UNIVERSITÀ DEGLI STUDI DI UDINE
Facoltà di Scienze
Matematiche Fisiche e Naturali

Tesi di Laurea in Scienze dell'Informazione

Automatic Classification of the Hadronic Decays of the Z^0 Boson

Laureando:
Gabriele Cosmo

Relatore:
Dott. A. De Angelis

Correlatore:
Dott. Paula Eerola

Anno Accademico 1990-1991

UNIVERSITA' DEGLI STUDI
DI UDINE

Gabriele Cosmo

AUTOMATIC CLASSIFICATION OF THE HADRONIC DECAYS OF THE Z° BOSON

Tesi di Laurea in Scienze dell'Informazione

ANNO ACCADEMICO 1990-1991

Preface

Neural Network models, inspired from biological data processing, represent a fundamentally new approach to information processing, and offer an alternative to "Programmed Computing". Several features have made them an object of growing interest for applications in the area of pattern recognition and classification: their capabilities to generalize from examples and their versatile adaptation to a wide range of different problems and tasks. Neural Networks are therefore interesting tools for High Energy Physics, where classification of events according to their physical origin is an essential task.

In this thesis work I have faced the problem to realise a Neural Network classifier which could separate the hadronic decays of the Z^0 boson into four quark classes, b, c, s and (u or d). The lack of experimental information of the decay probabilities of the Z^0 into each quark flavour is mainly due to the difficulty of separating events in which the Z^0 decays into a pair of light quarks s, u or d. Results obtained with four independent feed-forward networks are the first measurement of all the partial hadronic widths of the Z^0 [41].

I have studied the possibility to improve the event classification by using two different network structures. The first one is a single feed-forward network with four output nodes [42], whereas the second classifier is a

hybrid architecture based on a self-organized feature map, with a supervised learning stage for interpreting the distribution maps. Comparisons between the two networks indicate that the feed-forward network is more efficient, but the advantages of the hybrid network are faster training and topological visualisation of the feature classes.

In the first chapter some of the basic concepts of modern Particle Physics are discussed briefly. Chapter 2 gives an overview on Neural Computing and in chapter 3 topology and characteristics of feed-forward, self-organized and filter Neural Networks are discussed, including an introduction to the application of mapping networks to classification problems. Chapter 4 is focused to the main problem, the classification of events originating from decays of the Z^0 boson. The solutions proposed are discussed and compared. Finally, in chapter 5 the two software packages developed are briefly presented and explained.

□ □ □

Acknowledgements

I would like to state my appreciation to the staff of the Institute of Physics for the important support provided me during the draft of my thesis, and to INFN for the necessary support for travelling to CERN.

I wish to express my gratitude to Dr. Cinzia Bortolotto for her valued advice in the beginning of my work, and to Jussi Kalkkinen for the long period spent last summer working hard together in front of the VAX-Station consolle.

Special thanks to Dr. Lorenzo Santi for the useful suggestions in using PAW Graphic and LaTeX typesetting Environments.

Thanks are also due to Chiara Vignaduzzo, Mariano De Nardi, Paolo Garlatti-Costa, Gianluigi Girotto, Andrea Linussio and all the other last-year undergraduates of the Institute of Physics for their collaboration.

Last but not the least, I am lovingly thankful to my mother and my father for all the possibilities they gave me and for their understanding and patience during the completion of my studies.

G.C.

□ □ □

Contents

□ □ □

Chapter 1

Particle Physics

The study of the tiny constituent particles of matter and the forces acting between them is one of the main frontiers of physics research, the other being the investigation of the immense regions of outer space to try to understand the structure of the Universe. Although these two areas of research, one dealing with the very small and the other with the very large, are at first glance very different, the understanding of the elementary structure of particles also helps to shed light on the problems of astrophysics and the puzzle of the origin of the Universe. In this chapter, some of the basic concepts of modern particle physics will be introduced. See e.g. [1] for a more detailed discussion.

1.1 What we know about the Constituents of Matter

In the beginning of this century, experimental investigations of matter revealed that it is composed of atoms consisting of a small nucleus containing protons and neutrons, and electrons orbiting round the nucleus. In the past twenty years it has become clear that Nature has a still deeper layer of constituents. Protons and neutrons are not elementary particles, but are themselves built up of smaller entities, which have been given the name *Quarks*.

Our everyday world can now be explained in terms of two basic types of matter particles: Quarks and *Leptons*, while a third class of particles, the *Bosons*, are responsible for transmitting forces between quarks and leptons.

Leptons are particles (like the electron and the neutrino) which do not feel the strong force and do not seem to have any size.

Hadrons are particles (like the proton, neutron, pion and kaon) which feel all the forces and have a measurable size. There are hundreds of them, but they are composed of quarks, so they can be grouped into families with related properties.

Quarks have never been observed in isolation, and therefore they are believed to exist only inside hadrons. Combinations of three quarks give hadrons (like the proton) known as *Baryons*. Combinations of a quark and an antiquark gives hadrons (like the pion) known as *Mesons*. The short history of elementary particle physics in the 20^{th} century has been guided by experimental discoveries and has shown that quarks and leptons can themselves be related in three families of four *Fermions* generations(see Table 1.1). All the components of matter can be explained in terms of just one such family, containing the *up* and *down* quarks, the electron and the *electron-type* neutrino. At higher energies, which can be created artificially using particle accelerators or which can occur through some phenomena in outer space, two further quartets of quarks and leptons come into play. These quartets seem to be heavier copies of the first family. Some of these particles (the *top* quark and the *tau-type* neutrino) have not yet been directly observed, but are nevertheless thought to exist.

1.1.1 Particle Interactions

Particle interactions are considered to be caused by four different types of force (see Table 1.2):

- The **Strong** (or **Nuclear**) force is the most powerful of all. The exchange particle or mediator is the *Gluon*. Gluons are massless gauge particles and were observed for the first time in 1979 at e^+e^- collider PETRA at DESY in Hamburg. The strong force acts between particles carrying a quantum number called *colour*, which is analogous to the electric charge in electromagnetic interactions. Quarks are coloured particles, as well as gluons which carry a superposition of two colours. Colour, which in theory comes in three varieties, is

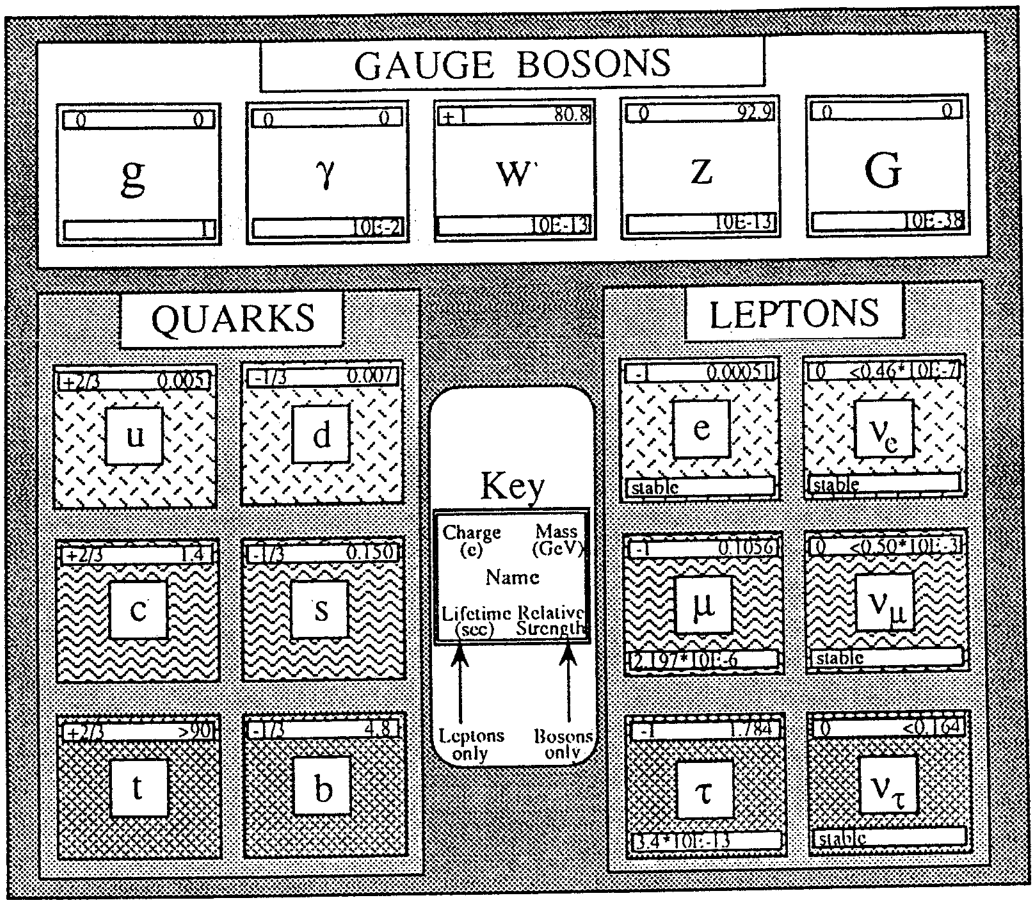


Table 1.1: *Quarks, Leptons and Bosons.*

Interaction	*Relative strengths at low energies*	*Exchange Particles*	*Manifestations*
Strong force	1	Gluons (g)	Nucleus
Electromagnetic force	10^{-3}	Photons (γ)	Atom
Weak force	10^{-5}	$W^{\pm}$,Z^0	Radioactive decay
Gravitation	10^{-38}	Gravitons (?) (G)	Planetary systems

Table 1.2: *The Forces in Nature.*

not observed in hadrons, since they are colour neutral. The residual strong force within protons and neutrons binds nuclei together.

- The **Electromagnetic** force acts between particles which carry electric charge. For example, it holds the cloud of negatively charged electrons around the positively charged protons in the nucleus, binding the atom together. The electric charge is the source of the electromagnetic field, while the quantum exchange particle or mediator is the *Photon.* Photons are massless particles with integral Spin in units of $\hbar$. According to their energy they can manifest themselves as radiowaves, visible light, X-rays and so on.

- The **Weak** force acts in the breakup or decay of particles. It is responsible for the radioactivity of a nucleus which emits an electron when a neutron breaks up. The weak force is over 10^5 times weaker than the strong force at low energies. $W^{\pm}$ and Z^0 bosons are the mediators of this kind of interaction. Discovered at CERN in 1983, two of them are electrically charged types (W^+ and W^-), and one is a neutral type, the Z^0. Weak interactions, in which electric charge is exchanged, are mediated by the weak intermediate bosons $W^{\pm}$. The neutral Z^0 boson can also act between electrically neutral particles like neutrinos.

- The **Gravitational** force acts between all particles. It pulls matter together and is the binding force of the Solar System. The *Graviton* is postulated to be the mediator of this interaction but so far it has not been observed. An adequate theory of quantum gravity has not been formulated yet. Between individual particles, however, gravitational force is extremely feeble, so it can usually be ignored.

The quantum theory of the strong interaction is known as Quantum ChromoDynamics (QCD) and describes the forces between quarks. Electromagnetic interaction is fully described by relativistic Quantum ElectroDynamics (QED) which has been extensively tested and is in agreement with all experimental data. The electromagnetic and weak interactions can be incorporated within a single model, the *Standard Model* of electroweak interactions [3,4,5]. It is called a model (rather than a theory) because parts of it remain purely empirical. The description of $W^{\pm}$ and Z^0 bosons has been one of the great successes of the Standard Model.

1.2 Large Electron-Positron Storage Ring

Progress in the experimental analysis of the structure of matter has been a consequence of higher and higher energies available at accelerators and storage rings. LEP (Large Electron-Positron collider) at CERN (Centre Européen pour les Recherches Nucleaires) is a large underground ring accelerator with a circumference of about 28 kms. Electron and positron beams are injected into LEP with an energy of 20 GeV. The beams consist of four bunches each, equally spaced around the ring. After accumulation, the beams are accelerated to energies of about 45.5 GeV by means of 128 copper RF-cavities, which also compensate the energy losses of the beam due to synchrotron radiation when running at the collision energy. As the beams circulate in opposite directions the electron and positron bunches meet at eight positions around the ring. In four of these collision points detection systems are installed – the detectors are called *ALEPH*, *DELPHI*, *L3* and *OPAL* (see Fig. 1.1). Around the collision points the circular form of the ring is interrupted and straight sections are introduced. At the centre of the straight sections, the particle bunches are focused by magnetic fields to achieve maximum number of collisions per second.

During a collision, a particle and its antiparticle can annihilate each other, producing a packet of energy which materialises into new particles. Outgoing particles are detected in the detection systems surrounding each collision point, except for cones of typically 10 degrees around the beam line.

The detectors [9] are built in layers (see Fig. 1.2). The inner layers take multiple samples along the tracks of charged outgoing particles, measuring their momentum by the track curvature in the magnetic field, and identifying some particle types by their rate of ionization or by Čherenkov radiation. The outer layers consist of *calorimeters* – massive absorbers in which primary particles interact and deposit their energy in showers of secondary particles. Electrons and high-energy photons (γ-rays) leave all of their energy in the fine-grained first layers of the calorimeters, whereas protons, neutrons and mesons deposit their energy in deeper layers. High-energy muons penetrate through all of the calorimeters and are recognized by their tracks in the outer parts of the experiment. Tau-leptons decay very characteristically into small numbers of charged particles. Neutrinos have a negligible probability to interact.

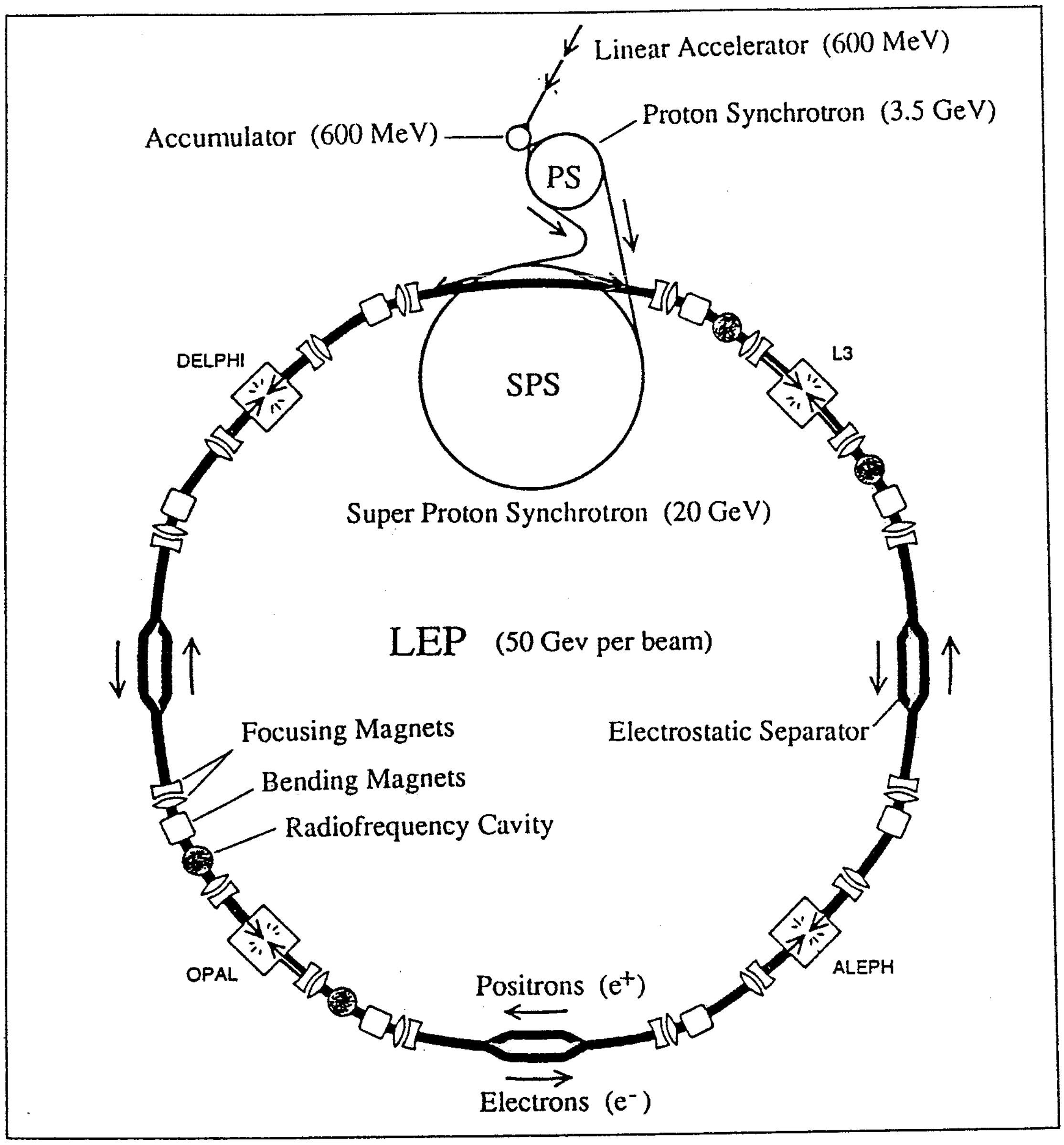


Figure 1.1: *LEP Storage Ring from a top view. The particle bunches meet at eight positions. Collisions take place inside the four detectors, while in the other four positions, collisions are prevented by electrostatic separators.*

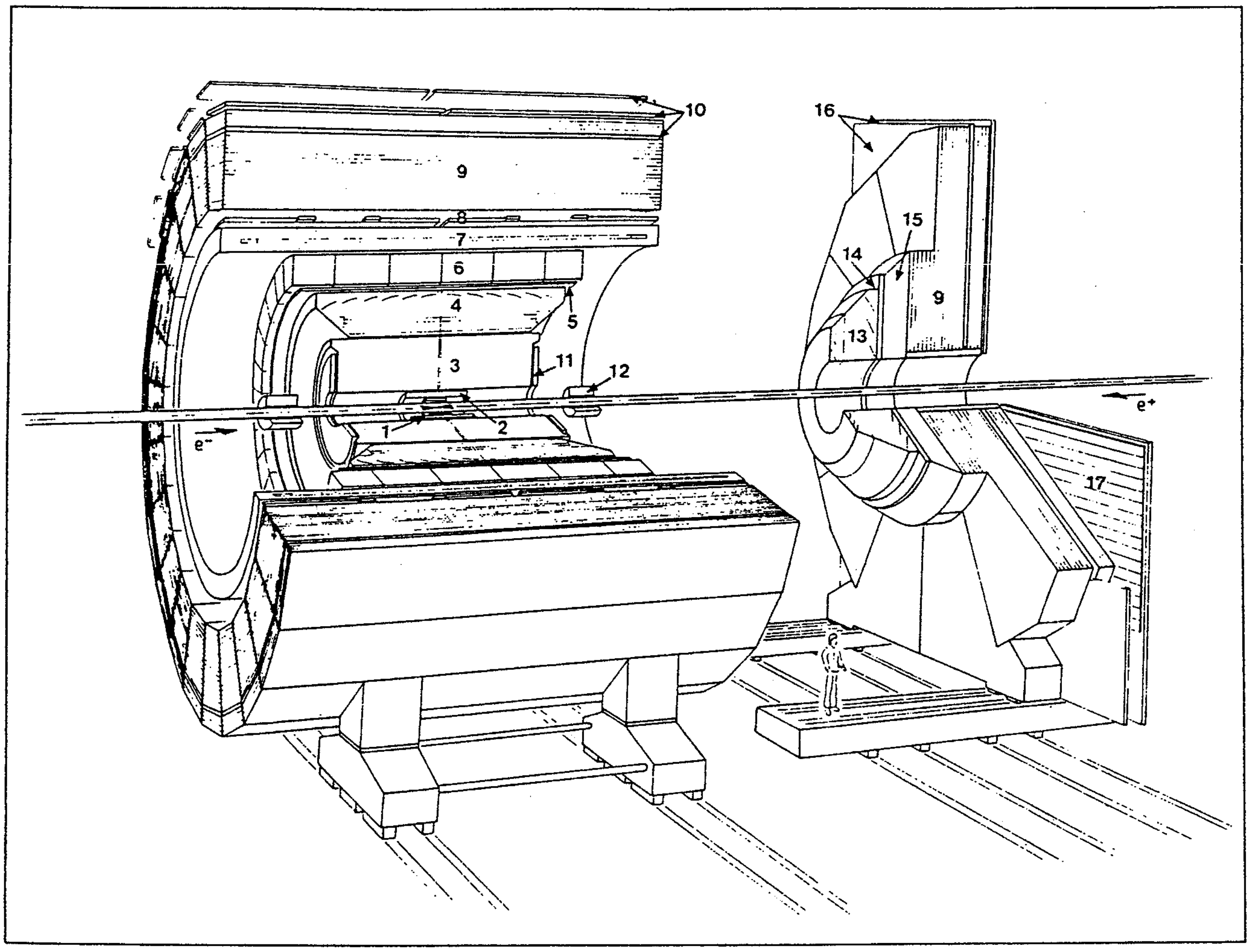


Figure 1.2: *Perspective view of the DELPHI detector.*

1=micro-vertex detector, 2=inner detector, 3=time projection chamber (TPC), 4=barrel ring imaging Čherenkov Counter (RICH), 5=outer detector, 6=high density projection chamber (HPC), 7=superconducting solenoid, 8=time-of-flight counters (TOF), 9=hadron calorimeter, 10=barrel muon chamber, 11=forward chamber A, 12=small angle tagger (SAT), 13=forward RICH, 14=forward chamber B, 15=forward electromagnetic calorimeter, 16=forward muon chambers, 17=forward scintillator odoscope.

1.3 Statistical Measurements

The Standard Model has been a great success with its predictions on the weak interaction, but on the other hand it contains a number of flaws: it does not make any prediction on the number of quark and lepton families, and neither is there an explanation for relations between the masses and the charges between quarks and leptons. These mysteries have led some theorists to propose the existence of a fourth class of quarks and leptons. This new family would change the flavour-mixing matrix by allowing a higher number of quark-to-quark transitions and introducing new free parameters to generate the violation of parity and charge (CP violation).

If the family structure is correct, then any bounds for the number of neutrinos hold for the number of quark and lepton generations, too. From a cosmological point of view, the number of light neutrino species has a profound effect on the light isotope generation in the primordial nucleosynthesis process. The expansion speed of the Universe depends on the number of the existing particle types. Studies in this field show that the number of neutrino species is limited by four or five at most [6].

The Z^0 boson can decay into all neutrino species lighter than half of its mass. The decay speed depends on the number of possible decay channels – the lifetime of the Z^0 decreases with increasing number of decay channels. By measuring the mean lifetime of the Z^0 boson, it is possible to find out the number of light neutrino families existing in Nature. The four LEP experiments have measured the *resonance width* of the Z^0 , which is the inverse of the lifetime. The results are consistent with the existence of only three types of light-neutrinos [7], and definitively exclude the existence of a fourth neutrino with a mass less than $\simeq$10 GeV, which is the required quantity to provide the "missing mass" from the critical mass of the Universe, which is needed to eventually stop the expansion of space.

Important topics of research at CERN are the searches of two particles predicted by the Standard Model: the top quark and the *Higgs*-boson. If these particles will not be found using the energies that LEP is now able to produce, it might be that their masses are simply higher than what can be kinematically explored with this accelerator. The precision measurements of the free parameters of the Standard Model, however, can be used to constrain the allowed range for the mass of the top quark. The partial decay width of the Z^0 into b quarks is particularly sensitive to the top quark mass. In addition, by measuring the quantum numbers of the b quark one can demonstrate, that this quark is a member of the third quark doublet and that the family needs to be completed by another particle, which is the missing top quark.

1.3.1 The Hadronic Decays of the Z^0

When an electron and a positron collide at high energies they may rebound unchanged (elastic scattering) or they may annihilate. If they annihilate they can form a short-lived intermediate particle, either a photon or a Z^0 boson. At LEP, the collision energy (about 91 GeV) is optimized to produce predominantly Z^0 bosons in the annihilation. The four experiments have already detected over $\simeq 10^6$ Z^0 events produced at LEP between 1989 and 1991.

The Z^0 has a mean lifetime of about 10^{-25} s, after which it decays into a pair of leptons or quarks. Interactions can be described in terms of *Feyman diagrams*. In Fig. 1.3 the four main visible decay channels of the Z^0 are presented. They are:

1. Electron-Positron (e^-e^+)
2. Muon-Antimuon ($\mu^+\mu^-$)
3. Tau-Antitau ($\tau^+\tau^-$)
4. Quark-Antiquark ($q\overline{q}$)

The produced quark and the antiquark are moving to opposite directions due to energy and momentum conservation. The strong colour force is linking the two quarks, and finally the potential energy of the colour field becomes so large that one or more $q\overline{q}$ pairs are created, "fragmenting" the original pair into *jets* of hadrons. All the available energy is shared among the final state particles, typically $\approx$30. The hadron jets originating from quarks are moving in the direction of the q or $\overline{q}$ – additional jets can be produced during the fragmentation process by energetic gluons radiating off the quarks.

In $q\overline{q}$ decay the snapping of the colour strings in the fragmentation process is essentially random, so it is appropriate to try to reproduce it with

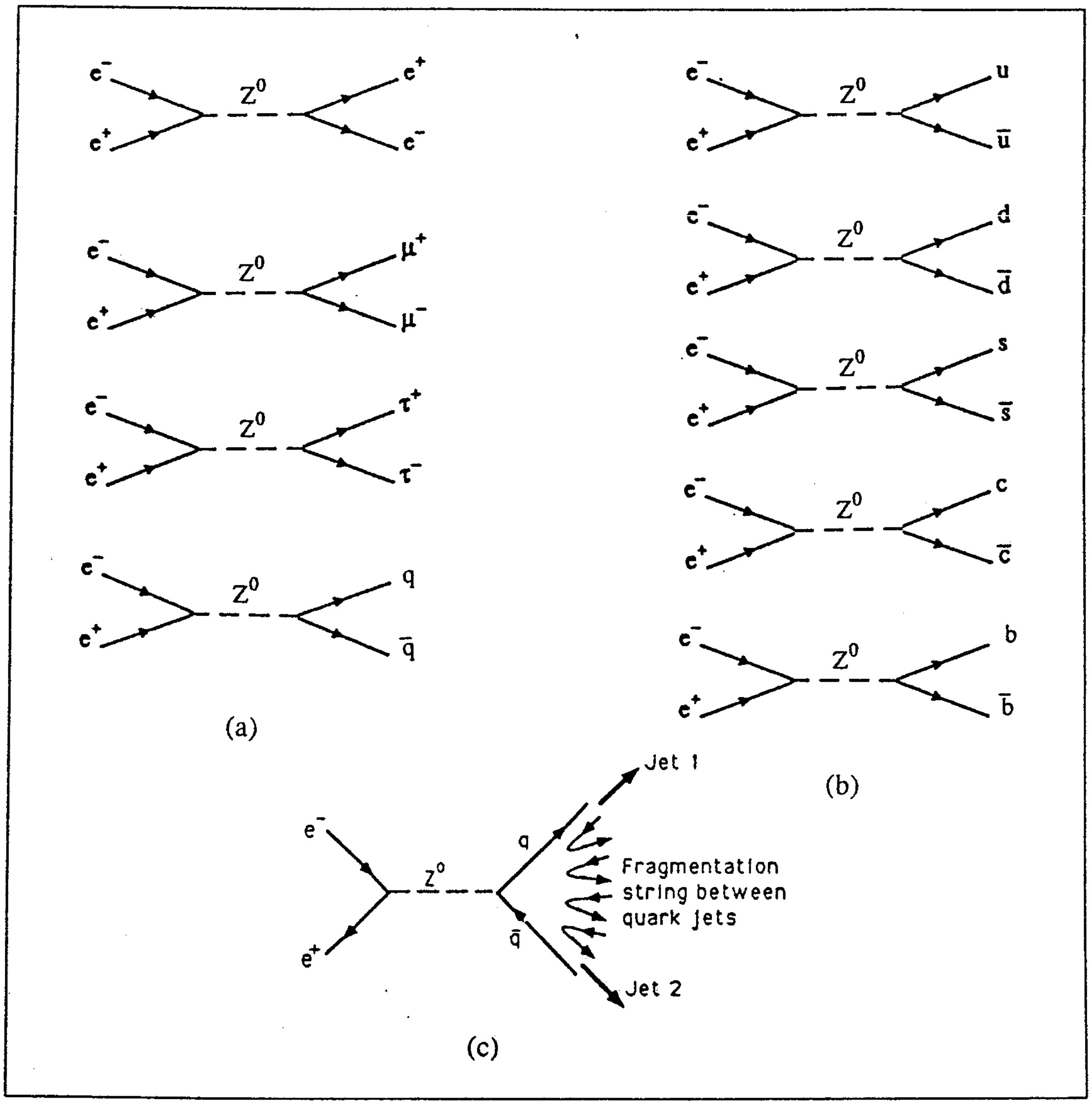


Figure 1.3: *(a) The four main visible decay channels of the Z^0 boson; (b) the 5 $q\bar{q}$ pairs; (c) the $q\bar{q}$ fragmentation.*

random number generators in a computer. *Monte Carlo* programs incorporating some theoretical constraints from QCD and some phenomenological parameters have been tuned to reproduce the particle multiplicities, momentum distributions and other statistical measures of the final state hadrons produced at lower energy machines and at LEP.

□ □ □

Chapter 2

Neural Computing

Neural Computing is a method for information processing that autonomously develops operational capabilities by an adaptive response to an information environment. This chapter begins with an overview of Neural Computing followed by a discussion of the relationship between Neural Computing and Neuroscience. After a brief history of the subject, a precise definition of a Neural Network will be given, and biological and artificial neuron models are compared.

2.1 Introduction

Neural Computing is the technological discipline describing parallel, distributed, adaptive information processing systems, which develop their capabilities in response to exposure to an information environment. It is a fundamentally new and different approach to information processing and it can be seen as the first alternative to *programmed computing*, which has dominated information processing for the last 45 years.

Programmed computing can be used in only those cases in which the processing to be accomplished can be described in terms of known procedures or a known set of rules. If the required algorithmic procedure or set of rules are not known, then they must be developed – an undertaking that, in general has been found to be costly and time consuming. Neural Computing is based on transformations, it does not require an algorithm or rule development, and therefore it often significantly reduces the quantity of software that must be developed. It also enables handling of problems for which the algorithms or rules are not known (data analysis, pattern recognition, control, etc.). These properties make Neural Computing an interesting alternative to programmed computing, at least in those areas where it is applicable.

One area of potential future development in Neural Computing that may radically alter the approach to its applications, is the reconstruction of an automated set of tools which would perform powerful general-purpose information processing functions.

The primary information processing structures in Neural Computing are *Neural Networks*. There is a multitude of neural network architectures that have been studied and characterized sufficiently in order to allow their use in solving practical problems. Each of these architectures has its own unique mixture of information processing capabilities, domains of applicability, techniques for use, required training data, training regimen, and so on. At present, however, they have been applied in only certain problem areas such as sensor processing, pattern recognition, data analysis, and control.

