UNIVERSITY OF ROME

“TOR VERGATA”

Faculty of Engineering

Ph. D. Thesis on Engineering of Sensory and Learning Systems:

“Neural network approach to Problems of Static/Dynamic Classification”

Supervisor
Prof. Mario Salerno

Assistant Supervisor
Ing. Giovanni Costantini

Ph. D. Student
Ing. Massimo Carota

Winter Session
Academic year 2006/2007

CONTENTS

Abbreviations and symbols

M	n° of *layers* in the *network*
l	*layer* index. $l = 0 \Rightarrow$ *input layer*, $l = M \Rightarrow$ *output layer*
N_l	*# of neurons* in the l^{th} *layer*. N_0 = *# of inputs*, N_M = *# of outputs*
n	*neuron* index
T	*length* of an *input sequence* (length of a *learning epoch*)
t	*time index*: $t = 1, 2, \ldots, T$
$x_n^{(l)}[t]$	n^{th} *neuron output sequence* in the l^{th} *layer* at time *t*. $x_0^{(l)} = 1$: *bias input* to next *layer*. $x_n^{(0)}[t],\ n = 1, \ldots, N_0$: *input signals* supplied to the *network*.
$L_{nm}^{(l)} - 1$	In a *IIR-MLP*, it's the *order* of the *MA* part of the *synapse* of the n^{th} *neuron* of the l^{th} *layer* relative to the m^{th} output of the $(l - 1)^{th}$ layer. $L_{nm}^{(l)} \geq 1$ and $L_{n0}^{(l)} = 1$ (*bias*).
$I_{nm}^{(l)}$	In a *IIR-MLP*, it's the *order* of the *AR* part of the *synapse* of the n^{th} *neuron* of the l^{th} *layer* relative to the m^{th} output of the $(l - 1)^{th}$ layer. $I_{nm}^{(l)} \geq 0$ and $I_{n0}^{(l)} = 0$ (*bias*).
$w_{nm}^{(l)}$	*static MLP*: *weights* of the n^{th} *neuron* $(n = 0, \ldots, N_l - 1)$ in the l^{th} *layer* $(l = 1, \ldots, M)$, m^{th} *neuron* output from the previous *layer* $(m = 0, \ldots, N_{l-1})$; $w_{n0}^{(l)}$ is the *bias*.
$w_{nm(p)}^{(l)}$	$\left(p = 0, 1, \ldots, L_{nm}^{(l)} - 1\right)$ In a *IIR-MLP*, are the *coefficients* of the *MA* part of the corresponding *synapse*. If , $L_{nm}^{(l)} = 1$, the *synapse* has no *MA* part and the *weight* notation becomes $w_{nm}^{(l)}$. $w_{n0}^{(l)}$ is the *bias*.
$v_{nm(p)}^{(l)}$	$\left(p = 1, \ldots, I_{nm}^{(l)}\right)$ In a *IIR-MLP*, are the *coefficients* of the *AR* part of the corresponding *synapse*. If $I_{nm}^{(l)} = 0$, the *synaptic filter* is purely *MA*.
υ	*w* or *v coefficient*
$\varphi(\bullet)$	*activation function*
$\varphi'(\bullet)$	*derivative* of $\varphi(\bullet)$
$y_{nm}^{(l)}[t]$	In a *IIR-MLP*, it's the *synaptic filter output* at time *t* relative to the *synapse* of n^{th} *neuron*, of the l^{th} *layer*, and the m^{th} input. $y_{n0}^{(l)} = w_{n0}^{(l)}$ is the *bias*.
$s_n^{(l)}[t]$	*activation sequence* relative to the n^{th} *neuron*, of the l^{th} *layer*, at time *t*. It's the input to the corresponding *activation function*.
$d_n[t]$	$(n = 1, \ldots, N_M)$ *desired output sequence* at time *t*.
$N + 1$	*spline activation function*: # of *control points*
Δx	*spline activation function*: abscissa *sampling step*
$i_n^{(l)}$	*spline activation function*: $(0 \leq i_n^{(l)} \leq N - 2)$ n^{th} *neuron*, l^{th} *layer*, index of the *curve span*
$u_n^{(l)}$	*spline activation function*: $(0 \leq u_n^{(l)} \leq 1)$ n^{th} *neuron*, l^{th} *layer*, *local parameter* for the *curve span*
$q_{n,k}^{(l)}$	*spline activation function*: $(0 \leq k \leq N)$ n^{th} *neuron*, l^{th} *layer*, ordinate of the k^{th} *control point*
$F_{n,i_n^{(l)}}^{(l)}(\bullet)$	*spline activation function*: n^{th} *neuron*, l^{th} *layer*, m^{th} *CR polynomial*
$C_{n,m}^{(l)}(\bullet)$	*spline activation function*: $(0 \leq m \leq 3)$ n^{th} *neuron*, l^{th} *layer*, $i_n^{(l)\,th}$ *curve span*

PREFACE

The purpose of my doctorate work has consisted in the exploration of the potentialities and of the effectiveness of different neural classifiers, by experimenting their application in the solution of classification problems occurring in the fields of interest typical of the research group of the "Laboratorio Circuiti" at the Department of Electronic Engineering in Tor Vergata.

Moreover, though inspired by works already developed by other scholars, the adopted neural classifiers have been partially modified, in order to add to them interesting peculiarities not present in the original versions, as well as to adapt them to the applications of interest.

These applications can be grouped in two great families. As regards the first application, the objects to be classified are identified by features of static nature, while as regars the second family, the objects to be classified are identified by features evolving in time. In relation to the research fields taken as reference, the ones that belong to the first family are the following:

- classification, by means of fuzzy algorithms, of acoustic signals, with the aim of attributing them to the source that generated them (recognition of musical instruments)
- exclusive classification of simple human motor acts for the purpose of a precocious diagnosis of nervous system diseases

The second family of application has been represented by that research field that aims to the development of neural tools for the Automatic Tanscription of piano pieces.

The first part of this thesis has been devoted to the detailed description of the adopted neural classification techniques, as well as of the modifications introduced in order to improve their behavior in relation to the particular applications. In the second part, the experiments by means of which I have estimated the before-mentioned neural classification techniques have been introduced.

It exactly deals with experiments carried out in the chosen research fields. For every application, the results achieved have been reported; in some cases, the further steps to perform have also been proposed.

After a brief introduction to the biological neural model, a description follows about the model of the artificial neuron that has afterwards inspired all the other models: the one proposed by McCulloch and Pitts in 1943. Subsequently, the different typologies of architectures that characterize neural networks are shortly introduced, as regards the feed-forward networks as well as the recursive networks. Then, a description of some learning strategies (supervised and unsupervised), adopted in order to train neural networks, is also given; some criteria by means of which one can estimate the goodness of an opportunely trained neural network are also given (errors made vs. generalization capability). A great part of the adopted networks is based on adaptations of the Backpropagation algorithm; the other networks have been instead trained by means of algorithms based on statistical or geometric criteria. The Backpropagation algorithm has been improved by augmenting the degrees of freedom to the learning ability of a feed-forward neural network with the introduction of a spline adaptive activation function. A wide description has been given of the recurrent neural networks and particularly of the locally recurrent neural networks, networks for dynamic classification exploited in the automatic transcription of piano music.

After a more or less rigorous definition of the concepts of classification and clustering, some paragraphs have been devoted to some statistical and geometric neural architectures, exploited in the implementation of static classifiers of common use and in particular in the application fields that have regarded my doctorate work.

A separate paragraph has been devoted to the Simpson's classifier and to the variants originated from my research work. They have revealed themselves to be static classifiers very simple to implement and at the same time very ductile and efficient, in many situations as well as regards the

problem of musical source recognition. Two have been the choices in this case. In the first one, these classifiers have been trained, by means of a pure supervised learning approach, while in the second the training algorithm, though keeping a substantially supervised nature, is prepared by a clustering phase, with the aim of improving, in terms of errors and generalization, the covering of the input space. Subsequently, the locally recurrent neural networks seen as dynamic classifiers are retrieved. However, their training has been rethought according to the effective reduction of the classification error instead of the classic mean-square error.

The last three paragraphs have been devoted to a detailed description, in terms of specifications, implementative choices and final results, of the aforesaid fields of applications. The results obtained in all the three fields of application can be considered encouraging. Particularly, the recognition of musical instruments by means of the adopted neural networks has shown results tha can be considered out comparable if not better than those obtained by means of other techniques, but with considerably less complex structures. In case of the Automatic Transcription of piano pieces, the dynamic networks I adopted have given good results. Unfortunately, the required computational resources required by such networks cannot be considered negligible. As far as the medical applications, we are still in an incipent phase of the research. However, opinions expressed by those people who work in this field can be considered substantially eulogistic.

The research activities my doctorate work is part of have been carried out in collaboration with the Department “INFOCOM” of the first University of Rome “La Sapienza”, as far as the recognition of musical instruments and the Automatic Transcription of piano pieces. The necessity to study the potentialities of neural classifiers in medical application has instead come from a profitable existing collaboration with the Istituto Superiore di Sanità in Rome.

INTRODUCTION

Given some set of whatever kind of objects (called *patterns*), the *classification* process is essentially based on the peculiar traits and/or mutual relationships existing among those objects, that lie behind the structure of that set and that allow the identification of significant subsets, called *classes*. Each one of these *classes* includes objects that share, so to speak, the same "values" of those traits, or, in other words, that that are in some relation among each other. The detection of the above mentioned structure is carried out by means of an abstraction process, that, for every *class*, leads to the definition of a "model" representing all the objects belonging to that *class*, as regards only the relevant peculiarities of its objects. On the contrary, the "model" has to necessarily set aside all the characteristics of the objects that can be somehow considered not essential. The more this "model" is accurate, the more it will be effective when used to associate a new object to the *class* to which this object belongs to. These concepts can be better clarified by the following example. Every people is more or less aware of what a car can be, that is to say, of what the common "idea" (the "model") of a car can be. If we face a collection of many sorts of objects, we will most probably be capable of identifying among them one or more objects that we can call "a car".
The identification process can be briefly summarized, as follows:

1. every object in the collection is compared with the personal "idea" of a car, an "idea" based on the experience previously acquired about
2. each object is labelled as being a car, or less, depending on its adherence to the model

Obviously, some characteristics, as for example the colour of the car, are not relevant in order to the classification process; thus, we shouldn't find any sign of it inside the abstract model of a car. On the contrary, the model should include information about the salient characteristics distinguishing a car from any other object, as for example: a car has four wheels, some doors, as well as proper dimensions and mass. The key to the successful solution of a classification problem consists mainly in the correct development of the above mentioned descriptive models. A problem strictly related to *classification* is *clustering*. Trivially speaking, in *classification* the knowledge about number and type of classes is previously given, while *clustering* is a process that aims to discover the presence of agglomerates of objects, identified by having common characteristics, inside a universe of them, agglomerates that can be subsequently named *classes*. In other words, in *classification* the structure of the universe is known in advance, while in *clustering* it must be identified by *clustering* operation itself. The first obstacle to overcome, in the design of an automatic *classification* and/or

clustering system, regards the correct method to adopt in creating the "model" that embeds the characteristics of the different *classes* involved.

A first approach can be the *analytical* one, according to which the designer incorporates his knowledge about the application at issue into the elaboration system, in the form of an algorithm, i.e. a set of rules that the system has to slavishly follow when it is asked to face up to that application. Particularly, this is the approach followed by a programmer, when he develops a program that implements a simulation algorithm, in one of the existing programming languages. Nevertheless, this strategy has many drawbacks. First of all, the resulting elaboration system would have potentialities and knowledge at most coincident with those of its creator; it would be rigorously static, that is, it would be structurally unable to "grow up". Moreover, the system would be unable to tackle situations that the designer was previously unable to foresee, or that cannot be faced starting from the instructions that the designer gave to it (induction inability). Furthermore, every problem to solve would require a well defined system typology, to be designed from scratch whenever a new application is proposed (system architecture strictly bounded to the problem). Last but not least, some problems would require calculation systems with a so high degree of complexity that every practical implementation would be inconceivable.

Anyway, we can imagine to turn to a radically different approach. If we take a look at how human brain constructs its mental categories, we jump to the following deduction: a *classification* system has to be able to "learn", that is to say, to build by itself (in an *unsupervised* way) abstract models, on the basis of examples (the *learning* or *training* process). Moreover, it must be provided with some kind of reasoning ability, thru which it can handle even situations that it never faced before (*generalization* capability, or *validation*). In some sense, this will be the approach that we will follow. Precisely, to efficiently solve problems of *classification* ad *clusterization* we will turn to a computational paradigm directly inspired to human brain: *neural networks.*

Neural networks are particularly suited to problems of *classification* ad *clusterization* thanks to their *generalization* capabilities. Actually, and according to what we stated before, the solution of a *classification* problem cannot be accomplished regerdless this capability. An important challenge in *classification* consists in extending some of the standard algorithms from simple *static data* (this is the situation until now considered, actually), to *time varying data*. In fact, it will be essential for our purposes to extend *classification* algorithms from *data sets* consisting of *static vectors* (points) to data sets consisting of *time varying vectors* (*trajectories*), in order to face problems of *dynamic classification*, based on some form of *time series processing*. We will consider the special case in which *time varying data* are collected by sampling trajectories from an underlying family of parameterized *dynamical systems*. *Dynamic classification* problems occur frequently in practice:

- *time series* prediction and modelling
- noise cancelling
- adaptive equalization of a communication channel
- adaptive control
- system identification

In order to describe, explore and control the behaviour of *linear systems*, *dynamical system* theory provides us with a very efficient tool, but it reduces to a local analysis tool when the hypothesis of linearity falls. Unfortunately, most of the *classification* problems are non linear. So, for this reason also, *neural networks* are an essential tool.

CHAPTER 1

NEURAL NETWORKS

1.1 The Brain, the nervous system and their model

Learning is a mental process by means of which experience influences human behaviour. At the foundation of this process, we can find an extremely complex structure: the brain. Much as been discovered about the real working of human brain, but a lot still remains to be explained. Anyhow, medical researches have established many of the processes that occur inside of it. Human brain is essentially a huge and highly complex network in the nodes of which we can find the so called nerve cells, or *neurons*. Human brain contains about 10 billion *neurons*. On average, each *neuron* is connected to other *neurons* through about 10000 *synapses*. The brain's network of *neurons* forms a massively parallel information processing system. This contrasts with conventional computers, in which a single processor executes one or more series of instructions. Just for comparison, let's consider the time taken for each elementary operation: *neurons* typically operate at a maximum rate of about 100 Hz, while a conventional CPU carries out several hundred million machine level operations per second. Nevertheless, despite of being built with very slow hardware, the brain has quite remarkable capabilities:

- In case of partial damage, its performances tend to degrade a little. In contrast, most programs and engineered systems are very little robust: if you remove some arbitrary parts, very likely the whole will cease to function.
- it can learn (reorganize itself) from experience.
- this is why partial recovery from damage is possible: healthy units can learn to take over the functions previously carried out by the damaged areas.
- it performs massively parallel computations with extreme efficiency. For example, complex visual perception occurs within less than 100 ms!
- it supports our intelligence and self-awareness (but nobody still knows how this occurs)

The brain is not homogeneous. At the largest anatomical scale, we distinguish cortex, midbrain, brainstem, and cerebellum. Each of these can be hierarchically subdivided into many regions and areas within each region, either according to the anatomical structure of the neural networks that form them or according to the functions they perform (Fig 1.1).

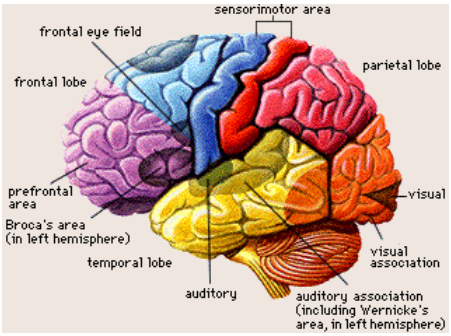


Fig. 1.1. the brain.

The overall pattern of projections (bundles of *neural* connections) between areas is extremely complex, and only partially known. The best mapped (and largest) system in the human brain is the visual system, where the first 10 or 11 processing stages have been identified. We distinguish *feedforward* projections that go from earlier processing stages (near the sensory input) to later ones (near the motor output), from *feedback* connections that go in the opposite direction. In addition to these long-range connections, *neurons* also link up with many thousands of their neighbours. In this sense they form very dense, complex local *networks* (Fig 1.2).

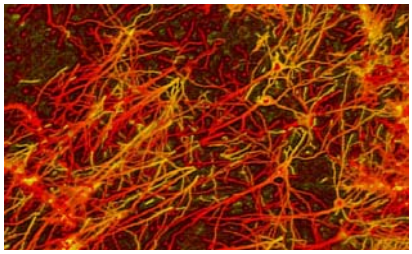


Fig. 1.2. the neural network.

1.1.1 The Biological Neuron

The basic computational unit in the nervous system is the nerve cell, or *neuron*. It's formed by:

- ***Dendrites*** (inputs)
- ***Cell body***
- ***Axon*** (output)

A *neuron* (Fig. 1.3) has a roughly spherical cell body called *soma*. Signals generated in the *soma* are transmitted to other *neurons* through an extension on the cell body called *axon,* or nerve fibre. Around the cell body, we can find other extensions, called *dendrites* (bushy tree shaped), which are responsible for receiving the incoming signals generated by other *neurons*. The *axon* (Fig. 1.3), has a length that varies from a fraction of a millimetre to a meter in human body and prolongs from the *cell body* at the point called *axon hillock*. At the other end, the *axon* splits into several branches, at the very end of which we find the *terminal buttons*. *Terminal buttons* are placed in special structures called the *synapses* which are the junctions transmitting signals from one *neuron* to

another. A *neuron* typically drives 103 to 104 *synaptic junctions*.

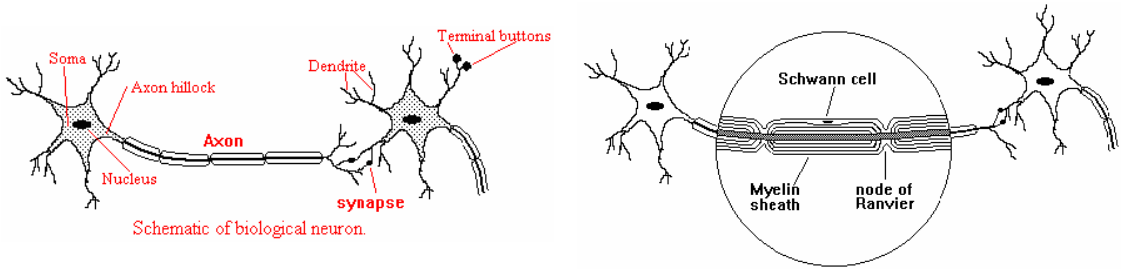


Fig. 1.3. the biological neuron.

A *neuron*'s *dendritic tree* is connected to a thousand neighbouring *neurons*; so, a *neuron* receives its input from other *neurons* (typically many thousands). When one of those *neurons* fires, a positive or negative charge is received by one of the *dendrites*. The strengths of all the received charges are added together and the aggregate input is then passed to the *soma* (*cell body*). The *soma* and the enclosed *nucleus* don't play a significant role in the processing of incoming and outgoing data. Their primary function is to perform the continuous maintenance required to keep the *neuron* functionalities. The part of the *soma* that participates to the elaboration of the incoming signals is the *axon hillock*: if the aggregate input is higher than the *axon hillock*'s *threshold* value, then the *neuron* fires (the *neuron* becomes *active*, otherwise it remains *inactive*) and an output signal is transmitted down the *axon*. In other words, once input exceeds the *threshold*, the *neuron* discharges a *spike* – an electrical pulse that travels from the *body*, down the *axon*, to the next *neuron*(s) (or other receptors). This spiking event is also called depolarization and is followed by a refractory period, during which the neuron is unable to fire. The *terminal buttons* (output zones) almost touch the *dendrites* or cell body of other *neurons*, leaving a small gap. Transmission of an electrical signal from one *neuron* to another is effected by *neurotransmitters*, chemicals transmitters which are released from the first *neuron* and which bind to receptors in the second. This link is called a *synapse* (Fig. 1.4). The *synaptic vesicles*, holding several thousands of molecules of *neurotransmitters*, are situated in the *terminal buttons*. When a nerve impulse arrives at the *synapse*, some of these *neurotransmitters* are discharged into the *synaptic cleft* – the narrow gap between the *terminal button* of the transmitting *neuron* and the *membrane* of the receiving *neuron*. In general, the *synapses* lie between an *axon branch* of a *neuron* and the *dendrite* of another *neuron*. Although it is not very common, *synapses* may also lie between two *axons* or two *dendrites* of different cells or between an *axon* and a *cell body*. Neurons are covered with a semi-permeable 5 nm *membrane*, that's able to selectively absorb and reject ions in the intracellular fluid. The *membrane* basically acts as an ion pump in order to maintain a different ion concentration between the intracellular fluid and extra cellular fluid. While the sodium ions are continually removed from the intracellular fluid to extra cellular fluid, the potassium ions are absorbed from the extra cellular fluid in order to

maintain an equilibrium condition.

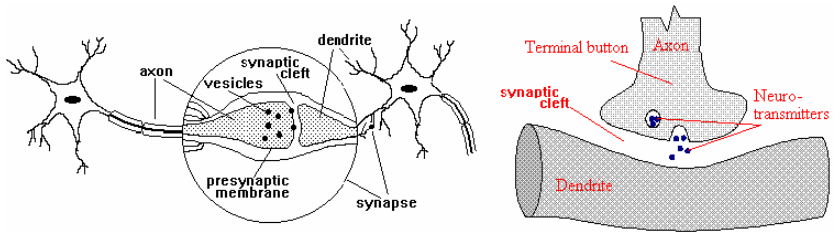


Fig. 1.4. the synapse.

Due to the difference in the ion concentrations inside and outside, the *cell membrane* become polarized. In equilibrium, the interior of the *cell* is observed to be 70 mV (*resting potential*) negative with respect to the outside of the *cell*. Nerve signals arriving at the *presynaptic cell membrane* cause *neurotransmitters* to be released in to the *synaptic cleft*; then they diffuse across the gap and join the *postsynaptic membrane* of the receptor site. The *membrane* of the *postsynaptic cell* gathers the *neurotransmitters*. This cause either a decrease or an increase in the efficiency of the local sodium and potassium pumps, depending on the type of the chemicals released in the *synaptic cleft*. The *synapses* whose activation decreases the efficiency of the pumps cause depolarization of the *resting potential*, while the *synapses* which increases the efficiency of pumps cause its hyper polarization. The first kind of *synapses* (encouraging depolarization) is called *excitatory*, while the others (discouraging depolarization) are called *inhibitory*. If the decrease in the polarization is adequate to exceed a *threshold* then the *post-synaptic neuron* fires. The arrival of impulses to *excitatory synapses* adds to the depolarization of *soma*, while *inhibitory* effect tends to cancel out the depolarizing effect of *excitatory impulse*. In general, although the depolarization due to a single *synapse* is not enough to fire the *neuron*, if some other areas of the *membrane* are depolarized at the same time by the arrival of nerve impulses through other *synapses*, it may be adequate to exceed the *threshold* and fire. The excitatory effects result in interruption of the regular ion transportation through the *cell membrane*, so that the ionic concentrations immediately begin to equalize as ions diffuse through the *membrane*. If the depolarization is large enough, the *membrane* potential eventually collapses, and for a short period of time the *internal potential* becomes positive (Fig. 1.5). The *action potential* is the name of this brief reversal in the potential, which results in an electric current flowing from the region at *action potential* to an adjacent region with a *resting potential*. This current causes the potential of the next resting region to change, so the effect propagates in this manner along the *membrane wall*. Once an *action potential* has passed a given point, it cannot be re-excited for a short period of time called *refractory period*. Because the depolarized parts of the *neuron* are in a state of recovery and cannot immediately become active again, the pulse of electrical activity propagates forward only.

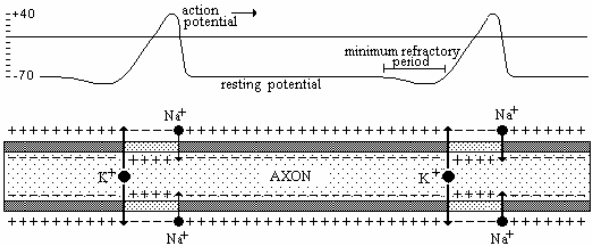


Fig. 1.5. The Action Potential on Axon.

The previously triggered region then rapidly recovers to the polarized resting state, due to the action of the sodium potassium pumps. The *refractory period* is about 1000 ms, and it limits the nerve pulse transmission, so that a *neuron* can typically fire and generate nerve pulses at a rate up to 1000 pulses per second. The number of impulses and the speed at which they arrive at the synaptic junctions determine whether the total excitatory depolarization is sufficient to cause the *neuron* to fire and send a nerve impulse down to its *axon*. The depolarization effect can propagate along the *cell wall* but these effects can be dissipated before they reach the *axon*. However, once the nerve impulse reaches the *axon hillock*, it will propagate until it reaches the *synapses* where the depolarization effect will cause the release of *neurotransmitters* into the *synaptic cleft*. The *axons* are generally enclosed by *myelin sheath*. The speed of propagation down the *axon* depends on the thickness of the *myelin sheath* that provides for the insulation of the *axon* from the extra cellular fluid and prevents the transmission of ions across the *membrane*. The *myelin sheath* is interrupted at regular intervals by narrow gaps called *nodes of Ranvier*, where extra cellular fluid comes into contact with *membrane* and the transfer of ions occur. Since the *axons* themselves are poor conductors, the action potential is transmitted as depolarizations occur at the *nodes of Ranvier*. This happens in a sequential manner, so that the depolarization of a node triggers the depolarization of the next one. The nerve impulse effectively jumps from a node to the next one along the *axon*, each node acting rather like a regeneration amplifier to compensate for losses. Once an action potential is created at the *axon hillock*, it is transmitted through the *axon* to other *neurons*. We conclude that signals in the *nervous system* are digital in nature, since the *neuron* is assumed to be either *fully active* or *inactive*. However, this conclusion is not that correct, because the intensity of a *neuron* signal is coded in the frequency of a train of invariant pulses of activity. In fact, the biological neural system can be better explained by means of a form of pulse frequency modulation that transmits information. The nerve pulses passing along the *axon* of a particular *neuron* are of approximately constant amplitude, but the number of generated pulses and their time spacing is controlled by the statistics associated with the arrival at the neuron's many *synaptic junctions* of sufficient *excitatory inputs*. The representation of *biophysical neuron output* behaviour is shown schematically in the diagram given in Fig. 1.6. At time $t = 0$, a *neuron* is excited; at time T (typically after about 50 ms), the *neuron* fires a *train of impulses* along its *axon*.

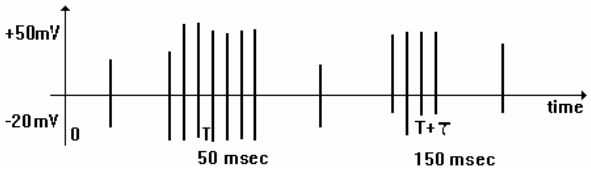


Fig. 1.6. Pulse Trains.

The extent to which the signal from one *neuron* is passed on to the next depends on many factors, e.g. the amount of *neurotransmitter* available, the number and arrangement of receptors, amount of *neurotransmitter* reabsorbed, etc. The strength of the output is almost constant, regardless of whether the input was just above the *threshold*, or a hundred times as great. The physical and neurochemical characteristics of each *synapse* determines the strength and polarity of the new input signal. This is where the brain is the most flexible, and the most vulnerable. Changes in the chemical composition of *neurotransmitter* increase or decrease the amount of stimulation that the firing *axon* imparts on the neighbouring dendrite. The alteration of the *neurotransmitters* can also change whether the stimulation is excitatory or inhibitory. Many drugs such as alcohol and LSD have dramatic effects on the production or destruction of these critical chemicals.

1.1.2 Synaptic Learning

From what we know about neural structures, we can argue that brain learns by altering the strengths of connections between its *neurons* and by adding or deleting connections between them. Brain learns "on-line", based on experience, and typically without the benefit of a benevolent teacher. Then, learning occurs by changing the efficacy of *synapses*, so changing the influence of a *neuron* on the others. The first notable rule that describes the actual functioning of learning through synaptic efficacy changing is the well known Hebbs rule, that states: *when an axon of cell A excites cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells, so that A's efficiency as one of the cells firing B is increased*. As a matter of fact, psychologists agree on the assumption that the mechanism behind children learning is simply based on associative comparisons. Let's suppose to instruct a child to distinguish between a chair and a table. During the *learning* phase, many *examples* of chair, as well as of table, will be submitted to the child, and he will always be informed about what *example* is that of a chair and what is that of a table. This way, sooner ore later, the child will have built inside is mind some criteria about how to distinguish between a chair and a table and how to associate all the proposed chairs to a *class* "chair" and all the proposed tables to a *class* "table". At the end of the training phase, the child will be able to associate a new example of chair or table to one of the two classes, eventually with some uncertainty. Actually, what has been modified during the *learning*

phase is just the brain of the child or, exactly, the *neural network* that constitutes his brain.

Many attempts have been made to simulate the behaviour of the biological brain by means of the so called *artificial neural networks* (*ANN*). Moreover, the approach followed to teach an *ANN* how to solve a *classification* problem has been derived from the biological counterpart. Firstly, a *training* phase is carried out: some elements belonging to a well defined *class* (*training set*) are supplied to the *ANN*, together with an explicit information (*label*) about the *class* every element is part of. Then, for every *example* (element + *label*), the *ANN* "calculates" a new configuration of its *artificial neurons* (*AN*), based on the input *example* received. After the *training phase*, the *network* passes thru a *validation* (or *generalization*) phase: the *ANN* is given an input element it never "saw" before, but belonging to one of the *classes* considered during the *learning phase*, in ordet to test whether it's able or not to correctly *classify* the new element. The data set used during this phase is called *validation set*. Every people is aware that, as far as computation power is concerned, even a small programmable calculator can be considered enormously stronger than human brain. Nevertheless, it's an experience of every day life that, at least at present technology level, our brain is much faster and efficient when facing identification problems. This is why it was obvious trying to imitate the functioning of the nervous system by implementing such a thing on ordinary computer systems, namely, the ability to learn by example. This is the starting point that has given birth to the *ANNs*.

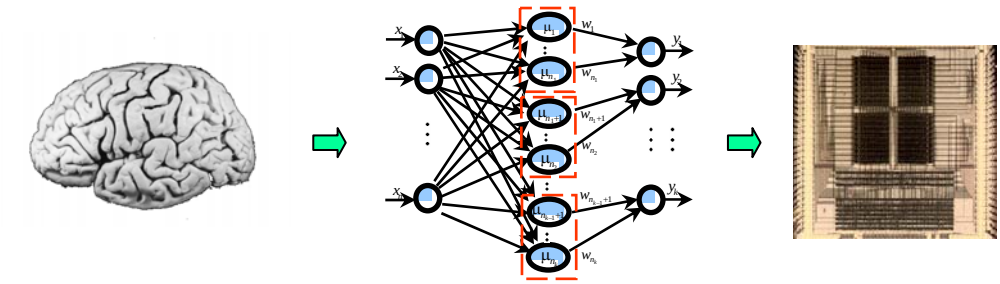


Fig. 1.7. from human brain to a neural microchip (design by Electronic Dpt – University of Rome "Tor Vergata").

