
                    {
                          Nptr->UpdateS();
                          Nptr = Nptr->GetNext();
                    }

                    Lptr = Lptr->GetNext();
              }
        }
}

void CEdgeDetectingNetworkView::BackwardPass()
{
        CalcLocalgrad();

        for(int i=1;i<15;i++)

              netptr[i]->ModifyWeights();

}

void CEdgeDetectingNetworkView::CalcLocalgrad()
{
        m_Localgrad = m_Outprime*m_Error;
        for(int i=1;i<15;i++)
        {
              netptr[i]->GetOUTlayer()->GetFirstNeuron()
                                          ->SetLocalgrad(m_Localgrad);
              netptr[i]->CalcLocalgrad();
        }
}
```

```
void CEdgeDetectingNetworkView::ReadImage(int offset)
{
      UINT nResource = imageIndex[offset];
      desired_out = outputIndex[offset];
      m_Bitmap.LoadBitmap(nResource);
      m_Bitmap.GetObject(sizeof(m_BM),&m_BM);
      m_Row = m_BM.bmHeight;
      m_Column = m_BM.bmWidth;
      image = new byte[m_Row*m_Column];
      m_Bitmap.GetBitmapBits(m_Row*m_Column, image);
      //copy array to currentImage, eqn used is y = 2x/255 - 1.
      //and subtract its mean from it
      //double mean = 0;
      for(int i=0;i<m_Row;i++)
      {
            for(int j=0;j<m_Column;j++)
            {
                  currentImage[i][j] =  (image[i*m_Column+j] –
                                         127.5)/255.0;
            }
      }

      m_Bitmap.DeleteObject();

}

void CEdgeDetectingNetworkView::CalcSampleAvg(UINT nResource)
{
      m_Bitmap.LoadBitmap(nResource);
      m_Bitmap.GetObject(sizeof(m_BM),&m_BM);
      m_Row = m_BM.bmHeight;
      m_Column = m_BM.bmWidth;
      image = new byte[m_Row*m_Column];
      m_Bitmap.GetBitmapBits(m_Row*m_Column, image);

      for(int i=0;i<m_Row;i++)
            for(int j=0;j<m_Column;j++)
                  avg[i][j]+= ((2.0*image[i*m_Column+j] –
                             255)/255.0)/48;

      m_Bitmap.DeleteObject();

}
```

```
char CEdgeDetectingNetworkView::TrainNetwork()
{
      char ch = 'f';
      int n = 10, m = 0, i = 0;
      double r_abs, r0_abs;
      BackProp(0);
      InitializeRS();
      r0_abs = CalcResidue();
      do
      {
            LineSearch(n);
            r_abs = CalcResidue();
            if(r_abs < EPSILON*r0_abs)
            {
                  ch = 's';
                  break;
            }

            ModifyWeights();

            if(m >= EPOCH_SIZE)
            {
                  m = 0;
                  RandomizeSamples();
            }

            BackProp(m);
            UpdateR();
            CalcBeta();
            UpdateS();
            m++, n++ ,i++;

      }while(i < MAX_ITERATION);

      return ch;

}

void CEdgeDetectingNetworkView::RandomizeSamples()
{
      int i,j;
      for(i=0;i<48;i++)
      {
            do
            {
                  j = int((double(rand())/RAND_MAX)*47);
```

```
			for(int k=0;k<i;k++)
			{
				if(sequence[k] == j)
				{
					j = 48;
					break;
				}
			}
		}while(j == 48);
		sequence[i] = j;
	}
	UINT* ptr1 = new UINT[48];
	double* ptr2 = new double[48];
	for(i=0;i<48;i++)
	{
		ptr1[i] = imageIndex[sequence[i]];
		ptr2[i] = outputIndex[sequence[i]];
	}
	for(i=0;i<48;i++)
	{
		imageIndex[i] = ptr1[i];
		outputIndex[i] = ptr2[i];
	}
	delete ptr1;
	delete ptr2;
}
```

## Bibliography:

[1] Simon Haykin, Neural Networks: A comprehensive foundation, Pearson Education, second edition.

[2] Rafael C. Gonzalez, Richard E. Woods, Digital Image processing, Pearson Education, second edition.

[3] Nick Efford, Digital Image Processing, a practical introduction using Java, Pearson Education

[4] Michael J. Young, Mastering Visual C++ 6, BPB Publications

[5] N. Ansari, Z.Z. Zhang, Generalized adaptive neural filters, IEE Electron. Lett. 29 (4) (1993) 342–343.

[6] D. de Ridder, R.P.W. Duin, P.W. Verbeek et al., The applicability of neural networks to non-linear image processing, Pattern Anal. Appl. 2 (2) (1999) 111–128, for edge enhancement, Pattern Recognition 26 (8) (1993) 1149–1163.

[7] D.T. Pham, E.J. Bayro-Corrochano, Neural computing for noise filtering, edge detection and signature extraction, J. Systems Eng. 2 (2) (1992) 111–222.

[8] V. Srinivasan, P. Bhatia, S.H. Ong, Edge detection using a neural network, Pattern Recognition 27 (12) (1994) 1653–1662.

[9] R.D. Dony, S. Haykin, Neural network approaches to image compression, Proc. IEEE 83 (2) (1995) 288–303.

[10] A. Ghosh, N.R. Pal, S.K. Pal, Image segmentation using a neural network, Biol. Cybernet. 66 (2) (1991) 151–158.

[11] A. Ghosh, S.K. Pal, Neural network, self-organization and object extraction, Pattern Recognition Lett. 13 (5) (1992) 387–397.

[12] Funahashi, K.-I. (1989). On the approximate realization of continuous mappings by neural networks. Neural Networks. 2(3), 183–192.

[13] Gader, P., Miramonti, J., Won, Y., and Coffield, P. (1995). Segmentation free shared weight networks for automatic vehicle detection. Neural Networks.
8(9), 1457–1473.

[14] Geman, S., Bienenstock, E., and Doursat, R. (1992). Neural networks and the bias-variance dilemma. Neural Computation. 4(1), 1–58.

[15] Gorman, R. and Sejnowski, T. (1988). Analysis of the hidden units in a layered network trained to classify sonar targets. Neural Networks. 1(1), 75–89.

[16] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. Neural Networks. 2(5), 359–366.

[17] Lawrence, S., Giles, C., Tsoi, A., and Back, A. (1997). Face recognition - a convolutional neural-network approach. IEEE Transactions on Neural Networks. 8(1), 98–113.

[18] Pal, N. and Pal, S. (1993). A review on image segmentation techniques. Pattern Recognition. 26(9), 1277–1294.

[19] Setiono, R. and Liu, H. (1997). Neural network feature selector. IEEE Transactions on Neural Networks. 8(3), 645–662.