2.2 The Relationship between Neural Computing and Neuroscience

Neuroscience can be defined as the scientific discipline concerned with understanding both the brain and the mind (the "Hardware" and "Software" aspects of the same object). The human brain is superior to the fastest digital supercomputer in many tasks, for example in visual information processing at recognizing any kind of objects. The brain has many other features that would be desirable in artificial systems:

- It is highly parallel.
- It is robust and fault tolerant. A small fraction of the nerve cells in the brain die every day without affecting its performance significantly.
- It can deal with information that is fuzzy, probabilistic, noisy or inconsistent.
- It is flexible. It can easily adjust to a new environment by "learning", without having to be programmed.

- It is small, compact, and dissipates very little power.

Only in tasks which require primarily simple arithmetic operations does the computer outperform the brain.

These are the main motivations for studying neural computation. Neural Computing has been inspired by progress in neuroscience, although it does not try to be biologically realistic in detail. The brain is composed of networks of neurons and these neurons are much more complicated than are the processing elements used in Neural Computing, and their functions are not yet fully understood. As the term "Neural Network" implies, it was originally mainly aimed at modelling networks of real neurons in the brain. These models are extremely simplified from a neurophysiological point of view, but they are still valuable for gaining insight into the principles of biological computation.

As with any science, progress in neuroscience has taken place by creating functional concepts and models based upon experimental results and then refining or refuting these by carrying out more experiments. Since the models are not adequate representations of the brain functions, Neural Computing systems based upon these ideas cannot be described as being "based upon the operation of the human brain". Nevertheless, the benefit to the Neural Computing community of this flow of ideas has been substantial.

On the other hand, Neural Computing may have valuable new insights to offer Neuroscience. New Neural Network architectures are constantly developing in Neural Computing, as well as new theories to explain the operation of these architectures. Many of these developments can be used by neuroscientists as new paradigms for model building of brain and mind.

2.3 Brief History

We can trace the origin of modern neural modelling to the paper [11] of McCulloch and Pitts (1943), which showed that even simple types of Neural Networks could, in principle, compute any arithmetic or logical function. In 1949 Donald Hebb pursued the idea [12] that classical psychological conditioning is ubiquitous in animals because it is a property of individual neurons. Hebb proposed a specific learning law for the synapses of neurons.

During the next fifteen years there was a considerable amount of work done on the detailed logic of threshold networks. In this era the first neurocomputer (the *Snark*) was constructed by Marvin Minsky in 1951 [13]. At the opposite extreme to detailed logic, continuum theories were also developed. Known as **Neurodynamics** or **Neural Field Theory**, this approach used differential equations to describe activity patterns in bulk neural matter.

The first successfull neurocomputer (the *Mark I Perceptron*) was developed in 1958 by Frank Rosenblatt and Charles Wightman. Rosenblatt [15] was able to prove the convergence of a **learning algorithm**, a way to change the weights iteratively so that a desired computation was performed. In the same period, very similar networks called *Adalines* were invented by Widrow and Hoff [16].

Many people expressed a great deal of enthusiasm and hoped that such machines could be the basis for artificial intelligence, but in 1969 Minsky and Papert [14] proved mathematically that a Perceptron could not perform some rather elementary computations such as the *XOR-problem*. Minsky and Papert doubted that one could overcome this problem and thought that it would be more profitable to explore other approaches to artificial intelligence. Due to this statement, most of the computer science community left the neural network paradigm for almost 20 years.

There were still some people who continued to develop Neural Network theory in the 1970's. A major theme was **Associative Content-Addressable Memory**, in which different input patterns become associated with one another if sufficiently similar.

In 1982 John Hopfield [19] was able to add some physical insight to neural modelling by introducing an *Energy Function*, and by emphasizing the notion of memories as dynamically stable attractors.

The most influential development in this decade, however, takes up the old thread of Rosenblatt's perceptrons, resuming an idea expressed by Werbos in 1974. An algorithm, known as *Backpropagation*, was independently developed in 1985 by Rumelhart, Hinton and Williams [17]. This algorithm adjusts the weights connecting units in successive layers of a multi-layer perceptron, in such a way that the perceptron can solve many problems which the simple one-layer perceptron can not.

In 1988 the INNS journal *Neural Networks* was founded, followed by *Neural Computation* in 1989 and the *IEEE Transactions on Neural Net-*

works in 1990.

Although Neural Computing has had an interesting history, the field is still at an early stage of development.

2.4 Neural Networks: Concepts and Definitions

As mentioned in section 2.1, the primary information processing structures in Neural Computing are *Neural Networks.*

Definition 2.1 *A* **Neural Network** *is a parallel distributed information processing structure consisting of* Processing Elements, *which can possess a local memory and can carry out localized information processing operations, and which are linked via unidirectional signal channels called* Connections. *Each processing element has a single output that is fanned out into as many lateral connections as desired.*

The processing function can be of any mathematical type. The information processing that takes place within the processing elements (see Fig. 2.1) must depend only on the current values of the input signals and on values stored in the processing element's local memory.

A Neural Network can be described as a *directed graph* (see Fig. 2.2), where:

- The nodes of the graph are called *processing elements* and links are called *connections.*
- Each processing element can have any number of *input connections* and any number of *output connections*, but the output connections of the same processing element must carry out the same output signal.
- Each processing element can have a *local memory* and possesses a *transfer function* which produces the processing element's output signal using the local memory and the input signals. The processors can operate continuously or episodically. If they operate episodically, there must be an input that *activates* the transfer function. This input arrives via a connection from a *scheduling* processing element that is part of the network.

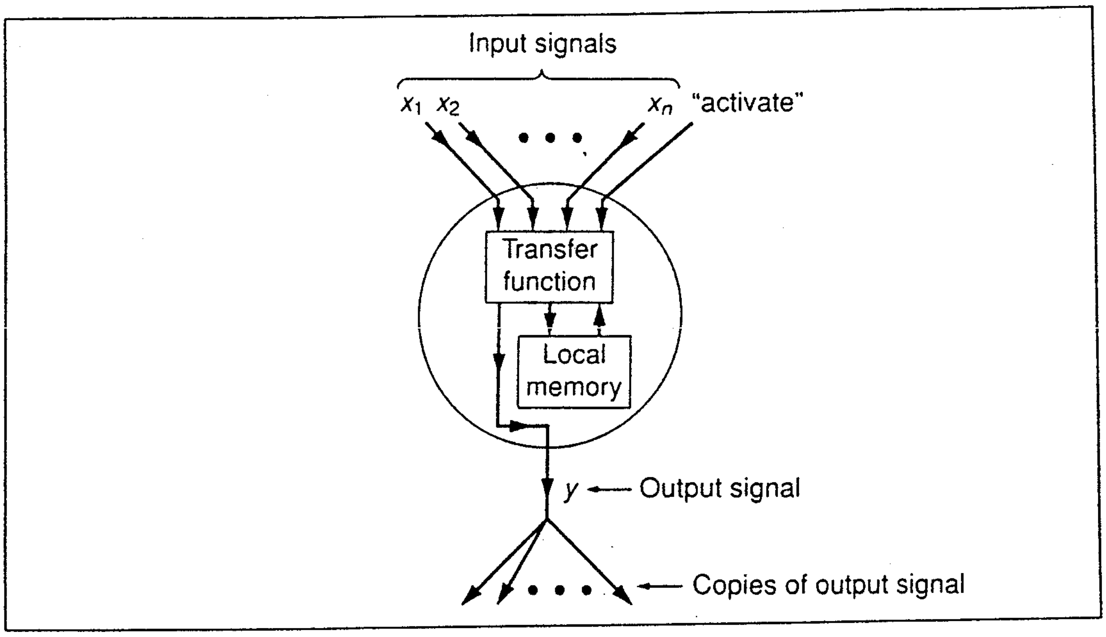


Figure 2.1: *A generic processing element or a neuron. Continuous-time processing elements do not have an "activate" input.*

- Input signals to a Neural Network arrive via connections that originate in the outside world, while outputs to the outside world are signals that leave the Network always via connections.

All known Neural Networks have their processing elements divided into disjoint subsets, called *layers*, in which all the processing elements possess the same transfer function.

Many Neural Networks contain a special *input layer*, which is made up of processing elements receiving exactly one input, which arrives from the outside world. These processing elements, or *fanout units*, typically have no local memory and their only function is to distribute the input signals to the processing elements of the following layers.

Transfer functions usually have a subfunction, called the *learning law*, that is responsible for adapting the input-output behaviour of the processor in response to the input signals. This adaptation is performed by modifying the values of variables stored in the processing element's local memory or by means of a process that creates or destroys connections between the

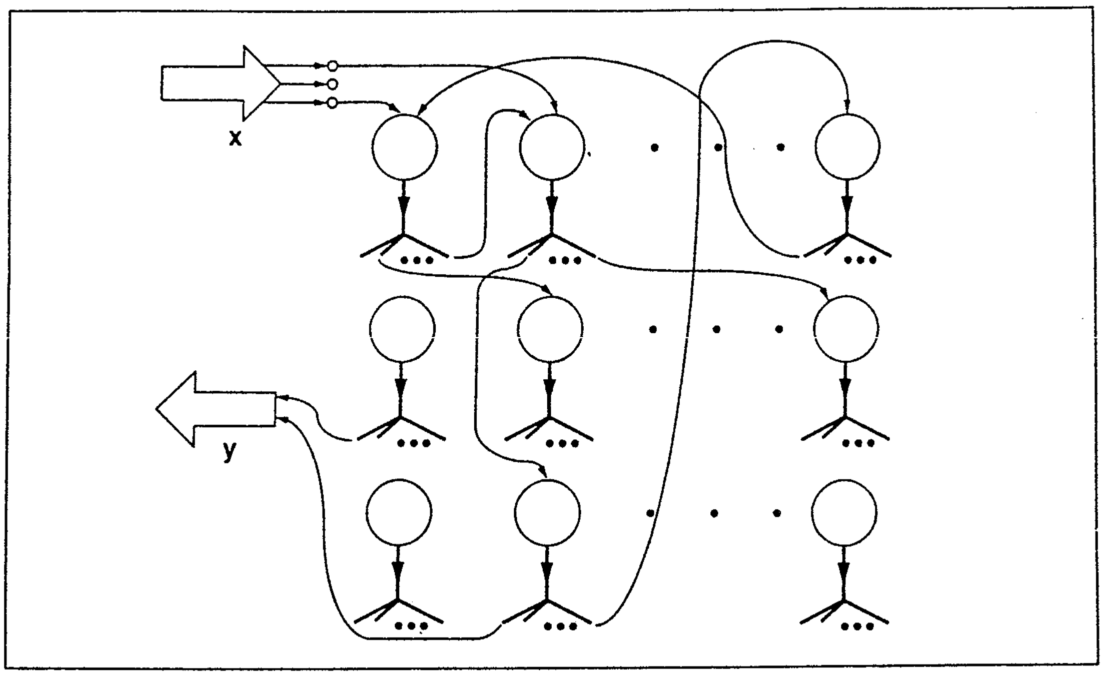


Figure 2.2: *A general Neural Network architecture scheme. The input to the network can be viewed as a data array* x *and the output as a data array* y.

processing elements.

Processing elements are also called *neurons* in analogy with the biological term. A historic example of a processing element is given in the next subsection.

2.4.1 The McCulloch-Pitts Neuron

In 1943 Warren McCulloch and Walter Pitts [11] proposed a simple model of a neuron as a binary threshold unit (see Fig. 2.3).

The neuron computes a weighted sum of its inputs from other units and outputs a one or a zero according to whether this sum is above or below a certain threshold:

$$\psi_i(t+1) = \Theta(\sum_j w_{ij}\psi_j(t) - \theta_i)\,. \tag{2.1}$$

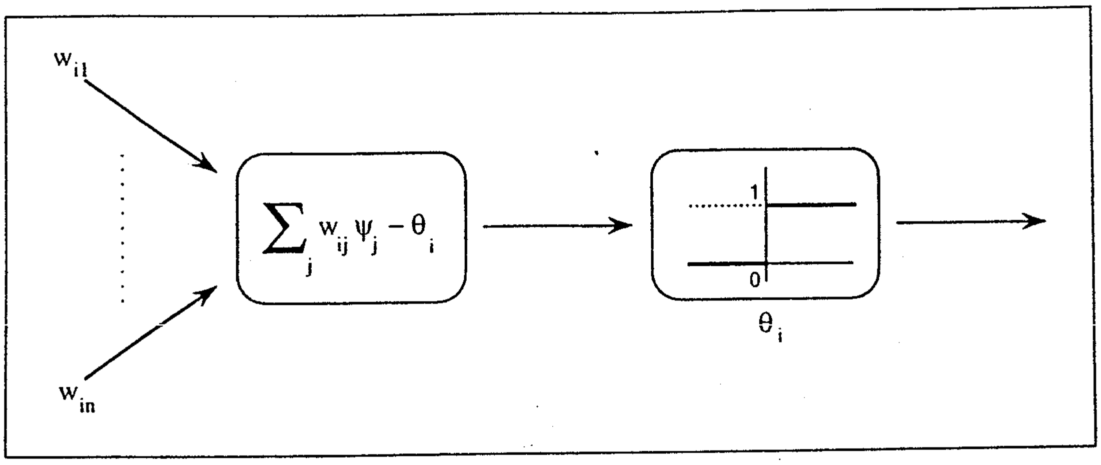


Figure 2.3: *Schematic diagram of a McCulloch-Pitts neuron. The unit fires when the weighted sum* $\sum_j$ *reaches or exceeds the threshold* θ_i.

Here ψ_i is either 1 or 0 and represents the state of neuron i as *firing* or *not firing*, respectively. Time t is taken to be discrete, with one time unit elapsing per processing step. $\Theta(x)$ is the *Step Function*:

$$\Theta(x) = \begin{cases} 1 & \text{if } x \geq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2.2}$$

A simple generalization of the McCulloch-Pitts equation (2.1) is:

$$\psi_i = g(\sum_j w_{ij}\psi_j - \theta_i)\,. \tag{2.3}$$

The threshold function $\Theta(x)$ of (2.1) has been replaced by a more general non-linear transfer function *g(x)* called the *activation function.* Consequently, ψ_i can now have a continuous value and it is called the *state* or *activation* of unit i. Rather than writing the time t or *t+1* explicitly, we must now simply give a rule when to update ψ_i.

A *weight* w_{ij} is a local memory variable that is assigned to each input connection and represents the strength of the connection (*Synapse*) between neuron j and neuron i. It can be positive or negative corresponding to an *excitatory* or *inhibitory* synapse respectively. It is zero if there is no synapse between i and j. The parameter θ_i is the threshold value for unit i – the weighted sum of inputs must reach or exceed the threshold for the neuron to fire.

McCulloch and Pitts proved that a synchronous assembly of such neurons is capable in principle of *universal computation* for suitably chosen weights. This means that it can perform any computation that an ordinary sequential computer can.

2.4.2 The Biological Neuron

The brain is composed of about 10^{11} *neurons* or *nerve cells* of many different types. The *soma* is the cell-body of the biological neuron, the cell *nucleus* is located inside it. Tree-like networks of nerve fibers called *dendrites* are connected to the soma.

Extending from the cell-body is a single long fiber called the *axon*, which eventually branches into strands and substrands. It is electrically active, unlike the dendrites, and serves as the output channel of the neuron. In the ends of its substrands are the *synaptic junctions*, or *synapses*, which are the interfaces to other neurons. The receiving junctions can be found both on the dendrites and on the cell-bodies themselves. Typically, an axon is connected to other neurons with a few thousand synapses.

The axon can be regarded as a non-linear threshold device, producing a voltage pulse, the *action potential*, consisting of a series of rapid voltage spikes. The transmission of a signal from one cell to another at a synapse is a complex chemical process in which specific transmitter substances, called *neurotransmitters*, are released from the sending side of the junction, when the synapse's potential is raised sufficiently by the action potential. The effect is to raise or lower the electric potential inside the body of the receiving cell. We then say that the receiving cell has "fired" when this potential reaches a threshold and a new pulse is sent further by the receiving cell.

Compared with the McCulloch-Pitts neurons, real neurons involve many complications, the most significant including:

- Real neurons are often not even approximately threshold devices. Instead they respond to their input in a continuous way (*graded response*). The non-linear relationship between the input and the output of a cell is a universal feature.

- Many real cells also perform a *non-linear* summation of their inputs. There can even be significant logical processing within the dendritic tree. This can in principle be taken care of by using several formal McCulloch-Pitts neurons to represent a single real one.
- Neurons do not have the same fixed delay ($t \rightarrow t{+}1$), nor are they updated synchronously.
- A real neuron produces a sequence of pulses, not a simple output level.
- The amount of neurotransmitters may vary unpredictably.

Some of these features are included in the generalized model (2.3) of McCulloch-Pitts neuron.

2.5 Parallel Processing

In computer science terms, we can describe the brain as a parallel system of about 10^{11} processors, each one computing a very simple program. Using the simplified model (2.3), each processor computes a weighted sum of the input data from other processors and then outputs a single number, which is a non-linear function of this weighted sum. The weights and the transfer functions can be thought of as local data stored in the processors.

The high connectivity of a network of such processors means that errors in a few terms will probably be inconsequential. Therefore, such a system can be expected to be robust and its performance will tolerate well noise and errors.

It is worth remarking that the typical cycle time of biological neurons is a few milliseconds, which is about a million times slower than their silicon counterparts, semiconductors gates. Nevertheless, the brain can perform very fast processing tasks, far beyond the capacity of a Cray supercomputer. This is due to the fact that the brain is a true parallel processing system with billions of neurons simultaneously operating.

□ □ □

Chapter 3

Mapping Networks

The general Mapping *problem is a central issue in several subjects, such as Pattern Recognition, Statistics and Control Theory. There are basically two types of mapping Neural Networks: the "feature" Networks based on* Supervised Learning *and the "prototype" Networks generally based on* Unsupervised Learning.

In this chapter, after introducing briefly the concept of mapping, three Neural Networks types will be discussed: the Supervised Feed-Forward Network, the Unsupervised Competitive-Learning Network and, finally, the Filter-Learning Network.

3.1 Introduction

Neural Network adaptation always takes place according to *training*. The Network is subjected to a particular information environment following a specified training scheme to achieve the desired end result. Training schemes can be divided into three categories: *supervised learning*, *graded* (or *reinforcement*) *learning* and *self-organisation*.

- **Supervised learning** implies that the training is done by comparing directly the output of the Network to the correct answers. The Network is thus told precisely what it should emit as its output.
- **Reinforcement learning** is a special kind of supervised learning, in which the only feedback to the Network is whether each output

is correct or incorrect, not what the correct answer is. The Network receives a score telling how well it has done over a sequence of multiple training trials. Graded training Networks are particularly applicable to control and process-optimisation problems, in which it is not known what the desired output should be.

- In **self-organisation**, or *unsupervised learning*, the Network modifies itself in response to its inputs. Sometimes the learning goal is not defined at all in terms of specific correct examples. The only available information is in the correlations of the input data or signals. The Network is expected to create categories from these correlations, and to produce output signals corresponding to the input category.

The possibility for training Networks by learning, not by giving rules, gives many exciting implications for computation. Instead of having to specify every detail of a calculation, we simply have to compile a training set of representative examples. This means that we can hope to treat problems where appropriate rules are difficult to know in advance.

More generally, the problem addressed by mapping Neural Networks is the implementation of a bounded mapping or function

$$\mathcal{F} : A \subset \Re^n \longrightarrow \Re^m \, ,$$

by utilising training examples $(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_k, \mathbf{y}_k), \ldots$ of the mapping's action, where $\mathbf{y}_k = \mathcal{F}(\mathbf{x}_k)$.

3.2 The Supervised Feed-Forward Neural Network

Networks can be considered as having separate inputs and outputs, in such a way that we can assume that we have a list or *training set* of correct input-output pairs as examples. In *supervised learning* the Network output is compared to the known correct answer, and the Network receives feedback about the result. The learning occurs by changing the connection strengths w_{ji} to minimise the difference between the actual output and the correct one. This is typically done incrementally, making small adjustments in response to each training pair.

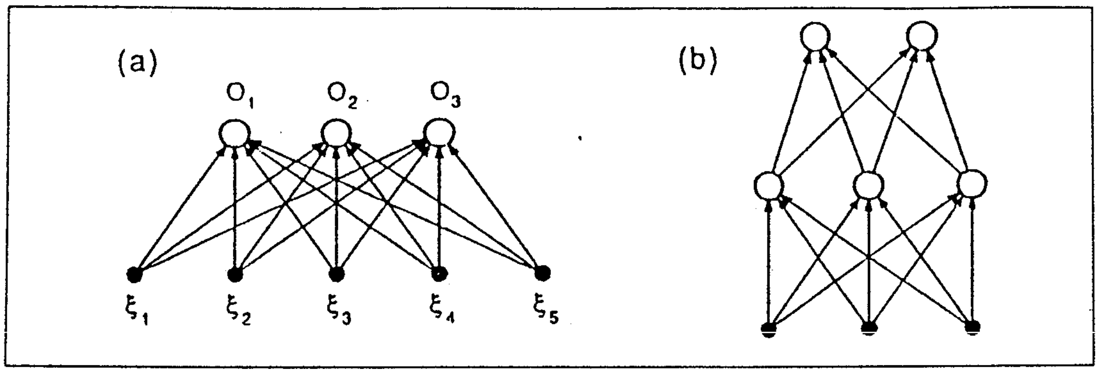


Figure 3.1: *(a) A simple perceptron and (b) a Network with one hidden layer.*

Layered feed-forward Networks were named *perceptrons* by Rosenblatt in 1962. There is a set of input terminals whose only task is to feed input patterns into the rest of the Network. Then comes one or more intermediate layers of *hidden* units, followed by a final output layer (see Fig. 3.1). There are no backward connections, i.e. connections leading from a unit to units in previous layers. A feed-forward Network without hidden units is called a *simple perceptron*, which has only an input and an output layer.

To illustrate the nature of Neural Networks, the classical simple-perceptron architecture will be described in the next subsection.

3.2.1 The Simple Perceptron

As illustrated in Fig. 3.1(a), the Network elements which perform computation are the output units O_j. Each of them makes a weighted sum of the inputs ξ_k. The general association task is cast in the form of asking for a particular output pattern ζ_j^μ in response to an input pattern ξ_k^μ. The aim is to have the *actual* output pattern O_j^μ to be equal to the *target* pattern ζ_j^μ, for each j and μ.

When the input ξ_k is given the pattern ξ_k^μ, the actual output is:

$$O_j^\mu = g\left(\sum_k w_{jk}\xi_k^\mu\right) , \tag{3.1}$$

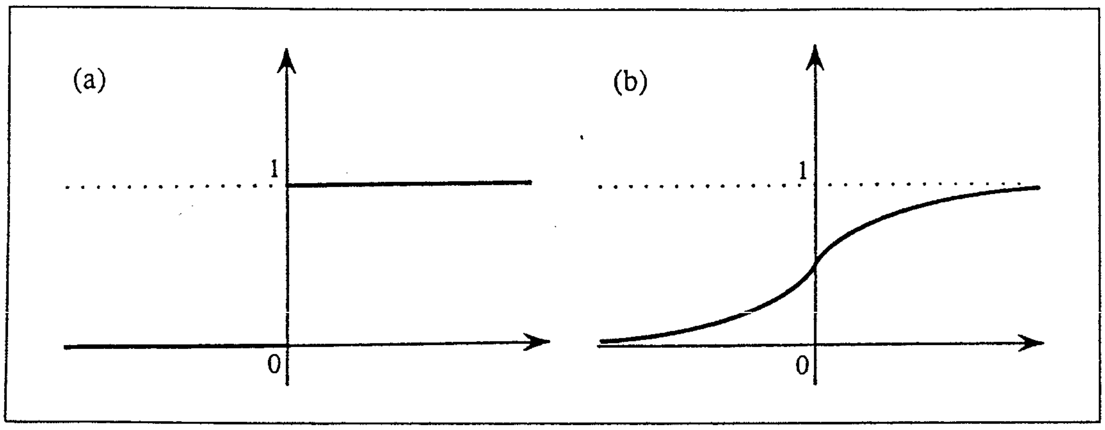


Figure 3.2: *Activation functions: (a) Threshold function and (b) Sigmoid function.*

where g is the activation function and $\mu = 1, 2, ..., p$, p is the number of input-output pairs in the training set.

The activation function g is usually taken to be non-linear. It can be a threshold function or a continuous *sigmoid* function (see Fig. 3.2). In most cases the activation function is the sigmoid *logistic function*:

$$g(a) = \frac{1}{1 + e^{-a}} \, . \tag{3.2}$$

The output units are independent, so it is allowed to consider only one at a time and drop the j subscripts. We can describe the weights w_{jk} with a *weight vector* $\mathbf{w} = (w_1, w_2, ..., w_N)$. Similarly, each input pattern ξ_k^μ can be considered as a *pattern vector* $\hat{\xi}^\mu$ in the same N-dimensional space.

On each training trial, the learning law modifies the weight vector $\mathbf{w}$ according to the equation:

$$\mathbf{w}^{new} = \mathbf{w}^{old} + (\hat{\zeta}^\mu - \hat{O}^\mu)\hat{\xi}^\mu \, . \tag{3.3}$$

An input pattern is a point in N-dimensional space, and we have to classify the point belonging to one of the two classes (0 or 1). The final goal is to find a set of weights for which the output of the perceptron will always match the class number of the point entered into the perceptron.

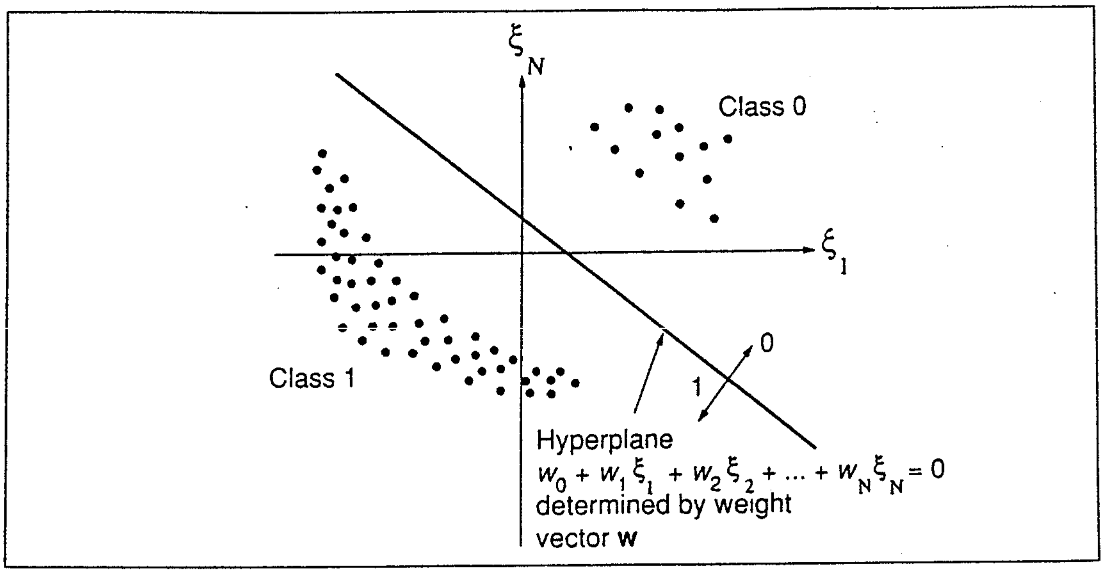


Figure 3.3: *A pattern classification problem in N-dimensional space.*

The perceptron weight vector **w** determines a hyperplane in N-dimensional space (see Fig. 3.3): if a point $(\xi_1, \xi_2, ..., \xi_N)$ lies on one side of the hyperplane, then the output of the perceptron is 0, and if the point lies on the other side, then the perceptron output is 1.

The learning law of equation 3.3 means that if the perceptron makes an error $(\hat{\zeta}^\mu - \hat{O}^\mu)$ in its output, the **w** hyperplane has to be reoriented so that the perceptron will not make an error on the particular $\hat{\xi}^\mu$ vector again. In Fig. 3.3 the perceptron will perform correctly because its hyperplane has been oriented properly relative to the two (*linear separable*) classes. A linearly separable problem is one in which a plane *can* be found in the ξ space separating the ζ^μ=+1 patterns from the ζ^μ=0 ones.

A problem is solvable by a simple perceptron if the problem is linearly separable. It is also possible to prove that if there exists a solution, the perceptron learning rule reaches it in a *finite* number of steps (*Convergence Theorem*). Problems not linearly separable (like the boolean XOR problem) necessarily need a multilayer perceptron to be solved.

3.2.2 The Multilayer Perceptron

Contrary to simple perceptrons, feed-forward Networks with intermediate or "hidden" layers between the input and the output layer are not limited by the linearity of the separation of the classes. In this kind of Network, each unit in the hidden layer and the output layer is like a simple-perceptron unit with a sigmoid thresholding function.

Although the greater power of multilayer networks was realised a long ago, it was only recently shown how to make them to learn a particular function. The learning rule for multilayer Networks is called the *generalised delta rule* or the *back-propagation rule* and was suggested by Rumelhart, McClelland and Williams [17] in 1986. The back-propagation learning rule allows one to find a set of weights by successive improvements from an arbitrary starting point. The network receives feedback about errors in order to minimise them during the training process.

We can define the error measure or *energy cost function* to be the *Mean squared* error:

$$E[\mathrm{w}] = \frac{1}{2}\sum_{j\mu}\left(\zeta_j^\mu - O_j^\mu\right)^2 , \tag{3.4}$$

for each hidden or output unit j of the Network. The mean squared error is not the only possible error function, but it is the most popular one. This error estimation scheme ensures that large errors receive greater attention than small errors. Besides, it takes into account the frequency of occurrences of particular inputs: it is much more sensitive to errors made on commonly encountered inputs than it is to errors on rare inputs.