1.2 Artificial Neuron Networks Models

Computational neurobiologists have constructed very elaborate computer models of *neurons* in order to run detailed simulations of particular circuits inside the brain. However, in the field of Computer Science, there's more interest in the general properties of *ANNs*, regardless of how they are actually "implemented". This means that we can use much simpler, abstract *ANs*, which (hopefully) capture the essence of neural computation, even if they leave out much of the details of how biological *neurons* work. People have implemented *AN* models in hardware, as electronic circuits, often integrated on VLSI chips, while others have run fairly large networks of simple *neuron* models as software simulations.

1.2.1 A Simple Artificial Neuron

As we have seen, the transmission of a signal from one *neuron* to another through *synapses* is a complex chemical process. The effect is to raise or lower the electrical potential inside the *body* of the receiving cell. If this potential reaches a *threshold*, the *neuron fires*. More or less, this is the only characteristic that the *AN model* proposed by *McCulloch and Pitts* [1943] attempts to reproduce (Fig 1.8).

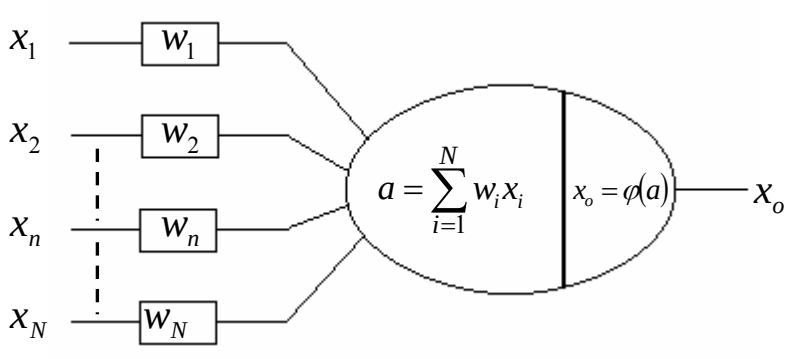


Fig. 1.8. the Artificial Neuron of McCulloch and Pitts.

The *AN* receives input from some other *neurons*, or in some cases from an external source. It has N inputs $(x_1, x_2, \ldots, x_N)$, and each input is associated to a *synaptic weight* $(w_1, w_2, \ldots, w_N)$. *Weights* in the artificial model correspond to the *synaptic connections* in biological *neurons* and can be modified in order to model *synaptic* learning. The *weighted sum* of its inputs is called *activation*. If θ is the *threshold*, then the *activation* is given by:

$$a = \sum_i w_i x_i + \theta \qquad (1.1)$$

Inputs and weights are real values. A negative value for a weight indicates an *inhibitory connection*, while a positive value indicates an *excitatory* one. θ is sometimes called *bias*. The *threshold* can be included into the summation, by adding a further input $x_0 = +1$ with a connection *weight* $w_0 = \theta$.

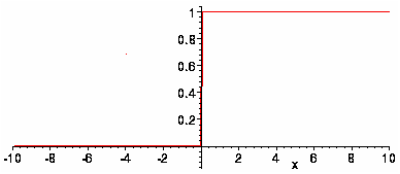
Hence the *activation* formula becomes:

$$a = \sum_{i=0}^{N} w_i x_i \tag{1.2}$$

That in vector notation can be written as: $a = \mathbf{w}^T \cdot \mathbf{x}$, where both vectors $\boldsymbol{w}$ and $\boldsymbol{x}$ are of size $N+1$. The output value of the *neuron* is given by a so called *activation function* (*AF*), where its argument is the *activation* and it is analogous to the firing frequency of the *biological neurons*:

$$x_o = \varphi(a) \tag{1.3}$$

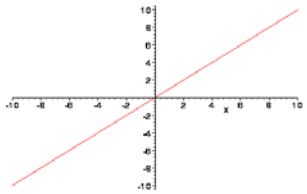
The original *AF* proposed by McCulloch Pitts was a *threshold function* (Fig 1.9).



$$\varphi(x) = \begin{cases} 0 & x \le 0 \\ 1 & x > 0 \end{cases}$$

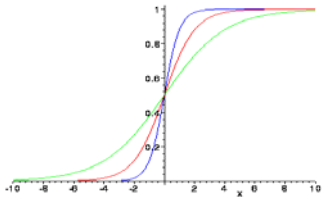
Fig. 1.9. Step (or threshold) function.

Subsequently, mainly for function and derivative continuity reasons, *linear*, *logistic*, *hyperbolic*, and *radial-basis* functions have been introduced (Fig. 1.10 to Fig. 1.13).



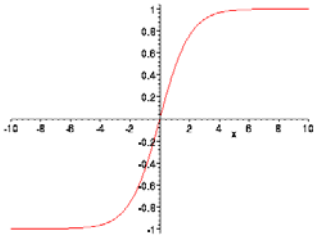
$$\varphi(x) = kx \qquad \frac{d\varphi}{dx} = k$$

Fig. 1.10. Linear (or identity) function.



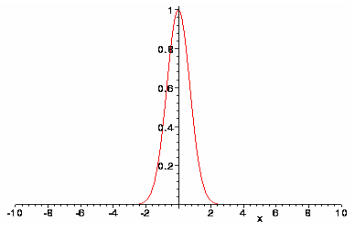
$$\varphi(x) = \frac{b}{1+e^{-ax}} \qquad \frac{d\varphi}{dx} = \varphi(x)[1-\varphi(x)]$$

Fig. 1.11. Logistic function (or sigmoid).



$$\varphi(x) = tgh(ax) \qquad \frac{d\varphi}{dx} = 1-\varphi^2(x)$$

Fig. 1.12. Hyperbolic tangent.



$$\varphi(x) = e^{-ax^2} \qquad \frac{d\varphi}{dx} = -2ax\varphi(x)$$

Fig. 1.13. Radial Basis Function.

The derivative of the step function is not defined and this is exactly why it isn't used. McCulloch and Pitts proved that a synchronous assembly of such *neurons* is capable in principle to perform any computation that an ordinary digital computer can do, though not necessarily so rapidly or conveniently. Namely, when the *threshold function* is used as *AF*, and binary input values 0 and 1 are assumed, the basic Boolean functions AND, OR and NOT of two variables can be implemented by choosing appropriate *weights* and *threshold* values, as shown in Fig. 1.14.

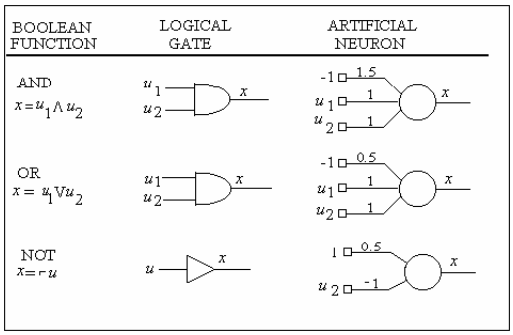


Fig. 1.14. Implementation of Boolean Functions by means of the AN.

1.2.2 Networks of Artificial Neurons

While a single *AN* could not be able to implement more complicated Boolean functions (the XOR, for example), the problem can be overcome by connecting more *neurons* in order to form a so called *neural network*. The *activation* of the *n*-th *neuron* is as follows:

$$a_n = \sum_{i=0}^{N} w_{ni} x_i \tag{1.4}$$

where x_i may be either the output of another *neuron*:

$$x_i = \varphi_i \left(a_i \right) \tag{1.5}$$

or an *external input* (*bias* included). Sometimes it may be convenient to think all the *network inputs* as being supplied to the *network* by means of the so called *input neurons*. They can be thought as *neurons* with a fixed null *bias weight*, with one input only and with a *linear AF*. We define a vector $\boldsymbol{x}$, the n^{th} component of which is the n^{th} neuron output. Furthermore, we define a *weight matrix* $\boldsymbol{W}$, the component w_{in} of which is the *weight* of the *connection* from *neuron i* to *neuron n*. The *network* can then be defined as follows:

$$\mathbf{x} = \mathbf{f}\left(\mathbf{W}^T \cdot \mathbf{x} \right) \tag{1.6}$$

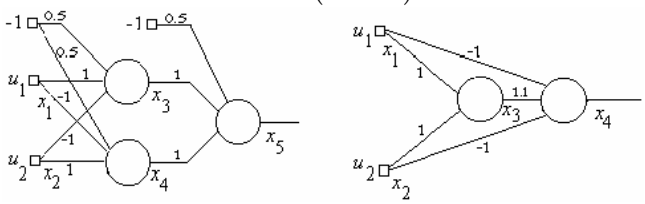


Fig. 1.15. various implementations of the XOR function by means of artificial neurons

For example, two *ANNs* that implement the behaviour of the boolean XOR function can be those depicted in Fig. 1.15. In vector notation, the second neural network of Fig. 1.15 can be expressed as:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \mathbf{f}\left(\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ -1 & -1 & 1.1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} u_1 \\ u_2 \\ 0 \\ 0 \end{bmatrix} \right) \tag{1.7}$$

where f_1 and f_2 are *identity functions*, f_2 and f_4 are *threshold functions*. In case of *binary input*, $u_i \in \{0,1\}$, or *bipolar input*, $u_i \in \{-1,1\}$, all of f_i may be chosen as *threshold function*. Note that the diagonal entries of the *weight matrix* are zero, since the *neurons* do not have *self-feedback*. Moreover, the *weight matrix* is upper triangular, since the *network* is *feedforward*.

1.2.3 Architectures for Networks of Artificial Neurons

Neural computing is an alternative to *programmed computing*. It's based on a *network* of *ANs* and a huge number of *interconnections* between them. According to the structure of those *connections*, we identify different classes of *network architectures*.

1.2.3.1 Feedforward Neural Networks

Neurons are grouped in what we call *layers*. All the *neurons* in a *layer* get *input* from the previous *layer* and feed their *output* to the subsequent *layer* (Fig. 1.16). *Connections* to *neurons* in the same or previous *layers* are not permitted. The last *layer* is called *output layer* and the *layers* between the *input* and *output layers* are called *hidden layers*.

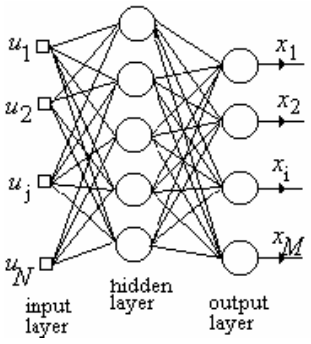


Fig. 1.16. Layered feedforward neural network

The *neurons* of the *input layer* serve just the purpose of transmitting the *applied external input* to the *neurons* of the first *hidden layer*. If there's no *hidden layer*, the *feedforward neural network* is considered as a *single layer network*. If there is at least one *hidden layer*, such *networks* are called *feedforward multilayer neural networks* (*FFNN*). For a *feedforward network*, the *weight matrix* is

triangular. Since *self-feedback neurons* are not allowed, the diagonal entries are zero. As the *connection graph* of a *FFNN* doesn't contain cycles, the non linear input/output transfer function will be *static*, that is the values returned by the output of the *network* at any time *t* will depend solely on the values presented to the input at $t - \Delta t$, where Δt equals the delay introduced by the propagation through the *network* itself.

1.2.3.2 Recurrent Neural Networks

The structures in which *connections* to the *neurons* of the same *layer* or to the previous *layers* are allowed are called *recurrent neural networks* (*RNN*). In other words, in a *RNN*, a *neuron* is permitted to get *input* from any *neuron* inside the structure and so *feedback connections* are allowed (Fig. 1.17). Anyway, even in this case *neurons* can be organized in *layers*: some neurons are *input neurons* and some others are *output neurons*. Al the others can be called *hidden neurons*. In case of *RNNs*, due to the presence of *feedbacks*, it is not possible to obtain a triangular *weight matrix* with any assignment of the indices. Furthermore, the diagonal entries can be non null.

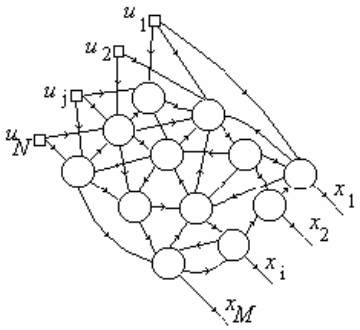


Fig. 1.17. Non-layered recurrent neural network

The behaviour of a *RNN* can be described in terms of a *dynamical system*. Because of the presence of *closed loops*, the affection of one *neuron output* to the *output* of another *neuron* must be considered in time. As a consequence, we can think of *RNNs* as dynamical systems for the processing of *sequences of patterns* (or shortly, *sequences*). In fact, given an *input pattern* at time *t*, the output of the *network* at the same time *t* doesn't depends on that *input pattern* at time *t* only, but even on the past history of the *network*, or, in other words, on the *input patterns* previously presented to the *network*. The *state* of the *network*, is a vector in which each entry corresponds to the *output* of a *neuron* in the *network*. The starting values of the entries of this vector form the *initial state* of the *network*. The *outputs* of the *neurons* change in time and if the *network* converges to a *final state*, which is not changing any more, the *network* is considered *asymptotically stable*. The *states* which are not changing are called *equilibrium states*. The *connection weights* and *threshold* values determine the *equilibrium states* of the *system*. Given an *equilibrium state*, the set

of neighbouring *states* converging to it is called the *basin of attraction* of that *equilibrium state*. The *connection weights* and the *threshold* values affect also the *basins of attraction*. The *energy function* of a *RNN* (Fig 1.18) is a bounded scalar function defined in terms of the *state* of the *network*. Each *local minimum* of the *energy function* corresponds to an *equilibrium state* of the *network*.

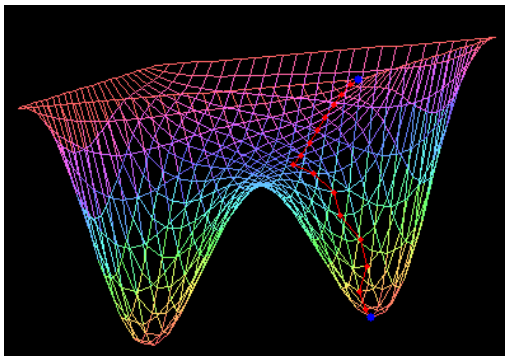


Fig. 1.18. energy function of a network with two neurons

The surface in Fig. 1.18 represents the *energy function* of a two *neuron network*, where *outputs* correspond to *x* and *y* axes. The *energy* value for a *state* is given by the height of the *energy surface*. The *energy surface* has two *local minima*, each corresponding to an *equilibrium state*. The sequence of red points represent a *trajectory*, that is the locus of the *states* passed through by the *network* during its evolution in time, while *energy* is decreasing. The blue points are used to represent the *initial state* (the upper) and *final state* (the lower), that's an *equilibrium state*. Obviously, we have two different *basins of attraction*, one for each *equilibrium state*.

1.2.4 Supervised and Unsupervised Learning

In general, *neural networks* are unreplaceable in all those applications in which we don't know the exact nature of the relationship existing between inputs and outputs (the model is unknown). In this case, the power of *neural networks* consists in that they *learn* the input/output relationship by *training*. A *neural network* can *learn* in two ways: by a *supervised training* or by an *unsupervised training*, of which the former is the most common.

Supervised learning is a machine learning technique for creating a function starting from a *training data set*. In other words, the *training* is based on a *training set* that contains *examples* formed by pairs (*input*, *desired output*, typically vectors) and the *network* learns to infer the relationship existing between the two. *Training data* are usually taken from historical records. The output of the function can be a continuous value (*regression* problem), or can predict a *class* label for the input object (*classification* problem). The task of the *supervised learner* is to predict the value of the function for any valid *input object* after having "studied" a number of *training examples* (i.e. pairs of *input* and *target* output). To this purpose, the *learner* has to *generalize* from the presented data to

unseen situations in a "reasonable" way. The best known *supervised learning algorithm* is *back propagation* (Rumelhart et. Al., 1986), which adjusts the *network*'s *weights* and *thresholds*, so as to *minimize* the *mean-square error* in its predictions on the *training set*. If the *network* is properly trained, it has then learned to model the (unknown) function that relates the input variables to the output variables, and can subsequently be used to make *predictions* where the output is not known.
In *unsupervised learning*, the purpose is to train a model to fit some *observations*. It is distinguished from *supervised learning* by the fact that there is no a priori output. In *unsupervised learning*, a *data set* of *input objects* is gathered. *Unsupervised learning* then typically treats *input objects* as a set of random variables. A joint density model is then built for the data set. In case of *unsupervised learning*, *training algorithms* adjust the *network weights* starting from *training data sets* that include the input values only. A form of *unsupervised learning* is *clustering* (partitioning of a *data set* into subsets called *clusters*), which is sometimes not probabilistic (*clustering problem*).

1.2.5 Fixed and variable architecture training algorithms

Fixed architecture training algorithms consider a *network* the structure of which is a priori defined. One the most relevant difficulties encountered in a *classification problem* is that, prior to training, we need to properly fix the number and the kind of interconnections of the *neurons* that compose the *network* itself. This kind of *training algorithms* suffer from the problem of avoiding the so called *overfitting* and *underfitting*, in relation to the specific problem to solve. In case of *overfitting*, we can compromise the *generalization capability* of the *network* and come up against systems too huge to be exploited, while in case of *underfitting*, it can happen that no *network parameter* configuration can guarantee the convergence of the *training algorithm* (the error remains too high).
The *variable architecture training algorithms*, on the contrary, don't require any a priori decision for what concerns the dimensions of the structure to adopt. In this case, the *network* automatically extends and shrinks, under provision of the algorithm, until an optimal - at least in theory - configuration is obtained, relatively to the particular problem to solve. In this case, besides the algorithm, an *optimization procedure* must be defined that finds the *network* that exposes the less structural complexity under the same performances.

1.3 Feedforward Multilayer Neural Networks (FFNN)

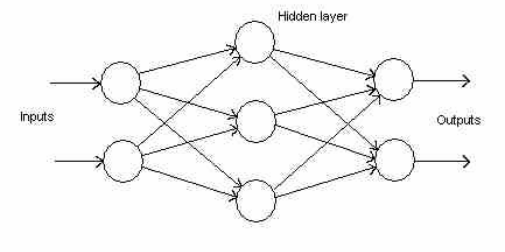


Fig. 1.19. FFNN architecture

These are perhaps the most popular *network architectures* in use today, where each units performs a *biased weighted sum* of its inputs and passes this *activation level* through an *AF* to produce its output. The units are arranged in a *layered feedforward* topology. The *weights* of the *connections* and the *thresholds* (*biases*) are the free parameters of the model. Such *networks* can model non-linear functions of almost arbitrary complexity, with the number of *layers* and the number of *neurons* in each *layer* that determine the function complexity. Important issues in the design of *FFNNs* include specifications about the number of *hidden layers* and the number of *neurons* in each *layer*. The number of *input* and *output neurons* is defined by the problem, while the number of *hidden units* to use is far from clear. A good starting point could be to use one *hidden layer*, with the number of *neurons* equal to half the sum of the number of *input* and *output units*.

1.3.1 The Back-Propagation Algorithm

It's a *training algorithm* that can be counted among the *supervised learning algorithms*. Once the number of *layers* and the number of *neurons* in each *layer* has been fixed, the *network*'s *weights* and *thresholds* must be *adjusted* in order to minimize the *prediction error* made by the *network*, once gathered historical *examples* are given. This is the role of the *training algorithms*. This process is equivalent to fitting the model represented by the *network* to the available *training data*. The *error* made by a particular *configuration* of the *network* can be determined by running all the *training examples* through the *network* and comparing the generated *actual output* with the *desired* or *target outputs*. The differences are combined together by an *error function* to give the *network error*. The most common *error function* is the *mean-square error*, where the individual errors of the *output neurons* on each *example* are squared and summed together. The target consists in algorithmically determining the model *configuration* that absolutely minimizes this *error*. Each of the *weights* and *thresholds* of the *network* (*free parameters* of the model) is taken as a dimension of a multidimensional space, on which the *network error function* is defined. For any possible *configuration* of *weights* and *thresholds*, the *error function* can be plotted forming an *error surface*.

The objective of *network training* is to find the lowest point in this multidimensional surface. In a linear model ($y = w_1 x + w_0$), this *error surface* is a parabola, that is a smooth bowl-shaped surface with a single minimum (Fig. 1.20).

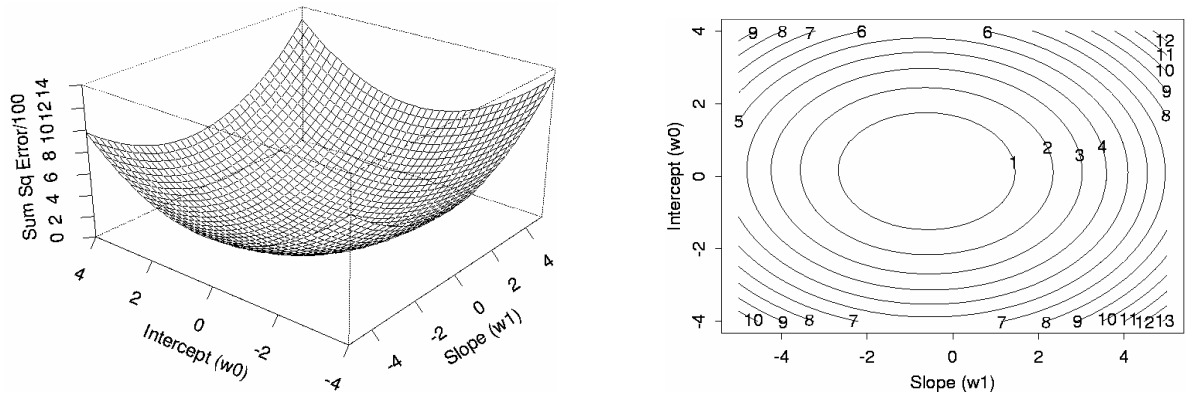


Fig. 1.20. Error surface in case of a linear model

In general, *neural network error surfaces* are much more complex and are characterized by a number of unhelpful features, such as *local minima* (which are lower than the surrounding terrain, but above the *global minimum*), flat-spots and plateaus, saddle-points, and long narrow ravines. It is not possible to analytically determine where the *global minimum* of the *error surface* is, and so *neural network training* is essentially an exploration of the *error surface*. From an initially random *configuration* of *weights* and *thresholds* (i.e., a random point on the *error surface*), the *training algorithm* incrementally seeks for the *global minimum*. Substantially, the *global minimum* of the *error surface* is searched by means of the *gradient* (slope) of the *error surface* itself, that's calculated at the current point and used to make a downhill move. Eventually, the algorithm stops in a low point, which may be a *local minimum* (but hopefully is the *global minimum*). Given a set of values for the *network* parameters, the *gradient* gives us the direction along which the *error surface* has the *steepest slope*. In order to decrease di *error*, we take a small step in the direction opposite to the one shown by the *gradient* (Fig. 1.21).

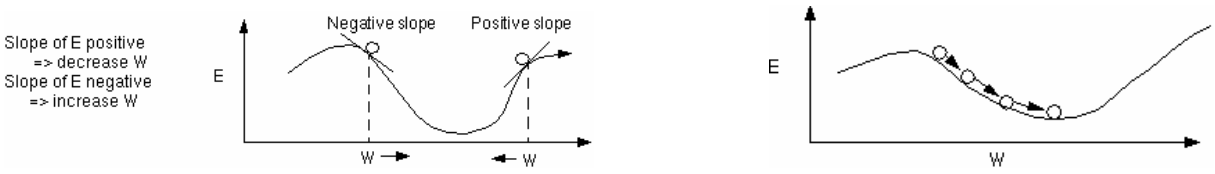


Fig. 1.21. The gradient of the Error function

By repeating this over and over, we move "downhill" until we reach a minimum where *gradient = 0*, so that no further progress is possible.

Let's now describe in details the *backpropagation* algorithm that's the most widely used *training* algorithm for *FFNN*s *learning*. Modern second-order algorithms, such as *conjugate gradient*

descent and *Levenberg-Marquardt* are substantially faster (an order of magnitude faster) for many problems, but *backpropagation* still has advantages in some circumstances and is the easiest algorithm to understand. There are also heuristic modifications of *backpropagation* that work well for some problem domains, such as *quick propagation*. Our goal is to find the *best network configuration* or, in other words, to find the values for the *parameters* that *minimize* di *error function*. To solve this problem, we calculate the *gradient* of the *error surface*, that's a *vector* that points towards the direction of steepest descent from the current point. Therefore, if we move along it a "short" distance, we will decrease the *error*, so that a sequence of such moves (slowing as we approach the bottom) will eventually find a *minimum* of some sort. Obviously, the focal point is to decide how large the steps should be. The step size can be chosen proportional to the slope (so that the algorithms settles down in a *minimum*) according to a constant called the *learning rate*. The *backpropagation* algorithm consists of two phases: the *forward phase*, during which the *activations* are propagated from the *input* to the *output layer*, and the *backward phase*, where the *error* between the *network output* and the *desired output* is propagated backwards, in order to modify the *weights* and *bias* values.

1.3.1.1 The Forward Phase

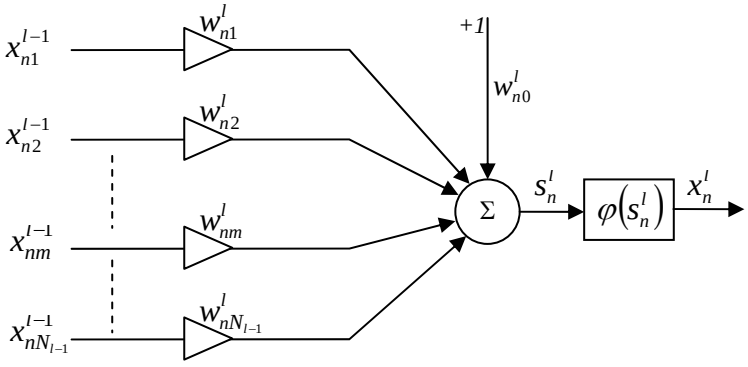


Fig. 1.22. The artificial neuron

In Fig. 1.22, the n^{th} *neuron* of the l^{th} *layer* of a *multilayer perceptron* is shown. In the *forward phase*, starting from the *input layer*, we need to calculate the output of all the *neurons* of each *layers*, up to the *output layer*. The output given by the *neurons* of the *output layer* will be the *actual output* of the *network*. So, the *forward phase* is formalized by the following equations ($l = 1, \ldots, M$ e $n = 1, \ldots, N_l$) :

$$x_n^{(l)} = \varphi\left(s_n^{(l)}\right) \tag{1.8}$$

$$s_n^{(l)} = \sum_{m=0}^{N_{l-1}} w_{nm}^{(l)}\, x_m^{(l-1)} \tag{1.9}$$

Note that before the output of n^{th} *neuron* could be calculated, the output of all its foregoing *neurons*

(the *neurons* of the preceding *layer*) must be already known. Since *feedforward networks* do not contain cycles, there is an ordering of *neurons* from input to output that respects this condition.

1.3.1.2 The Backward Phase (backpropagation of the error)

We are training an *FFNN* by *gradient descent*, based on some *training data* consisting of pairs $(\mathbf{x}^{(0)}, \mathbf{d})$. The vector of components $x_n^{(0)}\ (n=1,\ldots,N_0)$ is the *input pattern* (vector of *features*) used to feed the *network*, while the vector of components $d_n\ (n=1,\ldots,N_M)$ is the corresponding *target* (*desired output*). The overall *gradient* with respect to the entire *training set* is just the sum of the *gradients* for each *pattern*; in what follows we will therefore describe how to compute the *gradient* for just a single *training pattern*. Let's give some definitions ($l=1,\ldots,M$, $n=1,\ldots,N_l$, $m=1,\ldots,N_{l-1}$):

- *error signal*: $\delta_n^{(l)} = -\frac{\partial E}{\partial s_n^{(l)}}$ (1.10)
- (negative) *gradient* with respect to *weight* $w_{nm}^{(l)}$: $\Delta w_{nm}^{(l)} = -\mu \frac{\partial E}{\partial w_{nm}^{(l)}}$ (1.11)

where μ is the *learning rate*. Now, applying the *chain rule* to the *gradient*, we obtain:

$$\Delta w_{nm}^{(l)} = -\mu \frac{\partial E}{\partial s_n^{(l)}} \frac{\partial s_n^{(l)}}{\partial w_{nm}^{(l)}} \tag{1.12}$$

The first factor is the *error* of the n^{th} *neuron n* inthe l^{th} *layer*, while the second is:

$$\frac{\partial s_n^{(l)}}{\partial w_{nm}^{(l)}} = \frac{\partial}{\partial w_{nm}^{(l)}} \sum_{h=0}^{N_{l-1}} w_{nh}^{(l)}\, x_h^{(l-1)} = \sum_{h=0}^{N_{l-1}} \frac{\partial}{\partial w_{nm}^{(l)}} \left[w_{nh}^{(l)}\, x_h^{(l-1)} \right] = x_m^{(l-1)} \tag{1.13}$$

where $x_m^{(l-1)}$ is the output of the generic *neuron* of the *layer* that comes before. Putting all together, we get:

$$\Delta w_{nm}^{(l)} = \mu \delta_n^{(l)} x_m^{(l-1)} \tag{1.14}$$

We decide to adopt as *error function* (also called *cost function* or *objective function*) the *mean-square error*, as a function of the *parameters* of the *network*:

$$E = \frac{1}{2} \sum_{n=1}^{N_M} \left(d_n - x_n^{(M)} \right)^2 \tag{1.15}$$

The *error* for the generic *output neuron* $x_n^{(M)}$ is simply:

$$\delta_n^{(M)} = d_n - x_n^{(M)} \tag{1.16}$$

As regards the *hidden units*, we must propagate the *error* back, starting from the *output neurons* (hence the name of the algorithm). Again, using the *chain rule*, we can expand the *error* of a hidden unit in terms of the *errors* relative to the *neurons* of the *layer* that follows. The *output error* can be

considered as a function of $s_n^{(l)}$, through the dependence of $s_n^{(l)}$ on the *activations* $s_q^{(l+1)}$ of the *layer* that follows ($q = 1,\ldots,n,\ldots,N_{l+1}$), as shown in Fig. 1.23. In other words, it holds:

$$E = f\left(s_1^{(l+1)},\ldots,s_q^{(l+1)},\ldots,s_{N_{l+1}}^{(l+1)}\right) \tag{1.17}$$

$$s_q^{(l+1)} = f\left(s_n^{(l)}\right) \qquad q = 1,\ldots,n,\ldots,N_{l+1} \tag{1.18}$$

Therefore, the differential of *E* is equal to :

$$dE = \frac{\partial E}{\partial s_1^{(l+1)}} ds_1^{(l+1)} + \cdots + \frac{\partial E}{\partial s_{N_{l+1}}^{(l+1)}} ds_{N_{l+1}}^{(l+1)} = \sum_{q=1}^{N_{l+1}} \frac{\partial E}{\partial s_q^{(l+1)}} ds_q^{(l+1)} \tag{1.19}$$

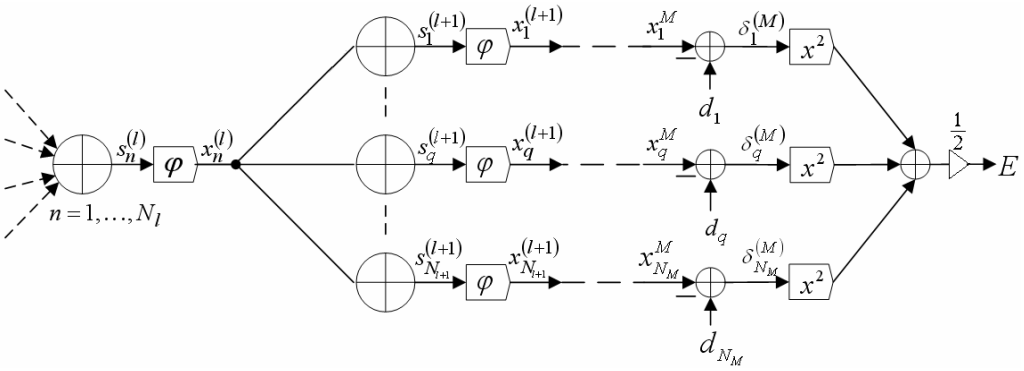


Fig. 1.23. The chain rule for a hidden layer

Anyway, thanks to the dependence of every $s_q^{(l+1)}$ on $s_n^{(l)}$, we can divide everything by the differential of $s_n^{(l)}$, and obtain:

$$\delta_n^{(l)} = -\frac{\partial E}{\partial s_n^{(l)}} = -\sum_{q=1}^{N_{l+1}} \frac{\partial E}{\partial s_q^{(l+1)}} \frac{\partial s_q^{(l+1)}}{\partial s_n^{(l)}} = \sum_{q=1}^{N_{l+1}} \delta_q^{(l+1)} \frac{\partial s_q^{(l+1)}}{\partial s_n^{(l)}} \tag{1.20}$$

Moreover, we can say that:

$$\delta_n^{(l)} = \sum_{q=1}^{N_{l+1}} \delta_q^{(l+1)} \frac{\partial s_q^{(l+1)}}{\partial x_n^{(l)}} \frac{\partial x_n^{(l)}}{\partial s_n^{(l)}} \tag{1.21}$$

where:

$$\begin{aligned} \frac{\partial s_q^{(l+1)}}{\partial x_n^{(l)}} &= \frac{\partial}{\partial x_n^{(l)}} \sum_{h=0}^{N_l} w_{qh}^{(l+1)} x_h^{(l)} = w_{qn}^{(l+1)} \\ \frac{\partial x_n^{(l)}}{\partial s_n^{(l)}} &= \frac{\partial \varphi\left(s_n^{(l)}\right)}{\partial s_n^{(l)}} = \varphi'\left(s_n^{(l)}\right) \end{aligned} \tag{1.22}$$

Putting all the pieces together we get:

$$\delta_n^{(l)} = \varphi'\left(s_n^{(l)}\right) \sum_{q=1}^{N_{l+1}} \delta_q^{(l+1)} w_{qn}^{(l+1)} \tag{1.23}$$

For *hidden units* that use the *tanh activation function*, we can make use of the special identity:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x) \tag{1.24}$$

giving us:

$$\varphi'\left(s_n^{(l)}\right) = 1 - \varphi^2\left(s_n^{(l)}\right) = 1 - \left[x_n^{(l)}\right]^2 \tag{1.25}$$

Note that, in order to calculate the *error* for *neuron n*, we must first know the *error* of all its *posterior neurons*. Again, as long as there are no cycles in the *network*, there is an ordering of *neurons* from the output back to the input that respects this condition. For example, we can simply use the reverse of the order in which *activity* was propagated forward.

For *fully connected FFNNs* – that is, each *neuron* in a *layer* connects to every *neuron* in the next layer – it is possible to write the *backpropagation* algorithm in matrix notation, where *bias weights*, *net inputs*, *activations*, and *error signals* for all *neurons* in a *layer* are combined into vectors, while all the *non-bias weights* from one *layer* to the next form a matrix **W**. *Layers* are numbered from 0 (the *input layer*) to *M* (the *output layer*). The *backpropagation* algorithm then looks as follows:

0. Choose some (random) initial values for the *network parameters*.
1. Initialize the *input layer*: $\mathbf{x}^{(0)}$
2. Propagate the output of each *layer* forward: for *l* = 1, 2, ..., *M*

   $$\mathbf{x}^{(l)} = \varphi\left(\mathbf{W}^{(l)} \cdot \mathbf{x}^{(l-1)}\right)$$

   The *input bias* is embedded in $\mathbf{x}^{(l-1)}$ and the *bias weight* in $\mathbf{W}^{(l)}$.
3. Calculate the *error* in the *output layer*:

   $$\boldsymbol{\delta}^{(M)} = \mathbf{d} - \mathbf{x}^{(M)}$$

4. Back-propagate the *error*, for *l = M-1, M-2, ..., 1*:

   $$\boldsymbol{\delta}^{(l)} = \varphi'\left(\mathbf{s}^{(l)}\right)\left(\left[\mathbf{W}^{(l+1)}\right]^T \cdot \boldsymbol{\delta}^{(l+1)}\right)$$

   where T is the matrix transposition operator.
5. Update the *weights*:

   $$\Delta\mathbf{W}^{(l)} = \mu\boldsymbol{\delta}^{(l)}\left[\mathbf{x}^{(l-1)}\right]^T$$

6. Come back to step 1., until some stop condition is satisfied (for example, the *error* goes under some acceptable threshold).

As a matter of fact, the real *error function* to be miminized is given by the sum of the *mean-square errors* calculated an all the *examples* in the *training set*:

$$E = \sum_p E^p, \qquad E^p = \frac{1}{2}\sum_{n=1}^{N_M}\left(d_n - x_n^{(M)}\right)^2 \tag{1.26}$$

Since differentiation and summation are interchangeable, we can likewise split the *gradient* into separate components for each *example*:

$$\frac{\partial E}{\partial w_{nm}} = \frac{\partial}{\partial w_{nm}} \sum_p E^p = \sum_p \frac{\partial E^p}{\partial w_{nm}} \tag{1.27}$$

In the calculations made before, the *gradient* has been computed for only one *example*, omitting the superscript *p* in order to make the notation easier to follow.

1.3.1.3 The Learning Rate

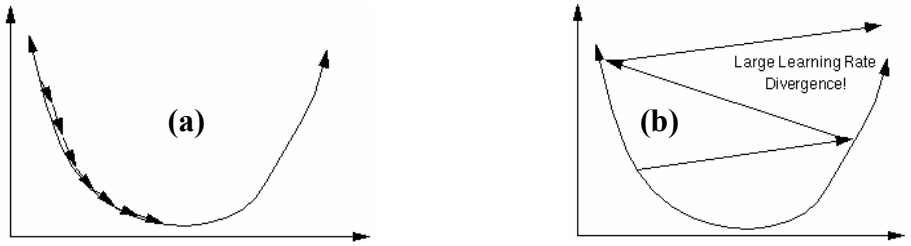


Fig. 1.24. (a) Small learning rate, slow convergence; (b) Large learning rate, divergence

The learning rate μ is fundamental, since it determines the amount we change the *weights w* at each step. If μ is too small, the algorithm may go in the correct direction, but will take a long time to converge, that is with a large number of iterations (Fig. 1.24 (a)). Conversely, if μ is too large, we may end up bouncing around the error surface out of control – the algorithm diverges (Fig. 1.24 (b)). This usually ends with an overflow error in the computer's floating-point arithmetic.

In other words, large steps may converge more quickly, but may also overstep the solution or (if the error surface is very eccentric) go off in the wrong direction. An example is that of an *error surface* that makes the algorithm progressing very slowly along a steep, narrow, valley, bouncing from one side across to the other. The correct setting for the *learning rate* is application-dependent, and is typically chosen by experiment; it may also be time-varying, getting smaller as the algorithm progresses. Otherwise, we can include a *momentum* term, so that, if several steps are taken in the same direction, the algorithm "picks up speed", acquiring the ability to (sometimes) escape *local minima*, and also to move rapidly over *flat spots* and *plateaus*.

1.3.2 Batch learning and On-line learning

In *batch learning*, the *FFNN* is *trained* according to an *epoch* based approach. An *epoch* is a single pass through the entire *training set*. On each *epoch*, the *training examples* are each submitted in turn to the *network*, and *target* and *actual outputs* compared and the *error* calculated. This *error*, together with the *error surface gradient*, is used to adjust the *weights*, and then the process repeats, that is to say, the algorithm progresses iteratively through a number of these *epochs*. The *initial network configuration* is random, and *training* stops when a given number of *epochs* elapses, or

when the *error* reaches an acceptable level, or when the *error* stops improving (*stopping conditions*). Therefore, in order to find the *gradient* for the entire *training set*, we sum the contribution given by $\partial E / \partial w_{nm}$ over all the *data points*. So, in *batch learning*, we accumulate the *gradient* contributions for all *data points* in the *training set* before updating the *weights*.
In *on-line learning*, the *weights* are updated immediately, after seeing each *data point*. The *stopping conditions* can be the same of the *batch learning* case. Since the *gradient* for a single *example* can be considered a noisy approximation of the overall *gradient*, this is also called *stochastic* (*noisy*) *gradient descent*. *On-line learning* has a number of advantages:

- it is often much faster, especially when the *training set* is redundant (contains many similar *examples*)
- it can be used when there is no fixed *training set* (new data keeps coming in)
- it is better for tracking non stationary environments (where the best model gradually changes over time)
- the noise in the *gradient* can help in avoiding *local minima* (a problem for *gradient descent* in *non linear models*)

The drawback is that many powerful *optimization* techniques are *batch* methods that cannot be used *on-line*. A compromise between *batch* and *on-line learning* could consist in updating the *weights* after every *n* data points, where *n* is greater than 1 but smaller than the *training set* size.

1.3.3 Over-learning vs. Generalization Power

One major problem with the approaches outlined above is that they don't actually *minimize* the *error* that we are really interested in: the *expected error* the *network* makes on *new examples* not included in the *training set*. In fact, the most desirable property of a *network* is its ability to *generalize* to new *examples*. In reality, the *network* is trained to *minimize* the *error* on the *training set*, and since we cannot have a perfect and infinitely large *training set*, this is not the same thing as minimizing the *error* on the real *error surface*, that is the *error surface* of the underlying and unknown model. The most important manifestation of this limitation is the problem of *over-learning*, or *over-fitting*. We illustrate this concept by means of the *polynomial curve fitting* problem. A *polynomial* is an equation with terms containing only constants and powers of the variables, as for example $y = 2x + 3$ or $y = 3x^2 + 4x + 1$. Different *polynomials* have different shapes, with larger powers (and therefore larger numbers of terms) and, as a consequence, a more precise interpolation power. Given a *data set*, we may want to fit a *polynomial curve* (i.e., a *model*)

to explain the data. The *data set* is probably noisy, so we don't necessarily expect the best *model* to pass exactly through all the points. Anyway, a *low-order polynomial* may not be sufficiently flexible to fit close to the points, whereas a high-order polynomial is actually too flexible, fitting the data exactly by adopting a highly eccentric shape that is actually unrelated to the underlying function (Fig. 1.25).

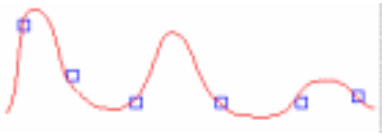


Fig. 1.25. Over-fitting polynomial

Neural networks show the same problem. A *network* with a lot of *weights* models a more complex function, and is therefore prone to *over-fitting*, even if it will eventually achieve a lower *error* on the *training set*. On the contrary, a *network* with less *weights* could not be sufficiently powerful to model the underlying function. So, a problem arises as for the selection of the right *complexity* of the *network*. This problem can be tackled by the use of a *selection set*, that's arranged by setting aside some *examples* that won't be uses for *training* with the *backpropagation* algorithm. The *examples* in the *selection set* are used to keep an independent check on the progress of the algorithm. Generally, the initial performance of the *network* on *training* and *selection sets* is the approximately the same (otherwise, it means that the distribution of the *examples* between the two sets is probably biased). As *training* progresses, the *training error* naturally drops, and provided *training* is really minimizing the *true error function*, the *selection error* must drop too. However, if the *selection error* stops dropping, or indeed starts to rise, this indicates that the *network* is starting to *over-fit* the data, and *training* should cease. In case *over-fitting* occurs during the *training* process, we are in the presence of what we call *over-learning*. In this case, it is usually advisable to decrease the number of *hidden units* and/or *hidden layers*, as the *network* is over-powerful for the problem at hand. In contrast, if the *network* is not sufficiently powerful to model the underlying function, neither *training* nor *selection errors* will drop to a satisfactory level. Unfortunately, this strategy implies experimenting with a large number of different *networks*, probably training each one a number of times (to avoid *local minima*), and observing individual performances. Moreover, the *selection set* plays a key role in selecting the model, which means that it is actually part of the *training* process. Its reliability needs always to be proved (with sufficient experiments, you may just hit upon a lucky *network* that happens to perform well on the selection set), to add confidence in the performance of the final model. This can be done (at least where the volume of *training data* allows it), by reserving a third set of *examples – the test set*. The final model is tested with the *test set* data, to ensure that the results on the *selection* and *training set* are real, and not artefacts of the *training* process. Of course, to fulfil this role properly the *test set* should be used only once – if it is in turn

used to adjust and reiterate the *training* process, it effectively becomes selection data! This division into multiple subsets is very unfortunate, given that we usually have less data than we would ideally desire even for a single subset. We can get around this problem by *resampling*. Experiments can be conducted using different divisions of the available data into *training*, *selection*, and *test sets*. There are a number of approaches to this division, including *random resampling*, *cross-validation*, and *bootstrap*. If we try to find the best configuration of a *neural network*, based upon a number of experiments with different subsets of *examples*, the results will be much more reliable. We can then either use those experiments solely to guide the decision as to which *network* types to use, and train such *networks* from scratch with new *examples* (this removes any *sampling* bias); or, we can retain the best *networks* found during the *sampling* process, but average their results in an ensemble, which at least mitigates the *sampling* bias. To summarize, *network design* (once the input and output variables have been selected) follows a number of stages:

- Select an *initial configuration* (typically, one *hidden layer* with the number of *hidden units* set to half the sum of the number of *input* and *output units*).
- Iteratively conduct a number of experiments with each *configuration*, retaining the best *network* (in terms of *selection error*) found. A number of experiments are required with each *configuration* to avoid being fooled if *training* locates a local minimum, and it is also best to *resample*.
- On each experiment, if *under-learning* occurs (the *network* doesn't achieve an acceptable performance level) try adding more *neurons* to the *hidden layer*(*s*). If this doesn't help, try adding an extra *hidden layer*.
- If *over-learning* occurs (*selection error* starts to rise) try removing *hidden units* (and possibly *layers*).
- Once you have experimentally determined an effective *configuration* for your *networks*, *resample* and generate new *networks* with that *configuration*.

1.4 FFNNs with Adaptive Spline Activation Functions

As regards both hardware and software implementations of *NNs*, their *complexity*, both *structural* (number of *interconnections*) and *computational* (number of multiplications), is a bottleneck for many real-time applications. However, under certain regularity conditions, the *representation capabilities* of *NNs* depend on the number of *free parameters*, whatever the structure of the network. Hence, *adaptable activation functions* can be introduced in order to add *free parameters* and as a consequence to reduce the number of *interconnections* as well as the overall *network complexity*. The classical *neuron model*, proposed by McCulloch and Pitts in 1943, is composed of a *linear combiner* followed by a nonlinear function (*activation function*) with hard-limiting characteristics. In order to develop *gradient based learning* algorithms, non linear differentiable *activation functions* have been introduced. It is well known that the *network* behaviour, as that shown by the *multilayer perceptron* (*MLP*), greatly depends on the shape of these *activation functions*. *Sigmoidal* functions are the most common; however, other kinds of functions, at times depending on some free parameters, are used to improve the *NN*'s representation capabilities. For example, an *MLP* with *adaptive-slope sigmoidal activation function*, or with *adaptive generalized hyperbolic tangent* function (two free parameters: slope, saturation level), or with *adaptive polynomial activation functions* have been proposed. Other more complex approaches have been proposed. Instead, our proposal is based on the so called *adaptive spline activation functions*. The *adaptive spline-based FFNN*'s (*ASNN*'s) are designed around a new kind of *neuron*, called *generalized sigmoidal neuron* (*GSN*), containing an *adaptive parametric spline activation function*. The basic *network* scheme is therefore very similar to classical *FFNN* structures, but with improved *nonlinear adaptive activation functions*. These functions have several interesting features: they 1) are easy to adapt; 2) retain the squashing property of the *sigmoid*; 3) have the necessary smoothing characteristics; and 4) are easy to implement both in hardware and in software. The *FFNNs* built around such *neurons* usually have a smaller *structural complexity*, but maintaining at the same time good *generalization capabilities*. Among the available *spline curves*, we will adopt the *cubic Catmull–Rom spline* (*CR*), suitable for implementing an *adaptive activation function* with some interesting features, due to its *local interpolation* (the adjustment of a single parameter doesn't change the overall shape of the function) and regularization characteristics. In fact, by means of a *LUT*, we are able to make a *local adaptation* process, that is, at every time step, the changing of the shape of the function concerns only a very small number of close points of the *LUT* itself (most of the adaptations made in the previous steps are preserved). After a certain number of steps, the shape of the function will reflect the information inside the *data set*. The *LUT* entries are the samples of the curve to be adapted. To reduce the number of the *free parameters*, we have to limit the length of

the LUT, as well as to adopt the right interpolation scheme. Above all, we will have to guarantee the continuity of the first derivative of the *activation function*.

Let a *LUT* be defined as a $N+1$ point array, $\{\boldsymbol{Q}_0, \boldsymbol{Q}_1, \ldots, \boldsymbol{Q}_N\}$, where each point is represented by the x and y coordinates $\boldsymbol{Q}_i = \begin{bmatrix} q_{x,i} & q_{y,i} \end{bmatrix}^T$. The *LUT* points are also constrained to be such that their projections over the x axis are always ordered in ascending order:

$$q_{x,0} < q_{x,1} < \ldots < q_{x,N} \tag{1.28}$$

to avoid loops (reverse ordering of abscissas) or multiple output values for a single abscissa (overlapping abscissas). This constraint is necessary, since it avoids representing non-injective functions. A *planar spline curve* is a two-dimensional array

$$\boldsymbol{F}(u) = \begin{bmatrix} \boldsymbol{F}_x(u) & \boldsymbol{F}_y(u) \end{bmatrix}^T \tag{1.29}$$

whose components are *piecewise polynomial univariate functions* of the same degree and of the same scalar variable u. Its mathematical formulation ensures both continuity and the existence of derivatives, along the curve as well as in correspondence to the *joining points* between the *curve spans*. Given a *LUT* as the one defined above, a general *spline* expression for that curve would be:

$$\mathbf{F}(u) = \mathop{\mathbf{C}}_{i=0}^{N-3} \mathbf{F}_i(u) = \mathop{\mathbf{C}}_{i=0}^{N-3} \begin{bmatrix} F_{x,i}(u) & F_{y,i}(u) \end{bmatrix}^T \tag{1.30}$$

where $\mathbf{C}$ is the *concatenation operator* and $\boldsymbol{F}_i(u)$ is the i^{th} *curve span* (or *patch*). The indices of the operator in (1.30) are valid only for *cubic polynomials*: the choice of using *cubic polynomials* was made because of the tradeoff between the requested properties and the computational complexity. The parameter u has the property of being *local* and its domain is $0 \le u \le 1$ for every *curve span*. Hence, given a value for the *abscissa global parameter* (x), we can define a unique mapping that allows us to calculate the *local parameter* (u), as well as the proper curve span (i). In this way, we can represent any point lying on the *spline curve* $\mathbf{F}(u)$ as a point belonging to the single i^{th} *curve span* $\mathbf{F}_i(u)$. It has been shown that the i^{th} *curve span* can be described as follows:

$$\mathbf{F}_i(u) = \begin{bmatrix} F_{x,i}(u) & F_{y,i}(u) \end{bmatrix}^T = \sum_{j=0}^{3} \mathbf{Q}_{i+j} C_j(u) \tag{1.31}$$

where $C_j(u)$ are the *spline polynomials*. As in (1.29), each coordinate is described by a univariate function, namely a *cubic polynomial* of the variable u.

1.4.1 The GS Neuron

Let s be the *linear combiner* output of a *neuron*, usually called "*activation*"; we have to make the dependence between s and $\boldsymbol{F}_i(u)$ explicit. There is a direct link between s and one of the two components (a *cubic polynomial function*), namely:

$$s = F_{x,i}(u) \tag{1.32}$$

Equation (1.32) plays a key role in finding a suitable *activation function* architecture, as it is the main bottleneck of the overall structure: in fact, it is the only link between the *linear combiner* output and *nonlinear neuron* output $y = F_{y,i}(u)$, which is unambiguously defined each time the following values have been fixed:

- $i \in [0, N]$ - which represents the *LUT* index
- $u \in [0,1]$ - which represents the offset of the i^{th} *curve span*

Let *y* be the *neuron* output, given the *activation s*; in order to calculate the *y* of a single *neuron*, we have to introduce a two-step procedure:

- calculate *u* and *i* from *s*, by inverting $s = F_{x,i}(u)$
- substitute these values of *u* and *i* in $y = F_{y,i}(u)$

Equation (1.31) resolves the link between the curve and the *sample points* $\mathbf{Q}_i$, which in literature are called *control points*, because they control the shape of the curve. The *cubic polynomial functions* $C_j(u)$, called *spline basis functions* or *blending functions*, characterize the way the curve moves along the path made up by the *control points*. The curve can *interpolate* or just *approximate* its *control points*, depending on which *blending function set* we rely on. In our case, we prefer an *interpolation scheme*, due to its local characteristics, which avoids the oscillatory behavior of the global adaptation of the *approximation scheme*. Actually, there are few splines that interpolate their control points; one that also has a very low computational overhead is the socalled *CR cubic spline basis*, which is described by the following *polynomials*:

$$\begin{aligned} C_0(u) &= \frac{1}{2}(-u^3 + 2u^2 - u) \\ C_1(u) &= \frac{1}{2}(3u^3 - 5u^2 + 2) \\ C_2(u) &= \frac{1}{2}(-3u^3 + 4u^2 + u) \\ C_3(u) &= \frac{1}{2}(u^3 - u^2). \end{aligned} \tag{1.33}$$

A quick inspection of (1.33) shows that all the multiplications, except for the powers of the parameter *u*, are by integer coefficients and that they should be easily implemented in hardware (just one or two shifts or sum-and-shift operations). This characteristics of the *CR spline* is important, as it simplifies the structure of the nonlinear block. A common expression of (1.31), is the matricial notation (1.34), which explicits the actual array structure of a *curve span*: a *parameter vector*, a *basis matrix* and a *control point vector*, combined using row by column multiplications. Fig. 1.26 shows the graphical representation of the single *CR* i^{th} *curve span*.

$$F_i(u) = [u^3 \quad u^2 \quad u \quad 1]\frac{1}{2}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}\begin{bmatrix} Q_i \\ Q_{i+1} \\ Q_{i+2} \\ Q_{i+3} \end{bmatrix} \tag{1.34}$$

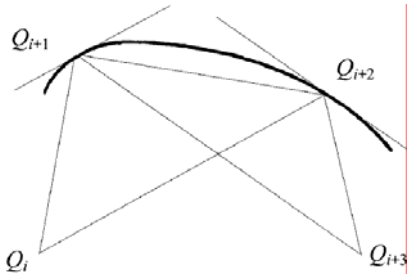


Fig. 1.26. Catmull–Rom spline curve span.

Deriving (1.34) with respect to *u*, we have that:

$$\frac{\partial F_i(u)}{\partial u} = [3u^2 \;\; 2u \;\; 1]\frac{1}{2}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} Q_i \\ Q_{i+1} \\ Q_{i+2} \\ Q_{i+3} \end{bmatrix} \tag{1.35}$$

where for $u = 0$ we have $\frac{\partial F_i(0)}{\partial u} = \frac{1}{2}(-Q_i + Q_{i+2})$ and for $u = 1$ we have $\frac{\partial F_i(1)}{\partial u} = \frac{1}{2}(-Q_{i+1} + Q_{i+3})$. This means that the curve tangency lines, at the points Q_{i+1} and Q_{i+2}, are parallel to the straight lines passing through the points Q_i and Q_{i+2} and through the points Q_{i+1} and Q_{i+3}, respectively (see Fig. 1.26). The continuity of the derivative is also useful for the definition of *gradient-based learning* algorithms. The matricial expression of $F_i(u)$ (1.35) can be expressed as a pair of *cubic polynomials* (*a*, *b*, *c* and *d* are appropriate constants):

$$\begin{aligned} F_{x,i}(u) &= a_{x,i}u^3 + b_{x,i}u^2 + c_{x,i}u + d_{x,i} \\ F_{y,i}(u) &= a_{y,i}u^3 + b_{y,i}u^2 + c_{y,i}u + d_{y,i} \end{aligned} \tag{1.36}$$

Fig. 1.27 shows the polynomials $F_{x,i}(u)$ and $F_{y,i}(u)$ used for implementing the *activation function* $y = f(s)$. In order to find the values of parameters *u* and *i*, which are in turn necessary to generate the *neuron*'s output, the inversion of

$$s = F_{x,i}(u) = a_{x,i}u^3 + b_{x,i}u^2 + c_{x,i}u + d_{x,i} \tag{1.37}$$

is needed. We could exploit the formulas for the solution of third-order equations (algebraically or iteratively), but a serious overhead in the calculations would be introduced. Moreover, this approach gives no valid answer to the ordering problem that affects the *abscissa control points*. There is, however, a regularization property common to most of the *polynomial spline basis sets* (*CR* included), called *variation diminishing property*, which ensures the absence of unwanted

oscillations of the curve between two consecutive *control points*, as well as the exact representation of linear segments.

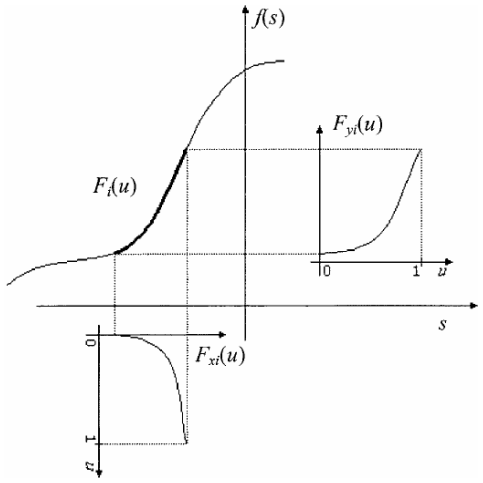


Fig. 1.27. Single curve span decomposed in its two components.

This second statement is particularly important, as it suggests a simple possible solution to the inversion problem (1.32): if we *uniformly sample* the abscissas along the *x*-axis, then the *cubic polynomial* $F_{x,i}(u)$ becomes a *first degree polynomial*. This approach ensures both a fast computation of the *local parameters u* and *i*, and the monotone ordering of the abscissas, in the case when they are kept fixed. In this case, let us consider a *fixed sample step* $\Delta x = q_{x,i+1} - q_{x,i}$. After substituting the proper values of the abscissa control points in (1.34), the function $F_{x,i}(u)$ has the following form:

$$F_{x,i}(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \frac{1}{2} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} q_{x,i} \\ q_{x,i} + \Delta x \\ q_{x,i} + 2\Delta x \\ q_{x,i} + 3\Delta x \end{bmatrix} \tag{1.38}$$

The solution to this equation is a linear mapping:

$$s = F_{x,i}(u) = u\Delta x + q_{x,i} + \Delta x = u\Delta x + q_{x,i+1} \tag{1.39}$$

in which all the parameters are known. Expression (1.39) is very fast to compute and it leads to a negligible computational overhead. Moreover, it is not necessary to store the $q_{x,i}$ table, as these values can now be determined algorithmically (see later). The (1.36) can now be written as:

$$\begin{aligned} F_{x,i}(u) &= s = c_{x,i}u + d_{x,i} \\ F_{y,i}(u) &= y = a_{y,i}u^3 + b_{y,i}u^2 + c_{y,i}u + d_{y,i} \end{aligned} \tag{1.40}$$

Operating the inversion $u = F_{x,i}^{-1}(s)$, the output *y* can be computed as a *cubic polynomial* which expresses a direct relation between the input *s* and the output *y* of the *CR*-based nonlinear block:

$$s = c_{x,i}u + d_{x,i} \quad \Rightarrow \quad u = F_{x,i}^{-1}(s) = \frac{s - d_{x,i}}{c_{x,i}}$$

$$\Downarrow$$

$$y = F_{y,i}(s) = \frac{a_{y,i}}{c_{x,i}^3}s^3 + \left(\frac{b_{y,i}}{c_{x,i}^2} - \frac{3a_{y,i}d_{x,i}}{c_{x,i}^3}\right)s^2 + \left(\frac{3a_{y,i}^2 d_{x,i}^2}{c_{x,i}^3} - \frac{2b_{y,i}d_{x,i}}{c_{x,i}^2} + \frac{c_{y,i}}{c_{x,i}}\right)s + \left(\frac{b_{y,i}d_{x,i}^2}{c_{x,i}^2} - \frac{a_{y,i}d_{x,i}^3}{c_{x,i}^3} - \frac{c_{y,i}d_{x,i}}{c_{x,i}} + d_{y,i}\right) \tag{1.41}$$

The geometric tangent continuity of the *CR* curve demonstrated above is clearly valid here, but we are now dealing with *polynomial function spans*, and not *curve spans*: this leads us to the conclusion that the continuity of the first derivative of the single *function span* (1.41) is preserved along the complete function. The expression (1.41), although simple to understand, cannot be efficiently calculated in this form: it is for the sake of computational efficiency that we will use (1.34) for the actual computation of $F_{y,i}(u)$. The result found in (1.39) is then employed to improve the efficiency of the complete structure. Therefore, we constrain the *control point* abscissas to be equidistant and, most important, not adaptable. Moreover, always for the sake of efficiency, another constraint is imposed on the *control points*, forcing the *sampling interval* to be centred on the *x*-axis origin. It is then possible to represent the abscissa of each point of the *activation function* using two parameters (the span index *i* and the local parameter *u*), without storing the *control point* abscissas $q_{x,i}$. All we need to know is how many *control points* the curve has, and the sampling step Δx.

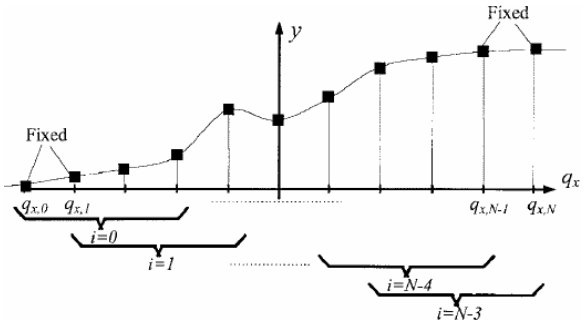


Fig. 1.28. Control points of the Catmull–Rom spline-based activation function with a fixed step Δx. The extreme points $q_{x;0}$, $q_{x;1}$, $q_{x;N-1}$ and $q_{x;N}$ are fixed.