Calling net_j^μ the activation of each unit j for pattern μ, we can write:

$$net_j^\mu = \sum_i w_{ji} O_i^\mu . \tag{3.5}$$

If $\mathcal{F}_j$ is the sigmoid function (continuously differentiable nondecreasing function) acting on unit j, then we can write the output from each hidden or output unit j as:

$$O_j^\mu = \mathcal{F}_j(net_j^\mu) . \tag{3.6}$$

Weights' *updating* is performed by using the *gradient descent method* on the mean squared error function E (3.4):

$$\Delta_\mu w_{ji} \propto -\frac{\partial E_\mu}{\partial w_{ji}} .$$

We can write:

$$\frac{\partial E_\mu}{\partial w_{ji}} = \frac{\partial E_\mu}{\partial net_j^\mu}\frac{\partial net_j^\mu}{\partial w_{ji}} \tag{3.7}$$

and, substituting in (3.5):

$$\frac{\partial net_j^\mu}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}}\sum_k w_{jk}O_k^\mu = \sum_k \frac{\partial w_{jk}}{\partial w_{ji}}O_k^\mu = O_i^\mu \ , \tag{3.8}$$

where $\frac{\partial w_{jk}}{\partial w_{ji}} = 0$ except when $k = i$ when it equals 1. Defining δ_j^μ as:

$$\delta_j^\mu = -\frac{\partial E_\mu}{\partial net_j^\mu} \ , \tag{3.9}$$

we can write (3.7) as:

$$-\frac{\partial E_\mu}{\partial w_{ji}} = \delta_j^\mu O_i^\mu \ , \tag{3.10}$$

so, the weight's changes will be:

$$\Delta_\mu w_{ji} = \eta \delta_j^\mu O_i^\mu \ , \tag{3.11}$$

where η is a proportionality coefficient called the *learning parameter.*

- If j is an output unit, then

$$\delta_j^\mu = -\frac{\partial E_\mu}{\partial net_j^\mu} = -\frac{\partial E_\mu}{\partial O_j^\mu}\frac{\partial O_j^\mu}{\partial net_j^\mu} \ , \tag{3.12}$$

where

$$\frac{\partial O_j^\mu}{\partial net_j^\mu} = \mathcal{F}_j'(net_j^\mu) \ . \tag{3.13}$$

Differentiating E_μ with respect to O_j^μ:

$$\frac{\partial E_\mu}{\partial O_j^\mu} = -(\zeta_j^\mu - O_j^\mu) \ , \tag{3.14}$$

where ζ_j^μ is the target on pattern μ. Thus

$$\delta_j^\mu = \mathcal{F}_j'(net_j^\mu)(\zeta_j^\mu - O_j^\mu) \ . \tag{3.15}$$

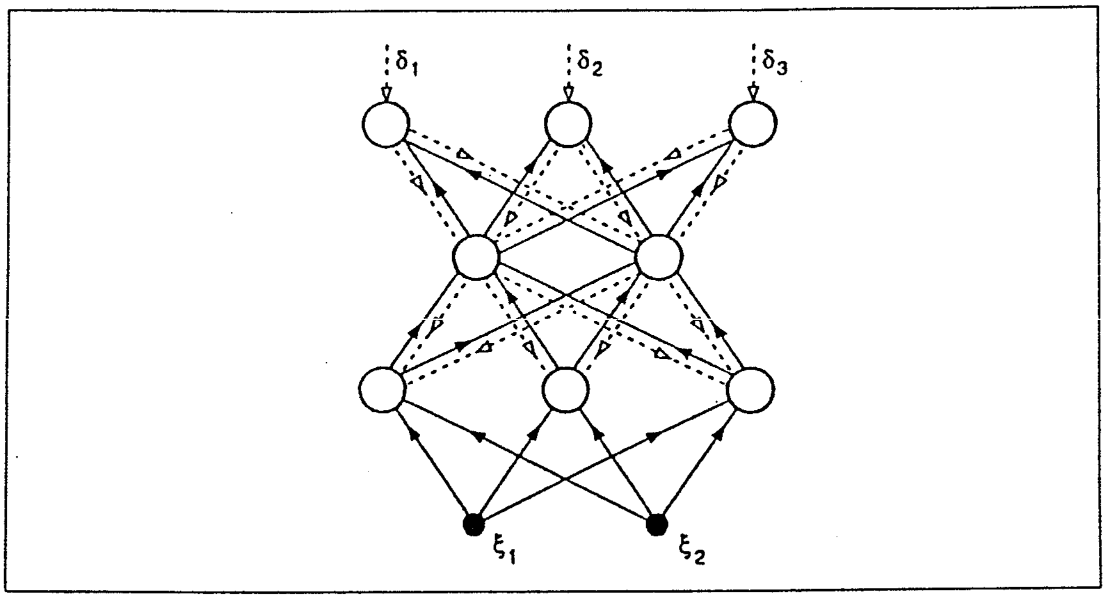


Figure 3.4: *Back-propagation in a Network with two hidden layers. Solid lines describe the signals and dashed lines the δ-errors.*

- Instead, if j is an hidden unit, we can write:

$$\frac{\partial E_\mu}{\partial O_j^\mu} = \sum_k \frac{\partial E_\mu}{\partial net_k^\mu} \frac{\partial net_k^\mu}{\partial O_j^\mu} = -\sum_k \delta_k^\mu w_{kj} \ , \tag{3.16}$$

then, substituting (3.16) in (3.12) we get finally:

$$\delta_j^\mu = \mathcal{F}_j'(net_j^\mu) \sum_k \delta_k^\mu w_{kj} \ . \tag{3.17}$$

Equation (3.17) allows one to determine the δ for a given hidden unit in terms of the δ's of the output units O that it feeds. The coefficients are just the usual "forward" weights w_{ji}, but here they are used for propagating errors (δ's) backwards instead of transmitting signals forwards – hence the name *error back-propagation* (see Fig. 3.4).

3.2.3 The Back-Propagation Learning Algorithm

The updating rule (3.11) is usually used incrementally: a pattern μ is presented at the input and then all weights are updated before the next pattern is considered. This clearly decreases the cost function (for small enough η) at each step, and successive steps adapt the Network to the local gradient. An alternative way of updating is the so called *batch mode* in which the weights are updated after all the patterns have been presented. This requires additional local storage for each connection. The first approach seems to be superior in most cases, especially for very regular or redundant training sets.

The fact that the cost function derivatives can be calculated by back-propagating errors has two important consequences:

- The update rule (3.11) is *local.* To compute the δ's for a given connection we only need quantities available at the two ends of the connection. This makes the back-propagation rule appropriate for parallel computation.

- The computational complexity is very low. Having a total of n connections, computation of the cost function (3.4) takes of the order of n operations. Calculating n derivatives directly would take of the order of n^2 operations.

Summarising, it is possible to express the back-propagation learning law in terms of a step-by-step algorithm:

1. Initialise the weights at small random values.

2. Present a pattern $\hat{\xi}^\mu$ and the associated target vector $\hat{\zeta}^\mu$ to the Network.

3. Propagate the signal forward by calculating the actual output O_j^μ for each hidden and output node j of the Network using equation (3.6).

4. Compute the δ's for the output layer using equation (3.15).

5. Compute the δ's for the hidden layers using equation (3.17).

6. Update the weights by using:

$$w_{ji}(t+1) = w_{ji}(t) + \eta \delta_j^\mu O_j^\mu \ ,$$

   where $w_{ji}(t)$ represents the weight from node i to node j at time t and η is the *gain-term* or *learning parameter.*

7. Goto step 2.

3.2.4 The Learning Parameter and the Momentum Term

The gradient descent rule (3.11) aims at minimising the error function E by adjusting the weights in the Network so that the energy surface is lowest. It changes the weight vectors $\mathbf{w}_j = (w_{j1}, ..., w_{jN})$ only in the direction of the pattern vectors $\hat{\xi}^{\mu}$. Thus any component of the weights orthogonal to the patterns is left unchanged by the learning. Within the pattern subspace, the gradient descent rule necessarily decreases the error if η is small enough, because it takes us in the downhill gradient direction. With sufficient number of iterations, we approach the bottom of the valley from any starting point. The value of η is limited by the largest eigenvalue ϱ^{max}, corresponding to the steepest curvature direction of the error surface: $\eta > 1/\varrho^{max}$. Otherwise we will end up jumping too far to the next valley[1].

The rate of approach to the optimum is limited by the smallest non-zero eigenvalue ϱ^{min}, corresponding to the shallowest curvature direction. If $\varrho^{max}/\varrho^{min}$ is large, progress along the shallow directions can be very slow. So, gradient descent can be very slow if η is small, and can oscillate widely if η is too large.

This problem can be solved by adding to each connection w_{ji} an *inertia* or *momentum term* α, so that the weights tend to change in the direction

[1]Using linear units and assuming that pattern vectors are linearly independent, the error function (3.4) can be written as

$$E = \sum_{k=1}^{M} \varrho_k (w_k - w_k^0)^2 \ , \tag{3.18}$$

where M is the total number of weights in the layer, $\varrho_k (\geq 0)$ and w_k^0 are constants depending on the pattern vectors and the w_k's are linear combinations of the weights. Performing the gradient descent on (3.18) we get:

$$\Delta w_k = -\eta \frac{\partial E}{\partial w_k} = -2\eta \varrho_k (w_k - w_k^0) \ .$$

Fixing $\delta w_k = w_k - w_k^0$,

$$\delta w_k^{new} = \delta w_k^{old} + \Delta w_k = \delta w_k^{old}(1 - 2\eta\varrho_k) \ .$$

Since $|1 - 2\eta\varrho_k| < 1$, then $0 < \eta < 1/\varrho_k$.

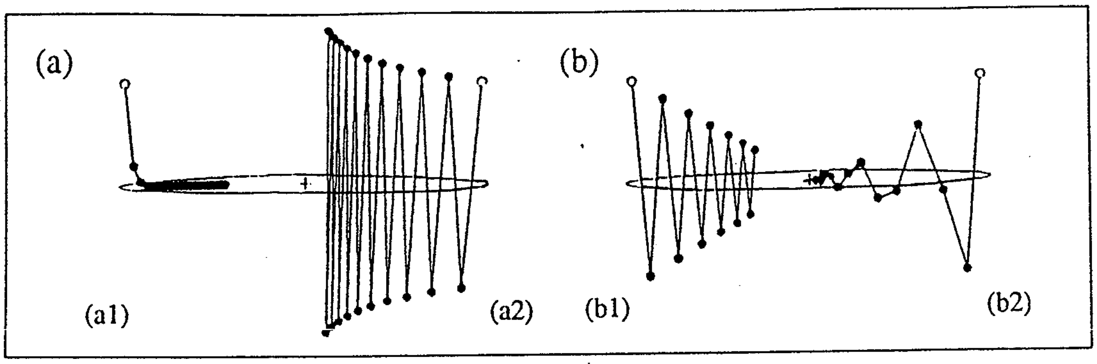


Figure 3.5: *Gradient descent on quadratic surface $E = x^2 + 20y^2$. The minimum is at the + and the ellipse shows the constant error contour (see text).*

of the average downhill "force", instead of oscillating wildly at every step:

$$\Delta w_{ji}(t+1) = -\eta \frac{\partial E}{\partial w_{ji}} + \alpha \Delta w_{ji}(t) \ , \qquad (3.19)$$

where α must be between 0 and 1.

In Fig. 3.5 a gradient descent on a quadratic surface $E = x^2 + 20y^2$ is represented when varying the value of η (the left and right parts are specular copies of the same surface), after 20 iterations in (a) and 12 iterations in (b). The value $y \approx 0$ is fairly quickly reached with $\eta = \frac{1}{50} = 0.02$, but the progress is slow in x (a1). At the other extreme, if $\eta > \frac{1}{20} = 0.05$ the algorithm produces a divergent oscillation in y (a2). The fastest approach is obtained by defining $\eta = \frac{1}{21} = 0.0476$ (b1). (b2) shows the result with $\eta = 0.0476$ and a momentum term $\alpha = 0.5$.

Choosing the appropriate values for the parameters η and α for a particular problem is quite difficult. Moreover, the best values at the beginning of training may not be so good at a later stage. Therefore, it is profitable to automatically adjust the parameters, decreasing η and increasing α as the learning progresses.

3.2.5 Local Minima

Neural Networks can settle into a stable solution that does not produce the correct output. In these cases the gradient descent has lead into a *local* *minimum* of the cost function (no *convergence theorem* exists for back-propagation). One way to partially solve this problem is to adjust the gain term and the momentum term during the learning process, as suggested in the previous subsection (3.2.4). The size of the initial random weights is also important: if they are too large, the sigmoids will saturate from the beginning and the system will become stuck in a flat plateau near the starting point. Weights w_{ji} should be taken to be of the order of $1/\sqrt{k_j}$, where k_j is the number of fan-ins to unit j, in such a way that the magnitude of the typical net input to unit j is less than - but not too much less than - unity.

A common type of a local minimum is one in which two or more errors compensate each other. If these minima are not very deep, just a little *noise* is needed to get out. Addition of noise during the learning process can be done by choosing the input patterns in a random order from the training set or by including a *noise parameter* T or *temperature* in the sigmoid function equation:

$$\mathcal{F}_T(x) = \frac{1}{1 + e^{-x/T}} \; . \tag{3.20}$$

The sigmoid function squeezes the output of a processing element between 0 and 1. The final output becomes more peaked at 0 or 1 as the temperature decreases, becoming exactly 0 or 1 in the asymptotic case in which the temperature is zero. The temperature term can also be adjusted during learning process by decreasing its value.

It is worth remembering that all these optional choices will increase the computation time needed for training considerably.

3.2.6 The Network Architecture

We have already seen (3.2.1) that only linearly separable functions can be represented with *no* hidden layers. Addition of *one* hidden layer is sufficient to represent *any* boolean function. If there are more than two units in the first hidden layer in a multilayer perceptron, pattern space is partitioned with a combination of more than 2 lines, producing *convex regions* or *convex hulls*. Adding more perceptron units in the first hidden layer, it is possible to define more edges for each convex region. In a Network with two hidden layers, units in the second hidden layer will receive as inputs, not lines, but convex hulls, and the combinations of these convex regions may overlap, intersect, or be separated from each other, producing arbitrary shapes. A Network with two hidden layers is therefore capable of separating any classes and the complexity of the shapes is limited by the number of nodes in the hidden layers of the Network, since these define the number of edges.

In Fig. 3.6 the capabilities of different Networks are illustrated. The second column in this figure indicates the types of decision regions that can be formed with different nets. The next two columns show examples of decision regions for the XOR-problem and a problem with meshed regions. The rightmost column presents the most general decision regions that can be formed with each Network.

No more than two hidden layers are required in a feed-forward Neural Network to generate arbitrarily complex decision regions. An astounding theorem was stated in 1957 by mathematician Andrei Kolmogorov, known as the **Kolmogorov's Mapping Neural Network Existence Theorem:**

Theorem 3.1 *Given any continuous function $\mathcal{F} : [0,1]^n \longrightarrow \Re^m$, where $\mathcal{F}(\mathbf{x}) = \mathbf{y}$, $\mathcal{F}$ can be implemented exactly by a three-layer feed-forward Neural Network having n fanout processing elements in the input layer, $(2n+1)$ processing elements in the middle layer, and m processing elements in the output layer.*

The *Kolmogorov mapping Network* consists of three layers of processing elements. The processing elements in the input layer are fanout units that simply distribute the input $\mathbf{x}$ vector components to the processing elements of the middle hidden layer, which implement the following transfer function:

$$z_k = \sum_{i=1}^{n} \lambda^k \Psi(x_i + k\epsilon) + k \ ,$$

where λ is a real constant and Ψ is a continuous real monotonic function. λ and Ψ are independent of $\mathcal{F}$ (although they do depend on n). ϵ is a rational number $0 < \epsilon \leq \delta$, where δ is an arbitrary chosen positive constant.

The m output units have the following transfer function:

$$y_j = \sum_{k=1}^{2n+1} g_j(z_k) \ ,$$

where the functions $g_j \ (j = 1, 2, ..., m)$ are real and continuous, depending on $\mathcal{F}$ and ϵ.

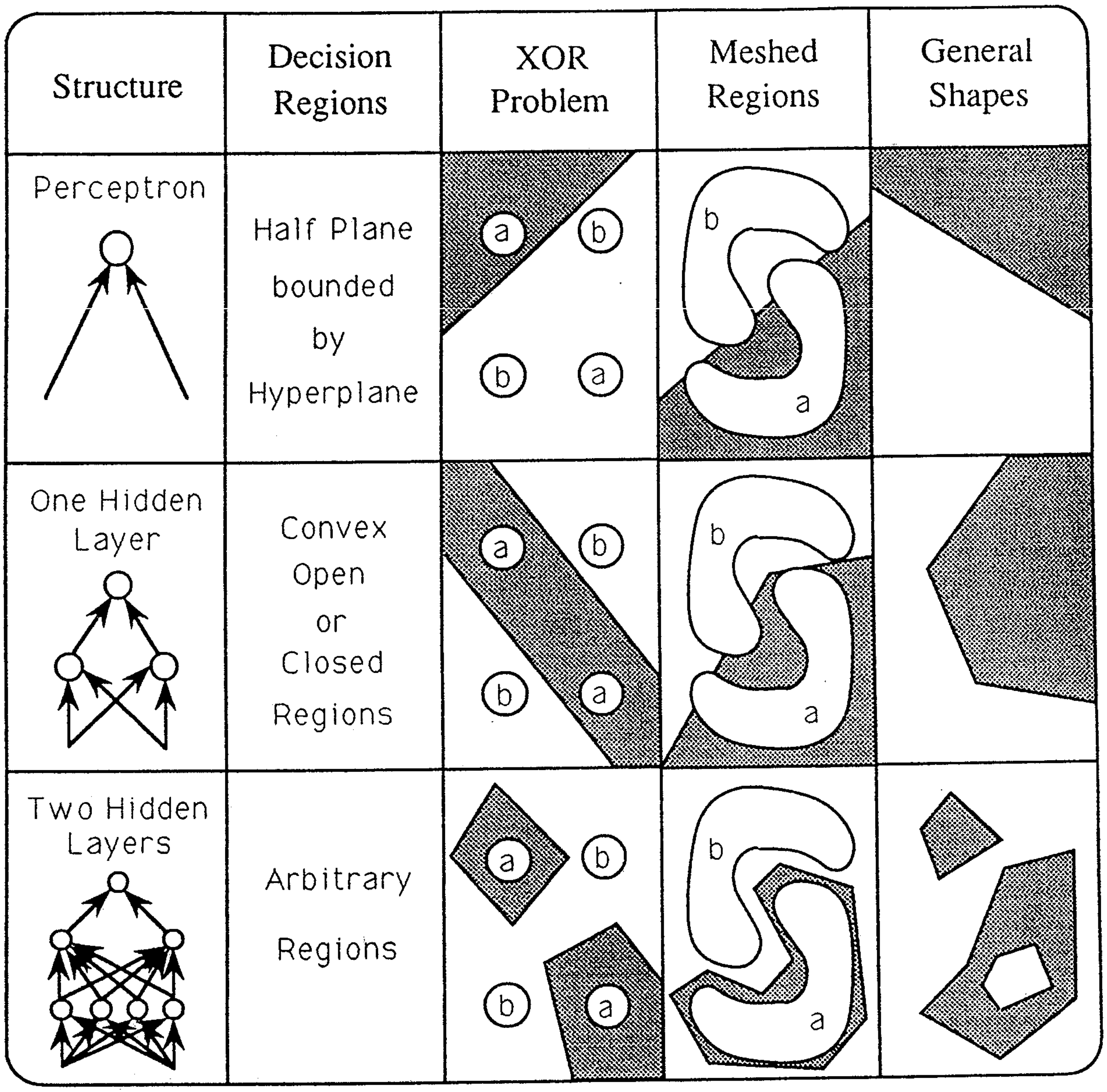


Figure 3.6: *Decision regions for different types of Neural Networks. Shading denotes decision regions for class A. Smooth contours enclose input distributions for classes A and B.*

The proof of the theorem is not constructive, it is strictly an existence theorem. It tells us that such a mapping Network must exist but it does not tell us how to find it. It is, however, an important result proving that whatever is done with two or more hidden layers, could also be done with one in principle. In practise, more than two hidden layers may sometimes permit a solution with fewer units in total, or may speed up the learning.

It is also possible to construct units that have a localised response, each becoming activated for inputs in a small region of the input space. Only *one* hidden layer of such units (*radial basis function units*) is needed to represent any well-behaving function.

Pruning and Weight Decay

To obtain a good generalisation ability one has to build into the Network as much knowledge about the problem as possible, and limit the number of connections appropriately, in order to realise the best topology of the input space within the *internal representation* of the Network. The goal is to obtain the best performance of the Network by using as few units as possible. This should not only reduce computational costs and perhaps training time, but should also improve generalisation.

It is therefore desirable to find algorithms that not only optimise the weights for a given architecture, but also optimise the architecture itself, reducing the number of layers and the number of units per layer. The natural way is to use the Network itself for removing non-useful connections during training. This can be accomplished by giving each connection w_{ji} a tendency to decay to zero, so that a connection disappears unless reinforced. The simplest method is to use:

$$w_{ji}^{new} = (1 - \epsilon_{ji}) w_{ji}^{old} \ , \tag{3.21}$$

after each update of w_{ji}, $\epsilon_{ji} \simeq 0$. This is equivalent to adding a penalty term to the original cost function E_0:

$$E = E_0 + \gamma \sum_{ji} \frac{w_{ji}^2}{1 + w_{ji}^2} \ , \tag{3.22}$$

and writing the ϵ_{ji} in (3.21) as:

$$\epsilon_{ji} = \frac{\gamma \eta / 2}{(1 + w_{ji}^2)^2} \ , \tag{3.23}$$

$\gamma \in \Re$, in such a way that small w_{ji}'s decay more rapidly than large ones.

Starting with an excess of hidden units, it is possible to discard those units which are not needed by using an appropriate pruning procedure in the training.

3.2.7 Examples and Applications

Backpropagation Networks have been applied to a wide variety of problems. Most of them employ a straightforward backpropagation learning by gradient descent method and the Network architecture contains only one hidden layer, with a full connectivity between layers.

The XOR Problem

As described in subsection (3.2.1), the Exclusive-OR problem can not be solved by a simple perceptron because it is not linearly separable. The Network should be able to represent the boolean XOR function with an output equal to "1" when one of the two inputs is on, and "0" when they are both on or both off. This can be achieved by using a Neural Network with one hidden layer.

Inputs		*Output*
ξ_1	ξ_2	ζ
0	0	0
0	1	1
1	0	1
1	1	0

In Fig. 3.7 two Networks with threshold units are presented. In (a) the two hidden units compute the logical OR (left) and AND (right) of the two binary inputs. Solution (b) is not a conventional feed-forward architecture – the hidden unit computes a logical AND to inhibit the output unit when both inputs are on.

The Encoding Problem and Image Compression

A general encoding problem involves finding a set of hidden unit patterns to encode input/output patterns with a large number of nodes. In order

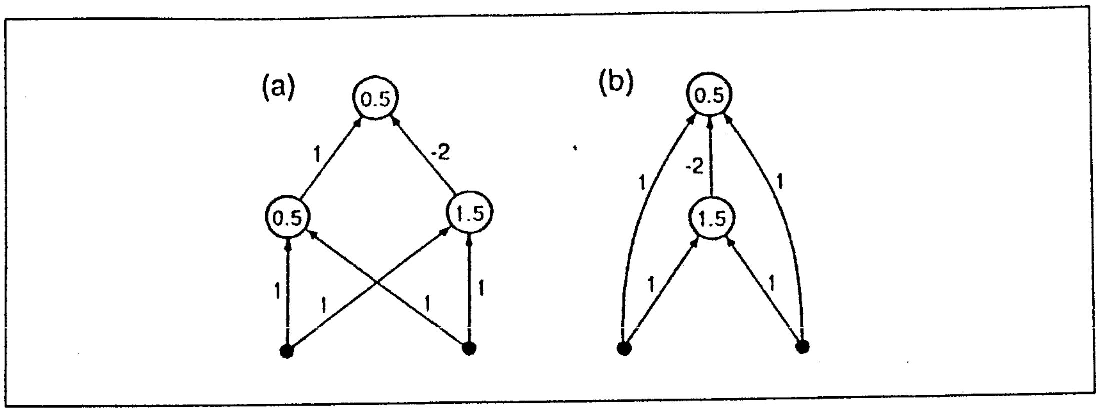


Figure 3.7: *Two possible solutions of the XOR problem using a Neural Network with one hidden layer.*

to achieve an efficient encoding, the number of hidden units must be quite small compared to the number of input/output nodes. The Network structure is made up of an input layer and an output layer of N units, and by a hidden layer of M units, with $M < N$ (see Fig. 3.8).

The training set is composed by exactly $p = N$ patterns, each having a *single* input and the corresponding target on, and the rest off: $\xi_i^\mu = \zeta_i^\mu = \delta_i^\mu$. Using a binary coding in the hidden layer, the activation pattern of the hidden units gives the binary representation of μ, the pattern number. With this scheme, the number of hidden units must be $M \geq log_2 N$.

Encoder Networks have practical applications for problems like *image compression*, in which a picture must be encoded or *compressed* into a much smaller number of bits than the total required to describe it exactly, and then decoded at the receiver into a complete picture. This can be formulated as a supervised learning problem by making the targets equal to the inputs, and taking the compressed signal from the hidden layer. The input-to-hidden connections perform the encoding and the hidden-to-output connections do the decoding.

An interesting feature of this example is that non-linearity in the hidden units confers no advantage. The use of linear units allows a detailed theoretical analysis, which shows that the Network projects the input onto the subspace of the first M principal components of the input. This means that a minimum amount of information is discarded, and those components

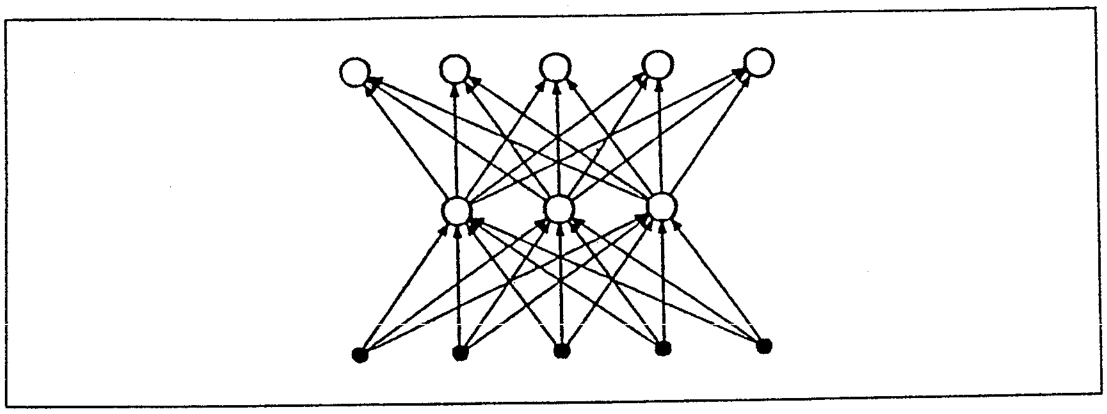


Figure 3.8: *A 5-3-5 encoder Network.*

of the input vector which vary the most [20] are retained.

3.3 The Unsupervised Competitive-Learning Neural Network

In *unsupervised learning*, the learning goal is not defined in terms of specific correct examples – the Network has no feedback from the environment telling what the outputs should be or whether they are correct. The Network must discover itself patterns, features, regularities, correlations, or categories in the input data and create an output coding. Therefore, processing elements and connections must possess some degree of *self-organisation.*

The mapping that a self-organising Neural Network is required to accomplish, is defined implicitly. Instead of learning a mapping

$$\mathcal{F} : A \subset \Re^n \longrightarrow \Re^m ,$$

where A is an arbitrary bounded subset belonging to $\Re^n$, by means of supervised training with explicit examples of the mapping, the self-organising map essentially learns a continuous topological mapping

$$\mathcal{F} : B \subset \Re^n \longrightarrow C \subset \Re^m ,$$

through self-organisation driven by examples in C, where B is a rectangular subset of $\Re^n$ and C is a bounded subset of $\Re^m$, upon which a probability density function ρ depends on.

Unsupervised learning needs redundancy in the input data in order to find patterns or features. Typical output information of an unsupervised learning Network could be:

- **Familiarity.** A single continuous-valued output telling how similar a new input pattern is to a typical or average pattern seen in the past. The Network would gradually learn what is typical.

- **Clustering.** A set of binary-valued outputs, only one on at a time, indicating to which of several categories an input pattern belongs to. These categories would have to be found by the Network on the basis of the correlations in the input patterns. Each cluster would then be classified as a single output class. Applications to clustering can be found in function approximation, image processing, statistical analysis, and combinatorial optimisation.

- **Encoding.** An encoded version of the input, keeping as much relevant information as possible. This could be used for data compression (assuming that an inverse decoding Network can also be constructed) through *vector quantisation*, in which an input data vector is replaced by the index of the output unit that it fires.

- **Feature Mapping.** A topographic map of the input, organising the output units in a fixed geometrical arrangement, and activating nearby output units for similar input patterns.

Unsupervised learning architectures are fairly simple, consisting often of only a single layer. Most of the Networks are essentially feed-forward, and except in the case of Feature Mapping, the number of output processing elements is usually much less than the number of input nodes.

Unsupervised learning may be useful even in situations where supervised learning were possible. For example, after having trained a Network with supervised learning, it may be advisable to allow some subsequent unsupervised learning so that the Network can adapt to gradual changes in its environment.

Optimisation approaches performed by unsupervised Network architectures and learning rules, are close to those of statisticians. There are similarities between many unsupervised learning Networks and standard statistical techniques of pattern classification and analysis [21]. Architectures of these Networks tend also to be more accurately modelled after neurobiological structures than elsewhere in neural computation.

3.3.1 Simple Competitive Learning

Learning laws belonging to the category of *competitive learning* have the property that a competition process, involving some or all of the processing elements of the Neural Network, always takes place before each episode of learning.

In a competitive learning Neural Network, only *one* output unit, or one unit per group, is on at a time. The output units *compete* for being the one to fire, thus they are often called the *"winner-takes-it-all"* units. The simplest competitive learning Networks are composed of a single layer of output processing elements O_j, fully connected to the input layer (see Fig. 3.9). Only one of the output units can fire at a time (*winner* unit) and

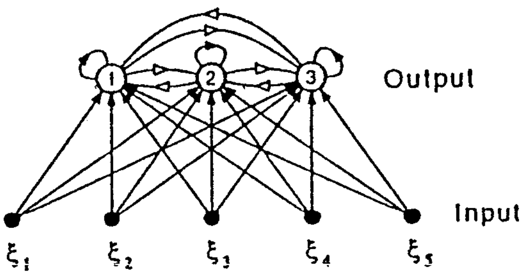


Figure 3.9: *A simple competitive learning Network. The open arrows represent inhibitory connections, the others are excitatory.*

is normally the unit with the largest net input:

$$h_j = \sum_i w_{ji}\xi_i = \mathbf{w}_j \cdot \hat{\xi} \ , \tag{3.24}$$

for the current input vector $\hat{\xi}$. $\mathbf{w}_j$ are excitatory connections.

Considering binary inputs and outputs, the following expression must be true for the winning output unit j^*:

$$\mathbf{w}_{j^*} \cdot \hat{\xi} \geq \mathbf{w}_j \cdot \hat{\xi} \quad (\forall j) \tag{3.25}$$

and $O_{j^*} = 1$. Normalising the weights so that $\mid \mathbf{w}_j \mid = 1$ for all j, we can rewrite (3.25) as:

$$\mid \mathbf{w}_{j^*} - \hat{\xi} \mid \leq \mid \mathbf{w}_j - \hat{\xi} \mid \quad (\forall j) \ . \tag{3.26}$$

Thus, the winner is the unit with normalised weight vector $\mathbf{w}$ closest to the input vector $\hat{\xi}$.

A "winner-takes-it-all" Network is a pattern classifier using (3.25) and (3.26), providing its output units with *lateral inhibitions* and *self-excitatory* connections (see Fig. 3.9) in such a way that each unit inhibits the others. Starting with small random values for the weights, a set of input patterns $\hat{\xi}^\mu$ is presented to the Network. For each input set the Network will find the winning output node j^* and then will update the weights w_{j^*i} for the winning unit only. This makes the $\mathbf{w}_{j^*}$ vector closer to the current input vector $\hat{\xi}^\mu$:

$$\Delta w_{ji} = \eta O_j(\xi_i^\mu - w_{ji}) \ , \tag{3.27}$$

where

$$O_j = \begin{cases} 1 & \text{if } j = j^* \\ 0 & \text{otherwise} \end{cases}$$

This solution is known as the *standard competitive learning rule.*

A problem arising with this learning scheme is that units with a weight vector $\mathbf{w}_j$ far from any input vector may never win, and therefore these units will never learn (so called *dead units*). There are several ways to solve this problem:

- *Using a different weight initialisation.* Weights can be initialised according to samples from the input itself, ensuring that they are in the right domain.

- *Using a BIAS term.* A threshold term θ_j can be subtracted from h_j in (3.24) and adapted periodically to make it easier for the frequently losing units to win. Units that win often should raise their bias terms θ_j, while the losers should lower them (*conscience learning*).

- *Updating all the weights.* Losers' weights can be updated as well as those of the winners, but with a much smaller η (*leaky learning*).

- *Adding a noise term.* Pattern vectors can be smeared with the addition of noise.

- *Using a geometrical distribution.* The output layer can be organised in a geometrical way, for example into a two-dimensional array of output units. The weights of the neighbouring losers can be updated as well as those of the winners. This method is used in *Kohonen feature mapping* that will be discussed in the next subsections.

3.3.2 Feature Mapping

The general concept in the Feature Mapping theory applied to Neural Networks is that nearby outputs must correspond to nearby input patterns. This can be explained by considering two input vectors $\hat{\xi}^1$ and $\hat{\xi}^2$ and the locations $\mathbf{p}_1$, $\mathbf{p}_2$ of the corresponding winning outputs in the Network: as $\hat{\xi}^1$ and $\hat{\xi}^2$ are made more similar, the $\mathbf{p}_1$ and $\mathbf{p}_2$ should get more closer. A Network performing such a mapping is called a *feature map.*

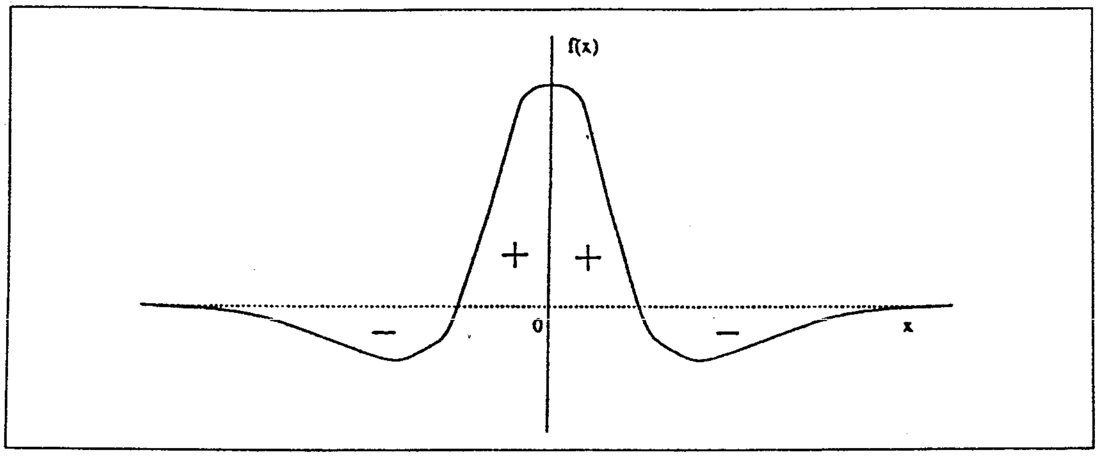


Figure 3.10: *The "Mexican hat" function for lateral connection weights.*

The feature map Neural Network consists of a rectangular array of N processing elements, each of which receives the same input vector $\hat{\xi}^{\mu}$. A weight vector $\mathbf{w}_j$ is assigned to each processing element and is updated after the winning output node has been found. Training of the Network then proceeds in a sequence of discrete self-organisation training trials, until the Network will present a *topographic map* of the input. The task of the Network is essentially to realise a mapping that preserves neighbourhood relations inherent to input patterns.

There are different ways to design an unsupervised Neural Network which organises itself into a feature map:

- The ordinary competitive learning method, adding lateral connections to each unit of the output layer. These connections will be excitatory between nearby units and inhibitory at longer range like a "Mexican hat" function (see Fig. 3.10).

- The ordinary competitive learning method, updating the weights belonging to the *neighbourhood* of the winning processing element (*Kohonen algorithm*).

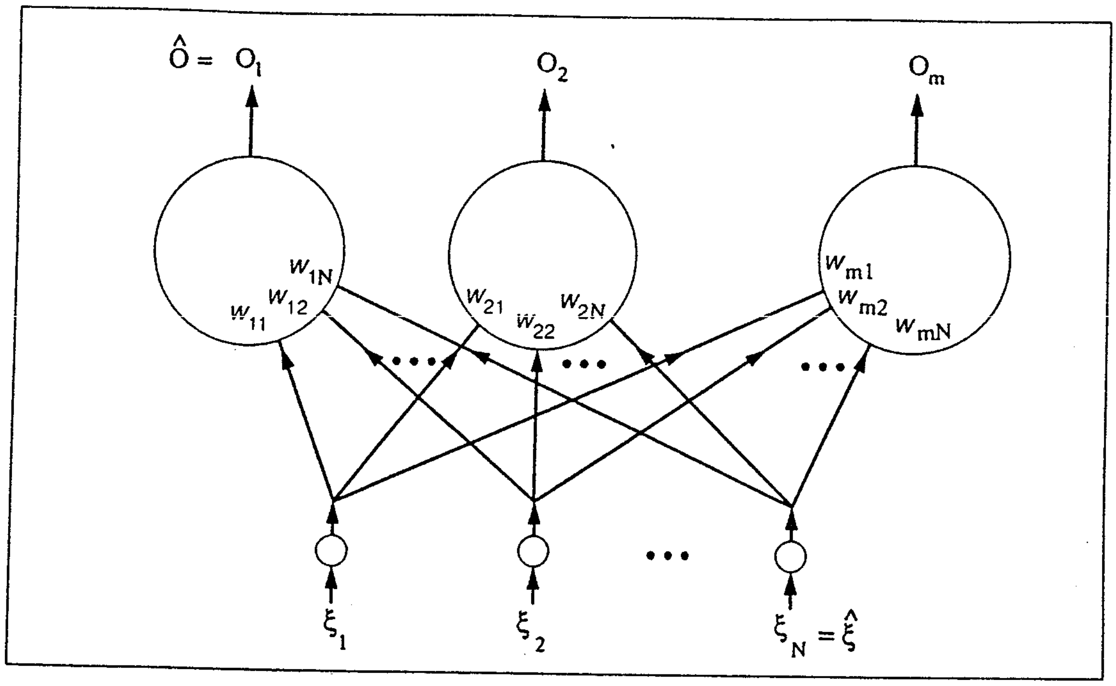


Figure 3.11: *The Kohonen layer: output units are typically arranged as a one or two-dimensional array.*

3.3.3 The Kohonen Layer

The *Kohonen* layer consists of m processing elements $O_1, O_2, ..., O_m$, each receiving N continuous valued input signals $\xi_1, \xi_2, ..., \xi_N$, defining a point $\hat{\xi}$ in a N-dimensional space. The ξ_i input to Kohonen output unit O_j has a real weight w_{ji} assigned to it. The output units are arranged into a one or two-dimensional array, and are fully connected with the inputs (see Fig. 3.11). Each Kohonen processing element calculates its *input intensity* I_j according to:

$$I_j = D(\mathbf{w}_j, \hat{\xi}) \ , \tag{3.28}$$

where $\mathbf{w}_j = (w_{j1}, w_{j2}, ..., w_{jN})^T$ and $\hat{\xi} = (\xi_1, \xi_2, ..., \xi_N)^T$, and $D(\mathbf{u}, \mathbf{v})$ is a function defining a distance. Two common choices for $D(\mathbf{u}, \mathbf{v})$ are the Euclidean distance ($d(\mathbf{u}, \mathbf{v}) = \mid \mathbf{u} - \mathbf{v} \mid$) and the spherical arc distance ($s(\mathbf{u}, \mathbf{v}) = 1 - \mathbf{u} \cdot \mathbf{v} = 1 - cos\theta$), where both $\mathbf{u}$ and $\mathbf{v}$ are assumed to be unit-length vectors and θ is the angle between them.

Once the output units have calculated their input intensities I_j, a competition takes place to see which unit has the smallest input intensity, i.e. a weight vector $\mathbf{w}_j$ closest to $\mathbf{x}$. If j^* is the winner unit, then:

$$\mid \mathbf{w}_{j^*} - \hat{\xi} \mid \leq \mid \mathbf{w}_j - \hat{\xi} \mid \qquad (\forall j) \,. \tag{3.29}$$

The Kohonen Learning Law

The input data vector $\hat{\xi}$ for the Kohonen layer is assumed to be drawn at random according to a fixed probability density function ρ. After determining the winning processing element j^*, the Network will set its output signals as:

$$O_j = \begin{cases} 1 & \text{if } j = j^* \\ 0 & \text{otherwise} \end{cases} \tag{3.30}$$

At this point, a weight modification takes place:

$$w_{ji}^{new} = w_{ji}^{old} + \eta \Lambda(j, j^*)(\xi_i - w_{ji}^{old}) \ , \ \forall \ i, j \ . \tag{3.31}$$

$\Lambda(j, j^*)$ is called the *neighbourhood function* which drives the adaptation according to the distance $\mid p_j - p_{j^*} \mid$ between the output units j and j^*. The shape of the function is such that the winning unit j^* and units close to the winner j^* have their weights changed appreciably, while those further away ($\Lambda(j, j^*)$ small) have low Δw_{ji}'s values. The rule (3.31) drags the weight vector $\mathbf{w}_{j^*}$ and the $\mathbf{w}_j$'s of the closest units towards input vector $\hat{\xi}$.

We can also define the neighbourhood function Λ and the learning step parameter η to depend on time, in such a way that we start with a wide range for $\Lambda(j, j^*, t)$ and a large value of $\eta(t)$, and then reduce both gradually as the learning proceeds (*elastic net*). A typical choice for $\Lambda(j, j^*, t)$ is:

$$\Lambda(j, j^*, t) = e^{\frac{-|p_j - p_{j^*}|^2}{2\sigma(t)^2}} \ , \tag{3.32}$$

where $\sigma(t)$ is a *width* parameter gradually decreasing as t increases.

3.3.4 Probability Density Function Estimation

As new $\hat{\xi}$ vectors are entered into the Network, the Kohonen unit weight vectors are drawn to them and form a cloud near where the $\hat{\xi}$ vectors actually appear, as determined by the probability density function ρ.

At the beginning of training, $\eta(t)$ is often set to a value of approximately 0.8, and is then lowered to 0.1 as the weight vectors $\mathbf{w}_j$ move into the area of the input data. As training progresses, the weight vectors become densest where input vectors are most common, and become least dense (or absent) where the $\hat{\xi}$ vectors hardly ever (or never) appear. The Kohonen layer adapts itself to conform approximately to ρ in a volume number density sense.

The Kohonen learning is similar to the statistical process of finding *k-means*. The k-means for a set of data vectors $\{\hat{\xi}^1, \hat{\xi}^2, ..., \hat{\xi}^p\}$, chosen at random with respect to a fixed probability density function ρ, comprise a set of k vectors $\{\mathbf{w}_1, \mathbf{w}_2, ..., \mathbf{w}_k\}$ that minimise the sum:

$$\sum_{i=1}^{p} D^2(\hat{\xi}^i, \mathbf{w}(\hat{\xi}^i)) \ ,$$

where $\mathbf{w}(\hat{\xi}^i)$ is the closest $\mathbf{w}$ vector to $\hat{\xi}^i$, calculated by using the distance measure D.

The Kohonen learning law can be used for finding k-means, as long as the η value used on each weight update is the fraction of $\hat{\xi}$ vectors which lies closest to the current winning weight vector. Like Kohonen weight vectors, k-means are distributed in the same area as the $\hat{\xi}$ data vectors. The k-means are not, however, equiprobable; compared to this statistical process, the simple competitive learning law is essentially the k-means incremental adjustment law [21].

A $\hat{\xi}$ vector, chosen randomly from a set of vectors distributed according to the probability density function ρ, will have the same probability of being closest to any of the weight vectors. The Kohonen learning law (3.31) produces a set of equiprobable weight vectors by means of the neighbourhood function Λ, which will supply the topological information of inputs: nearby units will receive similar updates and thus they will end up in responding to nearby input patterns.

Kohonen has presented a theory for a one-dimensional self-organising map [22], but a general theory does not exist. Any such theory has to take into account the fact that by changing the learning constants as a function of time, the Network's response to training input data alters consequently. This feature of the self-organising map Neural Network complicates the attempts to build a mathematical model of its operation.

3.3.5 Applications of Competitive-Learning

Competitive-learning Networks have been used for example for sensory mapping, speech recognition, front-end preprocessing, combinatorial optimisation and motor control. The most interesting application is, however, *vector quantisation* for data compression.

The idea is to categorise a given distribution of continuous-valued input vectors, with components ξ_i^μ, into M classes, and then represent each vector by the class which it belongs to. Normally the classes are defined by a set of *M prototype vectors* which divide the input space. By utilising a competitive learning Network, these prototype vectors can be represented by the weight vectors $\mathbf{w}_j$. When exposed to input data, the weights change their values in order to divide the input space, providing a discretised map of the input probability $P(\hat{\xi})$.

In applications like data compression, it is essential to have enough input vectors. Addition of more input samples gives a more precise division of the feature space and ensures an equitable distribution of units in the pattern space. The proper distribution can be maintained by using the conscience mechanism or a Kohonen feature mapping (as seen in 3.3.4).

Kohonen has also suggested a *supervised* version of vector quantisation called *learning vector quantisation* (LVQ), in which the updating rule depends on whether the class of the winner unit is correct or incorrect, and the learning rule is only applied if:

1. the input vector $\hat{\xi}$ is misclassified by the winning unit j^*;
2. the next-nearest neighbour unit j' has the correct class; and
3. the input vector $\hat{\xi}$ is sufficiently close to the decision boundary between $\mathbf{w}_{j^*}$ and $\mathbf{w}_{j'}$.

Both $\mathbf{w}_{j^*}$ and $\mathbf{w}_{j'}$ are updated by using the following rule:

$$\Delta w_{j^*i} = \begin{cases} +\eta(\xi_i^\mu - w_{j^*i}) & \text{if class is correct;} \\ -\eta(\xi_i^\mu - w_{j^*i}) & \text{if class is incorrect.} \end{cases} \tag{3.33}$$

The classification accuracy of the LVQ algorithm has been demonstrated to be very close to the decision-theoretical Bayesian limit even in difficult cases, and due to the very simple computation needed, its speed in learning as well as in classification can be significantly higher than what can be achieved by using other statistical approaches.

3.4 The Filter-Learning Neural Network

In *filter learning* (or *Grossberg learning*), the weights of the Network are determined by a filtering process in which one input to the processing element is treated as a time series signal and the designated weight (not necessary associated with this particular input) takes a value given by the output of a filter applied to this time series.

3.4.1 The Flywheel Equation

In order to understand the mathematical basis for the Grossberg Learning, we study the behaviour of the following scalar equation (the *Flywheel equation*):

$$z(t+1) = z(t) + \eta[\xi(t) - z(t)] \ , \tag{3.34}$$

or

$$z(t+1) - \nu z(t) = \eta\xi(t) \ , \tag{3.35}$$

where $0 < \eta < 1$, $\nu = 1 - \eta$ and $\xi(t)$ is the external input to the equation at time t. For solving this equation we can define $z(t) = \alpha(t)\beta(t)$:

$$\begin{aligned} z(t+1) - \nu z(t) &= \alpha(t+1)\beta(t+1) - \nu\alpha(t)\beta(t) \\ &= \alpha(t+1)(\beta(t) + [\beta(t+1) - \beta(t)]) - \nu\alpha(t)\beta(t) \\ &= \beta(t)[\alpha(t+1) - \nu\alpha(t)] + \alpha(t+1)[\beta(t+1) - \beta(t)] \ . \end{aligned} \tag{3.36}$$

Choosing, without loss of generality:

$$\alpha(t+1) - \nu\alpha(t) = 0 \ , \quad \forall t, \ t = 1, 2, \ldots$$

we obtain:

$$\alpha(t+1) = \nu^t\alpha(1) \ .$$

Substituting this to (3.37) and to (3.35), we get:

$$\alpha(t+1)[\beta(t+1) - \beta(t)] = \eta\xi(t) \ , \tag{3.37}$$

and further

$$\beta(t+1) - \beta(t) = \frac{\eta}{\nu^t\alpha(1)}\xi(t) \ . \tag{3.38}$$

Now we can write[2]

$$\beta(t+1)-\beta(1)=\sum_{k=1}^{t}\frac{\eta\xi(k)}{\nu^k\alpha(1)}=\frac{\eta}{\alpha(1)}\sum_{k=1}^{t}\frac{\xi(k)}{\nu^k}\ . \tag{3.39}$$

To allow a comparison between this discrete case and the continuous case, we can assume that $\xi(t)=\overline{\xi}+\Delta(t)$, where $\overline{\xi}$ is the average value of $\xi(t)$, and $\Delta(t)$ is the deviation of $\xi(t)$ from this average. Then, we get

$$\beta(t+1)-\beta(1)=\frac{\eta}{\alpha(1)}\left[\overline{\xi}\sum_{k=1}^{t}\left(\frac{1}{\nu}\right)^k+\sum_{k=1}^{t}\frac{\Delta(k)}{\nu^k}\right]\ . \tag{3.40}$$

Substituting in $z(t)$:

$$\begin{aligned} z(t) &= \alpha(t)\beta(t) \qquad\qquad (3.41)\\ &= [\nu^{t-1}\alpha(1)]\left(\left[\frac{\eta}{\alpha(1)}\right]\left[\overline{\xi}\sum_{k=1}^{t-1}\nu^{-k}+\sum_{k=1}^{t-1}\nu^{-k}\Delta(k)\right]+\beta(1)\right)\\ &= \eta\overline{\xi}\sum_{k=1}^{t-1}\nu^{t-1-k}+\eta\sum_{k=1}^{t-1}\nu^{t-1-k}\Delta(k)+\nu^{t-1}\alpha(1)\beta(1)\ . \end{aligned}$$

Since $0<\nu<1$,

$$\lim_{t\to+\infty}\nu^{t-1}\alpha(1)\beta(1)=0\ ,$$

while[3]

$$\lim_{t\to+\infty}\sum_{k=1}^{t-1}\nu^{t-1-k}=\frac{1}{1-\nu}=\frac{1}{\eta}\ .$$

$z(t)$ can now be expressed as:

$$z(t)\doteq\overline{\xi}+\eta\sum_{k=1}^{t-1}\nu^{t-1-k}\Delta(k)\ . \tag{3.42}$$

[2]Using the fact that:

$$\beta(t+1)-\beta(1)=\beta(t+1)-\beta(t)+\beta(t)-\beta(t-1)+\ldots+\beta(2)-\beta(1)\ .$$

[3]Note that:

$$\sum_{k=1}^{t-1}\nu^{t-1-k}=\nu^{t-2}+\nu^{t-3}+\ldots+\nu+1=\frac{1-\nu^{t-1}}{1-\nu}\ .$$

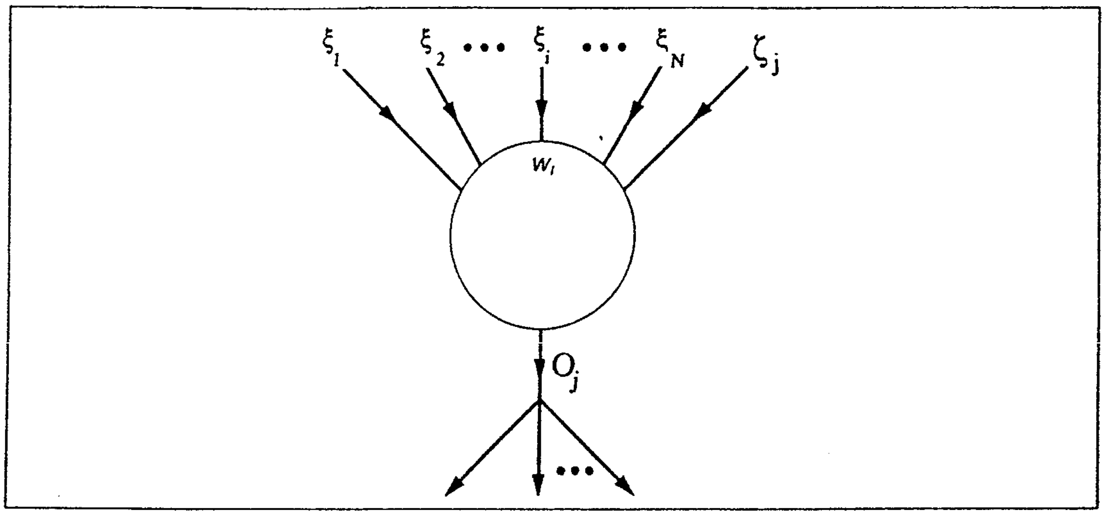


Figure 3.12: *A single instar processing element*

Assuming that the average value of $\xi(t)$ over intervals of time greater than some fraction of $1/\eta$ is always close to $\overline{\xi}$, equation (3.42) can be rewritten as:

$$z(t) \doteq \overline{\xi} \; , \qquad (3.43)$$

which is the solution of the flywheel equation.

3.4.2 The Grossberg Learning Law

The flywheel equation can be used to explain the Grossberg learning law [25]. A processing element j (*instar unit*) in Grossberg learning receives multiple signals (see Fig. 3.12), including inputs ξ_i (assumed to be non-negative real values) and a special input signal ζ_j.

Each input ξ_i has a weight w_{ji} associated with it. The idea is that, whenever the input ξ_i is active (that is, not 0), the weight w_{ji} for this connection will learn the average value of the concurrent ζ_j input multiplicatively weighted by ξ_i.

The Grossberg learning law is expressed by the equation:

$$w_{ji}^{new} = w_{ji}^{old} + \eta(\xi_i\zeta_j - w_{ji}^{old})\Theta(\xi_i) \; , \qquad (3.44)$$

where $0 < \eta < 1$, and Θ is the *step function*:

$$\Theta(\xi_i) = \begin{cases} 1 & \text{if } \xi_i > 0 \\ 0 & \text{otherwise} \end{cases} .$$

The weight will not change unless $\xi_i > 0$.

As in (3.43), after a long period of training, the weight w_{ji} will settle to the value:

$$w_{ji} \doteq \overline{\xi_i \zeta_j} , \tag{3.45}$$

where $\overline{\xi_i \zeta_j}$ is the time average of the quantity $\xi_i \zeta_j$ over the cases when $\xi_i > 0$.

3.5 Statistical Pattern Recognition

The investigation of the possibilities to use Mapping Networks in classification problems related to High Energy Physics experiments is the main aim of this thesis. The classical method to classify input samples is *Statistical Pattern Recognition.*

Statistical Pattern Recognition is an area in mathematical statistics that deals with random, metrically relatable stochastic vectorial variables. In its purest form it constitutes a particular discipline in the statistical decision theory.

Assume that each sample vector $\mathbf{x} = \mathbf{x}(t) \in \Re^n$, $t = 0, 1, 2, \ldots$ belongs to a class C_i $(i = 1, 2, \ldots, m)$; the goal is to define optimal **decision surfaces** (of dimensionality n-1) that divide $\Re^n$ into due class zones. A class is formed by all those objects fulfilling certain criteria which define the class itself. Each sample vector represents one of these objects, and each vector's component represents a criterion describing the class. The occurrence of a class C_i among the samples can be seen with the *a priori* probability $P(C_i)$, where

$$\sum_{i=1}^{m} P(C_i) = 1 .$$

For each sample vector $\mathbf{x}$ there exist m *joint conditional densities* $p(\mathbf{x}|C_i)$, each of which is the probability density function of input samples $\mathbf{x}$ belonging to class C_i. Using the *Bayes's rule* [21] we obtain:

$$p(\mathbf{x}|C_i) = \frac{p(C_i|\mathbf{x})p(\mathbf{x})}{P(C_i)} \ ,$$

where $p(\mathbf{x}) = \sum_{j=1}^{m} p(\mathbf{x}|C_j)P(C_j)$, and $p(C_i|\mathbf{x})$ is the *a posteriori* density.

In most real problems, the class distributions of the samples overlap, and therefore the above classification decision is bound to commit errors: every sample that belongs to class C_i but falls into the neighbouring zone will be counted as a misclassification.

If each classification decision is provided with equal weight, it can be shown [22] that the average misclassification cost is minimised if for every pair of "neighbouring" classes C_i and C_j the decision surface is analytically defined by the following equation in $\mathbf{x}$:

$$p(\mathbf{x}|C_i)P(C_i) = p(\mathbf{x}|C_j)P(C_j) \ . \qquad (3.46)$$

In other words, the optimal decision surfaces are defined by the crossings of the class distributions.

There exist many traditional approaches to approximate class distributions on the basis of available samples. In the **parametric methods**, the general form of the density function is fixed (e.g., multivariate normal distribution), and the class means and covariance matrices in the analytical expressions are then estimated on the basis of training samples. An example of a **non-parametric method** is the *Parzen window* method, in which a fixed "kernel" is defined and centered around every training sample; the "kernel" must be defined everywhere in $\Re^n$, and a multinormal distribution is the usual choice.

The manner in which mapping Networks approximate functions can be thought of as a generalisation of statistical regression analysis. In regression analysis, the specific form of a function to be fitted to data is first chosen and then fitting according to some error criterion (such as the mean squared error) is carried out.

The primary advantage of mapping Networks over classical statistical regression analysis is that Neural Networks have more general functional forms available than the statistical methods can effectively deal with. Neural networks are free from dependency on linear superposition and orthogonal functions, which linear statistical regression approaches must use. In linear statistical analysis the fitting functions can be non-linear functions of the input data, but only linear functions of the parameters. On the other hand, non-linear regression analysis resembles Neural Computing, because the fitting functions can be non-linear functions of both the input data and the parameters.

Enough experimental evidence has been gathered to state that mapping Networks are, in general, comparable to the best non-linear statistical regression approaches. The function approximations resulting from properly applied mapping Networks (provided that a sufficient amount of training data is available) are usually better than those provided by linear regression techniques. This difference is particularly important in high-dimensional spaces, where linear regression techniques often fail to produce an appropriate approximation.

□ □ □

Chapter 4

Event Classification in Hadronic Decays of the Z^0

In this chapter the problems related to the classification of the hadronic decays of the Z^0 are discussed. Then, a solution based on a four-output feed-forward Neural Network is described. Finally, a second solution based on a hybrid-scheme Network will be presented.

4.1 Introduction

As described in Section 1.3.1, a Z^0 boson can decay into a pair consisting of an elementary particle and its antiparticle. Decays into different charged lepton species are experimentally rather easy to distinguish, but the separation of the hadronic decays according to the ***quark flavour*** is rather difficult because quarks cannot be observed as free particles.

In a reaction $e^-e^+ \longrightarrow Z^0 \longrightarrow q\bar{q}$ five different quark-antiquark pairs can be generated:

1. $u\bar{u}$ pair (*up*-quarks);
2. $d\bar{d}$ pair (*down*-quarks);
3. $s\bar{s}$ pair (*strange*-quarks);
4. $c\bar{c}$ pair (*charm*-quarks);

5. $b\bar{b}$ pair (*beauty*-quarks).

The Standard Model predicts that the probability that the Z^0 boson decays into a $q\bar{q}$ pair depends on the quark flavour in such a way that, in first approximation, the relative hadronic *branching fractions* are:

$$\frac{\Gamma_{q\bar{q}}}{\Gamma_h} \simeq \begin{cases} 0.217 & \text{if } Q = -\frac{1}{3} \\ 0.175 & \text{if } Q = \frac{2}{3} \end{cases} ,$$

where $\Gamma_{q\bar{q}}$ is the partial width of the Z^0 into a $q\bar{q}$ pair, Γ_h is the total hadronic width of the Z^0, and Q is the charge of the quark.

The experimental determination of the hadronic branching fractions $\Gamma_{q\bar{q}}/\Gamma_h$ of the Z^0 is thus an important test of the Standard Model. In particular, the partial width of the Z^0 into $b\bar{b}$ pairs is especially sensitive to the top-quark mass.

The classification of the hadronic decays is problematic, because within a time during which it is not possible to make any measurement, the original $q\bar{q}$ pair fragments into jets of stable hadronic particles, hiding the nature of the primary decay. Several techniques have been proposed to identify the original quark flavour from the hadrons in the final state.

- **High Transverse Momentum Leptons**
  Heavy quarks b and c can be distinguished from *light* quarks u, d and s, by analysing their semileptonic decays into electrons and muons. The transverse momentum of the lepton with respect to the axis of the quark jet increases with increasing quark mass.

  By fitting the high transverse momentum (p_T) part of the lepton spectrum, one can measure the hadronic branching fractions of the Z^0 into heavy quarks, times the probability that the hadrons produced by these quarks decay into a final state including a lepton.

- **Impact Parameters**
  The b-quarks have a relatively long lifetime, so that many of the hadrons produced in $b\bar{b}$ events decay at a rather large distance (*secondary vertex*) from the point in which the Z^0 is produced (*primary vertex*). Therefore, the impact parameters of final state particles are on the average larger in b-quark decays than in case of light quark

decays. From the distribution of the impact parameters, one can measure the partial width of the Z^0 into $b\bar{b}$ pairs ($\Gamma_{b\bar{b}}$).

- **$D^{*\pm}$ decay**
  The production rate of $c\bar{c}$ events can be derived by identifying charmed mesons D^{*+} (and the charge conjugate particles) using the characteristic decay chain $D^{*+} \longrightarrow D^0\pi^+$, where the pion has a low transverse momentum with respect to the jet axis. The fraction of $c\bar{c}$ event can also be measured on the basis of excess of low transverse momentum pions.

- **Final State Radiation**
  The LEP measurements of the rate of the final state radiation from $q\bar{q}$ pairs can be used to compute the relative probabilities of the Z^0 decay into u- and d-type quarks. The absolute value of the charge of the u-type quarks is double with respect to that of the d-type quarks, and according to QED, the rate of production of photons is therefore times four larger. If the combined production rate of $u\bar{u}$ and $d\bar{d}$ pairs can be measured by some other methods, then the final state radiation can be used to calculate the individual rates of the light quarks.