An additional and important constraint is to force the *activation function* to be a *limiting function*, i.e., to be constant for $s \to \pm\infty$, while maintaining the ability to modify its shape inside some constant values. A practical implementation of this class of functions starts with the *logistic functions*:

$$f(s) = \frac{2c_1}{1 + e^{-c_2 s}} - c_3 \tag{1.42}$$

where c_1, c_2 and c_3 are suitable constant parameters (for example, $c_1 = 0.5$, $c_2 = 1$, and $c_3 = 0$ represent the *standard unsigned sigmoid*, while $c_1 = 1$, $c_2 = 1$, and $c_3 = 1$ represent the *signed*, or *bipolar, sigmoid*). Performing a uniform sampling of $f(s)$ with *N*+1 *control points* in an interval

centred on the *x*-axis origin and then fixing the first two and the last two control point abscissas, constant output values are obtained outside the sampling interval, while the *activation function* can be freely shaped between $q_{x,1}$ and $q_{x,N-1}$ (Fig. 1.28), starting from the *sigmoidal* shape as an initial condition.

1.4.2 Gradient-Based Learning for ASNN

Let's suppose we have an *M layer ASNN*, with N_l (l = 1, 2,…, M) *neurons* per *layer*. In what follows, the ordinate of the k^{th} *control point* will be symbolized by $q_{n,k}^{(l)}$ (previously $q_{y,n}$). The abscissas won't appear in the expressions, since we have assumed a uniformly sampled *x*-axis.

1.4.2.1 The FORWARD phase

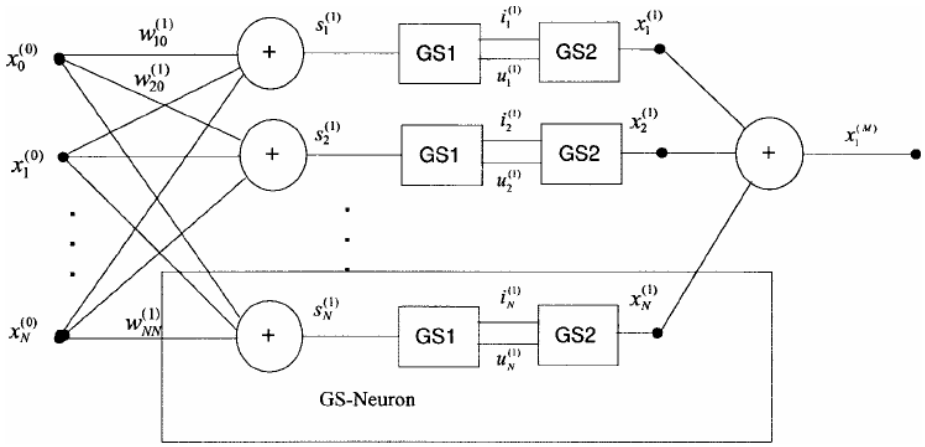


Fig. 1.29. An example of an adaptive spline neural network (ASNN).

The *inversion* operation $s = u\Delta x + q_{x,i+1}$ (1.39) can now be expressed by the following set of equations, that are intended to be evaluated in the proposed order ($n = 1,\ldots,N_l \quad l = 1,\ldots,M$):

$$z_n^{(l)} = \frac{s_n^{(l)}}{\Delta x} + \frac{N-2}{2} \qquad i_n^{(l)} = \lfloor z_n^{(l)} \rfloor \qquad u_n^{(l)} = z_n^{(l)} - \lfloor z_n^{(l)} \rfloor \tag{1.43}$$

where $\lfloor\ \rfloor$ is the floor operator, and $z_n^{(l)}$ is an internal dummy variable. As an implementation note, the second term in the second equation is needed to force $i_n^{(l)}$ to be always non negative. The last three equations solve the inversion problem: they are cheap in terms of computational demand, and provide the two parameters needed to calculate the output. This operation is performed as in (1.34), that now becomes

$$x_n^{(l)} = F_{n,i_n^{(l)}}^{(l)}\left(u_n^{(l)}\right) = \sum_{m=0}^{3} q_{n,\left(i_n^{(l)}+m\right)}^{(l)} C_{n,m}^{(l)}\left(u_n^{(l)}\right) \tag{1.44}$$

We call the two blocks expressed by (1.43) and (1.44) *GS1* and *GS2*, respectively (Fig. 1.29).

1.4.2.2 The BACKWARD Computation (Learning Phase)

When an iterative *learning* algorithm is employed, it can be noted that, at each iteration, all the *weights* are changed, whereas, for each *neuron*, only the four *control points* of the involved *curve span* are updated. This is a consequence of *locality* of the *spline interpolation scheme*. Let *t* be the iteration index, $d_k[t]$ the desired output, and $E_p[t] = \sum_{n=1}^{N_M} \left(d_n[t] - x_n^{(M)}[t]\right)^2$ the *instantaneous mean-square output error*, related to the p^{th} *learning pattern*; the *weights* and *control points* are adapted following the *anti-gradient* of the *error surface* as follows ($l = M,\ldots,1 \quad n = 1,\ldots,N_l$):

$$w_{nm}^{(l)}[t+1] = w_{nm}^{(l)}[t] - \mu_w \frac{\partial E_p[t]}{\partial w_{nm}^{(l)}[t]} \tag{1.45}$$

$$q_{n,\left(i_n^{(l)}+k\right)}^{(l)}[t+1] = q_{n,\left(i_n^{(l)}+k\right)}^{(l)}[t] - \mu_q \frac{\partial E_p[t]}{\partial q_{n,\left(i_n^{(l)}+k\right)}^{(l)}[t]} \qquad k = 0,\ldots,3 \tag{1.46}$$

where the parameters μ_w and μ_q represent the *learning rates* for the *weights* and for the *control points*, respectively. For each *layer*, following the *backpropagation* approach, we compute the expression:

$$e_n^{(l)}[t] = \begin{cases} d_n[t] - x_n^{(l)}[t] & l = M \\ \sum_{h=1}^{N_{l+1}} \delta_h^{(l+1)}[t] w_{hn}^{(l+1)}[t] & l = M-1,\ldots,1 \end{cases} \tag{1.47}$$

where the δ's take the values:

$$\delta_n^{(l)}[t] = e_n^{(l)}[t] \left(\frac{\partial F_{n,i_n^{(l)}}^{(l)}\left(s_n^{(l)}[t]\right)}{\partial s_n^{(l)}[t]} \right) \tag{1.48}$$

so that using the *chain rule* yields

$$\frac{\partial F_{n,i_n^{(l)}}^{(l)}\left(s_n^{(l)}[t]\right)}{\partial s_n^{(l)}[t]} = \frac{\partial F_{n,i_n^{(l)}}^{(l)}\left(s_n^{(l)}[t]\right)}{\partial u_n^{(l)}[t]} \frac{\partial u_n^{(l)}[t]}{\partial s_n^{(l)}[t]} = \frac{\partial F_{n,i_n^{(l)}}^{(l)}\left(s_n^{(l)}[t]\right)}{\partial u_n^{(l)}[t]} \frac{1}{\Delta x} \tag{1.49}$$

Since the external control points $q_{k,0}^{(l)}, q_{k,1}^{(l)}, q_{k,N-1}^{(l)}, q_{k,N}^{(l)}$ are a priori fixed, the following constant values of the derivatives are used for them:

$$\begin{aligned} \frac{\partial F_{n,i_n^{(l)}}^{(l)}\left(s_n^{(l)}[t]\right)}{\partial s_n^{(l)}[t]} &= \frac{q_{n,1}^{(l)} - q_{n,0}^{(l)}}{\Delta x} & s_n^{(l)} < q_{n,1}^{(l)} \\ \frac{\partial F_{n,i_n^{(l)}}^{(l)}\left(s_n^{(l)}[t]\right)}{\partial s_n^{(l)}[t]} &= \frac{q_{n,N}^{(l)} - q_{n,N-1}^{(l)}}{\Delta x} & s_n^{(l)} > q_{n,N-1}^{(l)} \end{aligned} \tag{1.50}$$

As for the other *control points*, we must compute the following derivatives:

$$\frac{\partial E_p[t]}{\partial q^{(l)}_{n,\left(i_n^{(l)}+k\right)}[t]} = \frac{\partial E_p[t]}{\partial x_n^{(l)}[t]} \frac{\partial x_n^{(l)}[t]}{\partial q^{(l)}_{n,\left(i_n^{(l)}+k\right)}[t]} \tag{1.51}$$

$$\frac{\partial x_n^{(l)}[t]}{\partial q^{(l)}_{n,\left(i_n^{(l)}+k\right)}[t]} = \frac{\partial F^{(l)}_{n,\left(i_n^{(l)}+k\right)}\left(s_n^{(l)}[t]\right)}{\partial q^{(l)}_{n,\left(i_n^{(l)}+k\right)}[t]} = C_{n,k}^{(l)}\left(u_n^{(l)}[t]\right) \qquad k = 0,\ldots,3 \tag{1.52}$$

Considering the *weights* $w_{n0}^{(l)}$ as offset values, (1.45) and (1.46) can be finally rewritten as ($l = M,\ldots,1 \quad n = 1,\ldots,N_l$):

$$w_{nm}^{(l)}[t+1] = w_{nm}^{(l)}[t] - \mu_w \delta_n^{(l)}[t] \begin{cases} x_m^{(l-1)} & m \neq 0 \\ 1 & m = 0 \end{cases} \tag{1.53}$$

$$q^{(l)}_{n,\left(i_n^{(l)}+k\right)}[t+1] = q^{(l)}_{n,\left(i_n^{(l)}+k\right)}[t] - \mu_q e_n^{(l)}[t] C_{n,m}^{(l)}\left(u_n^{(l)}[t]\right) \qquad k = 0,\ldots,3 \tag{1.54}$$

This algorithm reduces to the *standard backpropagation* rule, when the *control points* are not adapted (for example by setting the *learning rate* μ_q to zero).

1.5 Dynamic Locally Recurrent Neural Networks

We are here interested in *neural network* architectures that are able to *learn temporal features* in data for *time series prediction*. In other words, we intend to deal with the problem of *Time Series Processing*: field of statistics regarding the analysis of data characterized by both spatial and temporal dimensions. The *FFNN* is frequently used in *time series prediction*. *FFNN*, however, has the major limitation that it can only learn an input – output mapping which is static. Thus it can be used to perform a *nonlinear prediction* of a *stationary time series*. A *time series* is said to be *stationary*, when its statistics do not change with time. In many real world problems, however, the time when certain *feature* in the data appears contains important information. More specifically, the interpretation of a *feature* in data may depend strongly on the earlier *features* and the time they appeared. A common example of this phenomenon is speech. A good solution is to let time have an effect on the *networks* response. This can be achieved when the *network* has *dynamic properties* such that it will respond to *temporal sequences*. *Dynamic recurrent neural networks* have shown themselves to be really useful for *temporal processing*, particularly for *digital signal processing* (*DSP*) and *temporal pattern recognition*. Two main methods exist to provide a *static neural network* with *dynamic behaviour*: the insertion of a *buffer* somewhere in the *network*, i.e., implementing an explicit memory of the past inputs, or the use of *feedback*. In both approaches, an arbitrary input $x[t]$ may influence a future output $y[t+h]$, so that $\partial y[t+h]/\partial x[t]$ is not equal to zero for some *h*. In the case of asymptotic stability, this derivative goes to zero when *h* goes to infinity. The value of *h* for which that derivative becomes negligible is called *temporal depth*, while the number of *adaptable parameters* divided by the *temporal depth* is called *temporal resolution*. The first kind of *dynamic network* is a *buffered multilayer perceptron* (*BMLP*), in which *tapped delay lines* (*TDL*'s) of the inputs are used. The *buffer* can be applied at the *network inputs* only, keeping the *network* internally static, as in *buffered MLP* (Fig. 1.30):

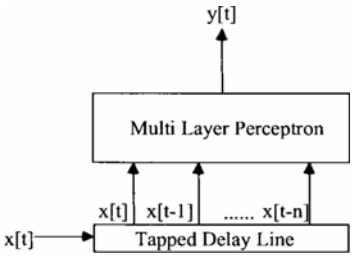


Fig. 1.30. Buffered multilayer perceptron with input buffer.

or at the input of each *neuron*, as in *MLP* with *Finite Impulse Response filter synapses* [*FIR-MLP*] (Fig. 1.31), sometimes called *time-delay neural network* (*TDNN*), and in *adaptive time-delay neural networks*. It can be shown that *BMLP* and *FIR-MLP* are theoretically equivalent, since internal buffers can be implemented as an external one.

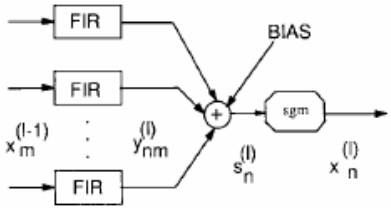


Fig. 1.31. Buffered multilayer perceptron with input buffer.

The problem with implementing *FIR-MLPs* as buffered *MLPs* is that first layers sub networks must be replicated (with shared weights) and so the complexity is much higher than considering the *buffer* internal. Therefore, *buffered MLP* and *FIR-MLP* are different architectures with regard to a real implementation. The main disadvantage of the *buffer* approach is the limited past history horizon thereby preventing modelling of arbitrary long time dependencies between inputs and *desired outputs*. It is also difficult to set the length of the *buffer*, given a certain application; moreover to have sufficient *temporal depth*, a long *buffer*, i.e., a large number of *input weights*, could be required, usually with a decrease in generalization performance and an increase in the overall computational complexity. In other words, the *buffer* approach with no *feedback* has the maximum *temporal resolution*, at the cost of a low *temporal depth*. To adaptively balance *temporal depth* with *temporal resolution*, another *buffer* type, called *gamma memory*, can be adopted, for which the *delay operator*, used in conventional *TDLs*, is replaced by a *single pole discrete time filter*. *Gamma memory* is a dispersive delay line with dispersion regulated by an adaptable *parameter*. In addition to these advantages of *temporal depth* and *temporal resolution* characteristics, it is known that *neural networks with feedback* have useful dynamic modelling behaviour. Feedback has been implemented for the first time with the introduction of the so called *fully recurrent neural networks* (*RNN*); they are formed by a *single layer* of *neurons* fully interconnected with each other, or several such *layers*. Such *RNNs*, however, exhibit some well known disadvantages: a large structural complexity (that is too many weights) and a slow and difficult *training*. As a matter of fact, they are very general architectures which can model a large class of dynamical systems. Nevertheless, on specific problems, simpler *dynamic neural networks*, which make use of available prior knowledge, can be better. Many efforts have been made with the aim of introducing *temporal dynamics* into the *multilayer perceptron* neural model. These efforts have paid in terms of less complex architectures and easier *training*, with respect to the *RNNs*. The major difference among the methods developed for the purpose lies in how *feedback* is included in the *network*.

Externally : As in the *Narendra–Parthasarathy MLP*, also known as *NARX network*, where *TDLs* are also used for the outputs that are then brought back to the input of the *network* (Fig. 1.32a). Another example is the *Elman's network* (Fig. 1.32b).

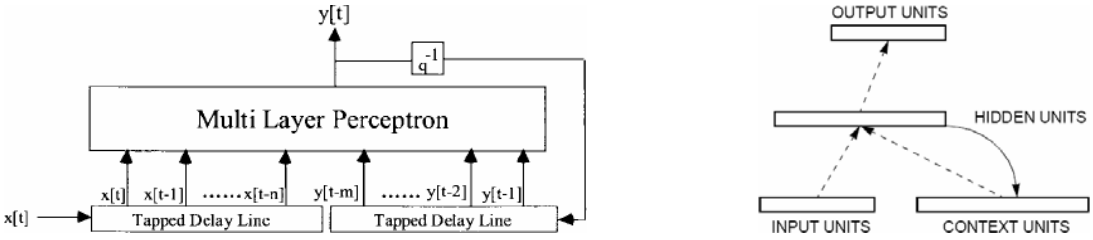


Fig. 1.32. (a) Narendra–Parthasarathy MLP; (b) Elman's network.

Internally: Inside each neuron.

The latter approach brings us to the so called *locally recurrent neural networks* (*LRNNs*) or *local feedback multilayer networks* (*LF-MLN*), that will be the architectures we will be concentrated on. In these structures, classical *infinite impulse response* (*IIR*) *linear filters*, also called *autoregressive moving average* (*ARMA*) models, are used, either directly, or with some modifications. Different architectures arise depending on how the *ARMA* model is included in the *network*.

The first architecture is the *IIR-MLP* proposed by *Back and Tsoi*, where *static synapses* are substituted by conventional *IIR adaptive filters* (Fig. 1.33).

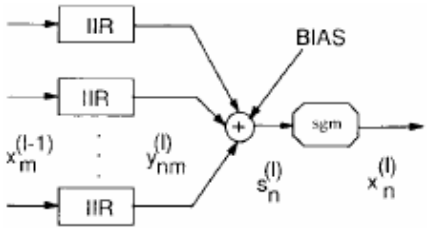


Fig. 1.33. IIR-MLP neuron structure

The second architecture is the *activation feedback multilayer perceptron network* (*AF-MLP*). The output of the *neuron summing node* is filtered by an *autoregressive* (*AR*) *adaptive filter* (all poles transfer function), before feeding the *activation function*. In the most general case, the synapses are FIR adaptive filters (Fig. 1.34). The *AF-MLP* is a particular case of the *IIR-MLP*, when all the *synaptic* transfer functions of the same *neuron* have the same denominator.

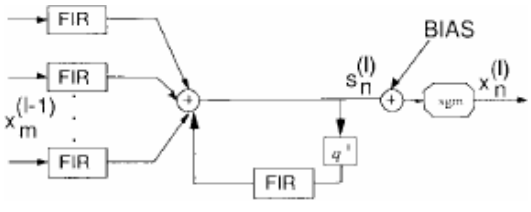


Fig. 1.34. AF-MLP neuron structure

The third structure is the *Output Feedback multilayer perceptron network* (*OF-MLP*), in which the *IIR filter* is modified in order to let the *feedback* loop pass through the *nonlinearity*, i.e., the one time step delayed output of the *neuron* is filtered by a *FIR filter* whose output is added to the inputs

contributions, providing the *activation*. Again, in the general model the *synapses* can be *FIR filters* (Fig. 1.35).

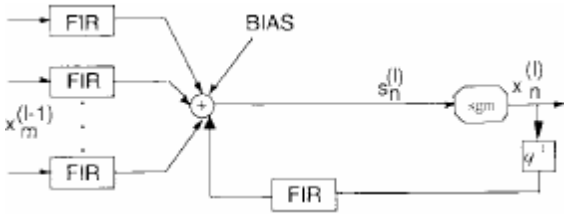


Fig. 1.35. OF-MLP neuron structure

Elman's network is based on the introduction of *context units* to include memory in a *network*, substituting the spatial metaphor of the *external buffer* with the *recurrent context* approach. *Context units* are dynamic recurrent neurons placed in the first *layer* to process the input signals, while the following *layers* are supposed to be static. This architectural constraint has been chosen basically to simplify the *learning* phase.

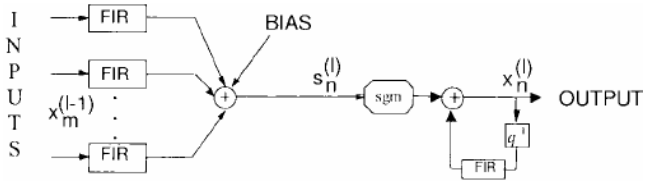


Fig. 1.36. auto

Fig. 1.37. auto-regressive MLP

At last, there is an architecture (Fig. 1.37) that's again a *multilayer network*, where each *neuron* has *FIR filter synapses* and an *AR filter* after the *activation function* (*ARMLP*). It is easy to see that this *network* is a particular case of the *IIR-MLP*, followed by *linear all-pole filters*.

The major advantages of *LRNNs* with respect to *buffered MLPs* or *fully recurrent networks* can be summarized as follows:

1) well-known *neuron interconnection* topology, i.e., the efficient and hierarchic multilayer;
2) small number of *neurons* required for a given problem, due to the use of powerful *dynamic neuron* models;
3) generalization of the popular *FIR-MLP* (or *TDNN*) to the *infinite memory* case;
4) pre-wired *forgetting behaviour*, needed in applications such as DSP, system identification and control;
5) simpler *training* than *fully recurrent networks*;
6) many *training* algorithms could be derived from filter theory and recursive identification.

In what follows, we will consider *LRNNs* only, particularly *IIR-MLP* which is the most general and interesting *architecture*. Their description will be accompanied even by a *gradient-based training*

algorithm, adjusted to fit the dynamic nature of *LRNNs*. It is called *Recursive Backpropagation* (*RBP*), while its *on-line* version is called *Causal Recursive Backpropagation* (*CRBP*).

1.5.1 Gradient-Based Learning Algorithms For LRNNs

Internally static networks, such as *buffered MLPs* (with *input buffer* only, no *recursion*) can be *trained* by the standard *BP* algorithm, while for the *NARX network* the so called "*open-loop*" approximation of the standard *BP* is usually employed. It consists in opening the *loop* during the *backward* phase, feeding the *network* with the *desired outputs*, instead of the true *network* outputs. In *IIR adaptive filter* theory, this is the equation error approximation of the true output error approach, while in *neural-network* theory, this is the teacher forced technique. The first attempts to extend to *recurrent networks* led to a case in which the *recurrent network* behaviour relaxes to a fixed point. However, if a general temporal processing is needed, two main gradient-based learning approaches exist for recurrent networks: *Backpropagation Through Time* (*BPTT*) and *Real-Time Recurrent Learning* (*RTRL*). *BPTT* is a family of algorithms which extends the *BP* paradigm to *dynamic networks*. There are two main points of view to understand the *BPTT* algorithm. The first is an intuitive one: *time unfolding* of the *recurrent network*, i.e., for single *layer* single *feedback* delay *fully recurrent networks*, one can think of the *network state* at time *t* as if it was obtained from the t^{th} layer output of a *multilayer network* with *T* layers, where *T* is the length of the *sequence*. The other point of view is a mathematical one and it is based on the Werbos theory of *ordered derivatives*. Werbos provided a mathematical tool to rigorously compute the derivatives of a certain variable with respect to another one in complex structures described by ordered mathematical relations (for example a *neural network*). Actually, with ordered derivatives, it is possible to derive both *BPTT* and *RTRL* algorithms in the same framework. The difference between *BPTT* and *RTRL* is in how the *chain rule* derivative expansion is applied. More specifically, during the *learning* phase, in *BPTT* the *neural network* is computed *backward* both in the *layer* and time dimensions, whereas in *RTRL* it is calculated *forward* (as in the *forward* calculation). Reversing the signal flow graph provides the great efficiency of *BPTT*, but the necessary reversion of time makes it *non-causal*, even when there is only one delay inside the *network* (i.e., after an adaptable *parameter* in the signal flow graph). Therefore *BPTT* can be implemented in *batch mode* only. For *on-line adaptation*, some approximations are needed, namely *causalization* and *truncation* of past history, as it will be explained in what follows for *LRNNs*. On the other hand, *RTRL* is computationally complex but intrinsically *on-line*. Most of the above methods were studied for general *RNNs*. Nevertheless, several *on-line learning* algorithms have been proposed for specific *dynamic multilayer neural networks*. Wan proposed a learning algorithm, named *Temporal Backpropagation*

(*TBP*), as an *on-line* version of the *batch mode BPTT* approach. However, it can be applied to the non-recurrent *FIR-MLP* only. *BPS* is a *learning* algorithm both for *OF-MLPs* and *AF-MLP Outputs local feedback MLNs* (*LFMLNs*). It is computationally simple and with small memory requirement, being only slightly more complex than standard BP. However, it can be applied only to *LF-MLN* with no dynamic units in *layers* other than the first one. *BPS* is basically the classical *BP* on a *multilayer network* with *recursive computation* concentrated inside each *dynamic neuron* only. Due to the architectural constraints, this algorithm does not implement *BP* through a dynamic structure. *Back and Tsoi* proposed a *learning* algorithm for *IIR-MLPs* that is similar to *BPS*, implementing both a *BP* and a *recursive computation*, but without any architectural restriction. However, to avoid *dynamic BP*, they proposed to use *static BP* even through a *dynamic neuron*.

The *on-line* algorithm we planned, i.e. *CRBP*, implements and combines together *BPTT* and *RTRL* paradigms for *LRNNs*. It works with the most general *LRNNs* and implements a more accurate computation of the *gradient* than the Back–Tsoi method. While *Back – Tsoi algorithm* uses an *instantaneous error* as *cost function*, *CRBP* can minimize the *global error*; this fact results in an improved *stability* of the algorithm. The name adopted, i.e. *Causal Recursive BP*, was chosen to remember the dual nature of the algorithm: *BPTT* style formulas are used to *back-propagate* the *error* through the *neurons* and *recursive computation* of derivatives inside each *neuron* is implemented to calculate *weights* variations. It is well known that *RTRL* and *BPTT* approaches are equivalent in *batch mode*: they compute the same *weights variations* using different *chain rule* expansions. Since *CRBP* uses another expansion of the same derivatives, it becomes equivalent to them when used in *batch mode* (*RBP*). Since *BPTT* is computationally simpler than *RTRL* or *RBP*, it is the best choice in *batch mode*, unless the memory requirement is an issue. In this case *RTRL* can be preferred, since for long *sequences* it requires less memory. However, the three methods are not equivalent in *on-line mode*. In this case *truncated BPTT* must be considered instead of *BPTT* and *CRBP* instead of *RBP*. It must be stressed that in *CRBP* each *local feedback* of a certain *neuron* is taken into account with no history truncation (necessary for *truncated BPTT*) for the adaptation of the coefficients of the same *neuron*, using *recursive* formulas instead of *non-causal* ones as in the *truncated BPTT* approach. In other words, *RBP* computes exact *gradient* and has the advantage that it can be efficiently implemented *on-line* (*CRBP*) at approximately the same cost, with a parameter that controls the trade-off between exactness of the *gradient* and computing time.

1.5.2 The RBP Algorithm for MLP with IIR Synapses

An *IIR-MLP* contains in each *synapse* a *linear filter* with *poles* and *zeros*, which are the *autoregressive* (*AR*) and *moving average* (*MA*) part, respectively. Due to the complexity of the

resulting structure, a rigorous notation is needed, where each index is explicitly written (see "Abbreviations and symbols" at the beginning of this thesis). This notation, which is a generalization of that used for *static MLP* and for *FIR-MLP*, is appropriate in this case where complex *architectures* of different kinds are defined and compared. To further clarify the notation, a simple *two-layer* ($M = 2$) *IIR-MLP* with *three inputs* ($N_0 = 3$), one *hidden neuron* ($N_1 = 1$), with no *MA* and *AR* parts in each *synapse* ($L_{1m}^{(1)} = 1$ and $I_{1m}^{(1)} = 0$ for $m = 1,\ldots,3$) and one output neuron ($N_2 = 1$), with both *MA* and *AR* parts in the *synapses* ($L_{11}^{(2)} = 3$ and $I_{11}^{(2)} = 2$), is shown in Fig. 1.38.

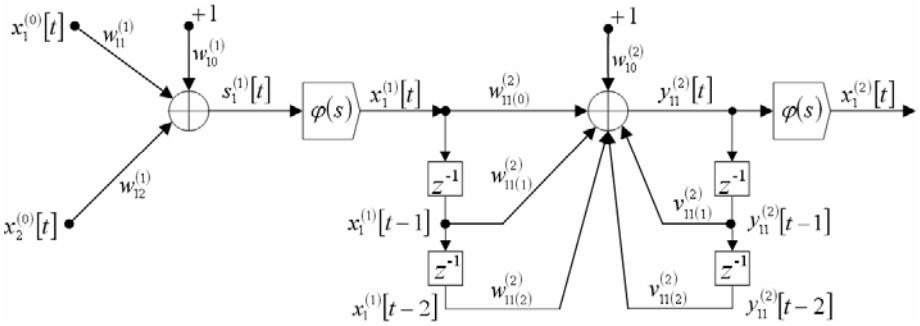


Fig. 1.38. A simple example of IIR-MLP network

1.5.2.1 The FORWARD phase

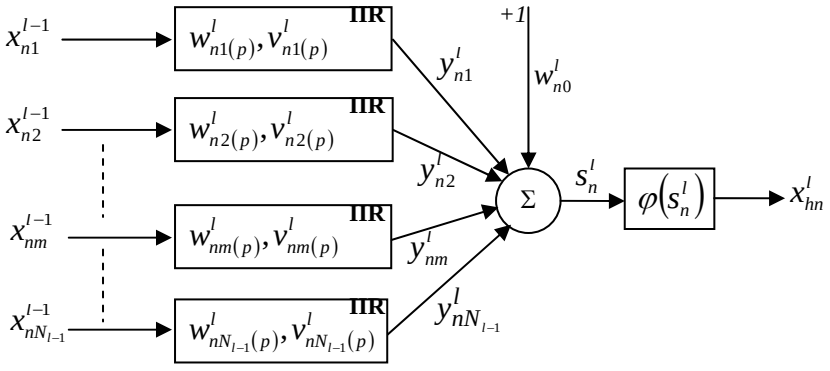


Fig. 1.39. l^{th} level, n^{th} neuron, m^{th} input (Back - Tsoi)

Let's consider an ***input sequence***:

$$\mathbf{x}[T],\ \mathbf{x}[T-1],\ \ldots,\ \mathbf{x}[t],\ \ldots,\ \mathbf{x}[1] \qquad\qquad \mathbf{x}[t] = \{x_1[t],\ \ldots,\ x_n[t],\ \ldots,\ x_{N_0}[t]\}$$

The *forward phase* at time *t* can be described by the following equations evaluated for $l = 1,\ldots,M$ and $n = 1,\ldots,N_l$ (Fig 1.39):

$$y_{nm}^{(l)}[t] = \sum_{p=0}^{L_{nm}^{(l)}-1} w_{nm(p)}^{(l)}\, x_m^{(l-1)}[t-p] + \sum_{p=1}^{I_{nm}^{(l)}} v_{nm(p)}^{(l)}\, y_{nm}^{(l)}[t-p] \tag{1.55}$$

$$s_n^{(l)}[t] = \sum_{m=0}^{N_{l-1}} y_{nm}^{(l)}[t] \qquad\qquad x_n^{(l)}[t] = \varphi\left(s_n^{(l)}[t]\right) \tag{1.56}$$

For (1.55), the *direct form I* of the *IIR filter* has been used, but other structures are possible. In particular, *direct form II* structures allow reduction in the storage complexity, as well as in the number of operations, both in *forward* and *backward computation*. For the sake of clarity, the expression corresponding to (1.55) in the *IIR filter* is reported in simplified notation (l^{th} *layer*, n^{th} *neuron*, m^{th} input):

$$y[t] = \sum_{p=0}^{L-1} w_p[t]\, x[t-p] + \sum_{p=1}^{I} v_p[t]\, y[t-p] \tag{1.57}$$

where $y[t]$ is the output, $x[t]$ the input of the *IIR filter*, *w* are the coefficients of the *MA* part, *v* of the AR part, $L-1$ and I, respectively, the orders of the *MA* and *AR* parts. If we define the *delay operator* as:

$$q^{-1} \;:\; q^{-p} s[t] = s[t-p] \tag{1.58}$$

we can adopt the following notation:

$$A(t,q) = \sum_{p=1}^{I} v_p[t]\, q^{-p} \qquad \text{and} \qquad B(t,q) = \sum_{p=0}^{L-1} w_p[t]\, q^{-p} \tag{1.59}$$

and write the expression of the *IIR filter output* as follows:

$$y[t] = \sum_{p=0}^{L-1} w_p[t]\, q^{-p} x[t] + \sum_{p=1}^{I} v_p[t]\, q^{-p} y[t] = B(t,q)\, x[t] + A(t,q)\, y[t]$$

$$\Downarrow$$

$$y[t] = \frac{B(t,q)}{1 - A(t,q)}\, x[t] = H(t,q) x[t] \tag{1.60}$$

In the above expression, the dependence of the *coefficients* on time is explicitly stated by index *t*, since we are considering the case in wich *filters* are adapted at every time step. However, in order to reduce the complexity of the notation from now on we will not use the explicit indication of time. By manipulating the last expression, it's easy to derive the formulation ot the output of the *adaptive IIR filter*:

$$y[t] = \frac{B(t,q)}{1 - A(t,q)}\, x[t] \quad \Rightarrow \quad \begin{cases} y[t] = B(t,z)\, u[t] \\ u[t] = \dfrac{1}{1 - A(t,z)}\, x[t] \end{cases}$$

$$\Downarrow$$

$$\begin{cases} u[t] = x[t] + \sum_{q=1}^{I} v_q[t]\, u[t-q] \\ y[t] = \sum_{q=0}^{L-1} w_q[t]\, u[t-q] \end{cases} \tag{1.61}$$

so, the structure of the *IIR filter* in *direct form II* becomes (l^{th} *layer*, n^{th} *neuron*, m^{th} input):

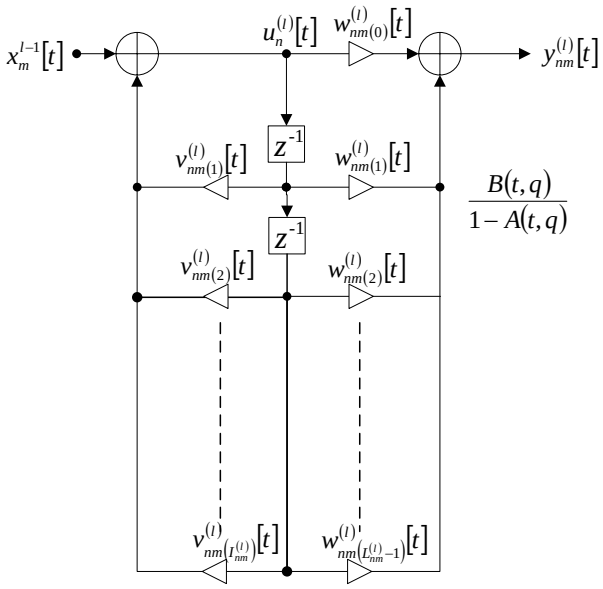


Fig. 1.40. structure of the IIR filter in direct form II

1.5.2.2 The Learning Algorithm (RBP)

Let's consider the *target sequence*:

$$\boldsymbol{d}[T],\ \boldsymbol{d}[T-1],\ \ldots,\ \boldsymbol{d}[t],\ \ldots,\ \boldsymbol{d}[1] \qquad\qquad \mathbf{d}[t]=\{d_1[t],\ \ldots,\ d_n[t],\ \ldots,\ d_{N_M}[t]\}$$

Every *example* to be used during the *training* process is a couple *input sequence - target sequence*. Training a *LRNN* consists in finding the best values for the *weights* that reduce the difference between the output $y_i[\tau],\ \tau=1,\ldots,T$ of the *network* and the corresponding *target sequence* $d_i[\tau]$, $\forall\tau$. Let's define the following quantities:

Instantaneous Error

evaluated at the output of the n^{th} neuron in the *output layer* (M^{th}) of the *network* at time *t*:

$$e_n[t]=e_n^{(M)}[t]=d_n[t]-x_n^{(M)}[t] \qquad\qquad n=1,\ldots,N_M \tag{1.62}$$

Instantaneous Global Squared Error

at time *t* is defined as:

$$e^2[t]=\sum_{n=1}^{N_M}e_n^2[t] \tag{1.63}$$

Global Squared Error

over the whole *training sequence*

$$E^2 \;=\; E^2[1\ldots T] \;=\; \sum_{t=1}^{T}e^2[t] \;=\; \sum_{t=1}^{T}\sum_{n=1}^{N_M}\left(d_n[t]-x_n^{(M)}[t]\right)^2 \tag{1.64}$$

where *T* is the duration of the *sequence*.

In the most general case, the *training set* of a *dynamic neural network* is composed of more than one *training sequence* (*runs*); therefore, the *error* to be minimized is the statistical average of the *error* over all the *runs*. To simplify the notation, we will consider only one *run*, but the extension is

straightforward. The *RBP learning algorithm* is based on the widely exploited *gradient descent* method. At any step, the adjustment of the *weights* (*coefficients*) of the *MA* in the generic *synapses* is accomplished as follows:

$$w_{nm(p)}^{(l)}[t+1] \;=\; w_{nm(p)}^{(l)}[t] + \Delta w_{nm(p)}^{(l)}[t+1] \;=\; w_{nm(p)}^{(l)}[t] - \mu \frac{\partial E^2}{\partial w_{nm(p)}^{(l)}[t]} \tag{1.65}$$

where μ is the *learning rate*. To compute the *gradient*, we can apply the *chain rule* expansion (Fig. 1.39):

$$\frac{\partial E^2}{\partial w_{nm(p)}^{(l)}[t]} = \frac{\partial E^2}{\partial s_n^{(l)}[t]} \frac{\partial s_n^{(l)}[t]}{\partial w_{nm(p)}^{(l)}[t]} \tag{1.66}$$

Let's define the following quantities, for a generic *layer l*:

Back-propagating Error

that is the derivative of *global squared error* with respect to the output of the n^{th} *neuron*:

$$e_n^{(l)}[t] = -\frac{1}{2} \frac{\partial E^2}{\partial x_n^{(l)}[t]} \tag{1.67}$$

Delta

that is the derivative of *global squared error* with respect to the *activation* of the n^{th} *neuron*:

$$\delta_n^{(l)}[t] = -\frac{1}{2} \frac{\partial E^2}{\partial s_n^{(l)}[t]} \tag{1.68}$$

As in the *static* case, it holds

$$\delta_n^{(l)}[t] = -\frac{1}{2} \frac{\partial E^2}{\partial s_n^{(l)}[t]} = -\frac{1}{2} \frac{\partial E^2}{\partial x_n^{(l)}[t]} \frac{\partial x_n^{(l)}[t]}{\partial s_n^{(l)}[t]} = e_n^{(l)}[t] \frac{\partial \varphi\left(s_n^{(l)}[t]\right)}{\partial s_n^{(l)}[t]}$$

$$\Downarrow$$

$$\delta_n^{(l)}[t] \;=\; e_n^{(l)}[t] \varphi'\left(s_n^{(l)}[t]\right) \tag{1.69}$$

On the basis of the *gradient descent* method, we have defined the *adjustment* of the *weights* of the *MA* of the generic *synapse* in the l^{th} layer as follows:

$$\Delta w_{nm(p)}^{(l)}[t+1] \;=\; -\; \mu \frac{\partial E^2}{\partial w_{nm(p)}^{(l)}[t]} \tag{1.70}$$

The *adjustment* is applied to the generic *MA coefficient* at the end of the presentation to the *network* of a whole *training sequence* (*accumulation* of all the Δ evaluated in each presentation time of the samples of the *input sequence*) and is equal to:

$$\Delta w_{nm(p)}^{(l)} = \sum_{t=1}^{T} \Delta w_{nm(p)}^{(l)}[t+1] = \sum_{t=1}^{T} \left(-\frac{\mu}{2} \frac{\partial E^2}{\partial w_{nm(p)}^{(l)}[t]} \right) = -\frac{\mu}{2} \sum_{t=1}^{T} \frac{\partial E^2}{\partial s_n^{(l)}[t]} \frac{\partial s_n^{(l)}[t]}{\partial w_{nm(p)}^{(l)}[t]}$$

$$\Downarrow$$

$$\Delta w_{nm(p)}^{(l)} = \mu \sum_{t=1}^{T} \delta_n^{(l)}[t] \frac{\partial s_n^{(l)}[t]}{\partial w_{nm(p)}^{(l)}[t]} \tag{1.71}$$

Likewise, it holds for the *AR coefficients*:

$$\Delta v_{nm(p)}^{(l)} = \sum_{t=1}^{T} \Delta v_{nm(p)}^{(l)}[t+1] = \mu \sum_{t=1}^{T} \delta_n^{(l)}[t] \frac{\partial s_n^{(l)}[t]}{\partial v_{nm(p)}^{(l)}[t]} \tag{1.72}$$

Actually, since Δ is applied to the *MA/AR coefficients* at the end of the *input sequence* (*batch* or *off-line* mode), those *coefficient* don't depend on time. So, during the application of the *input sequence* to the *network*, they can be considered as constants, that is $\upsilon_{nm(p)}^{(l)}[t] = \Delta\upsilon_{nm(p)}^{(l)}$. As a consequence, we can write:

$$\Delta w_{nm(p)}^{(l)} = \mu \sum_{t=1}^{T} \delta_n^{(l)}[t] \frac{\partial s_n^{(l)}[t]}{\partial w_{nm(p)}^{(l)}} \qquad \Delta v_{nm(p)}^{(l)} = \mu \sum_{t=1}^{T} \delta_n^{(l)}[t] \frac{\partial s_n^{(l)}[t]}{\partial v_{nm(p)}^{(l)}} \tag{1.73}$$

Expressions to compute the derivatives $\partial s_n^{(l)}[t] / \partial \upsilon_{nm(p)}^{(l)}$ must be provided, where $\upsilon_{nm(p)}^{(l)}$ is the generic *coefficient* of the *p*-order term of the *synaptic IIR filter MA/AR* (l^{th} *layer*, n^{th} *neuron*, output of the m^{th} *neuron* in the $(l-1)^{th}$ *layer*). The summation required by the calculation of the *activation* is extended to all the inputs to the *neuron* ($h = 0, \ldots, m, \ldots, N_{l-1}$). The output of the m^{th} *synapse IIR filter* depends on its *coefficients* only ($h = m$), while it is independent of the *coefficients* of the *IIR filters* in the other *synapses*:

$$\frac{\partial s_n^{(l)}[t]}{\partial \upsilon_{nm(p)}^{(l)}} = \frac{\partial}{\partial \upsilon_{nm(p)}^{(l)}} \sum_{h=0}^{N_{l-1}} y_{nh}^{(l)}[t] = \sum_{h=0}^{N_{l-1}} \frac{\partial y_{nh}^{(l)}[t]}{\partial \upsilon_{nm(p)}^{(l)}} \quad \Rightarrow$$

$$\Downarrow$$

$$\frac{\partial s_n^{(l)}[t]}{\partial \upsilon_{nm(p)}^{(l)}} = \frac{\partial y_{nm}^{(l)}[t]}{\partial \upsilon_{nm(p)}^{(l)}} \tag{1.74}$$

Therefore:

$$\Delta w_{nm(p)}^{(l)} = \mu \sum_{t=1}^{T} \delta_n^{(l)}[t] \frac{\partial y_{nm}^{(l)}[t]}{\partial \upsilon_{nm(p)}^{(l)}} \tag{1.75}$$

Let's compute the derivatives in the summation:

$$\frac{\partial y[t]}{\partial w_p} = \frac{\partial}{\partial w_p} \left(\sum_{h=0}^{L-1} w_h\, x[t-h] + \sum_{h=1}^{I} v_h\, y[t-h] \right) = \sum_{h=0}^{L-1} \frac{\partial}{\partial w_p} (w_h\, x[t-h]) + \sum_{h=1}^{I} \frac{\partial}{\partial w_p} (v_h\, y[t-h]) =$$

$$= \sum_{h=0}^{L-1} \frac{\partial w_h}{\partial w_p} x[t-h] + \sum_{h=0}^{L-1} w_h \frac{\partial x[t-h]}{\partial w_p} + \sum_{h=1}^{I} v_h \frac{\partial y[t-h]}{\partial w_p} + \sum_{h=1}^{I} \frac{\partial v_h}{\partial w_p} y[t-h]$$

$$\Downarrow$$

$$\frac{\partial y[t]}{\partial w_p} = x[t-p] + \sum_{h=1}^{I} v_h \frac{\partial y[t-h]}{\partial w_p} = \frac{q^{-p}}{1 - A(t,q)} x[t] \tag{1.76}$$

Likewise:

$$\frac{\partial y[t]}{\partial v_p} = \frac{\partial}{\partial v_p} \left(\sum_{h=0}^{L-1} w_h\, x[t-h] + \sum_{h=1}^{I} v_h\, y[t-h] \right) = \sum_{h=0}^{L-1} \frac{\partial}{\partial v_p} (w_h\, x[t-h]) + \sum_{h=1}^{I} \frac{\partial}{\partial v_p} (v_h\, y[t-h]) =$$

$$= \sum_{h=1}^{I} \left(\frac{\partial v_h}{\partial v_p} y[t-h] + v_h \frac{\partial y[t-h]}{\partial v_p} \right)$$

$$\Downarrow$$

$$\frac{\partial y[t]}{\partial v_p} = y[t-p] + \sum_{h=1}^{I} v_h \frac{\partial y[t-h]}{\partial v_p} = \frac{q^{-p}}{1 - A(t,q)} y[t] \tag{1.77}$$

In case of a *direct form II* implementation of the *synapse IIR filter*, we have:

$$u[t] = x[t] + \sum_{q=1}^{I} v_q[t] u[t-q] = x[t] + A(t,q) u[t]$$

$$\Downarrow$$

$$u[t] = \frac{x[t]}{1 - A(t,q)} \tag{1.78}$$

If we compare this expression with (1.77), we note that $\partial y[t]/\partial w_p$ agrees with $u[t]$ delayed a number of times that's equal to the index *p* of the *coefficient* respect to with the derivative has been evaluated. Therefore, the *adjustment* Δw can be acquired by picking up $u[t]$ directly from the inner nodes of the *IIR filter* (see Fig.). Similarly,

$$\frac{\partial y[t]}{\partial v_p} = \frac{q^{-p}}{1 - A(t,q)} y[t] = \frac{q^{-p}}{1 - A(t,q)} H(t,z) x[t] \tag{1.79}$$

agrees with the output of the *IIR filter* delayed a number of times that's equal to the index *p* of the *coefficient* respect to with the derivative has been evaluated and afterwards filtered by the *AR* of the same *filter* (Fig. 1.41).

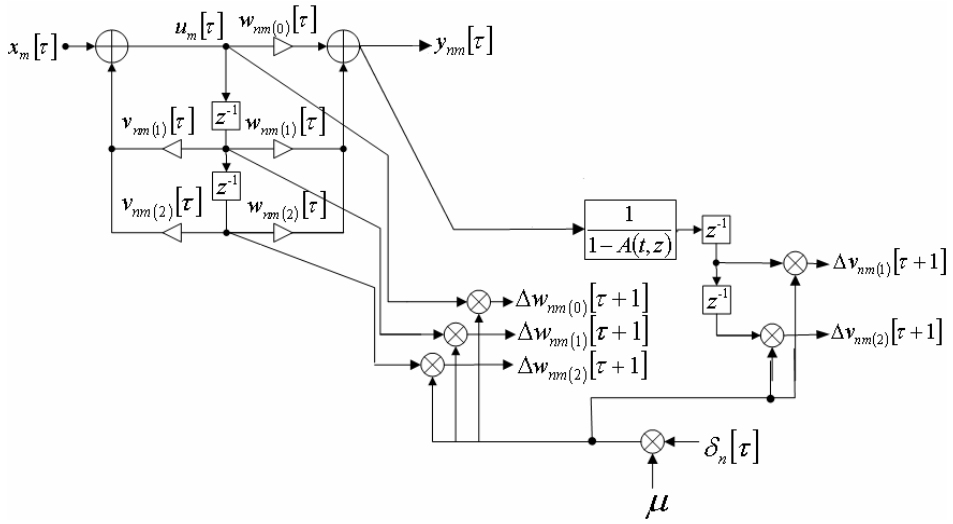


Fig. 1.41. computation scheme of Δw and Δv

The above formulas, as in the *IIR linear adaptive filter* context, are exactly true only if the *weights* (*w* or *v*) are not time-dependent, and so, the derivatives evaluation point is fixed. Alternatively, they can be considered approximately true, if they adapt slowly, i.e., the *learning rate* is sufficiently small. In *batch RBP*, the *weights update* is performed only at the end of the *learning epoch* (after the *network* has been supplied with a complete *input sequence*), using the *accumulated weight variations* computed at every time instant, so that the above expressions are exact and can be

computed iteratively, starting with null values of the initial derivatives. The next step consists in deriving the expression of (1.69). Being the computation of $\varphi'\left(s_n^{(l)}[t]\right)$ straightforward, we must concentrate on finding an expression for the *backpropagating error* (1.67). Let's start from the *output layer* ($l = M$), where:

$$e_n^{(M)}[t] = e_n[t] \qquad n = 1,\ldots,N_M \tag{1.80}$$

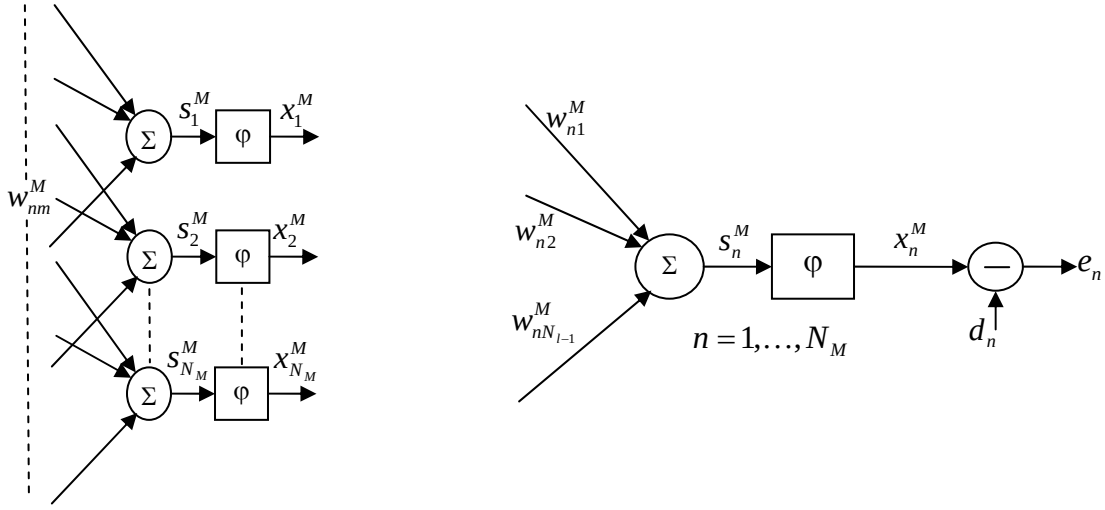


Fig. 1.42. output layer of a multilayer architecture

It holds ($\tau = 1,\ldots,t,\ldots,T$; $h = 1,\ldots,n,\ldots,N_M$):

$$e_n^{(M)}[t] = -\frac{1}{2}\frac{\partial E^2}{\partial x_n^{(M)}[t]} = -\frac{1}{2}\frac{\partial}{\partial x_n^{(M)}[t]}\sum_{\tau=1}^{T} e^2[\tau] = -\frac{1}{2}\sum_{\tau=1}^{T}\frac{\partial e^2[\tau]}{\partial x_n^{(M)}[t]}$$

$$\Downarrow$$

$$e_n^{(M)}[t] = -\frac{1}{2}\sum_{\tau=1}^{T}\frac{\partial}{\partial x_n^{(M)}[t]}\sum_{h=1}^{N_M} e_h^2[\tau] = -\frac{1}{2}\sum_{\tau=1}^{T}\sum_{h=1}^{N_M}\frac{\partial e_h^2[\tau]}{\partial x_n^{(M)}[t]} = -\sum_{\tau=1}^{T}\sum_{h=1}^{N_M} e_h[\tau]\frac{\partial e_h[\tau]}{\partial x_n^{(M)}[t]}$$

$$\Downarrow$$

$$e_n^{(M)}[t] = -\sum_{\tau=1}^{T}\sum_{h=1}^{N_M}\left(d_h[\tau] - x_h^{(M)}[\tau]\right)\frac{\partial\left(d_h[\tau] - x_h^{(M)}[\tau]\right)}{\partial x_n^{(M)}[t]}$$

$$\Downarrow$$

$$e_n^{(M)}[t] = d_n[t] - x_n^{(M)}[t] = e_n[t] \tag{1.81}$$

Before facing the computation of the same quantity in the *hidden layers*, we need to introduce some mathematical definitions and tools without which that computation could not be solved.

Let w be a function of $x, y, \ldots, z$, and let each $x, y, \ldots, z$ be a function of t. Suppose that the range of $[x(t), y(t), \ldots, z(t)]$ is contained in the domain of w. Then w is a function of t. If further w is totally differentiable and $x, y, \ldots, z$ are all differentiable, then w as a function of t is differentiable, and we have:

$$dw = \frac{\partial w}{\partial x}dx + \frac{\partial w}{\partial y}dy + \ldots + \frac{\partial w}{\partial z}dz \qquad \textbf{\textit{total differential}} \text{ of } w$$

$$\Downarrow$$

$$\frac{dw}{dt} = \frac{\partial w}{\partial x}\frac{dx}{dt} + \frac{\partial w}{\partial y}\frac{dy}{dt} + \ldots + \frac{\partial w}{\partial z}\frac{dz}{dt} \tag{1.82}$$

Moreover, let's give a new definition for the derivative of a composite function, called *rule of the ordered derivative*:

$$\frac{\partial^{+}\xi}{\partial z_i} = \frac{\partial \xi}{\partial z_i} + \sum_{j>i} \frac{\partial^{+}\xi}{\partial z_j}\frac{\partial z_j}{\partial z_i} \tag{1.83}$$

where the "+" derivatives are *partial ordered derivatives* and the others are the ordinary ones. This particular derivative of a composite function is valid only in the theory of *ordered systems*, where values are computed one after the other, in the order $z_1, z_2, \ldots, z_n, \xi$. The simple partial derivatives represent the *direct effect* of the z_i on z_j, thru the system equations that return the value of z_j. The *ordered derivative* represent the *total effect* of z_i on the *target* ξ, taking into account the direct effects as well as the *indirect effects*. For example, assume a simple system described by the following ordered equations:

$$\begin{cases} z_2 = 4z_1 \\ z_3 = 3z_1 + 5z_2 \end{cases}$$

The "simple" partial derivative of z_3 with respect to z_1 (*direct effect*) is $\partial z_3/\partial z_1 = 3$; to compute the simple effect, we look at the equations that returns z_3 only. Nevertheless, the *ordered derivative* of z_3 with respect to z_1 is 23, because of the *indirect effect* thru z_2:

$$\frac{\partial^{+} z_3}{\partial z_1} = \frac{\partial z_3}{\partial z_1} + \frac{\partial^{+} z_3}{\partial z_2}\frac{\partial z_2}{\partial z_1} = \frac{\partial z_3}{\partial z_1} + \left(\frac{\partial z_3}{\partial z_2} + 0\right)\frac{\partial z_2}{\partial z_1} = 3 + 5 * 4 = 23$$

The "simple" partial derivative measures what happens to z_3 when z_1 increases, assuming that everything else (z_2) in the equation that returns z_3 remains constant. On the contrary, the *ordered derivative* measures what happens to z_3 when z_1 increases, but this time taking into account even the variations of the other quantities (as z_2), considered as subsequent, with respect to z_1, in the *causal order* imposed on these variables by the structure of the system. This rule provides us with a method for computing the derivatives of a given *target variable* ξ with respect to all the inputs (and parameters) of a given *ordered differential system*, in only one passing thru the system itself.

For example, in classic *backpropagation*, the *target variable* is the *global squared error* E^2 and the computation to accomplish is the following:

$$\frac{\partial^{+} E^2}{\partial x_n^{(l)}} = \frac{\partial E^2}{\partial x_n^{(l)}} + \sum_{\lambda>l} \frac{\partial^{+} E^2}{\partial x_n^{(\lambda)}}\frac{\partial x_n^{(\lambda)}}{\partial x_n^{(l)}} \tag{1.84}$$

In a *neural network* (the *ordered system*), after the *output layer*, we can't find any other variable on which the *error* E^2 could depend and then the summation is equal to zero. In case of an *hidden layer*, the *global squared error* E^2 can be seen as a function of $x_n^{(l)}[t]$, thru its dependence on the *activation* of the *neurons* in the subsequent *layer* $s_q^{(l+1)}[t]$, in all the sample times in which the *input sequence* is supplied ($q = 1,\ldots,n,\ldots,N_{l+1}, t = 1,\ldots,T$).

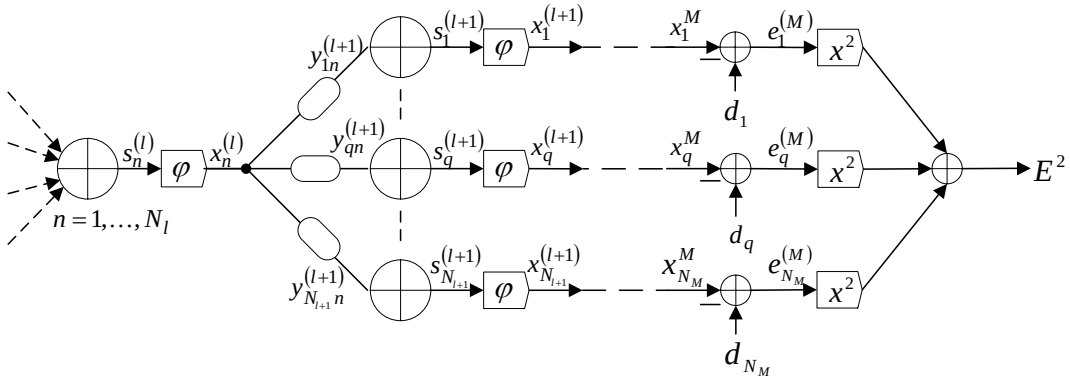


Fig. 1.43. The chain rule for a hidden layer

Actually, thru the *synaptic IIR filters*, the quantities $s_q^{(l+1)}[\tau]$, in all the sample times up to T ($t = \tau,\ldots,T$, or, in case of *non causal filters*, $t = 1,\ldots,T$) depend on the value of $x_n^{(l)}[\tau]$:

$$E^2 = f\left(s_q^{(l+1)}[\tau]\right) \qquad \forall\, \tau = 1,\ldots,T \qquad q = 1,\ldots,n,\ldots,N_{l+1} \tag{1.85}$$

$$s_q^{(l+1)}[\tau] = f\left(x_n^{(l)}[t]\right) \tag{1.86}$$

Therefore, the total differential of E^2 is equal to:

$$\begin{aligned} dE^2 = {} & \frac{\partial E^2}{\partial s_1^{(l+1)}[T]} ds_1^{(l+1)}[T] + \cdots + \frac{\partial E^2}{\partial s_{N_{l+1}}^{(l+1)}[T]} ds_{N_{l+1}}^{(l+1)}[T] + \\ & + \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots + \\ & + \frac{\partial E^2}{\partial s_1^{(l+1)}[1]} ds_1^{(l+1)}[1] + \cdots + \frac{\partial E^2}{\partial s_{N_{l+1}}^{(l+1)}[1]} ds_{N_{l+1}}^{(l+1)}[1] \end{aligned}$$

Being each $s_q^{(l+1)}[\tau]$ dependent on $x_n^{(l)}[t]$, we divide both members by the differential of the latter:

$$\begin{aligned} \frac{\partial E^2}{\partial x_n^{(l)}[t]} = {} & \frac{\partial E^2}{\partial s_1^{(l+1)}[T]} \frac{\partial s_1^{(l+1)}[T]}{\partial x_n^{(l)}[t]} + \cdots + \frac{\partial E^2}{\partial s_{N_{l+1}}^{(l+1)}[T]} \frac{\partial s_{N_{l+1}}^{(l+1)}[T]}{\partial x_n^{(l)}[t]} + \\ & + \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots + \\ & + \frac{\partial E^2}{\partial s_1^{(l+1)}[1]} \frac{\partial s_1^{(l+1)}[1]}{\partial x_n^{(l)}[t]} + \cdots + \frac{\partial E^2}{\partial s_{N_{l+1}}^{(l+1)}[1]} \frac{\partial s_{N_{l+1}}^{(l+1)}[1]}{\partial x_n^{(l)}[t]} \end{aligned}$$

$$\Downarrow$$

$$\frac{\partial E^2}{\partial x_n^{(l)}[t]} = \sum_{q=1}^{N_{l+1}} \sum_{\tau=1}^{T} \frac{\partial E^2}{\partial s_q^{(l+1)}[\tau]} \frac{\partial s_q^{(l+1)}[\tau]}{\partial x_n^{(l)}[t]} \tag{1.87}$$

But we know that by definition:

$$\frac{\partial E^2}{\partial s_q^{(l+1)}[\tau]} = -2\delta_q^{(l+1)}[\tau]$$

$$\Downarrow$$

$$e_n^{(l)}[t] = \sum_{q=1}^{N_{l+1}}\sum_{\tau=1}^{T}\delta_q^{(l+1)}[\tau]\frac{\partial s_q^{(l+1)}[\tau]}{\partial x_n^{(l)}[t]} = \sum_{q=1}^{N_{l+1}}\sum_{\tau=1}^{T}e_q^{(l+1)}[\tau]\varphi'\left(s_q^{(l+1)}[\tau]\right)\frac{\partial s_q^{(l+1)}[\tau]}{\partial x_n^{(l)}[t]} \qquad l < M \qquad (1.88)$$

By the last expression, under the hypothesis that the *IIR synaptic filter* be *causal* (only the $s_q^{(l+1)}[\tau]$'s evaluated in sample times after *t* up to *T* depend on $x_n^{(l)}[t]$) the internal summation can start from *t*. Therefore, the derivatives $\partial s_q^{(l+1)}[\tau]/\partial x_n^{(l)}[t]$ are different from zero for $\tau \geq t$ only:

$$e_n^{(l)}[t] = \sum_{q=1}^{N_{l+1}}\sum_{\tau=t}^{T}\delta_q^{(l+1)}[\tau]\frac{\partial s_q^{(l+1)}[\tau]}{\partial x_n^{(l)}[t]} \qquad (1.89)$$

Then, changing the variables as $\tau - t \rightarrow p$, the *backpropagation* through the *layers* can be derived:

$$e_n^{(l)}[t] = \begin{cases} e_n[t] & \text{per } l = M \\ \sum_{q=1}^{N_{l+1}}\sum_{p=0}^{T-t}\delta_q^{(l+1)}[t+p]\frac{\partial y_{qn}^{(l+1)}[t+p]}{\partial x_n^{(l)}[t]} & \text{per } l = (M-1),\ldots,1 \end{cases} \qquad (1.90)$$