- **Multidimensional Separation**
  As cited in Section 1.3.1, analysis algorithms are usually developed with the aid of simulated data generated by *Monte Carlo* programs reproducing all the statistical measures of the final state hadrons. The topology of an event can be represented in a fairly detailed manner through a set of event shape variables. The identification of flavours can then be performed by mapping this set of variables onto a feature space in which the different species are well separated.

  The simplest method to realise this kind of multidimensional separation is to use linear mapping [38]. A more powerful probe can be constructed by using non-linear Neural Network separators, in particular feed-forward Neural Networks [46,47].

4.2 Separation of Quark Flavours

The aim of this study is to develop a Neural Network program, which identifies the original flavour of a quark-antiquark pair produced in the decay of a Z^0. Such a Network can be considered as a filter, which implements a pattern classification algorithm to an environment in which the objects to be recognized are $q\bar{q}$ events. Each event belongs to one of four classes:

1. $(u\bar{u}+d\bar{d}\ )$[1] events class;
2. $(s\bar{s}\ )$ events class;
3. $(b\bar{b}\ )$ events class;
4. $(c\bar{c}\ )$ events class.

In the first stage of this study, it was tested whether topological properties of the event (i.e. properties related to the structure of multiparticle production) can be used by a Neural Network to classify not only $b\bar{b}$ events, but also $s\bar{s}$, $c\bar{c}$ and $(u\bar{u}+d\bar{d}\ )$-unresolved events [41]. Four Feed-Forward Neural Networks were constructed to identify each flavour independently, and the robustness of the separation against a wide range of systematic uncertainties related to the model-dependence of the classification was investigated.

A measurement of the hadronic branching fractions of the Z^0 was achieved with the following errors:

$$\begin{aligned}
\Gamma_{u\bar{u}+d\bar{d}}/\Gamma_h &= 0.417 \pm 0.015(stat) \pm 0.058(sys),\\
\Gamma_{s\bar{s}}/\Gamma_h &= 0.233 \pm 0.016(stat) \pm 0.051(sys),\\
\Gamma_{c\bar{c}}/\Gamma_h &= 0.139 \pm 0.010(stat) \pm 0.058(sys),\\
\Gamma_{b\bar{b}}/\Gamma_h &= 0.211 \pm 0.006(stat) \pm 0.020(sys).
\end{aligned}$$

The results are consistent with the Standard Model predictions.

In the next sections studies on two different kinds of Neural Networks (one feed-forward and one hybrid) will be described, including the implementation of the Networks. The new Networks were designed to have four output nodes to be able to classify any of the four event types in a single

[1]$u\bar{u}$ and $d\bar{d}$ separation is still unresolved

pass. We expect a single Network with four output nodes to be more efficient than the previous solution, and to take better care of correlations between distributions. We also tried to construct a feed-forward Neural Network with only two output nodes (in this case the output would be interpreted as a binary number), but this method did not seem to be sufficient to separate the flavours.

All the software packages were developed in FORTRAN 77 [56] which is still the most widely used programming language in the High Energy Physics software environment. The following software tools, developed in the Computing and Networking Division of CERN, were used for analysis and visualization of the obtained results:

- HBOOK [58], a subroutine package for handling statistical distribution analysis in a FORTRAN scientific computation environment;
- HPLOT [59], the HBOOK graphics interface;
- PAW [60], the Physical Analysis Workstation system for interactive analysis of data mainly oriented to High Energy Physics applications.

4.3 Simulation of Events

The simulation was performed by utilising the JETSET 7.2 Parton Shower Monte Carlo program (*JETSET PS*), which has proven after two years of activity of LEP, to reproduce well the main features of the hadronic decays of the Z^0 [9].

Charged particles fulfilling the followed criteria were used in the analysis:

1. momentum p larger than 0.1 GeV/c;
2. measured track length above 50 cm;
3. polar angle θ between 25° and 255°.

All particles were assumed to be pions.

Hadronic events were retained in the analysis if:

1. Each of the two hemispheres $cos\theta < 0$ and $cos\theta > 0$ contained a total energy of the charged particles $E_{ch} = \Sigma E_i$ larger than 3 GeV,where E_i are the particle energies;

2. The total energy of the charged particles seen in both hemispheres together exceeded 15 GeV;

3. there were at least 5 charged particles with momenta above 0.2 GeV/c;

4. The polar angle θ of the sphericity axis was in the range $40^o < \theta < 140^o$.

Two different sets of simulated data were used, each one composed of about 100,000 events. One set was used for training the Network (*training set*), and the other one was used for testing the Network with patterns which it had never seen before (*testing set*). In this way it was possible to measure the Network's ability to generalize.

4.4 Choice of the Discriminating Variables

Each input pattern supplied to the Network was described by 23 variables. Their choice came from examination of the literature, and from a study of flavour-dependent distributions based on the JETSET PS program. The particles in the event were clustered in jets according to the JADE/E0 algorithm [45], with $y_{cut} = 0.05$. In the following, the most energetic jet will be called "first jet", indicated by the superscript (f); the second most energetic jet will be called "second jet", indicated by the superscript (s). The description of the variables follows.

1. The sphericity $S^{(f)}$ of the first jet, calculated after a boost $\beta = 0.96$ along its axis. The axis of the jet was defined by the sum of the momenta of the particles belonging to it.

2. The directed sphericity $S^{(f)}_{1234}$ of the four most energetic particles in the first jet. For a set of Q tracks in a jet, this variable is defined as

$$S_Q = \frac{\sum_Q \vec{p_t}^2}{\sum_Q \mid \vec{p} \mid^2}$$

where the $|\vec{p}|$'s are the momenta in the rest frame of the set Q and the $\vec{p_t}$'s are their components perpendicular to the original jet direction in the laboratory frame.

3. The directed sphericity $S^{(s)}_{1234}$.

4. The invariant mass $M^{(f)}_{1234}$ of the four most energetic particles in the first jet.

5. The invariant mass $M^{(s)}_{1234}$ of the four most energetic particles in the second jet.

6..9. The products of the homologue direct sphericities for triplets of particles in the first and second jet, $S^{(f)}_{ijk} \times S^{(s)}_{ijk}$.

10..13. The products of the homologue invariant masses for triplets of particles in the first and second jet, $M^{(f)}_{ijk} \times M^{(s)}_{ijk}$.

14. The momentum of the slowest pion in the first jet, after a boost along the jet axis corresponding to a D^* energy equal to one half of the beam energy.

15. The momentum of the slowest pion in the second jet, with the same parametrisation as in 14.

16. The momentum $|\vec{p}|$ of the most energetic K^0 in the event (0 if no kaons reconstructed).

17. The momentum component perpendicular to the axis of the nearest jet $|\vec{p_t}|$ of the most energetic K^0 in the event (0 if no kaons reconstructed).

18. The sum over the jets of the ratios between the momentum of the leading particle and the momentum of the jet.

19. Sum of the track impact parameters, each one scaled by the error. Tracks with impact parameters greater than 2 mm are omitted because they are likely to come from secondary decays like K^0 and Λ.

20. Charge flow ($Q_f - Q_b$). The event is divided in two hemispheres by the thrust axis, and the axis is oriented so that it is pointing always to the forward hemisphere (polar angle $< 90^0$). The charge in the forward (backward) hemisphere is calculated by weighting the charge of each track by its momentum and dividing the weighted sum of charges by the sum of the absolute values of the track momenta, i.e.

$$Q_f = \frac{\sum_f \mid \vec{p} \mid Q}{\sum_f \mid \vec{p} \mid}$$

for all tracks in the forward hemisphere. This variable is sensitive to the quark charge, and the asymmetry of the quark polar angle (due to quark electroweak couplings).

21. Sum of the absolute values of the charges of the two most energetic jets, weighted by the momenta in the same way as in 19. This variable is sensitive to the quark charge.

22. Absolute momentum $\mid \vec{p} \mid$ of the most energetic lepton (0 if no leptons are reconstructed). The lepton can be a muon or an electron.

23. $|\vec{p_t}|$ of the most energetic lepton with respect to the closest jet.

All variables were rebinned in such a way that they were ranging from 0 to 1. An example of the sensitivity of a single variable to b-quarks is shown in Fig. 4.1.

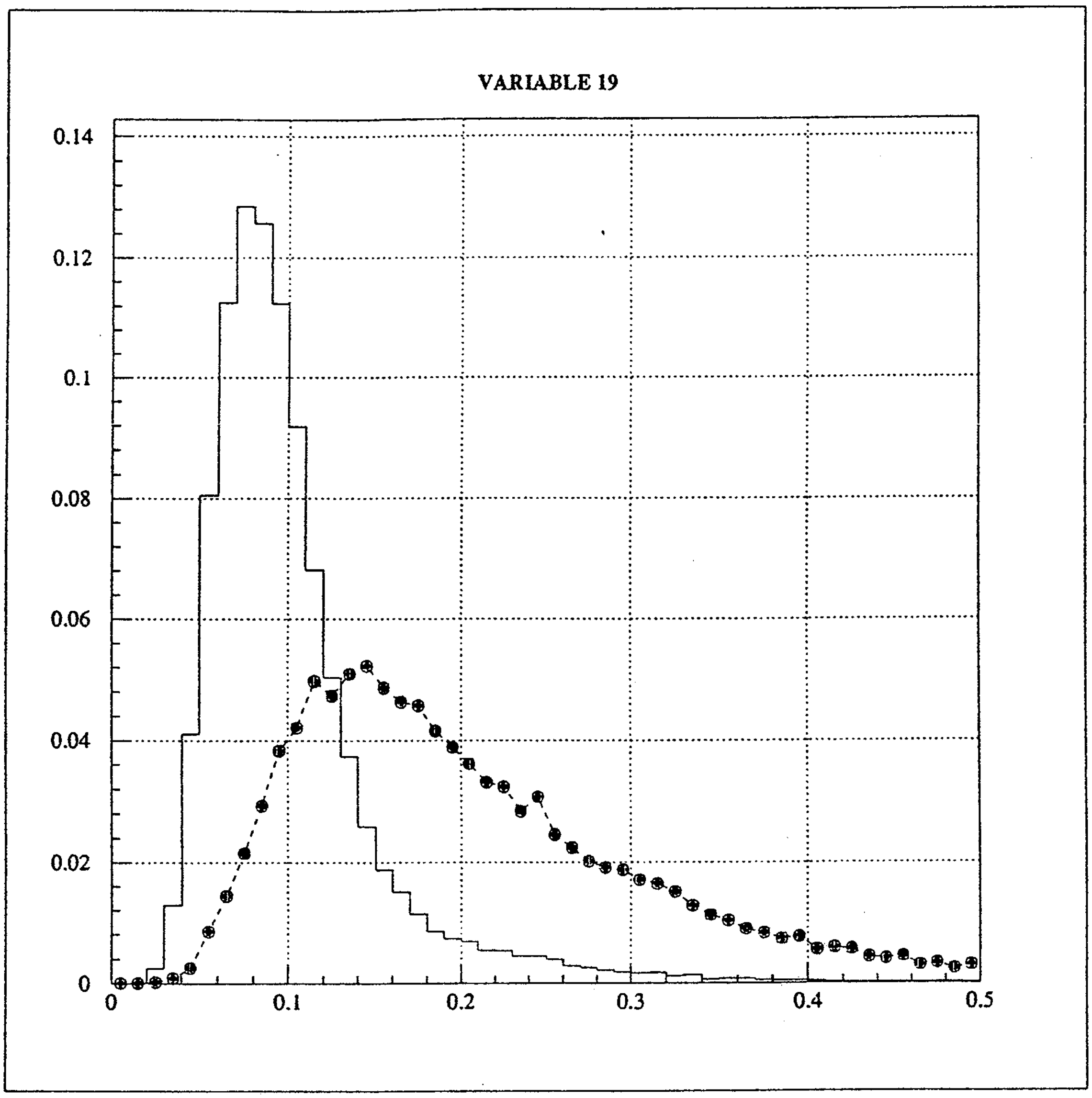


Figure 4.1: *Scaled impact parameter sum for $b\bar{b}$ (polymarker line) and non-$b\bar{b}$ (solid line) events.*

4.5 Measurement of the Performance of the Networks

Each event corresponds to one input pattern and is composed of 23 variables describing the shape and the particle contents of the event. In the simulated training data, an additional variable, the *target* of the event, was included. This variable was a real value k^μ, where $1 \leq \lfloor k^\mu \rfloor \leq 5$, and $\lfloor k^\mu \rfloor$ was the integer part of k^μ, indicating the flavour associated to the input pattern:

$\lfloor k^\mu \rfloor$	event flavour
1	$u\bar{u}$ event
2	$d\bar{d}$ event
3	$s\bar{s}$ event
4	$c\bar{c}$ event
5	$b\bar{b}$ event

The target value is necessary for calculating the *error* (δ) signals in the back-propagation learning algorithm. The true answer also gives a quantitative measure of the Network performance, when comparing this to the Network output after every input pattern.

We define three different measures for quantifying the general Network performance during both training and testing phase: (a) the *signal efficiency*; (b) the *background rejection* and (c) the *purity*.

- The **signal efficiency** is the rate of correct classifications performed by the Network, and can be defined as:

$$\epsilon_q = \frac{N_q^a}{N_q} , \tag{4.1}$$

where N_q is the total number of patterns given as input to the Network labelled as flavour q, while N_q^a is the number of patterns of this flavour correctly classified by the Network.

- As *background* we consider the complement set of input patterns to flavour q (N_{B_q}). The **background rejection** can be defined as:

$$\rho_q = 1 - \frac{N^a_{B_q}}{N_{B_q}} , \tag{4.2}$$

where $N^a_{B_q}$ is the number of input patterns wrongly labelled as q. The rejection will represent the rate of input patterns correctly *rejected* by the Network as background.

- The **Purity** can be interpreted as the rate of "genuine" input patterns labelled as q versus the total number of input patterns classified as q:

$$p = \frac{N^a_q}{N^a_q + N^a_{B_q}} = \frac{\epsilon_q}{\epsilon_q + (1-\rho_q)R} , \tag{4.3}$$

where $R = N_{B_q}/N_q$.

In order to obtain a good classification with the Network, these three parameters must be maximised.

4.6 The Back-Propagation Neural Network Solution

In this Section we will be describe the realisation of a feed-forward Neural Network capable of performing a good mapping $\mathcal{F} : P \subset \Re^{23} \longrightarrow F \subset \Re^4$ related to the classification problem presented above.

In the next subsection, we will explain the choice of the Network architecture, and describe how the involved problems were solved. In Section 4.6.2, the different training strategies adopted in order to obtain the best performance will be discussed, and then the results obtained after this first processing phase will be presented. Finally, in Section 4.6.3, the testing phase will be described and the final results will be presented.

4.6.1 The Neural Network Architecture

The choice of the Network architecture is a nontrivial problem. It has been shown that the performance of the Networks change considerably when using different Network architectures. On the other hand, one of our purposes was to construct an economic Network with the smallest possible structure, in terms of number of units. In fact, it has been proved that in order to realise the best possible topology of the input space within the internal representation of the Network, one has to limit the number of total connections appropriately.

Many attempts were made in order to find the exact number of hidden nodes, starting from structures with more than one hidden layer to structures with only 5 hidden units.

The Input Layer

There are 23 variables describing the input pattern, each one of them assigned as input to one of the input nodes. The input layer consists of 23 neurons or *fan-in* units that have a linear transfer function given by the identity function. Unique rule of these units is to distribute the variables constituting the input pattern x belonging to the pattern space P, to the subsequent layers of the Network.

The Necessary Number of Hidden Layers

As discussed in Section 3.2.6, it has been proved that only *one* hidden layer is sufficient to approximate any continuous function. A Network with more than one hidden layer is capable of separating any classes and therefore it may well permit a solution to the current problem.

Studies made by Lippmann [26] in 1987 demonstrate that the number of nodes in the second hidden layer (i.e., the one nearest to the output layer), must be greater than one, but generally lower than the number of units in the first hidden and input layer. Several two-layered Networks with *pyramid-like* structure (see Fig. 4.2) were tested, but none of them was found profitable. Performance obtained with these Networks was comparable with that obtained with the best one-layered one, but never better. On the contrary, it was found that a Network with two hidden layers reached a stable configuration in a more difficult way than a one-layered Network, slowing down the learning process. Therefore it was found reasonable to limit the number of hidden layers to one.

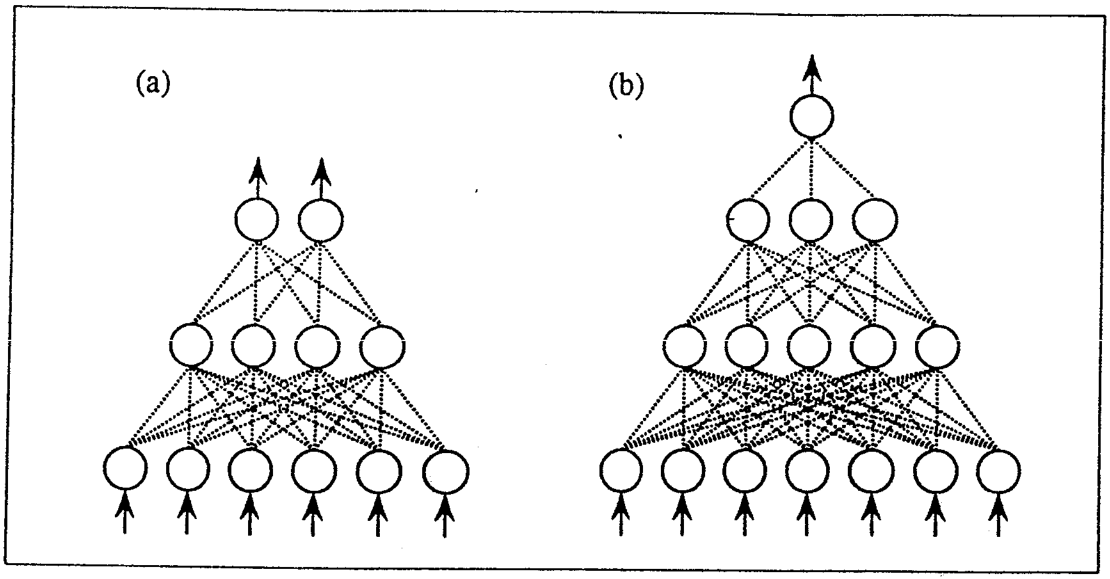


Figure 4.2: *General pyramid structures of layered-Networks.*

The Number of Hidden Units

- In a first stage of the Network structure study, it was found useful to use a pruning procedure in order to reduce the number of hidden nodes. Starting with a large number of hidden units (~50), the weights updating was performed by using the equation:

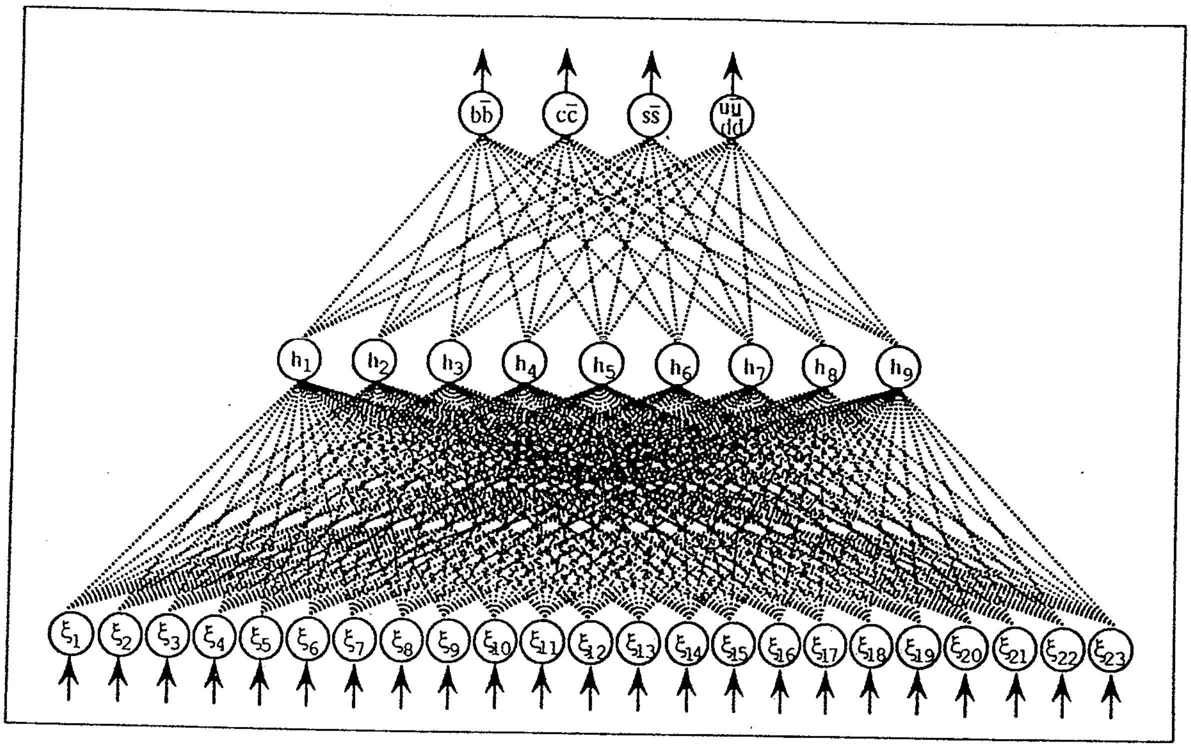
  $$w_{ji}^{new} = (1 - \epsilon_{ji}) w_{ji}^{old} ,$$

  where ϵ_{ji} is given by

  $$\epsilon_{ji} = \frac{\gamma\eta/2}{(1 + w_{ji}^2)^2} ,$$

  tuning γ in such a way that it is incremented ($\gamma \rightarrow \gamma + \nu$) whenever the function error value decreases with a very small value of ν.

  This updating rule has the effect that small weights decay more rapidly than large ones. Using a *pruning* procedure that disables all the neurons having no connections with absolute weight value greater than a fixed small threshold (0.1, in our case), it is possible to reduce the number of hidden neurons, and obtain a Network containing only the necessary weights needed to represent the problem.



$b\bar{b}$
$c\bar{c}$
$s\bar{s}$
$u\bar{u}$ $d\bar{d}$
h_1 h_2 h_3 h_4 h_5 h_6 h_7 h_8 h_9
ξ_1 ξ_2 ξ_3 ξ_4 ξ_5 ξ_6 ξ_7 ξ_8 ξ_9 ξ_{10} ξ_{11} ξ_{12} ξ_{13} ξ_{14} ξ_{15} ξ_{16} ξ_{17} ξ_{18} ξ_{19} ξ_{20} ξ_{21} ξ_{22} ξ_{23}

Figure 4.3: *The structural configuration of the constructed Neural Network.*

Using pruning it was possible to reduce the number of hidden neurons to 23, starting from a configuration of 50 nodes. Problems arose from the fact that this procedure seemed to keep into account only partially the general Network efficiencies needed for a fine tuning of the γ parameter.

- In a second stage, one-layered Networks with hidden nodes varying from 5 to 23 were studied. Structures with 15 and 9 hidden nodes were found to perform better than the others, so they were studied closer. The architecture with the smallest number of nodes was chosen and the final Network structure is represented in Fig. 4.3.

The Output Layer

The output layer of the Network consists of four neurons, each one associated with a flavour category (see Fig. 4.3). For every input pattern presented to the Network, the four output nodes produce four real numbers as result of the sigmoid transfer function associated to them. The Network output is then interpreted by using the *winner takes it all* technique: the flavour of the event is interpreted to be the one, the node of which gives the largest output value.

4.6.2 The Training Phase

The training phase of the Network was performed with a set of 86,990 simulated Z^0 decays into $q\bar{q}$ pairs in the proportions predicted by the Standard Model. The training set was composed by an equal number of patterns for each flavour type. This choice improves the classification, due to the fact that the back-propagation algorithm tries to minimise the error on the entire training set. If the number of patterns is not equal, the Network will concentrate on the classification of the flavour which occurs more frequently than the others.

If the teaching data is divided into too small parts, this may lead into overlearning of the most recent events and into forgetting the previous ones. Consequently, if we choose an epoch to consist of all the data we have got, the learning process does not favour any particular event and the Network is able to represent the desired classification function. The entire training process should therefore be divided into a defined number of sessions (*training epochs*). During each of them, the whole training set is filtered by the Network.

The Input Data

During the learning process, the representation of the input data is an important aspect. Improvements in Network's generalisation ability can be obtained by addition of noise, and this can be performed by choosing the input patterns in a random order from the training set.

The Sigmoid Functions and the Bias Value

The function computed by each neuron j in the Network can be summarised by the following expression:

$$O_j^{(l)} = \begin{cases} x_j & \text{if } l = 1 \\ g\left(\sum_{i=1}^{n_{l-1}} w_{ji} O_i^{(l-1)} - \Theta_j\right) & \text{if } l = 2, ..., L \end{cases} \tag{4.4}$$

where $O_i^{(l-1)}$ denotes the output of the i-th neuron of layer $l-1$, w_{ji} is the connection strength between the i-th neuron of layer $l-1$ and the j-th neuron of layer l, and g is the ***sigmoid*** or ***activation function.***

Sigmoid functions used for our purpose are

$$g(x) = \frac{1}{1+e^{-x/T}} , \tag{4.5}$$

or

$$g(x) = \frac{1}{2}(1+\tanh\frac{x}{T}) , \tag{4.6}$$

where $g : \Re \rightarrow [0,1]$ (see Fig. 4.4(a)). It is also possible to use a different activation function

$$g(x) = \tanh\frac{x}{T} , \tag{4.7}$$

where $g : \Re \rightarrow [-1,1]$ (see Fig. 4.4(b)). The output range is now from -1 to 1 for each hidden and output unit. In all cases, the output cannot reach its extreme values without infinitely large weights.

The ***threshold*** or *BIAS* value Θ_j in (4.4) is represented in the Network by an auxiliary ***bias-unit*** with a constant output ξ_0 connected to every hidden and output unit:

$$\Theta_j = w_{j0} \cdot \xi_0 ,$$

where w_{j0} is the strength of the connection between unit j and the *BIAS*-unit. It behaves as an ***adaptive threshold***-unit, changing its value Θ according to the weights updating during the learning process. We can rewrite equation (4.4) for unit j in layer l (with $l > 1$), as

$$O_j^{(l)} = g\left(\sum_{i=1}^{n_{l-1}} w_{ji} O_i^{(l-1)} - \Theta_j\right) = g\left(\sum_{i=0}^{n_{l-1}} w_{ji} O_i^{(l-1)}\right) , \tag{4.8}$$

where $O_0^{(l-1)} = \xi_0$ is a constant real value (*BIAS*).

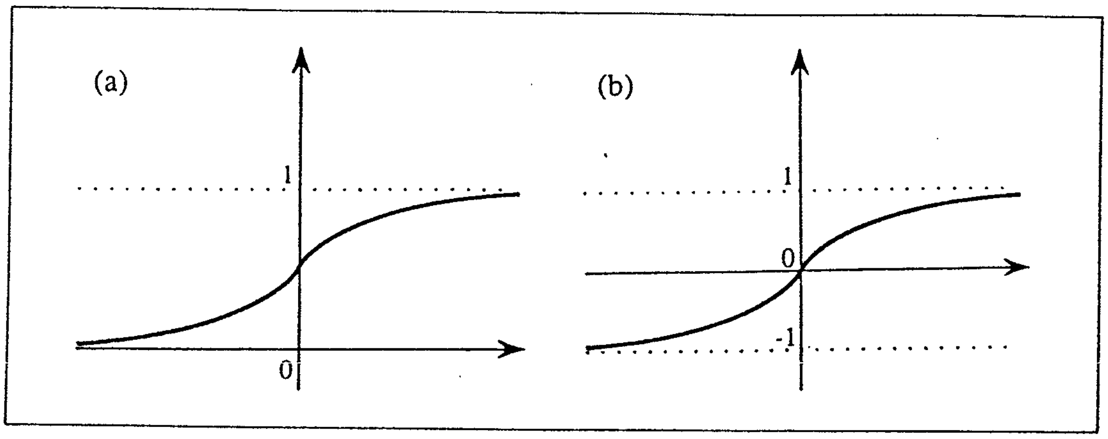


Figure 4.4: *(a) Sigmoid function for units with output ranging from 0 to 1; (b) Sigmoid function for units with output ranging from -1 to 1.*

Error Functions

Given a set of input pattern vectors ξ_i^μ, together with associated target values ζ_j^μ, the back propagation learning algorithm attempts to adjust the weights in order to minimise the *error* E to achieve these target values. Several types of error functions can be used, depending also on the transfer function associated to each processing element. The most commonly used error function is the *quadratic* cost measure

$$E = \sum_\mu E_\mu = \frac{1}{2}\sum_{j\mu}(\zeta_j^\mu - O_j^\mu)^2 \ , \tag{4.9}$$

where O_j^μ is the output of the j-th node when $\hat{\xi}^\mu$ is presented as input.

This error function has not been the only one used to train the Network. It is possible to replace the $(\zeta_j^\mu - O_j^\mu)^2$ factor in (4.9) by any other differentiable function $F(\zeta_j^\mu, O_j^\mu)$ that can be minimised when its arguments are equal, and then derive a corresponding update rule.

◇ *The Entropic Error*

Considering that ζ_j^μ can be only 0 or 1, let $f_j(\hat{\xi}, \hat{\theta})$ be the prediction of the probability of ζ_j^μ given the input vector $\hat{\xi}$, where $\hat{\theta}$ is a set of parameters determined by our learning algorithm. In the Neural Network case, the $\hat{\theta}$ are the connection weights, and

$$f_j(\hat{\xi}^\mu, \hat{\theta}) = f_j(\hat{\xi}^\mu, \{w_{ji}\}) = O_j^\mu \ .$$

Now lacking a priori knowledge of good $\hat{\theta} = \{w_{ji}\}$, the best one can do is to choose the parameters $\hat{\theta}$ to maximise the likelihood that the given set of patterns should have occurred [21]. The formula for this likelihood p, is:

$$p = \prod_\mu \left[\prod_{\{j|\zeta_j^\mu=1\}} O_j^\mu \prod_{\{j|\zeta_j^\mu=0\}} (1 - O_j^\mu) \right] ,$$

or

$$\log(p) = \sum_\mu \left[\sum_{\{j|\zeta_j^\mu=1\}} \log(O_j^\mu) \sum_{\{j|\zeta_j^\mu=0\}} \log(1 - O_j^\mu) \right] .$$

Applying this last expression to the case, where the $\hat{\zeta}$ are probabilities taking values [0,1], yields

$$\log(p) = \sum_{j\mu} \left[\zeta_j^\mu \log(O_j^\mu) + (1 - \zeta_j^\mu) \log(1 - O_j^\mu) \right] . \qquad (4.10)$$

This kind of expressions often arise in physics and information theory and are generally interpreted as *entropy* measures.