Therefore (*layer l*, time *t*), the *backpropagating error* $e_n^{(l)}[t]$ is given by the sum of all the *deltas* of the subsequent *layer*, processed by a *non causal filter* described in what follows. We need to find an expression for the derivatives $\partial y_{hn}^{(l+1)}[t+\tau]/\partial x_n^{(l)}[t]$. During the presentation to the *network* of the *input sequence*, the *MA*/*AR coefficients* remain constant (the *total variation* is applied at the end of the *learning epoch*). Therefore ((l + 1)th *layer*, q^{th} *neuron*, n^{th} input = l^{th} *layer* n^{th} *neuron* output):

$$\frac{\partial y[t+p]}{\partial x[t]} = \frac{\partial}{\partial x[t]}\left(\sum_{r=0}^{L-1}w_r\ x[(t+p)-r] + \sum_{r=1}^{I}v_r\ y[(t+p)-r]\right)$$

$$= \sum_{r=0}^{L-1}\frac{\partial}{\partial x[t]}\left(w_r\ x[(t+p)-r]\right) + \sum_{r=1}^{I}\frac{\partial}{\partial x[t]}\left(v_r\ y[(t+p)-r]\right) =$$

$$= \sum_{r=0}^{L-1}\frac{\partial w_r}{\partial x[t]}x[(t+p)-r] + \sum_{r=0}^{L-1}w_r\frac{\partial x[(t+p)-r]}{\partial x[t]}$$

$$+ \sum_{r=1}^{I}\frac{\partial v_r}{\partial x[t]}y[(t+p)-r] + \sum_{r=1}^{I}v_r\frac{\partial y[(t+p)-r]}{\partial x[t]}$$

$$\Downarrow$$

$$\frac{\partial y[t+p]}{\partial x[t]} = \sum_{r=0}^{L-1}w_r\frac{\partial x[(t+p)-r]}{\partial x[t]} + \sum_{r=1}^{I}v_r\frac{\partial y[(t+p)-r]}{\partial x[t]} \qquad (1.91)$$

The derivative inside the first summation differs from zero for $t + p - r = t \Rightarrow r = p$ only and till then $0 \leq p \leq L-1$. Therefore, we can write:

$$\frac{\partial y[t+p]}{\partial x[t]} = \sum_{r=1}^{I} v_r \frac{\partial y[(t+p)-r]}{\partial x[t]} + \begin{cases} w_p & 0 \le p \le L-1 \\ 0 & otherwise \end{cases} \tag{1.92}$$

Obviously, in case of a *causal IIR filter*, the output (1.55) depends only on *input samples* previously presented to the *network*, so, in the other summation, we must impose that $(t+p)-r > t \Rightarrow r < p$. Therefore, the upper bound for the summation will be the minimum between the order of the *AR* an *p*:

$$\frac{\partial y_{qn}^{(l+1)}[t+p]}{\partial x_n^{(l)}[t]} = \sum_{r=1}^{\min\left(I_{qn}^{(l+1)},\, p\right)} v_{qn(r)}^{(l+1)} \frac{\partial y_{qn}^{(l+1)}[(t+p)-r]}{\partial x_n^{(l)}[t]} + \begin{cases} w_{qn(p)}^{(l+1)} & 0 \le p \le L_{qn}^{(l+1)} - 1 \\ 0 & altrimenti \end{cases} \tag{1.93}$$

These derivatives have a very interesting interpretation. Consider the expression of a generic *causal linear filter output* as the *convolution* of the input $x[\tau]$ with an *impulse response* $h[t,\tau]$ (in general time variant case):

$$y[t] = \sum_{\tau=t_0}^{t} x[\tau] h[t,\tau] \tag{1.94}$$

where t_0 is the initial time instant. Differentiating we get:

$$\frac{\partial y[t+p]}{\partial x[t]} = \frac{\partial}{\partial x[t]} \sum_{\tau=t_0}^{t+p} x[\tau] h[t+p,\tau] = \sum_{\tau=t_0}^{t+p} \frac{\partial x[\tau]}{\partial x[t]} h[t+p,\tau] = h[t+p,t] \tag{1.95}$$

where *t* is a fixed parameter and *p* is the variable. Let's define the two *sequences*:

$$\begin{aligned} &\xi_t[p] = \frac{\partial y[t+p]}{\partial x[t]} \qquad 0 \le p \le L-1 \\ &w_t[p] = w_p \end{aligned} \tag{1.96}$$

where it's been highlighted that, in case of a *time dependent IIR filter*, both ξ and *w* depend on *t*. The sequence $\xi_t[\theta]$ can be seen as the *impulse response* of a *causal time dependent filter*:

$$\xi_t[p] = h[t+p,t] \qquad 0 \le p \le L-1 \tag{1.97}$$

We can write:

$$\begin{aligned} &\xi_t[p] - \sum_{r=1}^{\min(I,p)} v_r[t]\, \xi_t[p-r] = w_t[p] && 0 \le p \le L-1 \quad \Rightarrow \\ &\xi_t[p] - A(t,q)\xi_t[p] = w_t[p] && 0 \le p \le L-1 \quad \Rightarrow \\ &\xi_t[p] = \frac{1}{1-A(t,q)} w_t[p] && 0 \le p \le L-1 \end{aligned} \tag{1.98}$$

where the operator q^{-1} is now delaying the *p* index and not the *t* index, $A(t,q)$, previously defined, does not depend on *t* [$A(t,q) = A(q)$] and the upper bound of the summation is given by $\min(I,p)$. Obviously, if the *filter* is *time invariant*, the derivative does not depend on *t*. This means that the *sequence* $\xi_t[p]$ (the derivative) is obtained through *AR filtering* of the *sequence* of the *coefficients*

of the *MA* part with the *AR* part of the corresponding *IIR synaptic filter*. This is true, since, for the *causal filter*, the derivative inside the summation in (1.93) is zero if $r > p$, allowing the upper limit of the summation in (1.93) to be written also as $I_{qn}^{(l+1)}$. If the *learning* algorithm updates the *coefficients* at the end of the *learning epoch* only, (*batch adaptation*), then the *IIR filter* is *time independent* and its *coefficients* don't depend on *t*: $A(t,q) = A(q)$, $\xi_t[p] = \xi[p]$ e $w_t[p] = w[\theta]$. Therefore, the derivative doesn't depend on *t*:

$$\frac{\partial y[t+p]}{\partial x[t]} = \frac{1}{1-A(q)} w[p] \qquad 0 \le p \le L-1 \tag{1.99}$$

If the *learning rate* is small enough, even when *on-line adaptation* is performed, the derivative is slowly changing in time, i.e., with the *t* index in (1.93). Interpreting the sequence $\xi[p]$ as an *impulse response*, we can then assert that, for *MLP* with *IIR synapses*, each *backpropagating error* :

$$e_n^{(l)}[t] = \sum_{q=1}^{N_{l+1}} \sum_{p=0}^{T-t} \delta_q^{(l+1)}[t+p] \frac{\partial y_{qn}^{(l+1)}[t+p]}{\partial x_n^{(l)}[t]} = \sum_{q=1}^{N_{l+1}} \sum_{p=0}^{T-t} \delta_q^{(l+1)}[t+p] \xi_{qn}^{(l+1)}[p] \tag{1.100}$$

at layer *l* is a summation of all the *delta*'s at the following *layer* filtered by the *noncausal* version of the respective *IIR filter*, i.e., filtering by the *time reverted impulse response* of the *synaptic filter*. In fact, let's define:

$$e[t] = \sum_{q=1}^{N_{l+1}} \varsigma[t] \tag{1.101}$$

where

$$\varsigma[t] = \sum_{p=0}^{T-t} \delta[t+p] \xi[p] = \sum_{p=0}^{T-t} \delta[t+p] \frac{1}{1-A(q)} w[p] = \frac{1}{1-A(q)} \sum_{p=0}^{T-t} \delta[t+p] w[p] \tag{1.102}$$

and $0 \le p \le L-1$. Then:

$$\varsigma[t] = \sum_{p=0}^{L-1} w_p \delta[t+p] + \sum_{r=1}^{\min(I,p)} v_r \varsigma[t-r] \tag{1.103}$$

$\varsigma[t]$ is the output of the *noncausal IIR filter*. Note that, if all the *synapses* contain only the *MA* part ($I_{nm}^l = 0 : \forall n,m,l$), the architecture reduces to *FIR-MLP* and this algorithm reduces to the *temporal backpropagation* (*TBP*, *batch mode*). Obviously, if all the *synaptic filters* have no memory ($I_{nm}^l = 0, L_{nm}^l = 1 : \forall n,m,l$), this algorithm gives *standard Back propagation* (*batch adaptation*) for the *MLP*. Moreover, the *on-line* versions of *TBP* and *BP* are obtained as particular cases of *CRBP*. Fig. 1.44 shows the diagram of the *RBP* applied to the simple *IIR-MLP* example of Fig. 34, with a simplification of the recursive computation of derivatives.

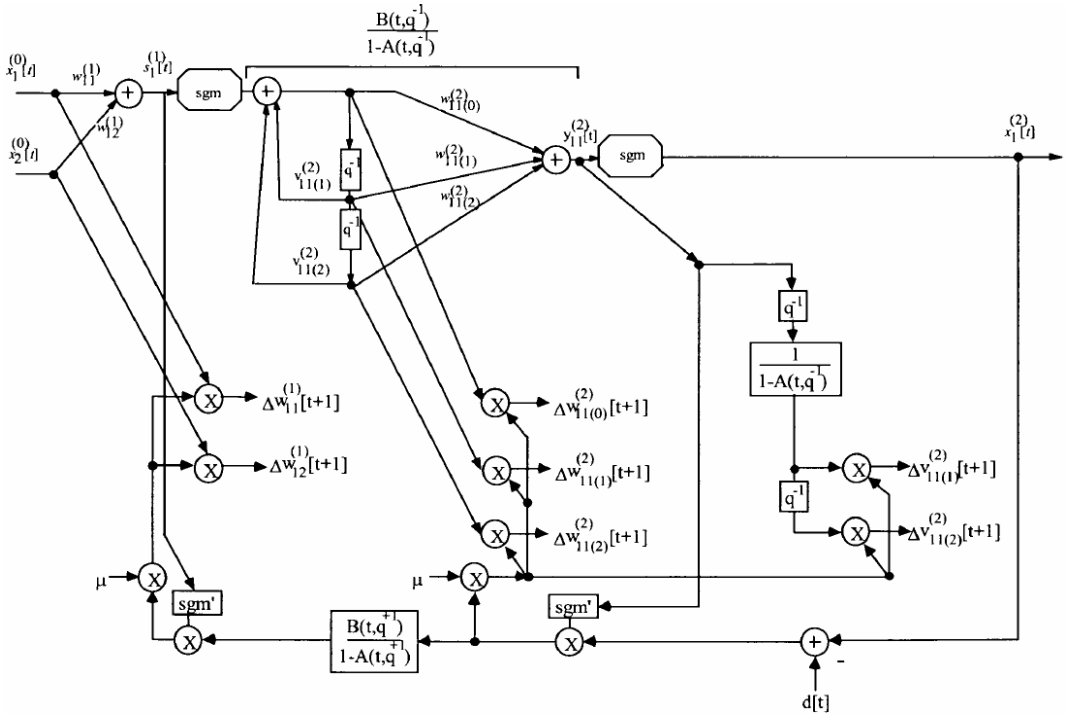


Fig. 1.44. RBP applied to the IIR-MLP

These are the steps of the algorithm for each *learning epoch*:

perform *forward pass* for the entire *input sequence*, saving the *states* of the *network* at all times, using (1.55) and (1.56);

- perform *forward pass* for the entire *input sequence*, saving the *states* of the *network* at all times:

  $t = 1, \ldots, T$

  $l = 1, \ldots, M$

  $n = 1, \ldots, N_l$

  $m = 1, \ldots, N_{l-1}$

$$y_{nm}^{(l)}[t] = \sum_{p=0}^{L_{nm}^{(l)}-1} w_{nm(p)}^{(l)}\, x_m^{(l-1)}[t-p] + \sum_{p=1}^{I_{nm}^{(l)}} v_{nm(p)}^{(l)}\, y_{nm}^{(l)}[t-p] \quad (*)$$

$$s_n^{(l)}[t] = \sum_{m=0}^{N_{l-1}} y_{nm}^{(l)}[t]$$

$$x_n^{(l)}[t] = \varphi\left(s_n^{(l)}[t]\right)$$

  $r = 0, \ldots, L_{nm}^{(l)} - 1$

$$\frac{\partial s_n^{(l)}[t]}{\partial w_{nm(p)}^{(l)}} = x_m^{(l-1)}[t-p] + \sum_{r=1}^{I_{nm}^{(l)}} v_{nm(r)}^{(l)} \frac{\partial s_n^{(l)}[t-r]}{\partial w_{nm(p)}^{(l)}} \quad (***)$$

  $r = 1, \ldots, I_{nm}^{(l)}$

$$\frac{\partial s_n^{(l)}[t]}{\partial v_{nm(p)}^{(l)}} = y_{nm}^{(l)}[t-p] + \sum_{r=1}^{I_{nm}^{(l)}} v_{nm(r)}^{(l)} \frac{\partial s_n^{(l)}[t-r]}{\partial v_{nm(p)}^{(l)}} \quad (***)$$

$t = 1, \ldots T$

$n = 1,\ldots N_M$

- start the *backward pass*, computing the *error* (for all the outputs and time instants):

$$e_n[t] = e_n^{(M)}[t] = d_n[t] - x_n^{(M)}[t]$$

$l = (M-1),\ldots,1$

- compute the derivatives of the *IIR filter* output iteratively

$q = 1,\ldots N_{l+1}$

$n = 1,\ldots N_l$

$p = 0,\ldots T-t$

$$\frac{\partial y_{qn}^{(l+1)}[t+p]}{\partial x_n^{(l)}[t]} = \sum_{r=1}^{\min\left(I_{qn}^{(l+1)},\, p\right)} v_{qn(r)}^{(l+1)} \frac{\partial y_{qn}^{(l+1)}[(t+p)-r]}{\partial x_n^{(l)}[t]} + \begin{cases} w_{qn(p)}^{(l+1)} & 0 \le p \le L_{qn}^{(l+1)} - 1 \\ 0 & altrimenti \end{cases} \quad (**)$$

with null initial conditions.

- compute:

$$e_n^{(l)}[t] = \sum_{q=1}^{N_{l+1}} \sum_{p=0}^{T-t} \delta_q^{(l+1)}[t+p] \frac{\partial y_{qn}^{(l+1)}[t+p]}{\partial x_n^{(l)}[t]}$$

- compute:

$$\delta_n^{(l)}[t] = e_n^{(l)}[t]\, \varphi'\left(s_n^{(l)}[t]\right)$$

- compute the *weights variations*:

$$\Delta w_{nm(p)}^{(l)} = \sum_{t=1}^{T} \Delta w_{nm(p)}^{(l)}[t+1] = \mu \sum_{t=1}^{T} \delta_n^{(l)}[t] \frac{\partial s_n^{(l)}[t]}{\partial w_{nm(p)}^{(l)}[t]}$$

$$\Delta v_{nm(p)}^{(l)} = \sum_{t=1}^{T} \Delta v_{nm(p)}^{(l)}[t+1] = \mu \sum_{t=1}^{T} \delta_n^{(l)}[t] \frac{\partial s_n^{(l)}[t]}{\partial v_{nm(p)}^{(l)}[t]}$$

- *update weights*

Since the *RBP* recursive expressions (**) and (***) have the same *feedback coefficients* as the corresponding *IIR filter* in the *forward* expression (1), the *learning* algorithm calculation will be stable, if all the *IIR filters* are stable.

1.5.3 The on-line RBP algorithm – CRBP

As previously stated, the *RBP* algorithm is used only as an intermediate step in the derivation of *CRBP*. It's clear that, in the computation of $e_n^{(l)}[t]$, the exact *RBP* algorithm is *non causal*, since

- $e_n^{(l)}$ at time t depends on the $\delta_q^{(l+1)}$ quantities, taken at future time instants t $\Rightarrow$
- the *weights update* can only be performed in *batch mode*

However, due to the recursive structure of

$$\frac{\partial s_n^{(l)}[t]}{\partial w_{nm(p)}^{(l)}} = x_m^{(l-1)}[t-p] + \sum_{r=1}^{I_{nm}^{(l)}} v_{nm(r)}^{(l)} \frac{\partial s_n^{(l)}[t-r]}{\partial w_{nm(p)}^{(l)}} \tag{1.104}$$

$$\frac{\partial s_n^{(l)}[t]}{\partial v_{nm(p)}^{(l)}} = y_{nm}^{(l)}[t-p] + \sum_{r=1}^{I_{nm}^{(l)}} v_{nm(r)}^{(l)} \frac{\partial s_n^{(l)}[t-r]}{\partial v_{nm(p)}^{(l)}} \tag{1.105}$$

the *RBP* algorithm can be easily approximated to obtain a very efficient *on-line learning* algorithm, indispensable to those cases in which signals to process have an excessive length (*T*), or, above all, have no predefined length and they have to be processed in *real time*, before we have them completely at our disposal. In other words, the *LRNN* is adjusted while it is processing the input signal. Therefore, let's consider *infinite length sequences* ($T \to \infty$). The *on-line* approximation consists of three steps:

1) *incremental* instead of *cumulative adaptation*;
2) *future convolution truncation*
3) *causalization*

1.5.3.1 Incremental Adaptation

An end for the *training sequence* cannot be considered; so, we can't take into account the *global quadratic error* on the whole *sequence*, as well as we cannot accumulate the Δs and apply them to the *IIR filter coefficients* at the and of the *sequence*. For this purpose, let's define Δ_u as the *adjustment* actually applied to the generic *weight* when required by the algorithm, and Δ as the instantaneous adjustment (computed at every time step). In case of *cumulative adaptation*, Δ_u is given by the *accumulation*, during the presentation of the whole *input sequence* to the *network* of all the Δs computed at every time step. Instead, in case of the *incremental adaptation*, the *weight adjustment* is accomplished every time step *t*, that is $\Delta_u = \Delta, \forall t \in [1,T]$:

$$\Delta_u \upsilon_{nm(p)}^{(l)}[t+1] = \Delta \upsilon_{nm(p)}^{(l)}[t+1] \tag{1.106}$$

$$\Downarrow$$

$$\upsilon_{nm(p)}^{(l)}[t+1] = \upsilon_{nm(p)}^{(l)}[t] + \Delta_u \upsilon_{nm(p)}^{(l)}[t+1] = \upsilon_{nm(p)}^{(l)}[t] + \Delta \upsilon_{nm(p)}^{(l)}[t+1] \tag{1.107}$$

This is the approximation that is usually proposed in literature for *dynamic networks*. However, a less obvious choice could be done (the Δs computed at the previous times are anyway accumulated):

$$\Delta \upsilon_{nm(p)}^{(l)}[t+1] = \mu \delta_n^{(l)}[t] \frac{\partial s_n^{(l)}[t]}{\partial \upsilon_{nm(p)}^{(l)}} + \sum_{\tau=1}^{t-1} \Delta \upsilon_{nm(p)}^{(l)}[\tau+1] \qquad t = 1,\ldots,T \tag{1.108}$$

This expression can be computed iteratively (with an approximation) that is equivalent to using a *momentum* term with *momentum parameter* equal to one. In fact, the *momentum* formula is

$$\Delta \upsilon_{nm(p)}^{(l)}[t+1] = \alpha\, \Delta \upsilon_{nm(p)}^{(l)}[t] + \mu\, \delta_n^{(l)}[t] \frac{\partial s_n^{(l)}[t]}{\partial \upsilon_{nm(p)}^{(l)}} \tag{1.109}$$

where α is the *momentum parameter*, that in general can be chosen in the range: $0 \le \alpha \le 1$. Here we will consider only the *incremental adaptation* without *momentum*:

$$\Delta \upsilon_{nm(p)}^{(l)}[t+1] = \mu \delta_n^{(l)}[t] \frac{\partial s_n^{(l)}[t]}{\partial \upsilon_{nm(p)}^{(l)}} \tag{1.110}$$

The computation of the derivatives $\partial s / \partial \omega$ is the same as those defined for the RBP algorithm:

$$\frac{\partial s_n^{(l)}[t]}{\partial w_{nm(p)}^{(l)}} = x_m^{(l-1)}[t-p] + \sum_{r=1}^{I_{nm}^{(l)}} v_{nm(r)}^{(l)} \frac{\partial s_n^{(l)}[t-r]}{\partial w_{nm(p)}^{(l)}} \tag{1.111}$$

$$\frac{\partial s_n^{(l)}[t]}{\partial v_{nm(p)}^{(l)}} = y_{nm}^{(l)}[t-p] + \sum_{r=1}^{I_{nm}^{(l)}} v_{nm(r)}^{(l)} \frac{\partial s_n^{(l)}[t-r]}{\partial v_{nm(p)}^{(l)}} \tag{1.112}$$

1.5.3.2 Truncation of the future convolution

We have previoulsy shown that the *backpropagating error* is given by a *noncausal convolution* of the δs computed at subsequent times with a well defined *IIR filter*. In order to work in *on-line mode*, a *learning* algorithm must be performed *in real time*; as a consequence, it has to leave the length *T* of the *input sequence* apart, being it unknown at Δ computation times. Theoretically, we could also admit that $T = \infty$ and have a *future convolution* for the computation of $e_n^{(l)}[t]$ formed by an infinite number terms, because of the infinite memory of the *synaptic IIR filters* (the derivatives $\partial y_{hn}^{(l+1)}[t+\tau] / \partial x_n^{(l)}[t] = h[\tau]$ are *impulse responses* of infinite length). Therefore, if a *causalization* is desired, a *truncation* of the *future convolution* is necessary. The computation of the *backpropagating error* was the following:

$$e_n^{(l)}[t] = \sum_{h=1}^{N_{l+1}} \sum_{\theta=0}^{\infty} \delta_h^{(l+1)}[t+\theta] \frac{\partial y_{hn}^{(l+1)}[t+\theta]}{\partial x_n^{(l)}[t]} \qquad l = (M-1), \ldots, 1 \tag{1.113}$$

As stated, we need to *truncate* the inner summation. The truncated formula is therefore:

$$e_n^{(l)}[t] = \sum_{h=1}^{N_{l+1}} \sum_{\theta=0}^{Q_{l+1}} \delta_h^{(l+1)}[t+\theta] \frac{\partial y_{hn}^{(l+1)}[t+\theta]}{\partial x_n^{(l)}[t]} \qquad l = (M-1), \ldots, 1 \tag{1.114}$$

where Q_{l+1} is appropriately chosen. This allows us to presume that $e_n^{(l)}[t]$ be dependent only on a well defined number of values of $\delta_h^{(l+1)}[t+\tau]$ in future times. In the particular case when in a given *layer l* we have $I_{nm}^{(l)} = 0 : \forall n, m$ (i.e., there's no *AR* part and so the *synapses* have finite memory), it

is useful to choose $Q_l = \max_{n,m}\left(L_{nm}^{(l)} - 1\right)$. In this way, Q_l is set to the maximum memory of the *synaptic filters* of the layer and no real *truncation* of the *filter response* is implemented.

1.5.3.3 Causalization

To remove *non causality*, we define expressions that introduce a *delay* in the *weight adjustments*. Particularly, the variation Δ_u to be used in the adjustment of the *generic weight* $\upsilon^{(l)}$ is supplied by a computation accomplished D_l times before the actual time *t*. Doing so, $e_n^{(l)}$ only seemingly depends of the values of $\delta_h^{(l+1)}$ at future times, since they are effectively computed in times preceding the actual one. In other words, the Δ_u s to be used in the *adjustment* of a *coefficient* at time $t+1$ won't be the one computed at time *t*, that depends on future times also, but the one computed at a proper time $t - D_l$:

$$\upsilon^{(l)}[t+1] \;=\; \upsilon^{(l)}[t] + \Delta_u \upsilon^{(l)}[t+1-D_l] \tag{1.115}$$

where D_l is a suitable integer number. In the *output layer*, $e_n[t]$ doesn't depend on any future times. In any *hidden layer*, on the contrary, $e_n[t]$ depends on the δ of the *forward layer*, computed at most at Q_{l+1} future times. Moreover, thru $e^{(l+1)}[t+Q_{l+1}]$, $\delta^{(l+1)}[t+Q_{l+1}]$ will in its turn depend on the δ of its *forward layer*, computed at further Q_{l+2} future times, and so on. After all, to be sure that, even in case of the first *layer* of the *network*, the computation of the *variations* be causal, we have to take care that:

$$D_l = \begin{cases} 0 & \text{per } l = M \\ \sum_{i=1+1}^{M} Q_i & \text{per } 1 \le l < M \end{cases} \tag{1.116}$$

The *causalized* formula for the computation of $e_n^{(l)}[t]$ can be obtained from (1.114) by reversing the order of the internal summation:

$$e_n^{(l)}[t] = \sum_{q=1}^{N_{l+1}} \sum_{p=0}^{Q_{l+1}} \delta_q^{(l+1)}[t+Q_{l+1}-p] \frac{\partial y_{qn}^{(l+1)}[t+Q_{l+1}-p]}{\partial x_n^{(l)}[t]} \tag{1.117}$$

and issuing the variable change $t+Q_{l+1} \to \tau$ $(t = \tau - Q_{l+1})$, where τ is the current time instant (present):

$$e_n^{(l)}[\tau - Q_{l+1}] = \sum_{q=1}^{N_{l+1}} \sum_{p=0}^{Q_{l+1}} \delta_q^{(l+1)}[\tau - p] \frac{\partial y_{qn}^{(l+1)}[\tau - p]}{\partial x_n^{(l)}[\tau - Q_{l+1}]} \qquad l = (M-1), \ldots, 1 \tag{1.118}$$

In (1.114) and (1.118), the trivial hypothesis $\delta[t]=0$ for $t \notin [1,T]$ has been done. Expression (1.118) is now causal, since it is evaluated at time τ, from *delta*'s up to time τ. The *impulse response* computed by

$$\frac{\partial y_{hn}^{(l+1)}[t+\theta]}{\partial x_n^{(l)}[t]} = \sum_{r=1}^{\min\left(I_{hn}^{(l+1)},\theta\right)} v_{hn(r)}^{(l+1)} \frac{\partial y_{hn}^{(l+1)}[t+\theta-r]}{\partial x_n^{(l)}[t]} + \begin{cases} w_{hn(\theta)}^{(l+1)} & 0 \le \theta \le L_{hn}^{(l+1)}-1 \\ 0 & altrimenti \end{cases} \tag{1.119}$$

is used by reversing the time scale since $\partial y_{hn}^{(l+1)}[t-\theta] / \partial x_n^{(l)}[t-Q_{l+1}] = h[t-\theta-(t-Q_{l+1})]$ in the time invariance hypothesis (or in that approximation), whereas *backpropagating errors* and *delta*'s are used in the normal time scale, as for standard *convolution*. The result is not assigned to the *backpropagating error* at time τ, but at time $\tau - Q_{l+1}$, as dictated by (1.114). For the sake of clarity, a diagram of *CRBP* is shown in Fig. 1.45.

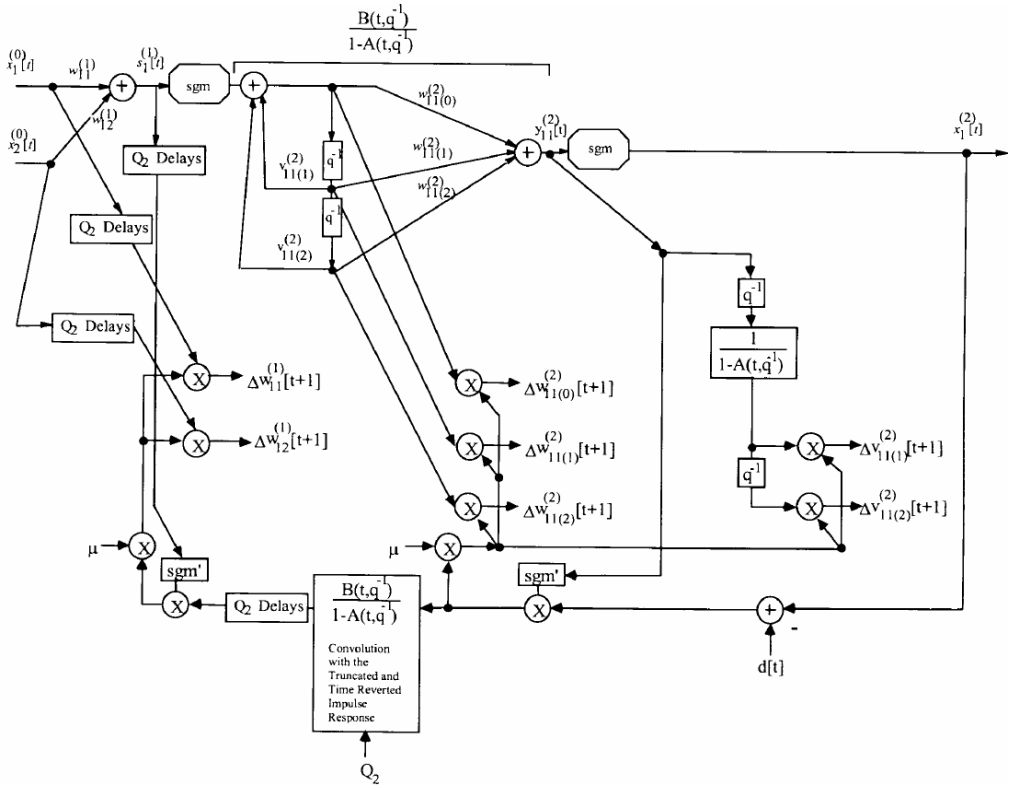


Fig. 1.45. CRBP applied to the IIR-MLP

The *causalization* and the *on-line update*, compared to the *batch mode* case, is not a strong approximation if the *learning rate* is small enough, because in this case the *weight variation* is small in the time interval of D_l instants. Instead, the *truncation* approximation can be justified by the following property: if a *linear time invariant IIR filter* is *asymptotically stable* (i.e., all the *poles* of the *transfer function* are inside the *unit circle*), then $\partial y[t+\theta] / \partial x[t] \to 0$, if $\theta \to \infty$, where $y[t]$ is the output of the *filter* and $x[t]$ the input at time *t*. The proof can be done in two ways that are

both interesting. The first is just considering that the derivative is the *impulse response* of the *filter* that must go to zero in the *stable case*. The second is considering that in the expressions

$$\frac{\partial y_{hn}^{(l+1)}[t+\theta]}{\partial x_n^{(l)}[t]} = \sum_{r=1}^{\min\left(I_{hn}^{(l+1)},\,\theta\right)} v_{hn(r)}^{(l+1)} \frac{\partial y_{hn}^{(l+1)}[t+\theta-r]}{\partial x_n^{(l)}[t]} + \begin{cases} w_{hn(\theta)}^{(l+1)} & 0 \le \theta \le L_{hn}^{(l+1)} - 1 \\ 0 & \textit{altrimenti} \end{cases} \tag{1.120}$$

$$\left(\frac{1}{1-A(t,q)}\right)\left(\begin{cases} w_\theta & \text{se } 0 \le \theta \le M-1 \\ 0 & \text{altrimenti} \end{cases}\right) = \frac{\partial y[t+\theta]}{\partial x[t]} = h[t+\theta-t] = h[\theta] \tag{1.121}$$

the *recursion coefficients* are the same of the corresponding *IIR filter*; therefore their *poles* must lie inside the *unit circle* for *stability*, i.e., the derivative goes to zero as $\theta \to \infty$. The second reasoning is more general and it is interesting because it shows a way to verify the validity of the *truncation* hypothesis, for any *locally recurrent network* architecture, e.g., *output feedback MLN*. For the derivative to go to zero, it is necessary and sufficient that the *feedback coefficients* in the calculation of the first of the above two expressions give *poles* inside the *unit circle*. Moreover, it is well known that *impulse responses* of *stable rational transfer functions* have an exponentially decaying behaviour. This means that the *truncation parameter* can be chosen quite small, even if setting the *truncation parameter* to zero is a too strong approximation that should be avoided. The previous property can be used to automatically select the desired *truncation parameter* Q_{l+1}, by taking into account the *impulse response* explicitly computed by the algorithm. The condition $\partial y[t+\theta]/\partial x[t] \to 0$ if $\theta \to \infty$, where $y[t]$ is the output (or the net) and $x[t]$ an input of a *recurrent neuron*, holds by definition for each *neuron* in *IIR-MLP*, *activation-output feedback MLN*'s and *AR-MLP*, in case each *neuron* exhibits *forgetting behaviour*. Instead in case of *latching behaviour* (possible only for *output feedback MLN*), that derivative does not go to zero and the corresponding linear system is unstable. In this case, the truncation of the internal summation in (1.90) can be too strong an approximation. However, it should be considered that the advantages of *networks* with *local feedback* over *fully connected* ones is especially in modelling a *forgetting behaviour*. *Latching behaviour* is outside normal working conditions for *LRNNs*.