The *entropy* error function can be defined as

$$E = -\log(p) . \qquad (4.11)$$

This measure has a natural interpretation in terms of learning the correct probabilities of a set of hypotheses represented by the output units. The advantage is, qualitatively, that the error function diverges if the output of one unit saturates at the wrong extreme. The quadratic measure (4.9) approaches a constant in this case, and therefore the learning can float around on a relatively flat plateau of E for a long time.

The back-propagation algorithm is now generalised from minimising the error to maximising the entropy or *log-likelihood* function. Direct differentiation shows that only the expression (3.15) for error δ-signals in the output layer changes, all the other equations of back-propagation remain unchanged.

Using the equation (4.5) as the activation function, the output for each neuron j for the input pattern μ is given by

$$O_j^\mu = g(net_j^\mu) = \frac{1}{1 + e^{-net_j^\mu / T}} , \tag{4.12}$$

where T is the noise parameter or *temperature*.

The derivative with respect to this unit, is

$$\begin{aligned} g'(net_j^\mu) &= \frac{1}{T} \frac{e^{-net_j^\mu/T}}{(1 + e^{-net_j^\mu/T})^2} & (4.13) \\ &= \frac{1}{T} g(net_j^\mu)(1 - g(net_j^\mu)) \\ &= \frac{1}{T} O_j^\mu (1 - O_j^\mu) . & (4.14) \end{aligned}$$

Differentiating the entropy cost function with respect to O_j^μ, we obtain:

$$\frac{\partial E_\mu}{\partial O_j^\mu} = -\frac{\zeta_j^\mu - O_j^\mu}{O_j^\mu (1 - O_j^\mu)} . \tag{4.15}$$

Then, substituting (4.14) and (4.15) in the equation of δ_j^μ of the output layer:

$$\delta_j^\mu = -\frac{\partial E_\mu}{\partial O_j^\mu} g'(net_j^\mu) = \frac{1}{T} (\zeta_j^\mu - O_j^\mu) , \tag{4.16}$$

which is a simple linear function of inputs and outputs.

The use of the entropy measure has been shown to be appropriate if the training set data are probabilistic or fuzzy as in our case. It also has seemed to speed up the learning process.

◇ *The Asymmetric Error*

If l identifies the hidden or output layer and n is the number of nodes belonging to that layer, then we can represent the activation of node j on layer l referred to the input pattern μ as $x^{\mu}_{l_j}$, where $1 \leq j \leq n$. Using the sigmoid function

$$O^{\mu}_{l_j} = g(x_{l_1}, x_{l_2}, ..., x_{l_n}, T) = \frac{e^{x_{l_j}/T}}{\sum_{i=1}^{n} e^{x_{l_i}/T}} , \tag{4.17}$$

a suitable error function for this representation can be:

$$E = \sum_{j\mu} \zeta^{\mu}_j \log \frac{\zeta^{\mu}_j}{O^{\mu}_j} , \tag{4.18}$$

called the ***asymmetric*** function or ***Kullback*** measure and giving the same error δ signal (4.16) as the entropic function, besides a constant factor.

Using this measure, each unit keeps into account the information related to all nodes in the entire layer to which it belongs. The Network performs its learning by considering the correlations existing among all data in a more detailed manner.

With the use of this error function, the performance is comparable to that achieved with the quadratic measure (4.9).

◇ *Other Error Measures*

Several alternative error functions have been studied, in order to find the one which could represent the best possible surface on which to perform the gradient descent. It has been shown that none of the following measures improves the learning.

For the sake of completeness, they will be briefly described.

- The quadratic cost function can be generalised in a *non-Euclidean* metric, in such a way that:

$$E = \frac{1}{r} \sum_{j\mu} (| O^{\mu}_j - \zeta^{\mu}_j |)^r , \tag{4.19}$$

  where $r \in \Re$ is the *power* value. Then, for the error δ:

$$\delta^{\mu}_j = | O^{\mu}_j - \zeta^{\mu}_j |^{r-1} O^{\mu}_j (1 - O^{\mu}_j) g'_j(net^{\mu}_j) \text{sgn}(O^{\mu}_j - \zeta^{\mu}_j) \tag{4.20}$$

For $r = 2$ the expression is equivalent to the standard back-propagation model.

It has been proved [50] that noise in the target domain may be reduced by using power values less than 2 in such a way that the error function will tend to model non-gaussian distributions where the tails of the distributions are more pronounced than in the gaussian. On the other hand, the sensitivity of partition planes to the geometry of the problem is increased with increasing power values.

- Another approach is to change the $(\zeta_j^\mu - O_j^\mu)$ term from the standard equation (3.15) of δ_j^μ, increasing the δ_j^μ when $\mid \zeta_j^\mu - O_j^\mu \mid$ becomes large, as:

  $$\delta_j^\mu = \text{arctanh}\frac{1}{2}(\zeta_j^\mu - O_j^\mu) \ . \tag{4.21}$$

  This rule must be used with activation function (4.7), with a range from -1 to +1 for each hidden and output neuron. In this case, $\delta_j^\mu \to \infty$ when $O_j^\mu \to -\zeta_j^\mu$.

- In order to improve the separation, it has been tried to strengthen the standard quadratic error measure by adding an extra penalty term of higher order:

  $$E = \frac{1}{2}\sum_{j\mu}\left[(\zeta_j^\mu - O_j^\mu)^2 + \left(\frac{\zeta_j^\mu - O_j^\mu}{s}\right)^{2n}\right] , \tag{4.22}$$

  where s is a small real value and $n \geq 1$ is an integer.

Updating of the Weights

The weights in the Network are initialised with small random values in the range of [-0.1,0.1]. It has been observed that an initialisation with low values speeds up the learning process, especially in cases where the input signals are quite different in magnitude.

The weights are then computed according to the generalised delta-rule discussed in Section 3.2.4:

$$\Delta w_{ji}(t+1) = -\eta\frac{\partial E}{\partial w_{ji}} + \alpha\Delta w_{ji}(t) \ . \tag{4.23}$$

The updating of the weights is performed after every fixed number of training passes. This is due to the fact that, in most cases, consequent positive and negative contributions in the weight changes may confuse the learning process of the Network. It is profitable for the updating to use the cumulative error from a fixed number of input patterns, in order to smooth out possible fluctuations.

The updating frequency was fixed to be every 10 input patterns; decreasing this frequency makes the learning more difficult.

A different updating rule was also tried, using a wider update frequency:

$$\Delta w_{ji}(t+1) = -\eta \cdot \operatorname{sgn}\left(\frac{\partial E}{\partial w_{ji}}\right) . \tag{4.24}$$

Using this method, the magnitude of the gradient of the error measure with respect to the weights does not matter, but only the sign of the gradient is used to determine the direction of the constant change for each weight. No improvement could be detected, and the mean error did not change considerably during the teaching phase.

Updating of the Parameters

After each training epoch it is possible to change the values of the three parameters η, α and T involved in the Network learning process.

As discussed in Section 3.2.4, η is the *learning strength* parameter and represents the gradient descent step size, while α is the *momentum* coefficient whose task is to speed up the learning process and dump out possible oscillations in the gradient descent. Tuning of these two parameters is essential to assure learning stability. Ideally, at the beginning of learning, weights' changes should be large and should decrease as learning proceeds. Since the larger η the larger the changes of the weights, η should be reduced as the learning proceeds. On the other hand, α should be increased in order to avoid awkward oscillations.

The temperature T is a measure of stochasticity in the gradient descent process. Too low values of T obstruct Network's learning, since the sigmoid function appears as a threshold step function. Also this parameter can be decreased as learning proceeds, starting from a high value.

Systematic updating of these parameters can be performed in two different ways:

	p_0	l	k
η	0.05	0.0001	0.05
α	0.4	0.9	0.14
T	3.0	0.8	0.71

Table 4.1: *Set values for η, α and T.*

- as **linear updating** – that is, at every learning epoch each parameter will be decreased or increased by a suitable constant, until the defined limit is reached.
- as **geometric updating** – each parameter p will be changed geometrically as learning proceeds starting from an initial value p_0:

$$p_{t+1} = p_t \cdot \epsilon_t \ , \qquad \epsilon_t = \left(\frac{l_p}{p_t} \right)^{k_p} \ , \tag{4.25}$$

  where $l_p \in \Re$ is the lower or upper limit for parameter p, and k_p is a real constant value adapted to parameter p.

In table 4.1 all the parameter features are presented. The temperature of the Network was varied systematically from 3.0 to 0.8 (the upper and lower limits; out of this range the Network performance was negatively affected), but the update of T during learning has found to be not convenient. The problem was to find the suitable value of T for each flavour classification – it was found, for example, that increasing the temperature the $b\bar{b}$ classification improved but the $s\bar{s}$ and ($u\bar{u}$+$d\bar{d}$) deteriorated.

It was also tried to use different temperatures in the hidden and output layers, fixing a higher value in the hidden layer in order to reduce noise in the final classification stage. The best results were obtained, however, by fixing T to 2.0 as a unique value during the entire training process.

On the other hand, parameters η and α were geometrically modified as learning proceeded. Their update process was temporarily stopped whenever at the end of every learning epoch the mean error value increased.

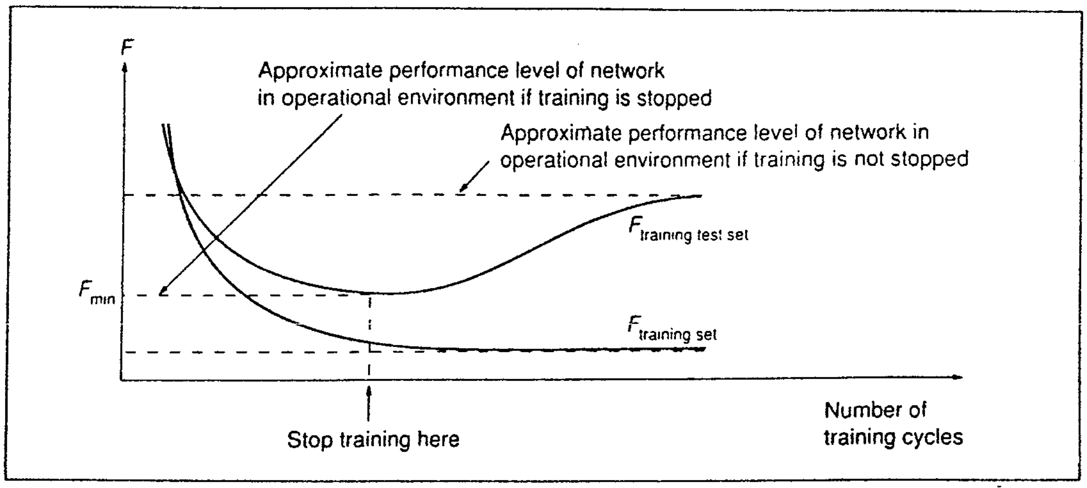


Figure 4.5: *Training set error vs. test set error as a function of the number of training epochs. This illustrates the phenomenon of overtraining.*

Overtraining

At every learning step, an indication of the performance of the Network can be inferred from the mean error function value. A plot of the evolution of this function must show a decreasing behaviour; if the evolution stabilises at a plateau value, it indicates a learning convergence or a local minimum. In the last case it is possible to "jump" out the local minimum, adding noise into the system or drastically resetting the learning parameters and starting again the training (we can call this, *reset-mode* training).

An unexpected and peculiar phenomenon occurring with feed-forward Networks using back-propagation algorithm is *overtraining* (see Fig. 4.5). This interesting phenomenon can be observed by looking at the mean error function plots in the training phase and in the testing phase together as learning proceeds. Typically, the error of the Network as measured using the training set constantly decreases, while the test set error decreases for a while and then begins to increase. The source of this problem seems to be related to the manner in which the Network forms its mapping approximation. As learning progresses there is almost always an intermediate stage at which the Network approximation reaches a good balance between accurately fitting the training patterns and yet still exhibiting a reasonably good interpolation capability between these examples. It is typically at this intermediate point that the training should be stopped.

The overtraining problem can be solved by performing a testing phase after every fixed number of training epochs and checking the mean test set error value $\overline{E}^{test}$. One can stop training or save the Network configuration when $\overline{E}^{test}$ reaches its minimum (*check-mode* training). The learning will be slowed down, but in this way we will obtain a more reliable performance of the Network.

Training Results

The study of the Network performance led us to use a fixed value of 2.0 for the temperature T, while η and α were geometrically updated as learning proceeded (parameter values are summarised in table 4.1). The selected error function was the entropic error function.

Training was performed in the *reset-mode* for a total of 1,000 iterations or training epochs, to ensure a true convergence of the Network. The behaviour of the error function is illustrated in Fig. 4.6(a). Then the Network was trained in the *check mode* for a total of 200 iterations, saving the best configuration defined by the test check. The error function behaviour is illustrated in Fig. 4.6(b).

In Fig. 4.7 the *signal* (solid line) and *background* (dotted line) efficiencies are plotted for each of the four classes as a function of the number of the training epochs. A good classification is achieved when the signal efficiency (ϵ_q) is represented by an increasing function and correspondingly, the background efficiency ($\epsilon_{B_q} = 1 - \rho_q$) is a decreasing function. One can observe a good performance for $b\overline{b}$ events, while a strong correlation between ($u\overline{u}+d\overline{d}$) and $s\overline{s}$ events is visible.

In table 4.2 the Network performance is summarised in terms of *efficiency*, *background rejection* and *purity* at the end of learning.

4.6.3 Testing Phase and Results

The testing phase of the Network was performed by using a different set of simulated events (*testing set*), never seen before by the Network. The Network with a weight configuration previously generated during a training phase was tested on the new test set and its real performance was evaluated.

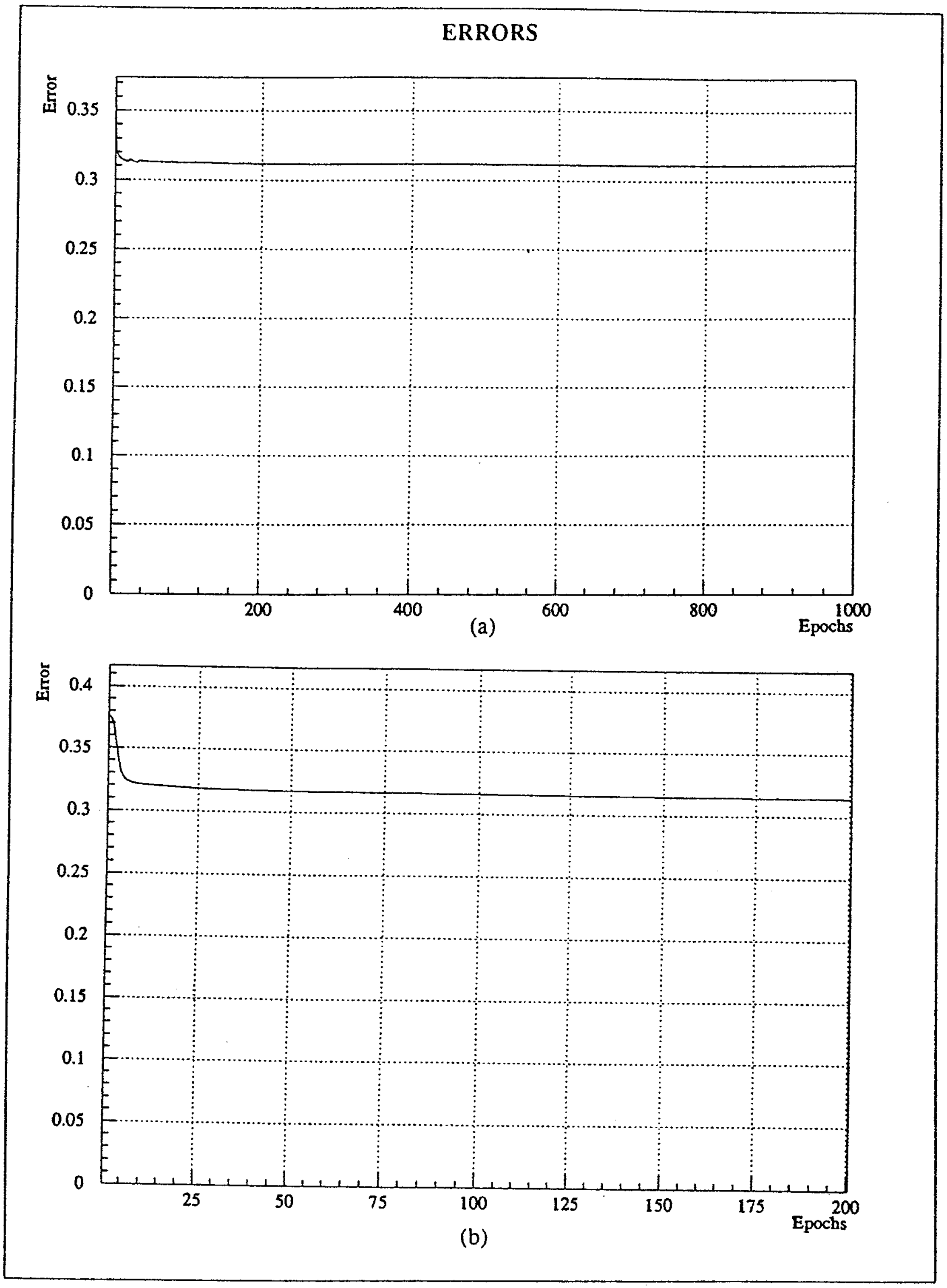


Figure 4.6: *The value of the error function as a function of the number of training epochs. (a) Reset-mode training on 1,000 iterations; (b) Check-mode training on 200 iterations.*

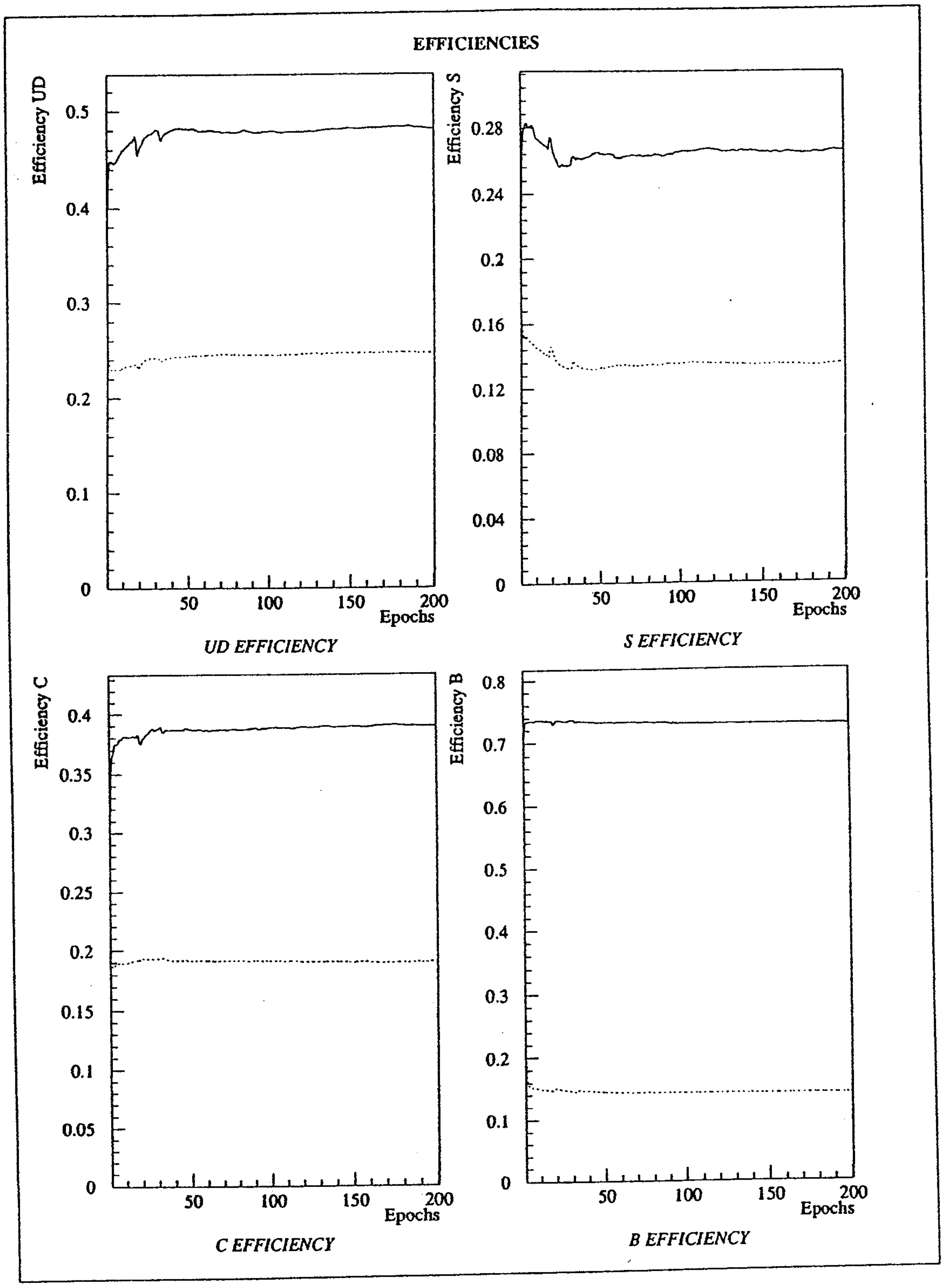


Figure 4.7: *Signal (solid line) and Background (dashed line) efficiencies as a function of the number of the training epochs for all flavours.*

Training	ϵ_q	ρ_q	p
$(u\bar{u}+d\bar{d})$	46.6 %	76.4 %	39.7 %
$s\bar{s}$	27.9 %	86.1 %	40.1 %
$c\bar{c}$	38.7 %	81.0 %	40.4 %
$b\bar{b}$	73.1 %	85.2 %	62.3 %

Table 4.2: *Network performance after training.*

The results presented here were obtained during a test session after a check-mode training of 200 iterations.

In Fig. 4.8 the output distributions of the desired winner nodes for each flavour are represented. Signal distributions are shown by the dotted line, while background distributions are described by the solid lines. A good classification is visible as a clear shape separation. The best results are obtained for $b\bar{b}$ classification. Also in this plot the strong correlation between the light quark shapes can be observed.

Plots in Fig. 4.8 cannot be considered as the real representation of the output distributions for each flavour, because they refer only to the winner nodes, and do not show the information related to the others. It is quite difficult have a precise graphic representation of distributions in a four-dimensional space.

It is, however, possible to obtain a more realistic representation of the Network performance by plotting the number of events as a function of the difference between the desired winner node output and the highest output of the remaining output nodes (Fig. 4.9). If the winner node corresponds to the desired one, the difference will be positive, otherwise it will be negative denoting a wrong classification by the Network. From this plot it is also possible to obtain signal efficiencies by estimating the ratio between the integrated area from 0 to 1 and the global integrated area:

$$\epsilon_q = \frac{\int_0^1 c_q(x)dx}{\int_{-1}^1 c_q(x)dx},$$

where c_q is the shape associated to the q-flavour distribution.

In Table 4.3 the final Network performance is summarised in terms of

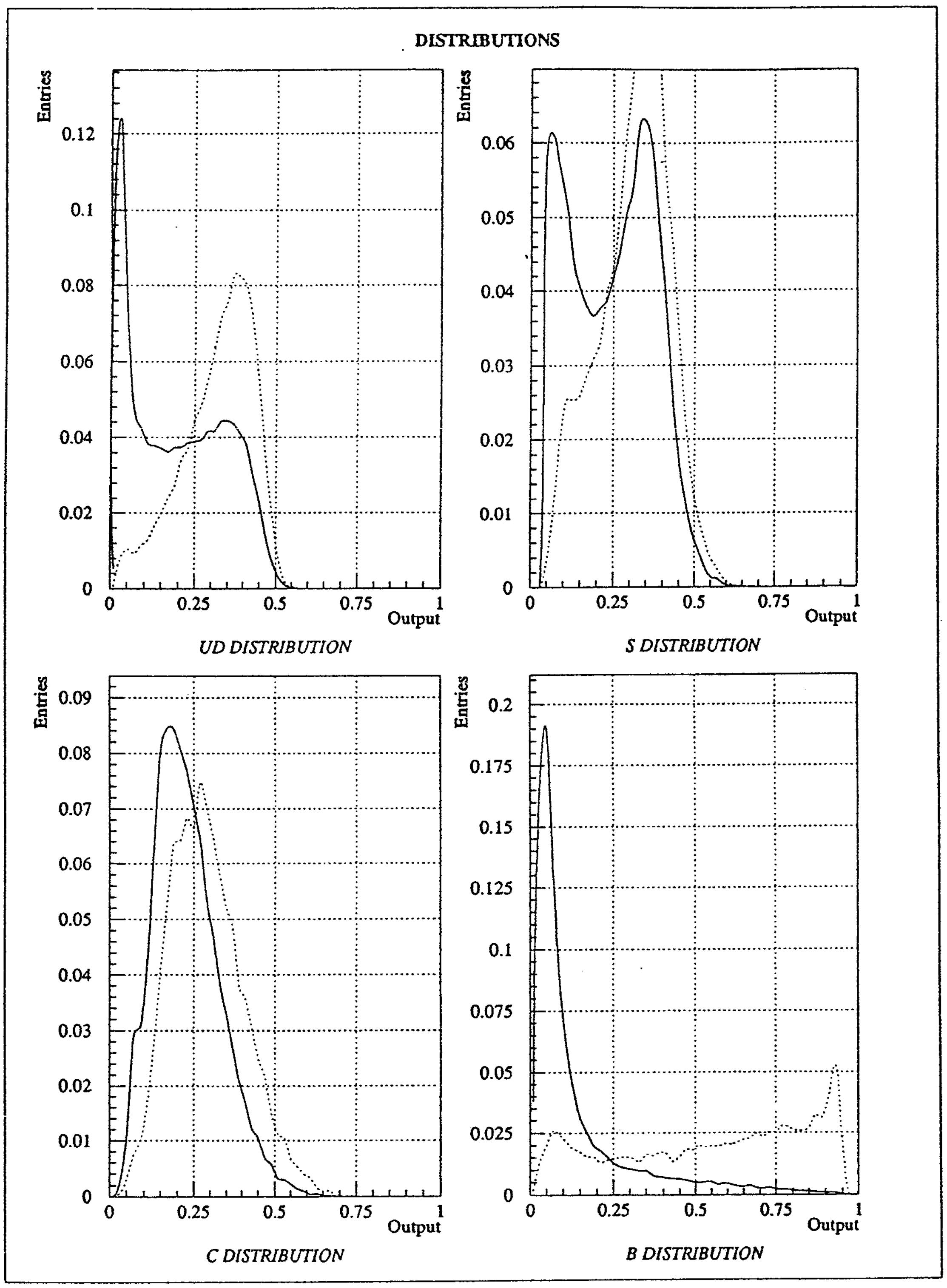


Figure 4.8: *Output distributions of the desired winner nodes. Signal (dotted line) and Background (solid line) distribution areas have been normalized to unity.*

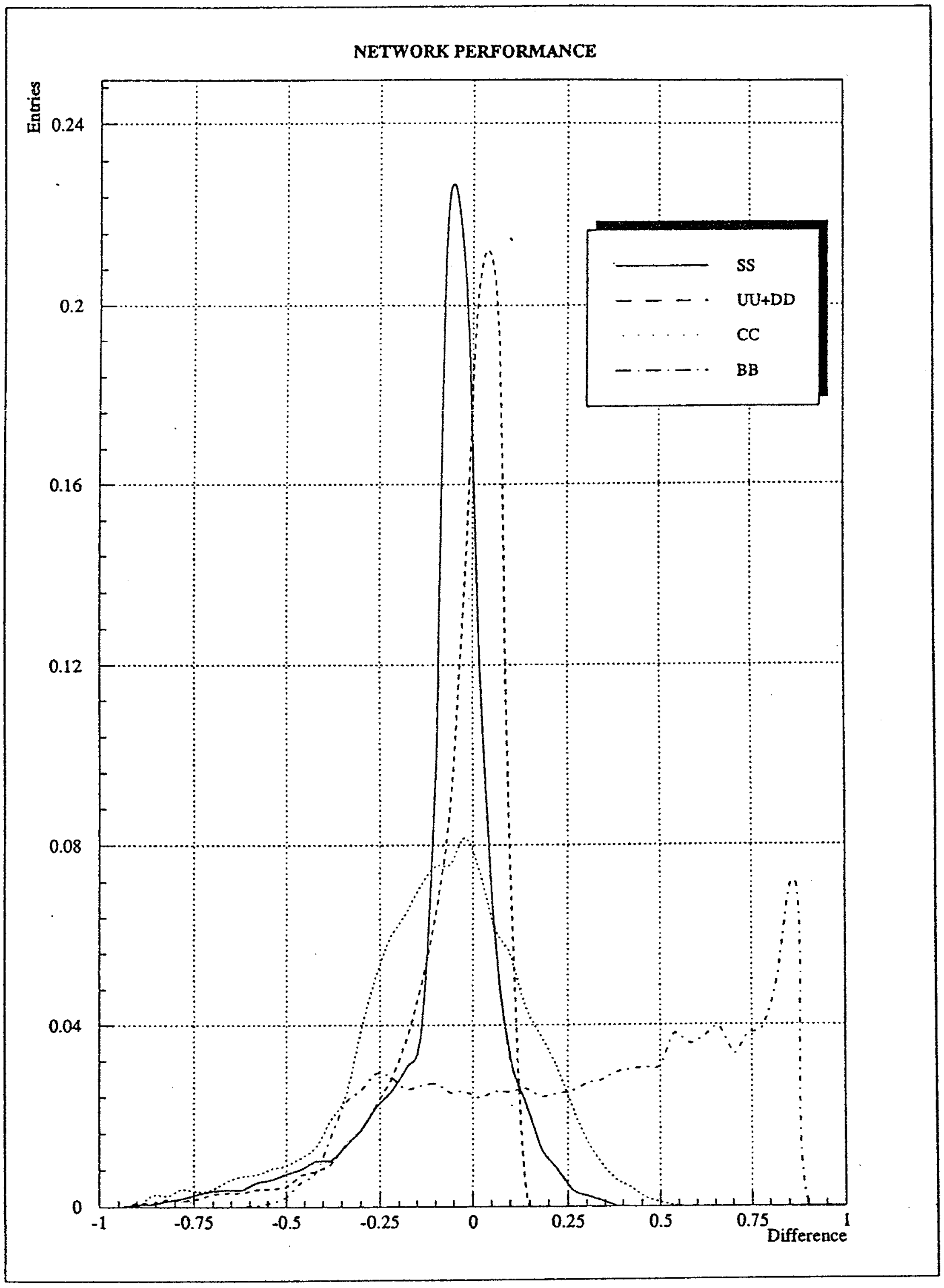


Figure 4.9: *Distributions of the difference between the output of the desired winner node and the output of the node among the rest having biggest value. The areas have been normalized to unity.*

Testing	ϵ_q	ρ_q	p
$(u\bar{u}+d\bar{d})$	44.9 %	77.8 %	55.8 %
$s\bar{s}$	28.4 %	83.0 %	31.4 %
$c\bar{c}$	36.0 %	81.7 %	29.7 %
$b\bar{b}$	75.7 %	85.5 %	60.1 %

Table 4.3: *The final Network performance.*

signal efficiency, *background rejection* and *purity*.