CHAPTER 2

CLASSIFICATION

2.1 The Classification Problem

The *Classification Problem* deals with the need to separate a large *class* of *objects* into smaller *classes*, and to give a criterion for determining whether a particular *object* belongs or not belongs to a particular *class*. *Classification* can be even considered as the arrangement of knowledge into specific groups or systems. One of the most famous example of *classification* in biology is given by the Carolus Linnaeus's famous *classification* of living things by class, order, genus, and species. Examples in physics are the *classification* of the physical world into matter and energy, the *classification* of matter according to its atomic number and of energy according to its wavelength. Whatever the application area, the *classification problem* ha been mathematically expressed by means of the *classification* of *vector spaces*. Moreover, the *classification problem*, together with the *clustering problem*, often considered as something preliminary to *classification*, are part of a wider field of interest and investigation: *data mining*. *Clustering* and *classification* analyses are two common *data mining* techniques for finding *hidden patterns* in data. Although *classification* and *clustering* are often mentioned in the same breath, they are different analytical approaches. Let's take as an example a database of customer *records*, where each *record* represents a customer's *attributes* (for example: identifiers such as name and address, demographic information such as gender and age, and financial attributes such as income and revenue spent). *Clustering* is an automated process aiming to group related records together, on the basis of similar values for *attributes* of the *records* themselves. This approach of segmenting the database via *clustering* analysis is often used as an exploratory technique, in all those situation in which the analyst doesn't know how *records* should be related together in advance. In fact, the objective of the analysis is often to discover segments or *clusters*, and then examine the *attributes* and *values* that define those *clusters* or segments. As such, interesting and surprising ways of grouping customers together can become apparent, and this in turn can be used to drive marketing and promotion strategies to target specific types of customers. There are a variety of algorithms used for *clustering*, but they more or less follow the same iteration process: they assign *records* to a *cluster*, calculate a measure (usually similarity, and/or distinctiveness), and re-assign *records* to *clusters* until the calculated measures don't change much, indicating that the process has converged to stable *clusters*. *Records* within a

cluster are more similar to each other, and more different from *records* that are in other *clusters*. Depending on the particular implementation, there are a variety of *measures of similarity* that are used (e.g. based on spatial distance, on statistical variability, etc.), but the overall goal is to converge to stable groups of related *records*. *Classification* is a different technique than *clustering*, and is similar to the latter in that it also segments records into distinct segments called *classes*. But, unlike *clustering*, a *classification* analysis requires that the analyst know in advance how *classes* are defined. In other words, *classification* is a *supervised* method, while *clustering* is *unsupervised*. For example, *classes* can be defined to represent the likelihood that a customer is a male (Yes/No). For a *classification* to be accomplished, it is necessary that each *record* in the *data set* used to build the *classifier* be already associated to a value for the *attribute* used to define *classes*. Because of this, and because the end-user decides on the *attribute* to use, *classification* is much less exploratory than *clustering*. The objective of a *classifier* is not to explore the data to discover interesting *clusters*, but rather to decide how new *records* should be classified (i.e. is this new customer likely to be a man?). *Classification* methods in *data mining* also use a variety of algorithms – and the particular algorithm used can affect the way *records* are classified. A common approach for *classifiers* is to use *decision trees* to partition and segment *records*. New *records* can be classified by traversing the *tree* from the root through branches and nodes, to a leaf representing a *class*. The *path* a *record* takes through a *decision tree* can then be represented as a rule. But due to the sequential nature of the way a *decision tree* splits *records* (i.e. the most discriminative attribute-values appear early in the *tree*) can result in a *decision tree* being overly sensitive to initial splits. Therefore, in evaluating the goodness of fitting accomplished by a *tree*, it is important to examine the *error rate* for each *leaf node* (proportion of *records incorrectly classified*). Generally speaking, *classification algorithms* can be subdivided into two main groups, *statistical* and *neural* or, from a different point of view, in *exclusive* (every *pattern* belongs to one *class* only) and *non-exclusive* (a *pattern* can be associated to more than one *class*).

2.1.1 Formal Definition of the Classification Problem

In an *n*-dimensional space, given two sets of vectors (*patterns*), *A* and *B*, respectively called *training set* and *test set*, and given a set, *K*, of objects called *labels*, let's suppose the existence of a relation:

$$\Psi : (A \cup B) \rightarrow K \tag{2.1}$$

assigning every object $\mathbf{x} \in (A \cup B)$ to one and only one object $k \in K$, where k labels the *class* C_k $\mathbf{x}$ belongs to:

$$\Psi(\mathbf{x}) = k \qquad \Leftrightarrow \qquad \mathbf{x} \in C_k \tag{2.2}$$

The solution of a *classification* problem is the same as determining, based uniquely on what we know about the relations existing among the *patterns* in *A* and the *labels* in *K*, a system *C* that implements a correspondence Ψ_C among the *patterns* in *B* and the *labels* in *K* and that minimizes an *error function* (or a *cost function*) defined on *B*. The *error function* can be defined as follows:

$$E(C, A, B) = \sum_{\mathbf{x} \in B} \delta\left(\Psi(\mathbf{x}), \Psi_C(\mathbf{x})\right) \tag{2.3}$$

where δ is the Kronecker delta:

$$\delta\ (i, j) = \begin{cases} 1, \text{if } i = j \\ 0, \text{if } i \neq j \end{cases} \tag{2.4}$$

2.1.1.1 The non-exclusive classification

In an *n*-dimensional space, given two sets of vectors (*patterns*), *A* and *B*, respectively called *training set* and *test set*, and given a set, *K*, of objects called *labels*, let's suppose the existence of a relation:

$$\Psi : (A \cup B) \rightarrow K \tag{2.5}$$

assigning every object $\mathbf{x} \in (A \cup B)$ to a *list* of objects $L_K = [k_1, k_2, ..., k_n]$, where $k_j \in K$ labels one of the *classes* **x** belongs to and $n \leq card(K)$, that is:

$$\Psi(\mathbf{x}) = L_k \quad \Leftrightarrow \quad \mathbf{x} \in C_{k_1}, C_{k_2}, \ldots, C_{k_n} \tag{2.6}$$

The solution of a *classification* problem is the same as determining, based uniquely on what we know about the relations existing among the *patterns* in *A* and the *lists* L_k, a system *C* that implements a correspondence Ψ_C among the *patterns* in *B* and the *list of labels* in *K*, and that minimizes an *error function* $d\left(L_k, \hat{L}_k\right)$. Such a function can be trivially defined starting from a binary representation of the *list*, where an object in *K* is associated with one or zero whether it belongs to the *list* or not. This way, the *list of labels* becomes a binary number and, as a result of this, the *error* can be seen as a distance between two binary numbers, that is between the *target list* and *list* computed by the *classifier*. For example, a Hamming distance can be used.

2.1.2 Formal Definition of the Clustering Problem

Clustering is the *unsupervised classification* of *patterns* (*observations*, *data items*, or *feature vectors*) into groups (*clusters*). In a *normed n*-dimensional space, given a set of vectors (*patterns*), *A*, and given a real positive number, *k*, let's suppose we need to decompose *A* into a family of *k* disjoint subsets A_i:

$$A = \bigcup_{i=1}^{k} A_i \ \text{ where } \ A_i \cap A_j = \varnothing \ ; \ i \neq j \tag{2.7}$$

so that *patterns* in *A* belonging to the same subset can be considered similar and at the same time different from *patterns* belonging to other subsets. In many cases, the closer two *patterns* are in the *normed space* they belong to, the more they are considered similar. Starting from this criterion, all the *patterns* in *A* will be gathered in *clusters*, based on the mutual distance in a *normed n-dimensional space*. This is the so called *k*-clustering *problem*, in which the number of *clusters k* is a priori fixed. The same problem can be formulated in a different manner, that is by not previously defining the number of *clusters k*, but by fixing a priori the maximum extension a *cluster* can achieve (*free-clustering problems*). A *cluster* can be defined in different ways; three amongst the most significant choices can be the following:

- a *cluster* is a collection of similar entities that at the same time are clearly different from entities belonging to another cluster
- a *cluster* is an aggregation of points in a space, such that the distance between any two of its points is less than the distance between a point belonging to it and one lying outside it
- a *cluster* can be described as a connected region of an *n-dimensional space* characterized by a relatively high density of points, and separated by other similar regions by a zone characterized by a low density of points

In the first two definitions, the concept involved is that of *distance*, while in the latter the idea the lies behind is that of *density*, that can be anyway traced back to the concept of *distance* (point belonging to high density regions are closer to each other that to points in other zones). It's very hard to give an operational definition of the process of *clustering*, since it can be accomplished in an indefinite number of modalities and can give results strictly depending on the application it has to face up to. Moreover, the way input data are analyzed during the process and the consequent number of *clusters* obtained depend on the resolution adopted. Unlike what happens for the *classification* process, the *clustering* process is *unsupervised* and therefore it doesn't try to generalize some rules on the basis of information previously known. Consequently, a *cluster* contains similar *patterns* only, while a *class*, being it defined during the *training* process by *labels* attached to each pattern, could be composed by more than one *cluster*. Let's clarify this situation by means of an example. Let's suppose we need to classify a group of patients distinguishing them on the basis of their state of health; in other words, we suppose the existence of two *classes*. A correctly trained *classifier* would associate one patient to one of the two classes, depending on his/her state of health. Two patients, suffering from different pathologies, will be both part of the *class* of diseased subjects but, if represented by *patterns* with coordinates that can highlight the characteristics of those pathologies, could even belong to different *clusters*.

2.2 The structure of a real classifier

The data flow that characterizes a typical *classification* process as a whole, from raw data to output results, is now introduced. A *classifier* is only the most important building block amongst those that constitute a *classification* process. In fact, most *classifiers* can accept input data in a well defined dominium of a *n*-dimensional space only. This is why, in most cases, any *classifier* must be preceded by a *data normalization system* that adjusts the raw data to fit the input of the *classifier* in question. Moreover, raw data are often produced by some sort of measurement and because of this suffer from the presence of errors. Finally, it often happens that raw data are incomplete and specifically that the value of some components of a *pattern* of the *training set* can be unknown. In these cases, *normalization* doesn't suffices anymore and so, before the operations made by the classifier, a more complex operation over the raw data is required, that is a *data pre-processing*, accomplished even by means of statistical methods. Likewise, in some situations the output of a *classifier* cannot be directly used as it is, as in the case of control systems. This is why it must be followed by a *post-processing* system that adapts the output of the *classifier* to the input of a system that is eventually based on its functionalities. For these reasons, the structure of a *real classifier* can be the one depicted in Fig. 2.1:

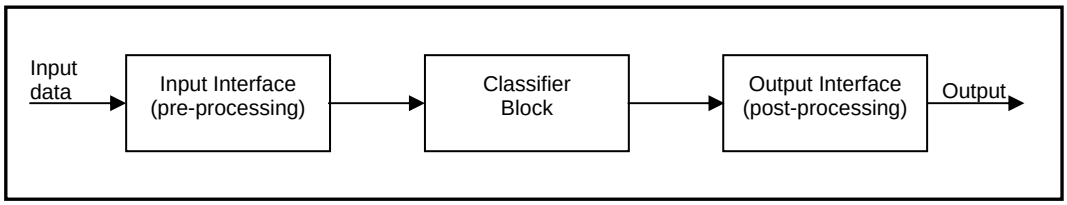


Fig. 2.1. The structure of a real classifier

In real applications, the *real classifier* consists in all of the three blocks depicted in Fig. 2.1. It means that a SW or HW implementation must find some way hiding the user from the three blocks above mentioned, and in particular from the *pre-processing* and *post-processing* blocks. Therefore, a *real classifier* appears as a black box accepting as input the raw data and producing at its output the "adapted" values required by the current application. This is why, for the *real classifier* to correctly operate, it is of primary importance that the *pre-processing* and *post-processing* blocks be designed in the right way. Moreover, the problem submitted to the *real classifier* must be valid and well posed. So, its abstract structure must be deeply analyzed, in order to define significant *training* and *test sets*. First of all, it's necessary to be sure that a *classification* operation really makes sense. For example, if data are distributed chaotically and have no significant statistics, *classification* is practically of no use. Secondly, those quantities that resemble all the information needed to exhaustively distinguish a *class* from another must be identified and considered as components of a generic *pattern*. Afterwards, a statistically meaningful population of examples, containing a suitable

degree of information about the “abstract” structure of the *classes*, must be chosen. Finally, the distribution of this population between *training set* and *test set* must be accomplished in a way that these two sets can be almost as perfectly considered as being statistically equivalent. It’s a duty of the of the central block –the *classifier* – the development, on the basis of previously given *examples*, of a structure that, after the *training* process has been accomplished, could give place to a *partition* of the whole *input space* in non generally connected dominions, each belonging to some *class* in the *training set*. The *Fuzzy classifier*, that will play a central role in what follows, after *training*, by a wise partition of the *input space*, will grant every point of it a *degree* of *membership* to each *class* in the *training set*. Afterwards, a point will be labelled as pertaining to the *class* for which the *membership degree* is maximum (a decision mechanism known as “*Winner Takes All*”).

2.3 Statistical classification

Statistical classification is a statistical procedure in which individual items are placed into groups, based on quantitative information on one or more characteristics incident to the items (called *patterns*, or vectors of *features*) and based on a *training set* of previously labelled *patterns*. Formally, the problem can be stated as follows: given the *training set* (pairs *pattern – label* of the *class* to which the item characterized by the *pattern* belongs)

$$\{(\mathbf{x}_1, c_1), (\mathbf{x}_2, c_2), \ldots, (\mathbf{x}_n, c_n)\} \tag{2.8}$$

produce a *classifier*

$$h : \mathsf{X} \to \mathsf{C} \tag{2.9}$$

that maps an object $x \in \mathsf{X}$ to its classification *label* (of the *class* it belongs to) $c \in \mathsf{C}$, based on the *probability* that $x \in \mathsf{X}$. For example, if the problem deals with some *classification* of animal species, then $\mathbf{x}_i$ is some representation of an animal and c_i could be either "mammal" or "Non-mammal ". There are many ways to define *statistical classification*, but the one interests us is the following: we try to estimate the *class-conditional probabilities* $P(\mathbf{x}|c)$ and then use the *Bayes' rule* in order to produce the *class probability* $P(c|\mathbf{x})$, that is the *conditional probability* that the *class* of belonging be *c*, provided that pattern **x** occurs.

2.3.1 Bayes Classifiers

The *Bayes Classifier* (*BC*) learns from *training data* the *conditional probability* of each attribute A_i, given the *class label C*. *Classification* is then done by applying *Bayes Rule* to compute the probability of *C*, given the particular instance $A_1, \ldots, A_n$ and predicting the *class* with the higher *posterior probability*, assuming the existence of *conditional dependency* among *attributes*. The *network* associated to a *BC* can be seen as a *directed acyclic graph* that allow efficient and effective representation of the *joint probability distributions* over a set of *random variables*. Each *vertex* in the *graph* represents a *random variable*, and *edges* represent direct correlations between the *variables*. More precisely, the *network* encodes the following *conditional independence* statements: each *variable* is independent of its non *descendants* in the *graph*, given the state of its *parents*. These *independencies* are then exploited to reduce the number of *parameters* needed to characterize a *probability distribution*, and to efficiently compute *posterior probabilities* given evidence. Probabilistic *parameters* are encoded in a set of tables, one for each *variable*, in the form of *local conditional distributions* of a *variable* given its *parents*. Using the *independence* statements encoded in the *network*, the *joint distribution* is uniquely determined by these *local conditional*

distributions. *Bayesian Network* learning from data is a form of *unsupervised learning*, in the sense that the learner does not distinguish the *class* variable from the *attribute* variables in the data. The objective is to induce a *network* (or a set of *networks*) that "best describes" the *probability distribution* over the *training data*. This *optimization* process is implemented in practice by using heuristic search techniques to find the best candidate over the space of possible *networks*. The search process relies on a *scoring function* that assesses the merits of each candidate *network*. A *BC* allows additional *edges* between *attributes* that capture correlations among them. This extension incurs additional computational costs. The induction of *Bayesian networks* requires searching the space of all possible *networks* – that is, the space of all possible combinations of *edges*.

2.3.1.1 Learning Bayesian networks

Consider a finite set $\mathbf{U} = \{X_1, \ldots, X_n\}$ of *discrete random variables* (sets of variables are denoted by boldface capital letters), where each variable X_i may take on values from a finite set, denoted by $Val(X_i)$. We use capital letters such as *X*, *Y*, *Z* for variable names, and lower-case letters such as *x*, *y*, *z* to denote specific values taken by those variables. Let *P* be a *joint probability* distribution over the variables in **U**, and let **X**, **Y**, **Z** be subsets of **U**. We say that **X** and **Y** are *conditionally independent*, given **Z**, if $\forall \mathbf{x} \in Val(\mathbf{X})$ (values taken by variables in any set are denoted by boldface lowercase letters), $\forall \mathbf{y} \in Val(\mathbf{Y})$, $\forall \mathbf{z} \in Val(\mathbf{Z})$, $P(\mathbf{x} \mid \mathbf{z}, \mathbf{y}) = P(\mathbf{x} \mid \mathbf{z})$ whenever $P(\mathbf{z}, \mathbf{y}) > 0$. A *Bayesian network* is an *annotated directed acyclic graph* that encodes a *joint probability distribution* over a set of *random variables* **U**. Formally, a *Bayesian network* for **U** is a pair $B = \langle G, \Theta \rangle$. The first component, *G*, is a *directed acyclic graph* whose *vertices* correspond to the *random variables* $X_1, \ldots, X_n$, and whose *edges* represent *direct dependencies* between the *variables*. The graph *G* encodes *independence* assumptions: each variable X_i is *independent* of its *non descendants*, given its *parents* in *G*. The second component of the pair, namely Θ, represents the set of *parameters* that characterizes the *network*. It contains a *parameters* like the following:

$$\forall x_i \in X_i, \qquad \theta(x_i, \mathsf{P}_{x_i}) = P_B(x_i \mid \mathsf{P}_{x_i}) \tag{2.10}$$

where $\mathsf{P}_{x_i} \in \mathsf{P}_{X_i}$ and where P_{X_i} denotes the set of parents of X_i in *G*. A *Bayesian network B* defines a unique *joint probability distribution* over **U** given by:

$$P_B(X_1, \ldots, X_n) = \prod_{i=1}^{n} P_B(X_i \mid \mathsf{P}_{X_i}) = \prod_{i=1}^{n} \theta(X_i, \mathsf{P}_{X_i}) \tag{2.11}$$

As an example, let $\mathbf{U} = \{A_1, \ldots, A_n, C\}$, where the variables $A_1, \ldots, A_n$ are the *attributes* and *C* is the *class* variable. Let's consider a *graph* structure where the *class* variable is the *root*, that is, $\mathsf{P}_C = \varnothing$,

and each *attribute* has the *class* variable as its unique parent, namely, $\mathsf{P}_{A_i} = \{C\}$, for all $1 < i < n$. This is the structure depicted in Fig. 2.2. For this type of *graph* structure, (2.1) yields

$$P(A_1,\ldots,A_n,C) = P(C)\prod_{i=1}^{n} P(A_i \mid C) \tag{2.12}$$

From the definition of *conditional probability*, we get:

$$P(C \mid A_1,\ldots,A_n) = \alpha\, P(C)\prod_{i=1}^{n} P(C \mid A_i) \tag{2.13}$$

where α is a normalization constant. This is in fact the definition of *Naïve Bayes* that will be described later. The problem of *learning* a *Bayesian network* can be informally stated as follows: given a *training set* $D = \{\mathbf{u}_1,\ldots,\mathbf{u}_N\}$ of instances of **U**, find a *network B* that, best matches *D*. The common approach to this problem is to introduce a *scoring function* that evaluates each *network* with respect to the *training data*, and then to search for the best *network* according to this function. In general, this *optimization* problem is intractable. Yet, for certain restricted classes of *networks*, there are efficient algorithms requiring polynomial time in the number of variables in the *network*. The two main *scoring functions* commonly used to learn *Bayesian networks* are the *Bayesian scoring function* (Cooper & Herskovits, 1992; Heckerman et al., 1995) and the function based on the principle of *Minimal Description Length* (*MDL*) (Lam & Bacchus, 1994; Suzuki, 1993). These *scoring functions* are asymptotically equivalent as the example size increases; furthermore, they are both asymptotically correct: with probability equal to one, the *learned distribution* converges to the underlying *distribution* as the number of *examples* increases.

2.3.1.2 Bayesian Networks as Classifiers

We can induce a *Bayesian network B*, that encodes a *distribution* $P_B(A_1,\ldots,A_n,C)$, from a given *training set*. We can then use the resulting model so that, given a set of *attributes* $a_1,\ldots,a_n$, the *classifier* based on *B* returns the label *c* that maximizes the posterior probability $P_B(c \mid a_1,\ldots,a_n)$. This approach is justified by the asymptotic correctness of the *Bayesian learning procedure*. Given a large *training set*, the *learned network* will be a close approximation for the *probability distribution* governing the domain (assuming that instances are sampled independently from a fixed *distribution*). However, in practice we may encounter cases where the *learning* process returns a *network* with a relatively good *MDL score* that performs poorly as a *classifier*.

2.3.2 Naïve Bayes Classifiers

A *Naïve Bayes classifier* (also known as *Idiot's Bayes*) is a simple *probabilistic classifier* based on

the application of *Bayes' theorem* with strong (naïve) independence assumptions (*independent feature model*). Depending on the precise nature of the probability model, *naive Bayes classifiers* can be trained very efficiently by means of *supervised learning*. The *Naïve Bayes Classifier* learns from *training data* the *conditional probability* of each attribute A_i, given the *class label C*. *Classification* is then done by applying *Bayes Rule* to compute the probability of *C*, given the particular instance $A_1,\ldots,A_n$ and predicting the *class* with the higher *posterior probability*. This computation is simplified by making a strong independence assumption: all the *attributes* A_i are *conditionally independent*, given the value of the *class C*. When represented as a *Bayesian network*, a *Naïve Bayesian classifier* has the simple structure depicted in Fig. 2.2.

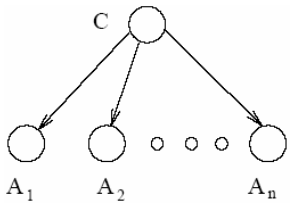


Fig. 2.2 The structure of a Naïve Bayes Network.

This *network* captures the main assumption behind the *Naïve Bayesian classifier*, namely, that every *attribute* (every *leaf* in the *network*) is independent form the rest of the *attributes*, given the state of the *class* variable (the root in the *network*). In this case, the *class* variable is a *parent* of every *attribute*, that is to say, in the learned *network*, the *probability* $p(C \mid A_1, A_2,\ldots,A_n)$, the main term determining the *classification*, will take every *attribute* into account. In many practical applications, parameter estimation for *Naïve Bayes models* uses the method of *maximum likelihood*; in other words, one can work with the *Naïve Bayes model* without believing in Bayesian probability or using any Bayesian methods. In spite of their naive design and apparently over-simplified assumptions, *Naïve Bayes classifiers* often work much better in many complex real-world situations than might be expected.

2.3.2.1 The Naïve Bayes probabilistic model

Generally speaking, the probability model for a *classifier* is a *conditional model*:

$$p(C \mid x_1, x_2,\ldots,x_n) \tag{2.14}$$

over a dependent *class variable C*, with a small number of outcomes or *classes*, conditional on several *feature* variables $x_1, x_2,\ldots,x_n$. The problem is that if the number of *features n* is large or when a *feature* can span over a large number of values, then basing such a model on probability tables is infeasible. So, reformulating the model to make it more tractable can be compulsory. Using *Bayes' theorem*, we write:

$$p(C \mid x_1, x_2, \ldots, x_n) = \frac{p(C)p(x_1, x_2, \ldots, x_n \mid C)}{p(x_1, x_2, \ldots, x_n)} \tag{2.15}$$

Actually, we are only interested in the numerator of that fraction, since the denominator does not depend on C and the values of the features x_i are given, so that the denominator is effectively constant. The numerator is equivalent to the *joint probability* model $p(C, x_1, x_2, \ldots, x_n)$ which can be rewritten as follows, using repeated applications of the definition of *conditional probability*:

$$\begin{aligned} p(C, x_1, x_2, \ldots, x_n) &= p(C)p(x_1, x_2, \ldots, x_n \mid C) = \\ &= p(C)p(x_1 \mid C)p(x_2, \ldots, x_n \mid C, x_1) \\ &= p(C)p(x_1 \mid C)p(x_2 \mid C, x_1)p(x_3, \ldots, x_n \mid C, x_1, x_2) = \\ &\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ &= p(C)p(x_1 \mid C)\ldots p(x_{n-1} \mid C, x_1, x_2, \ldots, x_{n-2})p(x_n \mid C, x_1, x_2, \ldots x_{n-1}) \end{aligned} \tag{2.16}$$

and so forth. Now the "naive" *conditional independence* assumptions come into play: assume that each feature x_i is *conditionally independent* of every other feature x_j for $j \neq i$. This means that

$$p(x_i \mid C, x_j) = p(x_i \mid C) \tag{2.17}$$

and so the *joint model* can be expressed as

$$p(C, x_1, x_2, \ldots, x_n) = p(C)\prod_{i=1}^{n} p(x_i \mid C) \tag{2.18}$$

This means that, under the above *independence* assumptions, the *conditional distribution* over the *class* variable *C* can be expressed like this:

$$p(C \mid x_1, x_2, \ldots, x_n) = \frac{1}{Z} p(C)\prod_{i=1}^{n} p(x_i \mid C) \tag{2.19}$$

where *Z* is a scaling factor dependent only on $x_1, x_2, \ldots, x_n$, i.e., a constant if the values of the *feature* variables are known. Models of this form are much more manageable, since they factor into a so-called *class prior* $p(C)$ and *independent probability distributions* $p(x_i \mid C)$. If there are *k* *classes* and if a model for $p(x_i)$ can be expressed in terms of *r parameters*, then the corresponding *Naïve Bayes model* has $(k - 1) + nrk$ *parameters*. In practice, often $k = 2$ (*binary classification*) and $r = 1$ (*Bernoulli* variables as features) are common, and so the total number of *parameters* of the *Naïve Bayes model* is $2n + 1$, where *n* is the number of *binary features* used for prediction.

2.3.2.2 Parameter estimation

We want to estimate the *parameters* of the *probability model* by means of a *supervised learning algorithm*. Because of the *independent feature* assumption, it suffices to estimate the *class prior* and the *conditional feature models* independently, using the method of *maximum likelihood*, *Bayesian*

inference or other *parameter* estimation procedures.

2.3.2.3 Derivation of a classifier from the probability model

The *Naïve Bayes classifier* can be derived by combining the *Naïve Bayes probability model*, that's *independent feature model*, with a *decision rule*. One common *rule* is to pick the hypothesis that is most probable; this is known as the *maximum a posteriori* or *MAP decision rule*. The corresponding *classifier* can be described by the following function:

$$f(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_n) = \max_c p(C = c) \prod_{i=1}^{n} p(x_i = \hat{x}_i \mid C = c) \tag{2.20}$$

2.3.2.4 Considerations

Despite the fact that the far-reaching independence assumptions are often inaccurate, the *Naïve Bayes classifier* has several properties that make it surprisingly useful in practice. In particular, the decoupling of the *class conditional feature distributions* means that each *distribution* can be independently estimated as a one dimensional *distribution*. This in turn helps to alleviate problems stemming from the curse of dimensionality, such as the need for data sets that scale exponentially with the number of *features*. Like all *probabilistic classifiers* based on the *MAP decision rule*, it gives a correct classification, as long as the correct *class* is more probable than any other class; hence *class probabilities* do not have to be estimated very well. In other words, the overall *classifier* is robust enough to ignore serious deficiencies in its underlying *naive probability model*.

2.4 Neural Classification

Examples of *neural classification algorithms* include:

- *Linear classifiers*
  - *Perceptron*
- *NON Linear classifiers*
  - *Neural networks / SOFM*
  - *Support Vector Machines*
  - *Fuzzy MIN-MAX Neural Networks*

2.4.1 Linear classifiers

A *linear classifier* is a *classifier* that uses a *linear function* of its *inputs* to base its decision on. That is, if the *input feature vector* to the classified is a real vector $\mathbf{x}$, then the estimated output score (or probability) is

$$y = f(\mathbf{w} \cdot \mathbf{x}) = f\left(\sum_n w_n x_n\right) \tag{2.21}$$

where $\mathbf{w}$ is a real vector of *weights* and f is a function that converts the dot product of the two vectors into the desired output. Often f is a simple *threshold function* that maps all values above a certain *threshold* to "yes" and all other values to "no". For a *two-class classification problem*, one can visualize the operation of a *linear classifier* as splitting a high-dimensional input space with a *hyperplane* (Fig. 2.3): all points on one side of the *hyperplane* are classified as "yes", while the others are classified as "no".

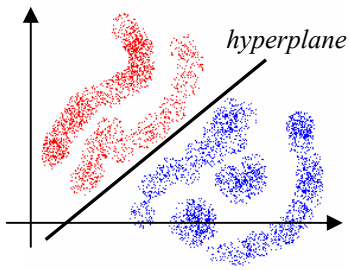


Fig. 2.3 Two linear separable sets

A *linear classifier* is often used in situations where the speed of *classification* is an issue. In fact, it is often the fastest *classifier*, especially when $\mathbf{x}$ is sparse. Moreover, *linear classifiers* often work very well when the number of dimensions in $\mathbf{x}$ is large, as in document classification, where each element in $\mathbf{x}$ is typically the number of counts of a word in a document. *Linear classifier* algorithms can be converted into non-linear algorithms, operating on a different input space, using a *kernel* function $\varphi(\mathbf{x})$. For what concerns the methods that can be used to train these kind of

classifiers, we can refer to the chapter dedicated to *neural networks*.