4.6.4 Use of Multivariate Discriminant Analysis

At the end of each testing phase, the Network outputs for input patterns and their respective targets were collected and then used to obtain a measure of the prototype distributions for each flavour. The Network architecture and configuration was tested on a set of simulated patterns, consisting of 73,691 Z^0 decays, and it was tried whether one could measure the hadronic branching fractions of the Z^0 boson in this simulated test sample.

The results were obtained by minimising a χ^2 measure of the discrepancy between the four flavour densities from simulated Monte Carlo data and the test data in a four-dimensional hypercube. The technique [53] involves Multivariate Discriminant Analysis, using an optimisation algorithm to fit a discriminant surface between the events of one flavour and all the others. This have been done four times, one for each flavour, leading to the following results:

$$\begin{aligned}
\Gamma_{u\bar{u}+d\bar{d}}/\Gamma_h &= 0.421 \pm 0.013 \pm 0.015 \\
\Gamma_{s\bar{s}}/\Gamma_h &= 0.173 \pm 0.011 \pm 0.007 \\
\Gamma_{c\bar{c}}/\Gamma_h &= 0.178 \pm 0.006 \pm 0.003 \\
\Gamma_{b\bar{b}}/\Gamma_h &= 0.227 \pm 0.003 \pm 0.0009
\end{aligned}$$

These values are consistent with the input values in the Monte Carlo model used.

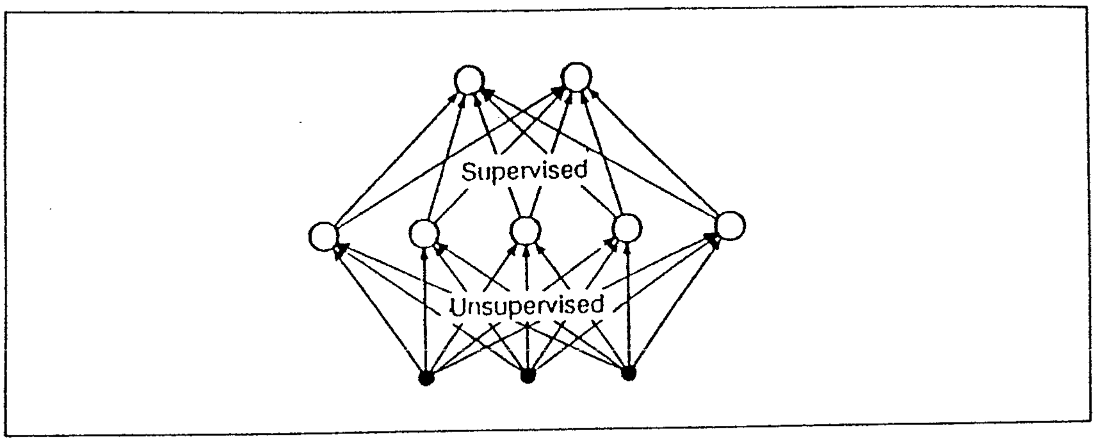


Figure 4.10: *A general hybrid learning Network scheme.*

4.7 The Counter-Propagation Neural Network Solution

Combining supervised and unsupervised learning in the same Network can be useful for some problems. The general idea is to have one layer that learns in a unsupervised way, followed by one (or more) layers trained by back-propagation or any other kind of supervised learning method (see Fig. 4.10). Networks of this kind are called *hybrid* Neural Networks.

In a self-organising Neural Network the only available information is given by correlations in the input data. In the previous Section, we saw that the implemented feed-forward Neural Network had difficulties in separating the light quarks due to the strong correlations between these classes, and therefore it is interesting to analyse if a hybrid Network is able to perform better in classifying light quarks. The hybrid schemes are not optimal in the sense that back-propagation is, since the hidden layer responses are not optimised with respect to the output performance. It can be anticipated that more hidden units are needed in order to get comparable results.

Many applications of hybrid Networks have been investigated, including pattern classification, data compression, statistical analysis and function approximation. The greatest appeal of these Networks is their speed: for a given problem the training is typically a factor of 10-100 times faster than for conventional back-propagation Networks, with comparable results. Moreover, they do not suffer of problems like "overtraining" in back-propagation Networks, so in principle they can be trained with an unlimited number of learnings, reaching a very stable final configuration.

4.7.1 The Neural Network Architecture

By combining a portion of a self-organising map (Section 3.3.3) and an instar/outstar structure of a Filter-Network (Section 3.4), a new type of mapping Neural Network is obtained.

The topology of a full *counter-propagation* Network is presented in Fig. 4.11. Input vectors $\hat{\xi}$ and associated target vectors $\hat{\zeta}$ are entered at opposite ends of the Network. The inputs *propagate* through the Network in opposite directions in a *counterflow* arrangement (thus the name "counter-propagation") producing output vectors O' and O'' that are approximations of $\hat{\xi}$ and $\hat{\zeta}$.

The Network is designed to approximate a continuous function $\mathcal{F} : A \subset \Re^n \longrightarrow B \subset \Re^m$, defined on a compact set A. It is assumed that the $\hat{\xi}$ input vectors are drawn according to a fixed probability density function $\rho(\hat{\xi})$.

We will consider a *forward-only* variant of the counter-propagation Network (see Fig. 4.12), that consists of three layers:

1. **Input Layer**: contains n fan-in units each assigned to one variable of the input pattern $\hat{\xi}^\mu$, and m units corresponding to the associated target vector $\hat{\zeta}^\mu$ (one unit for each component of the target vector).

2. **Kohonen Layer**: contains N processing elements performing a Kohonen feature mapping. They can be represented as a bidimensional array $X \cdot Y$ ($N = X \cdot Y$), defining a plane on which to perform a topological interpretation of input distributions.

3. **Grossberg Layer**: contains m output units, each receiving connections from the units in the Kohonen layer and a connection from the respective component in the target vector $\hat{\zeta}^\mu$. The output vector can be considered as the the Network approximation of the target vector.

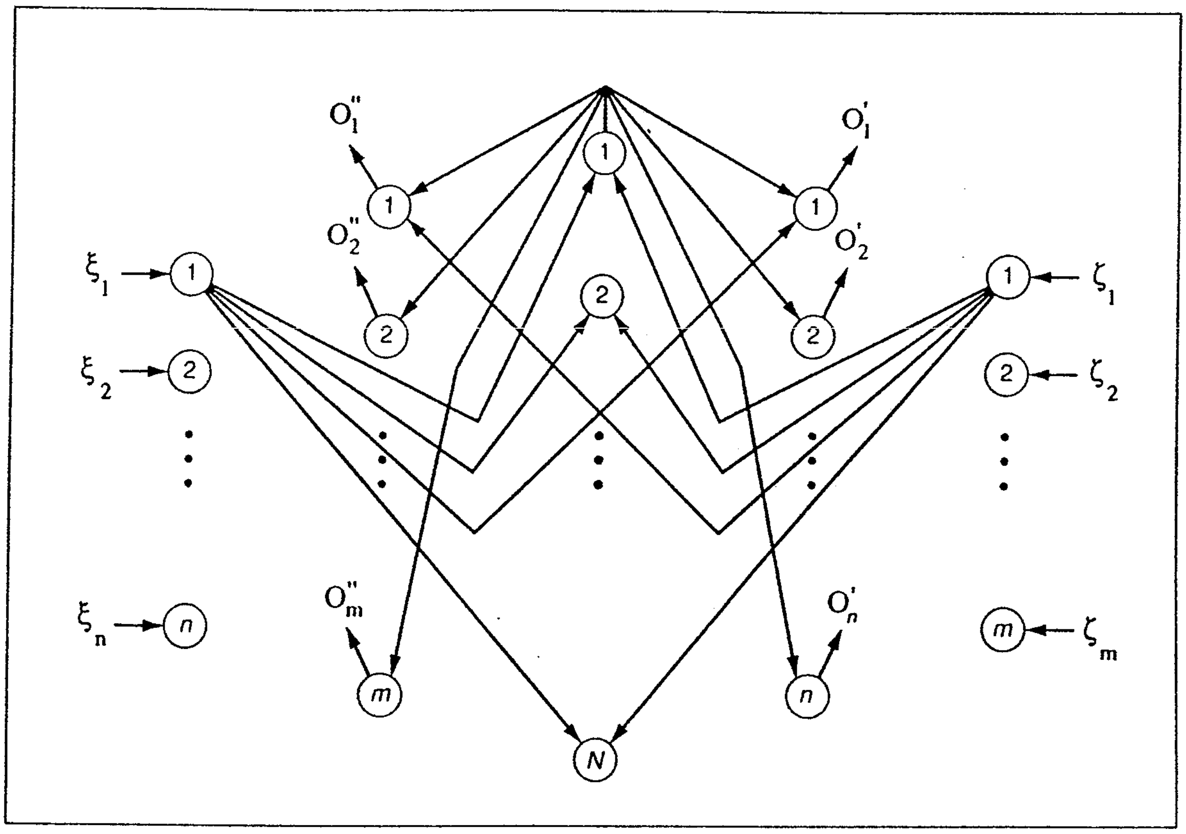


Figure 4.11: *Topology of the full counter-propagation Neural Network.*

4.7.2 The Network Training

During training the Network is exposed to patterns of the mapping F. After each $\hat{\xi}^{\mu}$ is selected, $\hat{O}^{\mu} = \mathcal{F}(\hat{\xi}^{\mu})$ is determined. Output values of the second layer are computed by performing a competition among all the units z_j according to the Kohonen learning law $(1 \leq j \leq N)$. For every input pattern μ, only one of the units (the winner) in the second layer will take the output value 1, the others will be set to 0. The output layer of the Network receives the z-signals from the second layer and the components of the target vector associated to the input vector $\hat{\xi}^{\mu}$.

The connection weights between the first and the second layer are updated according to the Kohonen learning law: only the winning unit and the units belonging to a defined *neighbourhood H* of the winner have their

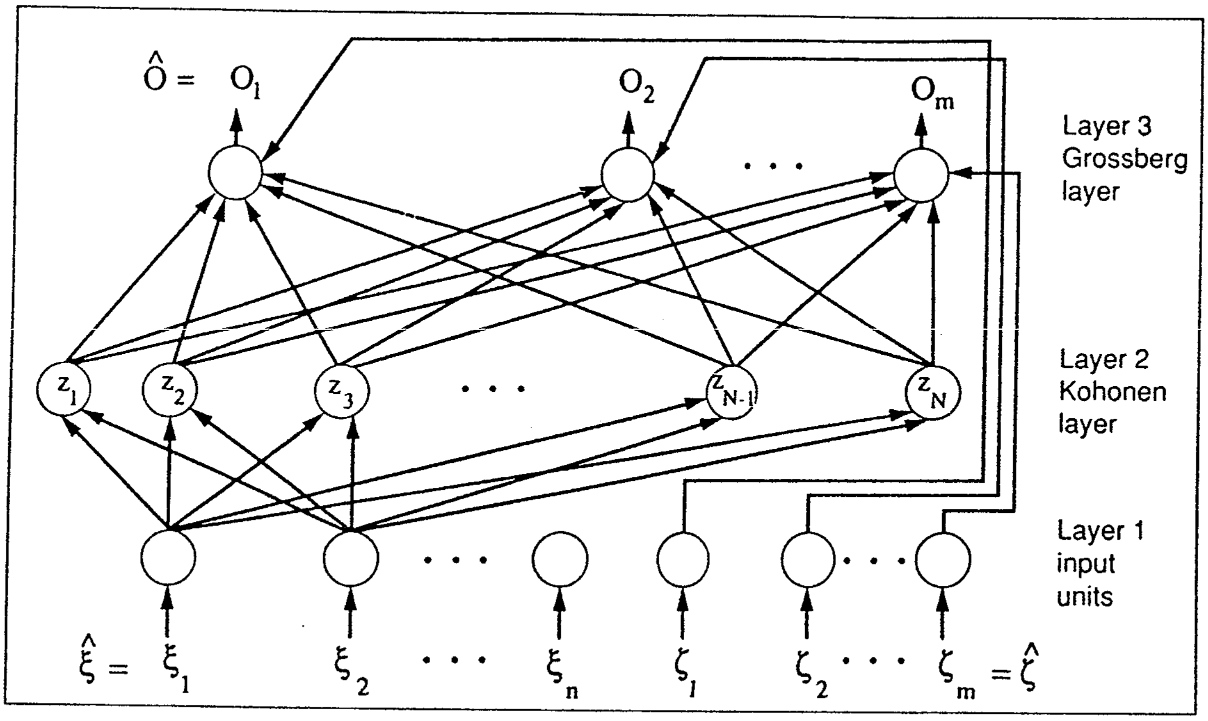


Figure 4.12: *The forward-only counter-propagation Network architecture.*

weights modified.

If we define I_1 to be the set of weights associated to the connections between first and second layer, and I_2 the set of weights associated to the connections between second and third layer, we can write the learning algorithm for the counter-propagation Neural Network as a step-by-step procedure:

1. Initialise the weights w_{ji} in set I_1 with small random values and assign the weights u_{kj} in set I_2 to 1.

2. Initialise the Network parameters by defining the initial value of the learning parameter η and the size of the neighbourhood zone H.

3. Present a pattern μ to the Network as an input vector:

$$\hat{\xi}^{\mu} = \xi_1^{\mu}, \xi_2^{\mu}, \ldots, \xi_n^{\mu} \, .$$

4. For each node j in the Kohonen-layer, compute the distance between the input vector components and the weights w_{ji} assigned to node j as

$$D_j = \sum_{i=1}^{n} (\xi_i^\mu - w_{ji}^{old})^2 \ . \tag{4.26}$$

5. Select the winner node j^* as the node having the lowest distance D_{j^*}: $D_{j^*} \leq D_j \quad \forall j, \ 1 \leq j \leq N$. Define output values z_j^μ for all the units j in the Kohonen-layer as:

$$z_j^\mu = \begin{cases} 1 & \text{if } j = j^* \\ 0 & \text{otherwise} \end{cases} \ . \tag{4.27}$$

6. Compute output values O_k^μ in Grossberg layer as:

$$O_k^\mu = \sum_{j=1}^{N} u_{kj}^{old} z_j^\mu \ . \tag{4.28}$$

7. Select the winner node in the output layer by utilising the "winner takes it all" technique.

8. Update the weights of the set I_1 by using:

$$w_{ji}^{new} = w_{ji}^{old} + \eta_1(t)(\xi_i^\mu - w_{ji}^{old}) \ , \quad \forall j \in H_{j^*}(t) \ . \tag{4.29}$$

9. For each unit k in the output layer, update the weights of the set I_2 by using:

$$u_{kj}^{new} = u_{kj}^{old} + \eta_2(-u_{jk}^{old} + \zeta_k^\mu) z_j^\mu \ . \tag{4.30}$$

10. Goto step 3.

$H_{j^*}(t)$ is the neighbour zone defined around the winner unit j^* at time t, $\eta_1(t)$ is the learning parameter for the Kohonen layer at time t, and η_2 is the step parameter for the output layer.

$H_{j^*}(t)$ and $\eta_1(t)$ are geometrically modified as the learning proceeds (i.e., t increases):

$$\eta_1(t) = \eta_{min} + \eta_{max} e^{-t/T_1} \ , \quad T_1 = \frac{p}{2} \ , \tag{4.31}$$

where p is the learning set dimension (number of total input patterns). $\eta_{min} < \eta_1(t) < \eta_{min} + \eta_{max}$, the learning parameter will be reduced as t increases. $H_{j^*}(t)$ is defined as a square centered around the node j^* with side $2 \cdot l(t)$ that will be reduced as the learning proceeds:

$$l(t) = l_{min} + l_{max} e^{-t/T_2} \ , \quad T_2 = \frac{2p}{5} \ , \tag{4.32}$$

where $l_{min} < l(t) < l_{min} + l_{max}$. If $N = N_X \cdot N_Y$ is the total number of units in the second layer (a bidimensional matrix with N_X rows and N_Y columns), we take $l_{max} = \frac{min(N_X, N_Y)}{2}$.

After a large number of training inputs, the $\mathbf{w}_j$ vectors will arrange themselves in $\Re^n$ in such a way that they are approximately equiprobable with respect to $\hat{\xi}$ input vectors. Once the Kohonen layer has equilibrated, the units in the third (output) layer begin to learn the averages $\mathbf{v}_j$ of the target vectors $\hat{\zeta}$ associated with each weight vector $\mathbf{w}_j$. In particular, after sufficient training, the Network will produce a vector $\mathbf{v}_j = (u_{1j}, u_{2j}, ..., u_{mj})$ whenever the processing element j wins the competition in the second layer. $\mathbf{v}_j$ vectors will equilibrate by using small values of the step parameter η_2.

The training phase of the counter-propagation Neural Network is performed in two stages (*differentiated training*):

1. *Kohonen layer learning only*: updating is performed only on weights belonging to I_1, while units in the Grossberg's layer are disabled. This partial training is continued until the Kohonen layer has equilibrated with equiprobable weights.

2. *Full learning*: Units in the Grossberg's layer are enabled and the updating of weights is extended to those of set I_2.

4.7.3 Use of the Network and Results

Once the training phase has been completed, the Network functions exactly as a nearest-match lookup table. The input $\hat{\xi}^\mu$ to the Network is compared with weight vectors $\mathbf{w}_j$ to find the closest match $\mathbf{w}_l$, and the output vector $\mathbf{v}_l$ associated to $\mathbf{w}_l$ is emitted.

Compared to other mapping Networks, the counter-propagation Network requires orders of magnitude less training trials to achieve its best performance. Its power is the capability of displaying typical features of

η_1	l	η_2	$N = N_X \cdot N_Y$	n
$\eta_{min} = 0.1$ $\eta_{max} = 0.8$	$l_{min} = 0$ $l_{max} = 16$	0.02	$N_X = 30$ $N_Y = 30$	23

Table 4.4: *Configuration parameters of the Network.*

Testing	ϵ_q		ρ_q		p	
$(u\bar{u}+d\bar{d})$	35.6 %	-9.3 %	79.1 %	+1.3 %	51.6 %	-4.2 %
$s\bar{s}$	30.7 %	+2.3 %	77.2 %	-5.8 %	27.2 %	-4.2 %
$c\bar{c}$	35.4 %	-0.6 %	80.8 %	-0.9 %	27.9 %	-1.8 %
$b\bar{b}$	71.2 %	-4.5 %	81.8 %	-3.7 %	53.2 %	-6.9 %

Table 4.5: *Performance of the counter-propagation Neural Network and comparison to the back-propagation one.*

the input data in a transparent manner through the internal representation, without needing to know in advance the number of feature classes.

The Network was trained and tested on the same data sets used for the back-propagation Network, with a configuration described in Table 4.4. After 20 epochs of partial-training and 40 epochs of full-training, its final performance in terms of *signal efficiency*, *background rejection* and *purity* is summarised in Table 4.5. In this Table the differences with respect to the previous back-propagation Network approach are also presented.

It was found useful to look at the final map obtained with the Kohonen layer. The Network detects the significant features of each flavour and is able to group the different flavours topologically. In Fig. 4.13 the distribution maps for each quark flavour are represented. Every cell in the plane can be regarded as a Kohonen processing element; the corresponding box represents the number of events activating this cell.

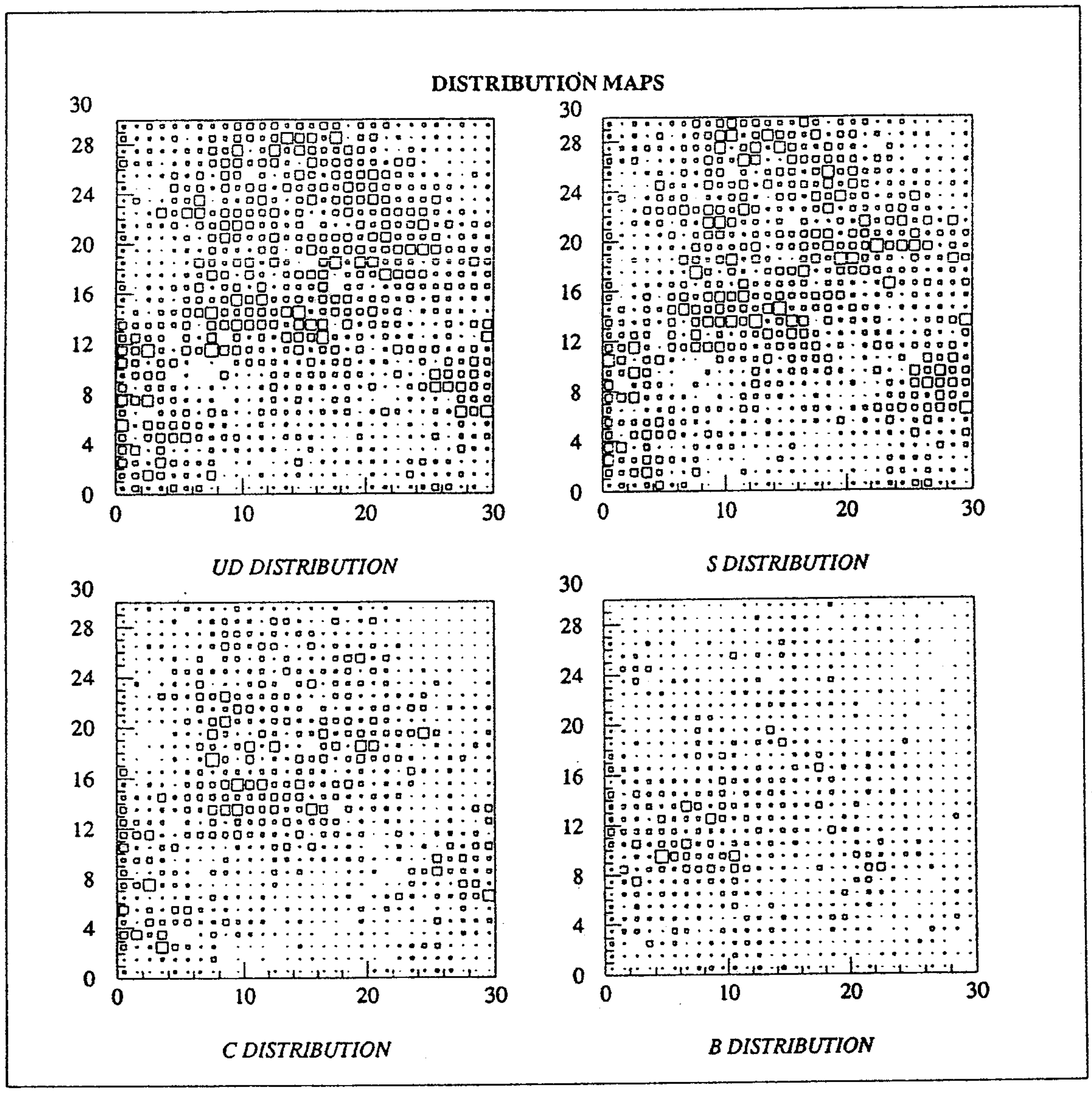


Figure 4.13: *Distribution maps for each flavour.*

Chapter 5

Implementation

In this chapter the software packages developed for the implementation of both back- and counter-propagation Neural Networks are briefly described.

5.1 Introduction

Two independent software packages BPNETRIG and CPNETRIG have been developed, including the Network features discussed in the previous chapters. The packages implement a back-propagation Network and a counter-propagation Network, respectively.

BPNETRIG and CPNETRIG are organised in a similar hierarchy of four modules as presented in Fig 5.1.

- **I/O_MANAGER** - This module collects the main program and the user interface subroutines, and distributes to the other modules all the user commands and environment information needed to initialise and control the process.
- **NET_TOOLS** - This module contains the subroutines implementing the Network functions.
- **NET_TRAIN** - This module implements the Network training session. It receives parameters and commands from the I/O_MANAGER and collects the subroutine and function calls to module NET_TOOLS.
- **NET_TEST** - This module implements the Network testing session.

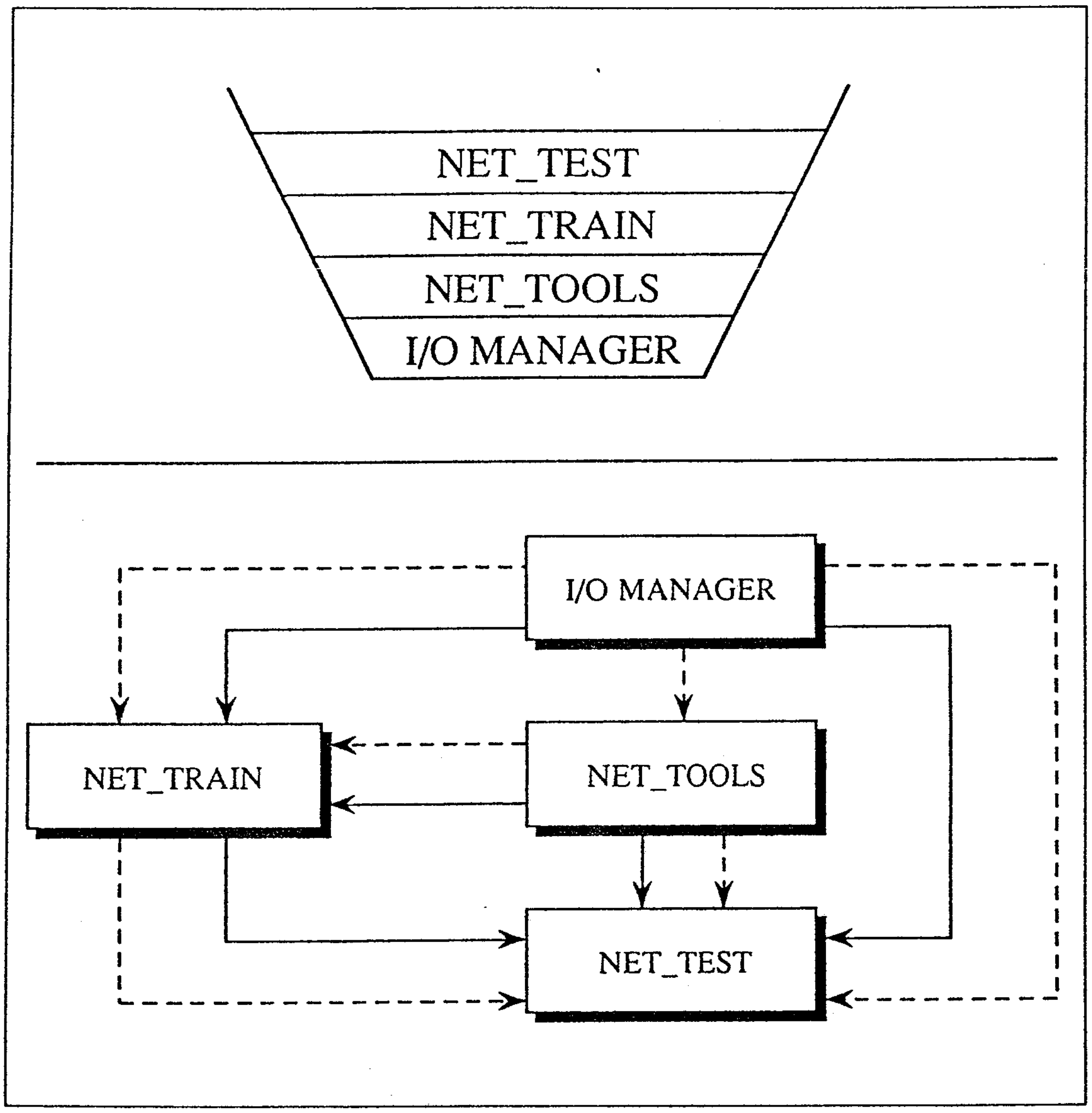


Figure 5.1: *(a) Hierarchical structure of the modules, modules in the upper layers can call subroutines and refer to parameters defined in the modules belonging to lower layers. (b) Relations between the modules in terms of subroutines-passing (solid line) and parameters dependence (dashed line).*

The subroutines have been written in FORTRAN 77 programming language.

5.2 BPNETRIG Implementation

The external functions used in the package are:

- *HBOOK1*(id,chtitl,nx,xmi,xma,vmx): HBOOK function defining a 1-dimensional histogram;
- *HBOOK2*(id,chtitl,nx,xmi,xma,ny,ymi,yma,vmx): HBOOK function defining a 2-dimensional histogram;
- *HDELET*(id): HBOOK function deleting a histogram *id* and releasing the corresponding memory space;
- *HFILL*(id,x,y,weight): HBOOK function filling a 1- or 2-dimensional histogram;
- *HLIMIT*(m): HBOOK function defininig the total size *m* of common /PAWC/ containing the HBOOK memory area;
- *HMEMOR*(m): HBOOK working space declaration;
- *HOPERA*(id1,choper,id2,id3,c1,c2): HBOOK function for performing logical operations on histograms.
- *HRPUT*(id,file,ty): HPLOT function for writing histograms into a file;
- *RNDM*(n): random generator function.

All these external functions are defined in the CERN libraries *KERNLIB* and *PACKLIB* [57], to which BPNETRIG must be linked.

5.2.1 Common Blocks

The *Common* blocks defined in the package are */PAWC/* and */NETWORK/*. The first one is the HBOOK memory area, and the second one contains all the variables and arrays which define the Network structure. Common */NETWORK/* includes:

- *LAYER*: an integer type variable indicating the number of layers in the Network;
- *NEURONS(MAXLAY)*: an integer type array indicating the number of units in each layer; *MAXLAY* is an integer type constant defining the maximum number of layers;
- *NETW(MAXLAY,MAXNEU)*: a real type array for representing the *activation* values for each neuron (as defined in Section 3.2.2). For example:
$$NETW(i,j) = net_j^\mu \ ,$$
represents the *activation* of node j in layer i for pattern μ. Input vector $\hat{\xi}$ is put in $(NETW(1,1),...,NETW(1,NEURONS(1)))$.
- *DELTA(MAXLAY,MAXNEU)*: a real type array for storing the error delta signals δ_j as described in Section 3.2.2.
- *WEIGHT(MAXLAY,MAXNEU,MAXNEU)*: a real type array for storing the connections weights w_{ji}. WEIGHT(L,j,i) is the value of a weight associated to the connection between node i in layer L and node j in layer $L+1$.
- *BW(MAXLAY-1,MAXNEU)*: a real type array for storing the weights associated to the bias unit.
- *DELTAW(MAXLAY,MAXNEU,MAXNEU)*: a real type array for storing the current weight updates $\Delta w_{ji}(t+1)$.
- *OLDDELTAW(MAXLAY,MAXNEU,MAXNEU)*: a real type array for storing the old weight updates $\Delta w_{ji}(t)$.
- *DELTABW(MAXLAY-1,MAXNEU)*: a real type array for storing the weight updates for bias unit connections.

5.2.2 I/O_MANAGER Module

I/O_MANAGER is the *main* module, collecting 7 subroutines for I/O operations (*INPUT_PAR()*, *INFORMATIONS()*, *FILL_VECTOR()*, *FETCH_TRAIN_DATA()*, *FETCH_TEST_DATA()*, *FETCH_TEST_REALDATA()*, *FETCH_DATA_RND()*), and the central body of the program.

1. *INPUT_PAR()*: this subroutine reads the input parameters specified by user, fixes the Network structure and its initial parameters, defines the size of learning and testing sets, defines the type of training in terms of error function and parameters updating, specifies the data fetching mode (randomised or not) and the type of testing (on simulated or real data).

2. *INFORMATIONS()*: this subroutine stores on a file all the information, parameters and Network characteristics specified by the user in *INPUT_PAR()*.

3. *FILL_VECTOR()*: reads the data from the training file in equal proportions for each flavour, and stores them in a big internal array *EVENTS*.

4. *FETCH_TRAIN_DATA()*: reads 24 real type variables (describing an event) from the array *EVENTS* created in *FILL_VECTOR()*, and defines an input vector of 23 real type variables ($\hat{\xi}^{\mu}$) plus a target vector of 4 integer type values ($\hat{\zeta}^{\mu}$).

5. *FETCH_TEST_DATA()*: reads from a file of simulated test data a set of 24 variables (describing an event) and defines an input vector of 23 variables and a target vector as in *FETCH_TRAIN_DATA()*.

6. *FETCH_TEST_REALDATA()*: reads from a file of experimental test data a set of 23 variables without target value and defines the input vector.

7. *FETCH_DATA_RND()*: reads from the array *EVENTS* a set of 24 variables as in *FETCH_TRAIN_DATA()*, but in a random way.

In the *main* program, the first subroutine to be called is *INPUT_PAR()*. The Network's weights will be initialised with the subroutine *INIT()* or with *LOADWG()* if it has been requested to load weights previously saved. All the Network features are displayed and memorised on a file with a call to *INFORMATIONS()*. Finally, with *TRAINING()* and *TESTING()*, learning and testing sessions will begin.

5.2.3 NET_TOOLS Module

This module collects all the subroutines performing the primitive functions of the Neural Network:

- *INIT()*: the weights of the Network are initialised at random values in a defined interval [MINW,MAXW].
- *SQUARED_ERROR()*: implements the squared error function (4.9).
- *MINKOWSKY_ERROR()*: implements the generalised error function (4.19).
- *ENTROPIC_ERROR()*: implements the entropic error function measure (4.11).
- *POTTS_ERROR()*: implements the asymmetric error function or Kullback measure (4.18).
- *G()*: implements the sigmoid function $g(x) = \frac{1}{2}(1+\tanh\frac{x}{T})$ for output nodes ranging from 0 to 1.
- *G2()*: implements the sigmoid function $g(x) = \tanh\frac{x}{T}$ for output nodes ranging from -1 to 1.
- *G3()*: implements the sigmoid function (4.17) for the asymmetric error measure.
- *GDERIV()*: implements the derivative of sigmoid function (4.6) as: $\frac{1}{2}(1-\tanh^2(\frac{x}{T}))$.
- *G2DERIV()*: implements the derivative of sigmoid function (4.7) as: $1-\tanh^2(\frac{x}{T})$.
- *SIGNUM()*: implements function sgn(x).
- *NORM()*: normalises a value x in a interval [min..max] depending on nodes output range:

$$x = \begin{cases} \frac{x-min}{max-min} & \text{if } [0,1] \\ \frac{2(x-min)}{max-min} - 1 & \text{if } [-1,1] \end{cases} .$$

- *FORWARD()*: performs the forward propagation of input signals through the Network. It computes the weighted sum (activation signals) and modifies *NETW* with respect to the current input.

- *BACKWARD()*: performs the backward propagation of the error-delta signals through the Network. It computes the error signals in *DELTA* depending on the error function used, and then it computes in *DELTAW* the weight changes.

- *WUPDATE()*: performs the weight updating according to the generalised delta-rule discussed in Section 3.2.3.

- *PRUN_WUPDATE()*: performs the weight updating by adding a penalty term (3.23) for pruning.

- *ETAUPDATE(), ALPHAUPDATE(), TUPDATE()*: parameters η (*ETA*), α (*ALPHA*) and T (*T*) are linearly updated.

- *GEOMETAUPDATE(), GEOMALPHAUPDATE(), GEOMTUPDATE()*: parameters *ETA*, *ALPHA* and *T* are geometrically (4.25) updated.

- *BACK()*: this function has a value equal to one when the back-propagation process has to be done. Back-propagation is performed if the absolute difference between the Network output value and the target is greater than a fixed threshold.

- *PRUNING()*: all the connections of each neuron are checked. Any neuron with no connection having an absolute weight value greater than 0.1 will be removed.

- *OUTMAX()*: Network's output is interpreted by means of the "winner takes it all" technique.

- *SAVEWG(), LOADWG()*: saves/restores the Network configuration, memorising/loading all the Network weights and updatings.

5.2.4 NET_TRAIN Module

The Network training session is executed when the subroutine *TRAINING()* is called from the main program.

For every input pattern the following operations are performed:

1. Call *FETCH_TRAIN_DATA()* or *FETCH_DATA_RND()* to collect the input variables and define the input and target vectors.

2. Call *FORWARD()* to propagate the input signals through the Network and then compute the specified error function, defining a quantity *ERR*.

3. Call *OUTMAX()* to interpret the Network output obtained from *FORWARD()*.

4. Perform, if possible (*BACK()* test), the error signal back-propagation calling *BACKWARD()*.

Updating of the weights is performed after a fixed number of fetched events (10 in our case), calling *WUPDATE()* or *PRUN_WUPDATE()*.

Training is divided into a user-defined number of learning epochs, each one filtering the training set with the specified dimension. At the end of every learning epoch it is possible to update the parameters *ETA*, *ALPHA* and *T* linearly or geometrically by using the mentioned subroutines, and resetting, if necessary, their values (*reset-mode* training).

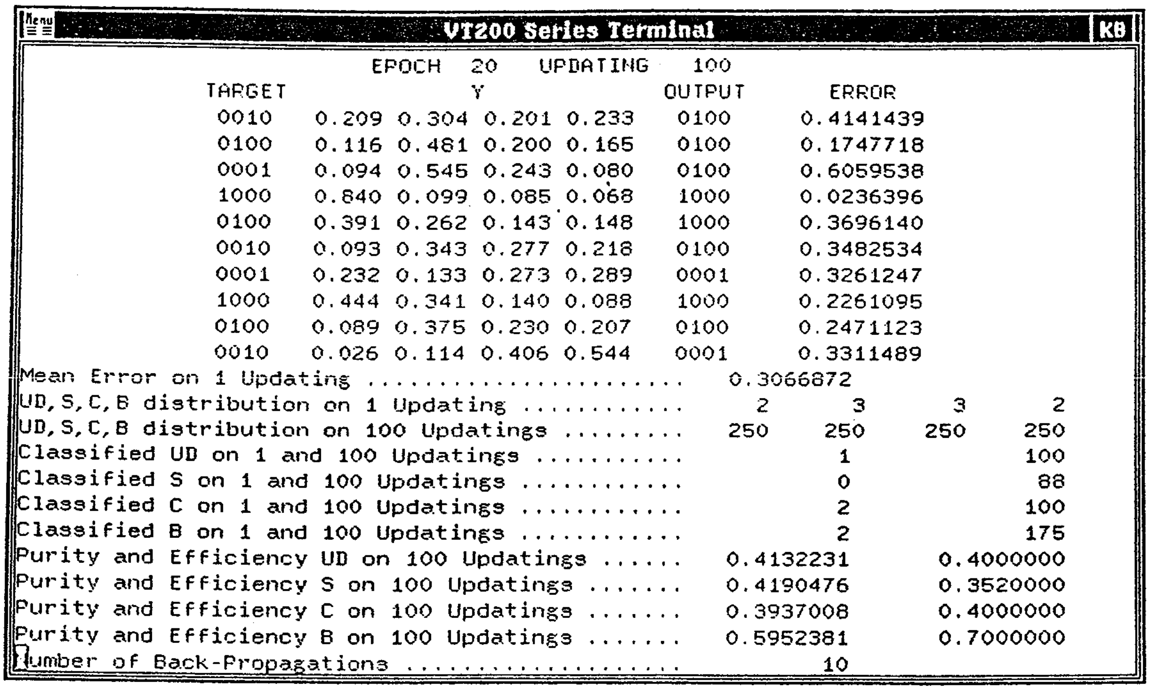
After every 5 learning epochs, it is possible to perform a test phase calling subroutine *TESTING()* (*check-mode* training).

After every 100 updatings, the training process is visualised by showing the current error measure and the general performance (see Fig. 5.2).

5.2.5 NET_TEST Module

The Testing phase can be performed after the training or by using a previously saved configuration of the Network. The testing is done by calling the subroutine *TESTING()*.

The structure of this subroutine is similar to that of *TRAINING()*, except that testing is performed only once on the entire test set without computing the errors or updating the weights.



```
                          EPOCH  20    UPDATING   100
          TARGET                    Y                  OUTPUT       ERROR
           0010     0.209 0.304 0.201 0.233   0100      0.4141439
           0100     0.116 0.481 0.200 0.165   0100      0.1747718
           0001     0.094 0.545 0.243 0.080   0100      0.6059538
           1000     0.840 0.099 0.085 0.068   1000      0.0236396
           0100     0.391 0.262 0.143 0.148   1000      0.3696140
           0010     0.093 0.343 0.277 0.218   0100      0.3482534
           0001     0.232 0.133 0.273 0.289   0001      0.3261247
           1000     0.444 0.341 0.140 0.088   1000      0.2261095
           0100     0.089 0.375 0.230 0.207   0100      0.2471123
           0010     0.026 0.114 0.406 0.544   0001      0.3311489
Mean Error on 1 Updating .......................   0.3066872
UD,S,C,B distribution on 1 Updating ............     2        3         3        2
UD,S,C,B distribution on 100 Updatings .........   250      250       250      250
Classified UD on 1 and 100 Updatings ...........            1                100
Classified S on 1 and 100 Updatings ............            0                 88
Classified C on 1 and 100 Updatings ............            2                100
Classified B on 1 and 100 Updatings ............            2                175
Purity and Efficiency UD on 100 Updatings ......   0.4132231        0.4000000
Purity and Efficiency S on 100 Updatings .......   0.4190476        0.3520000
Purity and Efficiency C on 100 Updatings .......   0.3937008        0.4000000
Purity and Efficiency B on 100 Updatings .......   0.5952381        0.7000000
Number of Back-Propagations ....................          10
```

Figure 5.2: *Display during the training process.*

Input signals are propagated forward by means of subroutine *FORWARD()* and then the network output is interpreted by calling *OUTMAX()*.

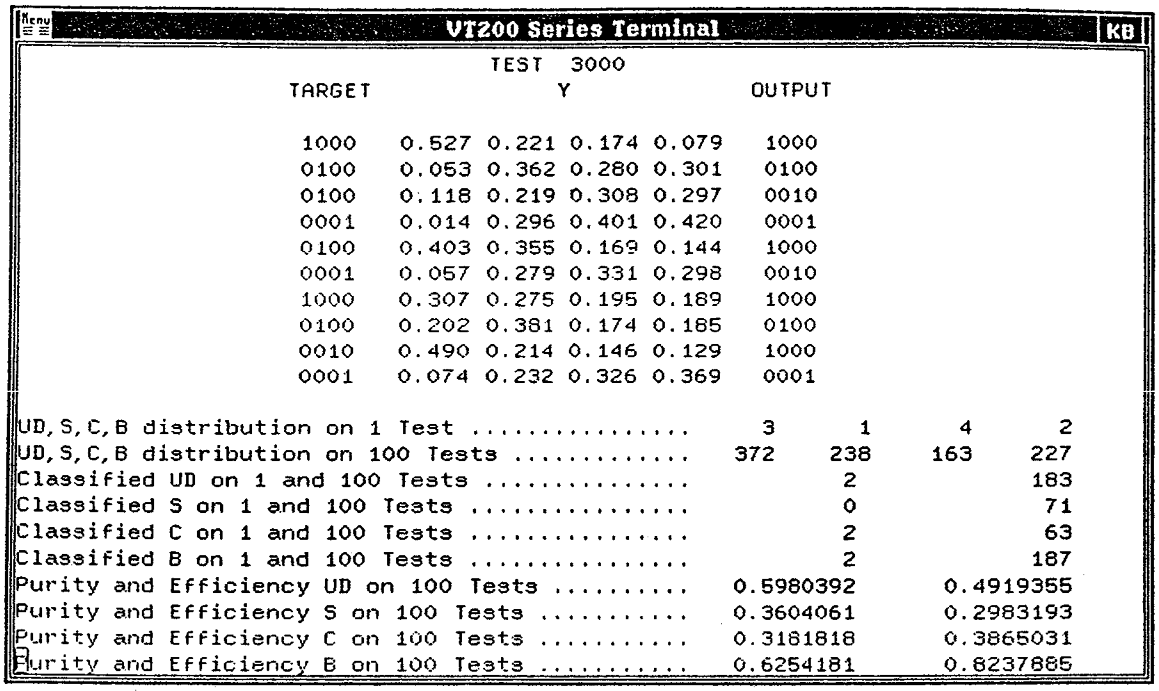
Testing can be performed on simulated data, in which case a visualisation of the running process is given (see Fig. 5.3), or on experimental data, on which it is not possible to make any statistics.

5.3 CPNETRIG Implementation

The implementation structure of this package is similar to BPNETRIG. The external functions are the same used in BPNETRIG (Section 5.2).

5.3.1 Common Blocks

The *Common* blocks defined are */PAWC/* and */NETWORK/*, the first one referring to the HBOOK memory area, the second one containing all



```
                         TEST  3000
       TARGET                 Y                    OUTPUT

        1000     0.527 0.221 0.174 0.079     1000
        0100     0.053 0.362 0.280 0.301     0100
        0100     0.118 0.219 0.308 0.297     0010
        0001     0.014 0.296 0.401 0.420     0001
        0100     0.403 0.355 0.169 0.144     1000
        0001     0.057 0.279 0.331 0.298     0010
        1000     0.307 0.275 0.195 0.189     1000
        0100     0.202 0.381 0.174 0.185     0100
        0010     0.490 0.214 0.146 0.129     1000
        0001     0.074 0.232 0.326 0.369     0001

UD,S,C,B distribution on 1 Test ...............      3       1        4        2
UD,S,C,B distribution on 100 Tests ............    372     238      163      227
Classified UD on 1 and 100 Tests ..............              2               183
Classified S on 1 and 100 Tests ...............              0                71
Classified C on 1 and 100 Tests ...............              2                63
Classified B on 1 and 100 Tests ...............              2               187
Purity and Efficiency UD on 100 Tests .........    0.5980392        0.4919355
Purity and Efficiency S on 100 Tests ..........    0.3604061        0.2983193
Purity and Efficiency C on 100 Tests ..........    0.3181818        0.3865031
Purity and Efficiency B on 100 Tests ..........    0.6254181        0.8237885
```

Figure 5.3: *Display during testing process.*

the variables and arrays defining the Network structure. Common */NETWORK/* includes:

- *KOHOW(MNUMZX,MNUMZY,MNUMX)*: a real type array for storing the connection weights w_{ji}, where i is a unit (X) of the input layer and j is a unit in the Kohonen-map layer matrix indexed by (ZX, ZY).
- *GROSW(MNUMZX,MNUMZY,MNUMY)*: a real type array for storing the connection weights u_{kj}, where j is a unit in the Kohonen-map layer matrix indexed (ZX, ZY) and k is an output unit.
- *NETKOHO(MNUMZX,MNUMZY)*: a real type array related to the output values z_j in the Kohonen-map layer matrix.
- *NETGROS(MNUMY)*: a real type array related to the Network output O_k in the Grossberg-output layer.

5.3.2 I/O_MANAGER Module

I/O_MANAGER is the *main* module, collecting 3 subroutines for I/O operations (*INPUT_PAR()*, *INFORMATIONS()*, *FETCH_DATA()*) and the central body of the program.

1. *INPUT_PAR()*: this subroutine reads all the input parameters specified by user, fixes the Network structure and its initial parameters.
2. *INFORMATIONS()*: this subroutine stores on a file the information, parameters and Network characteristics specified by user in *INPUT_PAR()*.
3. *FETCH_DATA*: reads from a file of simulated data a set of 24 variables (describing an event) and defines an input vector of 23 variables and a integer target vector.

The structure of the main body is equal to that of BPNETRIG package.

5.3.3 NET_TOOLS Module

This module collects all the subroutines performing the primitive functions of the Neural Network:

- *INIT_NET()*: the second layer weights w_{ji} of the Network are initialised with small random numbers and can be normalised by calling subroutine *NORMALIZE()*, while the others weights u_{kj} are set to 1.0.
- *INIT()*: opens the data-files for training and testing.
- *NORMALIZE()*: normalises a weight vector $\mathbf{w_j}$ as:

$$w_{ji} = \frac{w_{ji}}{\sqrt{\sum_i w_{ji}^2}} .$$

- *NORM()*: normalises a value x in a interval [min..max].
- *MATCHING()*: Finds out the winner node in Kohonen layer by computing the distances between the input vector and the weight vector (4.26). *WINROW* and *WINCOL* will be the winner node indexes in the Kohonen-map layer matrix.

- *ETAK()*: updates the step parameter η_1 (4.31).
- *NEIGHBOUR()*: updates the neighbourhood of the winner unit after computing the square's side (4.32).
- *BORDER()*: defines the new neighbourhood boundaries around the winner unit.
- *WKOUPDATE()*: The weight w_{ji} updating is performed according to the Kohonen learning law (4.29). The new weight vectors can be normalised by calling subroutine *NORMALIZE*.
- *WGRUPDATE()*: Performs the updating of weights u_{kj} according to the instar-learning law (4.30).
- *OUTPUT_NET()*: Network's output is computed according to the equation (4.28) and is interpreted by means of the "winner takes it all" technique.
- *STATISTICS()*: Network classifications and errors are counted and stored.
- *PERFORMANCES()*: Network's performance is computed, displayed and memorised in a file.
- *SAVEWG(), LOADWG()*: saves/restores the Network configuration, memorising/loading Network weights.

5.3.4 NET_TRAIN Module

The Network training session is executed when the subroutine *TRAINING()* is called from the main program.

For every input pattern the following operations are performed:

1. Call *FETCH_DATA()* to collect the input variables and define the input and target vectors.
2. Begin competition in the Kohonen layer by calling *MATCHING()*, select the winner unit $j^* = (WINROW, WINCOL)$ and set: $NETKOHO(WINROW, WINCOL) = 1$.

3. Compute the Network output by calling *OUTPUT_NET()*.
4. Update the winner node neighbourhood by calling *NEIGHBOUR()* and define the new square zone by calling *BORDER()*.
5. Update the η_1 parameter by calling *ETAK()*.
6. If the Network is running in *full* training, then update all the weights with *WKOUPDATE()* and *WGRUPDATE()*. Otherwise, use only *WKOUPDATE()*.
7. If in *full* training call *STATISTICS()*.
8. Set *NETKOHO(WINROW,WINCOL) = 0.*

Training is divided into a user-defined number of learning epochs, each one filtering the entire training set. At the end of every learning epoch in *full* training it is possible to obtain statistics on the Network performance by calling subroutine *PERFORMANCES()*.

At the end of the training the Network configuration can be saved by calling *SAVEWG()*.

5.3.5 NET_TEST Module

The Testing phase can be performed by calling the subroutine *TESTING* after the training or starting with a previously saved configuration of the Network.

The structure of this subroutine is similar to that of *TRAINING*, except that the testing is performed only once without updating the weights. Input signals are propagated forward and then the network output is interpreted by calling *OUTPUT_NET()*.

□ □ □

Conclusions

The thesis considers the problem of classifying the hadronic decays of the Z^0 boson with Mapping Neural Networks.

Three different approaches have been studied.

1. Use of four different supervised feed-forward Neural Networks, each one of them classifying only one flavour against all the others, and therefore having only one output unit;

2. Use of a single supervised feed-forward Neural Network with four output units, each one associated to one flavour;

3. Use of a hybrid Neural Network (Counter-Propagation Neural Network) composed by a self-organising hidden layer and a supervised output layer.

After a detailed research of many possible techniques, and a deep study of Networks' parameters and results, the first and the second choices have been found to be the most convenient in terms of efficiency.

Results obtained by using the counter-propagation Neural Network look promising considering that the study is in a preliminary phase. For this kind of problem, however, the "supervised back-propagation" solution seems to be the best one.

The modular construction of the counter-propagation Neural Network illustrates that existing Neural Networks can be viewed as building block components which can be assembled into new configurations offering new and different information processing capabilities.

The four-output unit back-propagation Network has been utilised to evaluate the Z^0 hadronic branching fractions on a set of simulated data. The results indicate the feasibility of the measurement with an equivalent statistics of real data.

□ □ □

List of Tables

□ □ □

List of Figures

□ □ □

Bibliography

[1] F.Halzen and A.D.Martin, *"Quarks and Leptons: An Introductory Course in Modern Particle Physics"*, John Wiley & Sons, New York 1984.

[2] P.Eerola, *"A Search for New Heavy Particles in Z^0 Decays at LEP"*, SEFT Report Series, Helsinki 1990.

[3] S.Weinberg, Physics Review Letter **19** (1967) 1264.

[4] A.Salam, Proceedings of the VIII Nobel Symposium, ed. N.Svartholm p.367, Stockholm 1968.

[5] S.Glashow, Nuclear Physics **22** (1961) 579.

[6] J.Ellis and K.Olive, *"Neutrino Bounds from Astrophysics and Cosmology"*, Physics Letter **B193** (1987) 525.

[7] ALEPH Coll., *"A Precise Determination of the Number of Families with Light Neutrino and of the Z^0 Boson Partial Width"*, CERN-EP/89-169 (1989).

[8] J.E.Dodd, *"The Ideas of Particle Physics"*, Cambridge University Press, Oxford 1984.

[9] P.Aarnio et al. (DELPHI coll.), *"The DELPHI Detector at LEP"*, Nuclear Instruments and Methods in Physics Research **A303** (1991) 233.

[10] D.J.Miller, *"Particle Physics after a year of LEP"*, Nature **349** (1991) 379.

[11] W.S.McCulloch and W.Pitts, *"A Logical Calculus of the Ideas Immanent in Nervous Activity"*, Bulletin of Mathematical Biophysics **5** (1943) 115. Reprinted in Anderson and Rosenberg (1988).

[12] D.C.Hebb, *"The Organization of Behaviour"*, John Wiley & Sons, New York 1949.

[13] M.Minsky, *"Neural nets and the brain - Model problem"*, Doctoral Dissertation, Princeton University, Princeton 1954.

[14] M.Minsky and S.Papert, *"Perceptrons"*, MIT Press, Cambridge 1969.

[15] F.Rosenblatt, *"Principles of Neurodynamics"*, Spartan Books, Washington 1961.

[16] B.Widrow, *"Generalization and Information Storage in Networks of Adaline neurons"*, Self-Organizing Systems, Spartan Books, Washington 1962.

[17] D.E.Rumelhart, G.E.Hinton and R.J.Williams, *"Learning Internal Representation by Error Propagation"*, Parallel Distributed Processing **1**, MIT Press, Cambridge 1986.

[18] R.Beale and T.Jackson, *"Neural Computing: An Introduction"*, Adam Hilger IOP, Bristol 1990.

[19] J.J.Hopfield and D.W.Tank, *"Neural Computation of Decisions in Optimization Problems"*, Biological Cybernetics **52** (1985) 141.

[20] P.Baldi and K.Hornik, *"Neural Networks and Principal Component Analysis: Learning from Examples without Local Minima"*, Neural Networks **2** (1989) 53.

[21] R.O.Duda and P.E.Hart, *"Pattern Classification and Scene Analysis"*, John Wiley & Sons, New York 1973.

[22] T.Kohonen, *"Self-Organization and Associative Memories"*, Springer-Verlag, Berlin 1989.

[23] T.Kohonen, *"An Introduction to Neural Computing"*, Neural Networks **1** (1988) 3.

[24] G.Venkataraman and G.Athithan, *"Spin Glass, The Travelling Salesman Problem, Neural Networks and all that"*, Journal of Physics **36** (1991) 1.

[25] S.Grossberg, *"Studies of Mind and Brain: Neural Principles of Learning, Perception, Development, Cognition, and Motor Control"*, Reidel Press, Boston 1982.

[26] R.P.Lippmann, *"An Introduction to Computing with Neural Nets"*, IEEE ASSP Magazine (1987) 4.

[27] E.Domany, *"Neural Networks: A Biased Overview"*, Journal of Statistical Physics **51** (1988) 743.

[28] B.Widrow and R.Winter, *"Neural Nets for Adaptive Filtering and Adaptive Pattern Recognition"*, IEEE Magazine (1988) 25.

[29] D.M.Clark and K.Ravishankar, *"A Convergence Theorem for Grossberg Learning"*, Neural Networks **3** (1990) 87.

[30] D.Cohen and J.S.Taylor, *"Feedforward Networks - A Tutorial"*, Proceed. on Neural Computing, London 1989.

[31] W.Y.Huang and R.P.Lippmann, *"Neural Net and Traditional Classifiers"*, in Neural Information Processing Systems, ed. D.Z.Anderson (1988) 377.

[32] A.Lapedes and R.Farber, *"How Neural Nets Work"*, in Neural Information Processing Systems, ed. D.Z.Anderson (1988) 442.

[33] B.S.Wittner and J.S.Denker, *"Strategies for Teaching Layered Networks Classification Tasks"*, in Neural Information Processing Systems, ed. D.Z.Anderson (1988) 850.

[34] F.M.Silva and L.B.Almeida, *"Speeding up Backpropagation"*, in Advanced Neural Computers, Elsevier Science Pub. (1990) 151.

[35] K.Hornik, *"Approximation Capabilities of Multilayer Feedforward Networks"*, Neural Networks **4** (1991) 251.

[36] R.Hecht-Nielsen, *"On the Algebraic Structure of Feed-Forward Network Weight Space"*, in Advanced Neural Computers, Elsevier Science Pub. (1990) 159.

[37] Yoh Han Pao, *"Adaptive Pattern Recognition and Neural Networks"*, Addison-Wesley Publishing, 1990.

[38] P.Henrard et al. (ALEPH), *"Using Multivariate Analysis to Measure the Z^0 Partial Width into $b\bar{b}$ "*, presented at the 4th International Symposium on Heavy Flavour Physics, Orsay, June 1991.

[39] A.De Angelis, *"Hadronic Branching Fractions of the Z^0 Boson"*, presented at the XXI International Symposium on Multiparticle Dynamics, Wuhan (Popular Republic of China), September 1991.

[40] L.Bellantoni, J.S.Conway, J.E.Jacobsen, Y.B.Pan and Sau Lan Wu, *"Using Neural Networks with Jet Shapes to Identify b Jets in e^+e^- Interactions"*, submitted to Nuclear Instruments and Methods in Physics Research A, CERN-PPE/91-80, May 1991.

[41] C.Bortolotto, G.Cosmo, A.De Angelis, P.Eerola, J.Kalkkinen and A.Linussio, *"A Measurement of the Partial Hadronic Widths of the Z^0 using Neural Networks"*, to be published in the Proceedings of the Workshop on Neural Networks: from Biology to High Energy Physics, INFN/AE-91/12, Isola d'Elba, June 1991.

[42] G.Cosmo, A.De Angelis, P.Eerola, J.Kalkkinen and A.Linussio, *"DELPHI results on the Measurement of the Partial Hadronic Widths of the Z^0 using Neural Networks"*, to be published in the Proceedings of the II International Workshop on Software Engineering, Artificial Intelligence and Expert Systems for High Energy and Nuclear Physics, UDINE REPORT/92/02/AA, l'Agelonde sur Marne (France), January 1992.

[43] C.Bortolotto, A.De Angelis and L.Lanceri, *"Tagging the Decays of the Z^0 Boson into b quark pairs with a Neural Network Classifier"*, Nuclear Instruments and Methods in Physics Research **A306** (1991) 459.

[44] C.Bortolotto, A.De Angelis, L.Lanceri, N.De Groot and J. Seixas *"Neural Networks in Experimental High-Energy Physics"*, Preprint 91/09/CB, December 1991.

[45] W.Bartel el al. (JADE Coll.), Z.Physics **C33** (1986) 23.

[46] L.Lönnblad, C.Peterson and T.Rögnvaldsson, *"Using Neural Networks to Identify Jets"*, Nuclear Physics **B349** (1991) 675.

[47] C.Peterson, *"Neural Networks and High Energy Physics"*, presented at the International Workshop on Software Engineering, Artificial Intelligence and Expert Systems for High Energy and Nuclear Physics, Lyon, March 1990.

[48] C.Peterson, *"Track Finding with Neural Networks"*, Nuclear Instruments and Methods in Physics Research **A279** (1989) 537.

[49] L.Lönnblad, C.Peterson, Hong Pi and T.Rögnvaldsson, *"Self-Organizing Networks for Extracting Jet Features"*, submitted to Computer Physics Communications, Lund, March 1991

[50] S.J.Hanson and D.J.Burr, *"Minkowski-r Back-Propagation: Learning in Connectionist Models with Non-Euclidian Error Signals"*, in Neural Information Processing Systems, ed. D.Z.Anderson (1988) 348.

[51] B.Denby,*"Neural Network Tutorial for High Energy Physicists"*, FERMILAB-Conf-90/94, Batavia, May 1990.

[52] J.Kalkkinen, *"D^* Identification using Neural Networks"*, Internal Report, Helsinki, July 1991.

[53] A.Linussio, *"Nonlinear Multivariate Discriminant Analysis for High Energy Physics"*, Thesis, Università degli Studi di Udine, Udine, March 1992.

[54] T.Akesson and O.Bärring, *"Jet Classification with a Neural Network"*, DELPHI 90-59 PHYS 78, Lund, December 1990.

[55] P.Bhat, L.Lönnblad, K.Meier and K.Sugano, *"Using Neural Networks to Identify Jets in Hadron-Hadron Collisions"*, presented at the 1990

Summer Study on High Energy Physics, Snowmass (Colorado), July 1990.

[56] M.Metcalf, *"Effective FORTRAN 77"*, Claredon Press, Oxford 1987.

[57] CERN Program Library D421 (1989).

[58] R.Brun and D.Lienhart, *"HBOOK Users Guide"*, CERN Program Library Y250, Geneva 1988.

[59] R.Brun and N.Cremel Somon, *"HPLOT Users Guide"*, CERN Program Library Y251, Geneva 1988.

[60] R.Brun et al., *"PAW Physics Analysis Workstation"*, CERN Program Library Q121, Geneva 1990.

□ □ □