2.4.1.1 Perceptron

The *perceptron* is a *FFNN* that can be seen as the simplest kind of a *linear classifier*. It is a *binary classifier* that maps its input $\mathbf{x}$ to an output value $f(\mathbf{x})$:

$$f(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b \tag{2.22}$$

where $\mathbf{w}$ is a vector of real-valued *weights* (the dot product computes a weighted sum). b is the *bias*, a constant term that does not depend on any input value. The sign of $f(\mathbf{x})$ is used to classify $\mathbf{x}$ as either a positive or a negative instance (*binary classification* problem). The *bias* alters the position (though not the orientation) of the *decision boundary*. In order to describe the *training* procedure, let

$$D_m = \{(\mathbf{x}_1, c_1), (\mathbf{x}_2, c_2), \ldots, (\mathbf{x}_m, c_m)\} \tag{2.23}$$

denote a *training set* of *m examples*, where $\mathbf{x}_i$ denotes the input and c_i denotes the *desired output* for input $\mathbf{x}_i$. We use y_i to refer to the output of the *network* presented with *training example* $\mathbf{x}_i$. For convenience, we assume binary values for the desired outputs c_i, i.e. c_i = 1 for positive examples and c_i = 0 for negative ones. When the *training set* D_m is *linearly separable*, there exists a *weight* vector $\mathbf{w}$ such that:

$$y_i = \langle \mathbf{w}, \mathbf{x}_i \rangle + b > 0 \qquad \forall i \tag{2.24}$$

and the *perceptron* can be trained by a simple *on-line learning algorithm*, in which *examples* are presented iteratively and corrections to the *weight* vectors are made each time a mistake occurs. If the *training set* is *not linearly separable*, the *on-line algorithm* will never converge.

2.4.2 NON Linear classifiers

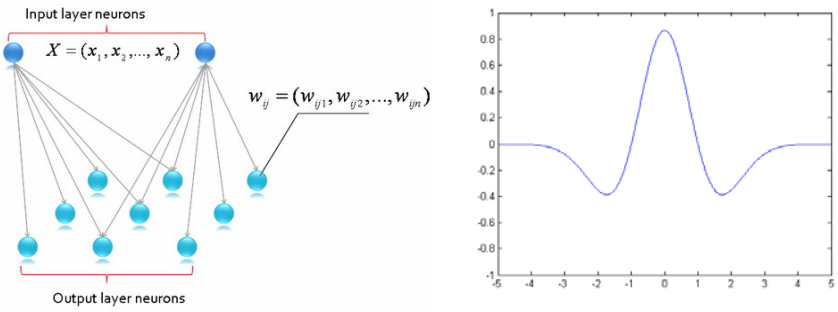
As regards *NON Linear classifiers*, we are interested only in *Multilayer Perceptron* / *SOFM* and *Support vector machines*. A separate chapter will be devoted to the *Fuzzy MIN-MAX classifier*.

2.4.2.1 Multilayer Perceptron and SOFM

As concerns the *architecture* and the *training strategies* adopted in case of a *multilayer perceptron* as a *classifier*, we refer to the above chapter dedicated to *FFNNs*. Obviously, a *multilayer perceptron* offers a more incisive *classification* power than the simple *perceptron*, since using more *neurons* and more *layers* it's possible to construct *decision boundaries* capable of separating even *non-linear separable classes*. So, *multilayer perceptron* trained by means of *gradient descent*

algorithm minimizing the *mean- square error* is a good candidate as a *classifier*. Nevertheless, it can be shown that the minimization of the *mean- square error* not always leads to the best *classifier*. We will deal with this problem later. A *two-class problem* is usually encoded using a single *output neuron*. *Many-class problems* use one *output neuron* per *class*. In case of a *two-class network*, the *target output* is either 1.0 (indicating *membership* relative to one *class*) or 0.0 (representing *membership* relative to the other one). In case of a *multi-class problem*, the *target output* is 1.0 for the *correct class* and 0.0 for all the others. When we use *NNs* in *classification* problems, a question arises on how we can interpret the numeric level on the *output neuron*(*s*). There are two different approaches. In one of these, the *output* of the *output layer units* determines the *class*, usually by interpreting this *output* as a *confidence measure*, and finding the highest confidence *class*. That approach is used in most *neural network* types.

An alternative to this approach is given by the so called *Self Organizing Feature Maps* (*SOFM*). *SOFMs* are *competitive neural networks* in which *neurons* are organized in a 2-D grid (in the most simple case) representing the *feature space*. According to the *learning rule*, vectors that are similar to each other in a n-dimensional space (*input space*) will be similar in the 2-D space. *SOFMs* are often used just to visualize an n-dimensional space, but its main application is data *classification*. Let's suppose we have a set of *n*-dimensional vectors, describing some *patterns*. Each vector element is a *feature* of a *pattern*. A *SOFM neural network* consists of two *layers* of *neurons* (Fig. 2.4 (i)). The first *layer* receives only the input and transfers it to the second *layer*. Let's consider the case of a second *layer* with *neurons* combined into a 2-D grid (other structures, like 3-D spheres, cylinders, etc. can be used). Each *neuron* of the second *layer* connects with each *neuron* of the first *layer*. The number of *neurons* in the second *layer* can be chosen arbitrary and differs from task to task. Each *neuron* of the second *layer* has its own *weight vector*. The *neurons* are connected to *adjacent neurons* by a *neighbourhood relation*, which dictates topology or structure of the *map* and is assigned by a special function called *topological neighbourhood* (Fig. 2.4 (ii)).



(i) (ii)

Fig. 2.4 (i) The architecture of SOFM neural network; (ii) The Mexican hat topological neighbourhood.

The *learning rule* is the following. In the beginning, all *weights vectors* of the second *layer*'s

neurons are set to a random values. After that, some *input-vector* from the set of *training set* vectors is selected and set to the input of the *neural network*. At this step, the distance between the input vector and all *neurons' weight vectors* are calculated. The *neural network* chooses the *winner-neuron*, i.e. the *neuron* whose *weights vector* is the most similar to the *input vector*. Then, a correction of the *weight vectors* of the *winner* and all *adjacent neurons* is made, according to the *topological neighbourhood function*. When a *SOFM* has been correctly learned and a new *input vector* is presented to it, the *SOFM* calculates the distance between the new *input vector* and all *neurons' weight vectors* (we suppose that each of these *neurons* has a *class label*). The *class label* of the *winning* (smallest distance from input case) *neuron* is typically used as the output of the *network* ("*winner takes all*" algorithm). The standard algorithm can be extended using the *KL nearest neighbour algorithm*, according to which the *class* assigned by the *network* is the most common *class* among the *K winning neurons*, provided that at least *L* of them agree. The input case might actually be very distant from any of the *weight vectors*: this case is defined as "*unknown*". In a following chapter, we will resume and deeply develop the approach to *classification* by means of *FFNNs*, when we will introduce the *discrimination* concept and the way we have chosen to solve *discrimination* problems.

2.4.2.2 Support vector machines

A *Support Vector Machine* (*SVM*) performs *classification* by constructing an *N-dimensional hyperplane* that optimally separates the data into two categories. *SVM* models are closely related to *NNs*, since an *SVM* model using a *sigmoid kernel function* is equivalent to a *two-layer FFNN*. Using a *kernel function*, *SVMs* are an alternative *training* method for *polynomial*, *radial basis function* and *multi-layer perceptron classifiers*, in which the *weights* of the *network* are found by solving a *quadratic programming problem* with linear constraints, rather than by solving an unconstrained *minimization* problem. In the parlance of *SVM* literature, a *predictor* variable is called an *attribute*, and a transformed *attribute* that is used to define an *hyperplane* is called a *feature*. The task of choosing the most suitable representation is known as *feature* selection. A set of *features* that describes one case (i.e., a row of *predictor* values) is called a *vector*. The goal of *SVM* modelling consists in finding the *optimal hyperplane* that separates *classes* of *vectors*, in such a way that cases in one category of the target variable are on one side of the *hyperplane* and cases in the other category are on the other size of the *hyperplane*. The *vectors* closest to the *hyperplane* are called *support vectors*. Fig. 2.5 shows an overview of the *SVM* process. Before considering *N-dimensional hyperplanes*, let's look at a simple 2-dimensional example. Assume we wish to perform a *classification*, and our data has a categorical *target variable* with two *classes*. Let's even assume that there are two *predictor variables* with continuous values.

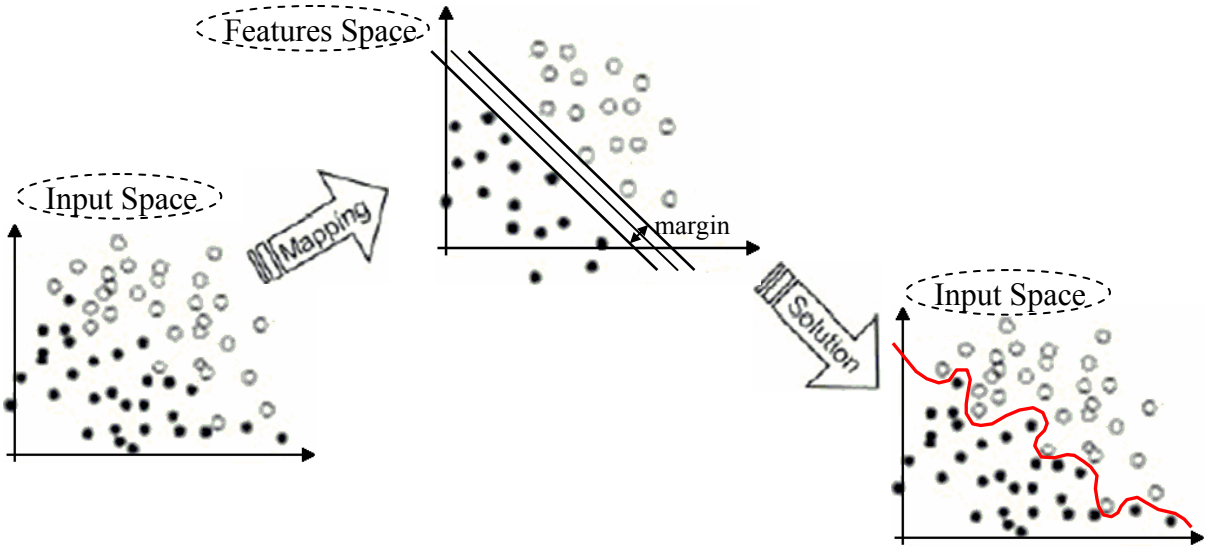


Fig. 2.5 A Two-Dimensional Example

If we plot the data points using the value of one *predictor* on the *X* axis and the other on the *Y* axis, we might end up with an image such the one shown in Fig. 2.6, where one *class* of the *target variable* is represented by squares while the other *class* is represented by circles.

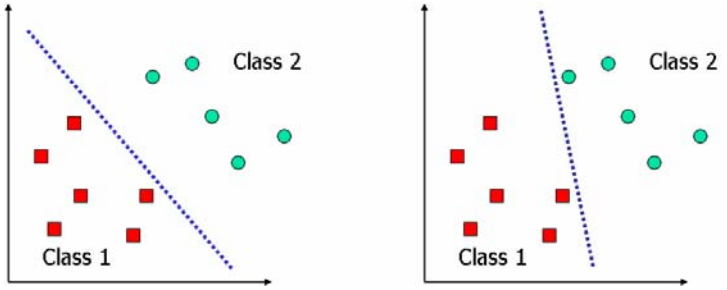


Fig. 2.6 two possible separation hyperplanes

In this example, the two *classes* are completely separated. *SVM* analysis attempts to find a *1-dimensional hyperplane* (i.e. a line) that separates the two *classes* based on their *target categories*. There are an infinite number of possible lines; two candidate lines are shown in Fig. 2.6. The question is which line is better, and how do we define the optimal line. In Fig. 2.7, the dashed lines drawn parallel to the separating line mark the distance between this line and the *vectors* closest to it.

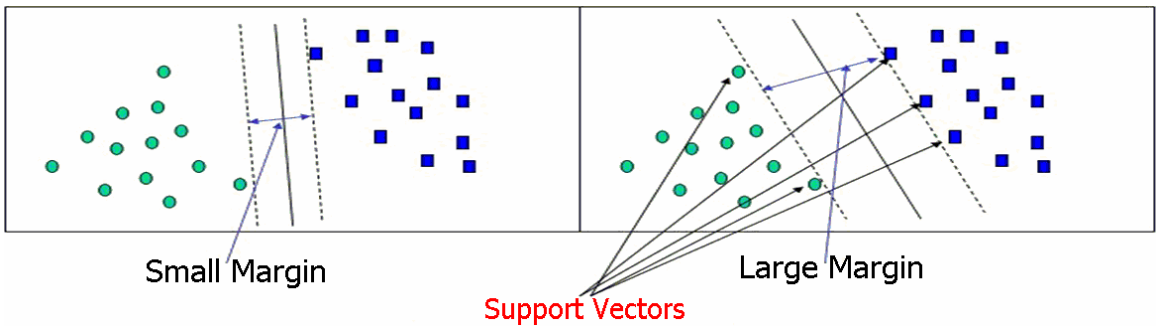


Fig. 2.7 separation hyperplanes and support vectors

The distance between the dashed lines is called *margin*. The *vectors* (points) that constrain the

width of the *margin* are the *support vectors*. An *SVM* analysis finds the line (or, in general, the *hyperplane*) that is oriented so that the *margin* between the *support vectors* is maximized. In Fig. 2.7, the line in the right panel is superior to the line in the left panel. If the *classification* problem consists in two *class target variables* only, with two *predictor* variables and with *clusters* of points that can be divided by a straight line, it can be easily solved. Unfortunately, this is not generally the case, since *SVM* must deal with

1. more than two *predictor* variables
2. separating the points with non-linear curves
3. handling the cases where *clusters* cannot be completely separated
4. handling classifications with more than two categories.

As we add more *predictor* variables (*attributes*), the data points will be represented in an *N*-dimensional space, and an (N-1)-dimensional *hyperplane* will try to separate them. But what can we do if the points are separated by a nonlinear region such as shown in Fig. 2.8?

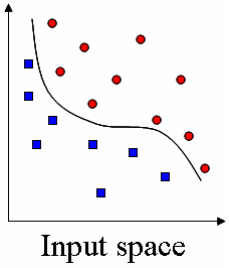


Fig. 2.8 two non linearly separable classes

In this case we would need a non linear separating line. Anyway, rather than fitting nonlinear curves to the data, *SVM* handles this situation by exploiting a *kernel function* Φ that *maps* the input data into another space, where this time an *hyperplane* can be used to do the separation, as in Fig. 2.9.

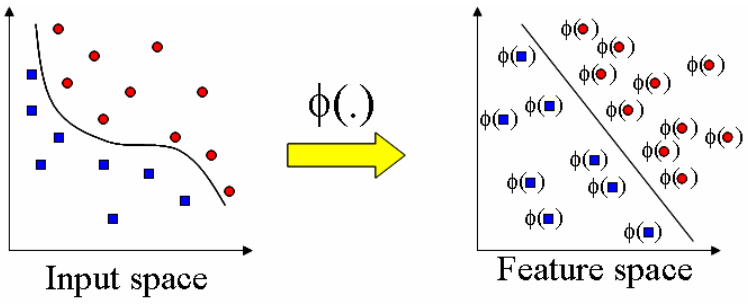
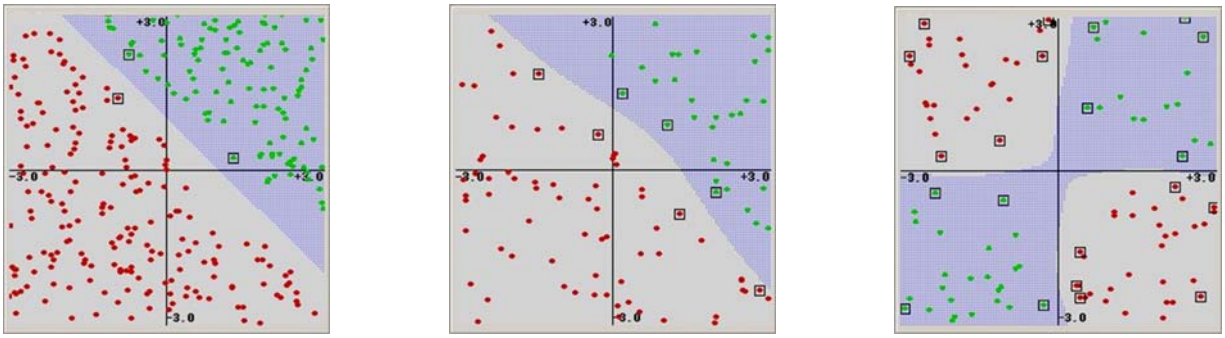


Fig. 2.9 non linear mapping between input and feature space

The *kernel function* transforms input data into a higher dimensional space, in order to make it possible to perform the separation. Many *kernel mapping functions* can be used – probably an infinite number. But only a few *kernel functions* have been found to work well for a wide variety of applications. The most common *kernel functions* are: *linear*, *polynomial* and the widely used *Radial*

Basis Function (*RBF*). They have been depicted in Fig. 2.10



(a) (b) (c)

Fig. 2.10 Frequently used Kernel functions: (a) linear; (b) polynomial; (c) RBF

Ideally, an *SVM* analysis should produce a *hyperplane* that completely separates the *feature vectors* into two *non-overlapping* groups. However, perfect separation may be impossible, or it may result in a model with so many *feature vector* dimensions that the model does not *generalize* well to other data; this is known as *over-fitting* (Fig. 2.11).

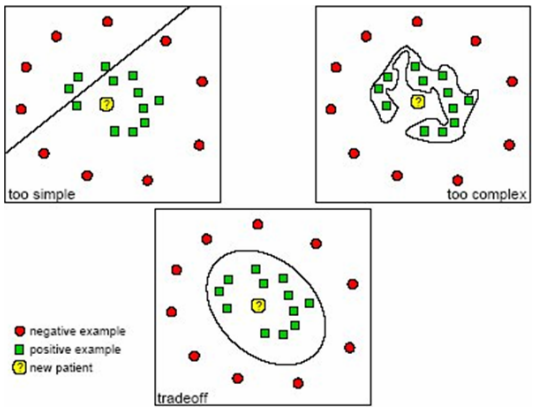


Fig. 2.11 Under-fitting and Over-fitting

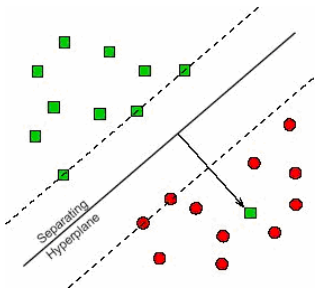


Fig. 2.12 Non separable training sets

The idea of using *hyperplanes* to separate the *feature vectors* into two *classes* works well when there are two *classes* only; but how does *SVM* handle the case where the *target variable* has more than two *classes*? Several approaches have been suggested, but two are the most popular:

1. "one against many": one *class* is split out and all the others are merged
2. "one against one" where $k(k\text{-}1)/2$ models are constructed where k is the number of *classes*. This technique is more accurate, but more computationally expensive.

The accuracy of an *SVM* model is largely dependent on the selection of the *kernel parameters*. To avoid *over-fitting*, *cross-validation* is used to evaluate the fitting provided by each *parameter* value set tried during the *training* process.

In order to define the *Support Vector Classification*, without loss of generality, we will restrict ourselves to considering the *two-class problem*, in which the goal is to separate the two *classes* by a function which is induced from available *examples*, or else, to produce a *classifier* that predicts well whether an unseen *test sample* belongs to one of the two classes (i.e. it *generalises* well). In the example in Fig. 2.13, there are many possible *linear classifiers* that can separate the data, but there is only one that maximizes the *margin* (distance between it and the nearest *support vector*). This *linear classifier* is termed the *optimal separating hyperplane*.

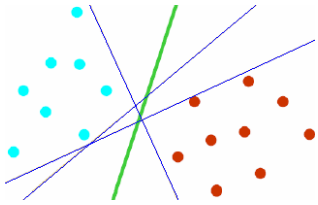


Fig. 2.13 Optimal Separating Hyperplane

Intuitively, we would expect this *hyperplane* to be the one that best *generalises*, with respect to the other possible *hyperplanes*. Let's suppose we have a set of *training vectors* belonging to two separate *classes*,

$$D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}, \qquad \mathbf{x} \in R^n, \; y \in \{-1, 1\} \tag{2.25}$$

We call $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ the *input vectors* and $\{y_1, \ldots, y_m\}$ the *class labels*. We consider the simple case in which the *vectors* in the two *classes* are *linearly separable*: we can define an *hyperplane*, or *decision surface*

$$f(\mathbf{x}) = \mathbf{w}^T \cdot \mathbf{x} - b \tag{2.26}$$

such that all cases with y_i = -1 fall on one side and have $f(\mathbf{x}_i) < 0$ and cases with y_i = +1 fall on the other and have $f(\mathbf{x}_i) > 0$. At this point, we can *classify* new *test cases*, according to the rule

$$y_{test} = sign[f(\mathbf{x}_{test})] \tag{2.27}$$

We will choose the *hyperplane* that solves the separation problem for our *training data*, considering that we may have different performances on the newly arriving *test cases*. For instance, if we put the *hyperplane* very close to members of one particular *class*, say y = -1, when *test cases* arrive, we will not make many mistakes on those that should be *classified* with y = +1, but we will make very easily mistakes on those with y = -1 (for example, in cases where new data are simply small perturbations of the *training data*). This consideration suggests us to choose the *hyperplane* as far away from both y = -1 and y = +1 *training cases* as we can, i.e. right in the middle. Geometrically,

vector **w** is directed orthogonal to the *hyperplane* defined by $\mathbf{w}^T \cdot \mathbf{x} = b$. This can be understood as follows. First take $b = 0$. Obviously, all *vectors* orthogonal to **w** satisfy this equation. In we translate the *hyperplane* away from the origin over a *vector* **a**, the equation for the *hyperplane* now becomes $\mathbf{w}^T \cdot (\mathbf{x} - \mathbf{a}) = 0 \Rightarrow \mathbf{w}^T \cdot \mathbf{x} = \mathbf{w}^T \cdot \mathbf{a}$, where the offset $b = \mathbf{a}^T \cdot \mathbf{w}$ is the projection of **a** onto the vector **w**. Without loss of generality, we may thus choose a perpendicular to the *hyperplane*, in which case the length $\|\mathbf{a}\| = |b|/\|\mathbf{w}\|$ represents the shortest, orthogonal distance between the origin and the *hyperplane*. We now define two more *hyperplanes*, parallel to the *separating hyperplane*, that cut through the closest *training examples* on either side, and we will call them "*support hyperplanes*", because the *data-vectors* they contain support the *hyperplane* itself. We define the distance between these *hyperplanes* and the *separating hyperplane* to be d_+ and d_- respectively. The *margin* is defined to be

$$\gamma = d_+ + d_- \tag{2.28}$$

Our goal is now to find a the *separating hyperplane* so that the *margin* is the largest, while the *separating hyperplane* is equidistant from both *classes*. We can write the following equations for the *support hyperplanes*:

$$\begin{aligned} \mathbf{w}^T \cdot \mathbf{x} &= b + \delta \\ \mathbf{w}^T \cdot \mathbf{x} &= b - \delta \end{aligned} \tag{2.29}$$

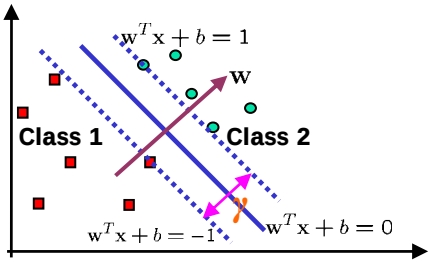


Fig. 2.14 Optimal Separating Hyperplane and its support hyperplanes

We note that, if we scale **w**, b and δ by a constant factor α, the equations for x are still satisfied. To remove this ambiguity we will require that $\delta = 1$. We can now also compute the values for d_+ (distance between the a point belonging to a line and another line):

$$d_+ = \frac{\|b+1| - |b\|}{\|\mathbf{w}\|} = \frac{1}{\|\mathbf{w}\|} \tag{2.30}$$

(this is true if only $b \notin (-1, 0)$ since the origin doesn't fall between the *hyperplanes* in that case. If $b \in (-1, 0)$ we use $d_+ = \|b+1| + |b\|/\|\mathbf{w}\| = 1/\|\mathbf{w}\|$). Hence the *margin* is equal to twice that value:

$$\gamma = \frac{2}{\|\mathbf{w}\|} \tag{2.31}$$

With the above definition of the *support hyperplanes*, we can write down the following constraint

that any solution must satisfy:

$$\begin{aligned} \mathbf{w}^T \cdot \mathbf{x}_i - b \le -1 \quad & \forall y_i = -1 \\ \mathbf{w}^T \cdot \mathbf{x}_i - b \ge +1 \quad & \forall y_i = +1 \end{aligned} \tag{2.32}$$

or in one equation:

$$y_i\left(\mathbf{w}^T \cdot \mathbf{x}_i - b\right) - 1 \ge 0 \tag{2.33}$$

We now formulate the *primal problem* of the *SVM*:

$$\text{minimize} \qquad \frac{1}{2}\|\mathbf{w}\|^2 \tag{2.34}$$

$$\text{subject to} \qquad y_i\left(\mathbf{w}^T \cdot \mathbf{x}_i - b\right) - 1 \ge 0 \quad \forall i \tag{2.35}$$

Thus, we maximize the *margin*, subject to the constraints that all *training examples* fall on either side of the *support hyperplanes*. The *examples* that lie on the *hyperplane* are called *support vectors*, since they support the *hyperplanes* and hence determine the solution to the problem. The *primal problem* can be solved by a quadratic program. However, it is not ready to be kernelised, because its dependence is not only on inner products between *data-vectors*. Hence, we transform to the dual formulation by first writing the problem using a Lagrangian (Lagrange multipliers method),

$$\mathsf{L}\left(\mathbf{w}, b, \boldsymbol{\alpha}\right) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{N} \alpha_i \left[y_i\left(\mathbf{w}^T \cdot \mathbf{x}_i - b\right) - 1\right] \tag{2.36}$$

The solution that minimizes the *primal problem* subject to the constraints is given by $\min_{\mathbf{w}} \max_{\boldsymbol{\alpha}} \mathsf{L}\left(\mathbf{w}, \boldsymbol{\alpha}\right)$, i.e. a saddle point problem. When the original *objective-function* is convex, (and only then), we can interchange the minimization and maximization. Doing that, we find that we can find the condition on **w** that must hold at the saddle point we are solving for. This is done by taking derivatives with respect to **w** and *b* and solving,

$$\begin{aligned} & \mathbf{w} - \sum_i \alpha_i y_i \mathbf{x}_i = 0 \quad \Rightarrow \mathbf{w}^* = \sum_i \alpha_i y_i \mathbf{x}_i \\ & \sum_i \alpha_i y_i = 0 \end{aligned} \tag{2.37}$$

Inserting this back into the Lagrangian, we obtain what is known as the *dual problem*:

$$\text{maximize} \qquad \mathsf{L}_D = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{ij} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \tag{2.38}$$

$$\text{subject to} \qquad \sum_i \alpha_i y_i = 0 \qquad \alpha_i \ge 0 \quad \forall i \tag{2.39}$$

It is also a quadratic program, but the number of variables α_i is equal to the number of *examples*, N. This problem only depends on $\mathbf{x}_i$ through the inner product $\mathbf{x}_i^T \mathbf{x}_j$. This is readily kernelised through the substitution $\mathbf{x}_i^T \mathbf{x}_j \to k\left(x_i, x_j\right)$. Next we turn to the conditions that must necessarily hold

at the saddle point and thus the solution of the problem (*KKT conditions*). They are necessary in general, and sufficient for convex optimization problems. They can be derived from the *primal problem* by setting the derivatives with respect to **w** to zero. Also, the constraints themselves are part of these conditions and we need that for inequality constraints the Lagrange multipliers are non-negative. Finally, an important constraint called "complementary slackness" needs to be satisfied,

$$\begin{aligned} \partial_{\mathbf{w}} L_p = 0 \quad &\rightarrow \quad \mathbf{w} - \sum_i \alpha_i y_i \mathbf{x}_i = 0 \\ \partial_b L_p = 0 \quad &\rightarrow \quad \sum_i \alpha_i y_i = 0 \end{aligned} \tag{2.40}$$

constraint - 1 $$y_i\left(\mathbf{w}^T \cdot \mathbf{x}_i - b\right) - 1 \geq 0 \tag{2.41}$$

multiplier condition $$\alpha_i \geq 0 \tag{2.42}$$

complementary slackness $$\alpha_i \left[y_i\left(\mathbf{w}^T \cdot \mathbf{x}_i - b\right) - 1 \right] = 0 \tag{2.43}$$

The function $f(\cdot)$ that can be used to *classify* future *test examples* is the following:

$$f(\mathbf{x}) = \mathbf{w}^{*T}\mathbf{x} - b^* = \sum_i \alpha_i y_i \mathbf{x}_i^T \mathbf{x} - b^* \tag{2.44}$$

As an application of the *KKT conditions*, we derive a solution for b^* by using the *complementary slackness condition*:

$$b^* = \sum_j \alpha_j y_j \mathbf{x}_j^T \mathbf{x}_i - y_i \qquad \text{where } i \text{ is a } \textit{support vector} \tag{2.45}$$

where we used $y_i^2 = 1$. So, using any *support vector* one can determine *b*, but for numerical stability it is better to average over all of them (although they should obviously be consistent). The most important conclusion is again that this function $f(\cdot)$ can thus be expressed solely in terms of inner products $\mathbf{x}_i^T \mathbf{x}_i$, which we can replace with *kernel* matrices $k(x_i, x_j)$ to move to high dimensional non-linear spaces.

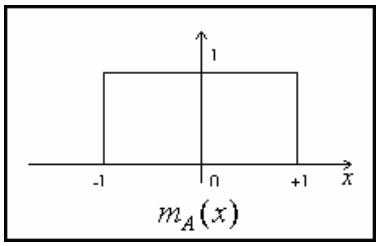
2.5 Fuzzy MIN-MAX Neural Networks (FMMNN)

FMMNNs can be seen as *supervised learning neural network classifiers* that utilizes *fuzzy sets* as *pattern classes*. As described later, each *fuzzy set* is an aggregate (union) of *fuzzy set hyperboxes*. The use of a *fuzzy set* approach to *pattern classification* inherently provides *degree of membership* information that is extremely useful in higher level decision making. This is what is called *non-exclusive classification*. *Fuzzy logic* gives a very high degree of robustness to *classification*, above all in terms of error rejection, since *class boundaries* are never clear but structurally fuzzy, that is to say, *patterns* that lie close to those *boundaries* can have peculiarities that can be considered common to more than one *class*.

2.5.1 Fuzzy Logic

Fuzzy sets were introduced by Zadeh as a means of representing and manipulating data that are not precise, but rather *fuzzy*. Zadeh's extension of set theory provides a mechanism for representing linguistic constructs such as "many," "few," "often,", "sometimes," "cold" and "hot" and it provides new tools to be applied to *pattern recognition*, by allowing the degree to which a *pattern* is present or a situation is occurring to be measured. On the contrary, traditional set theory describes crisp events, events that either do, or do not, occur. In order to explain whether an event will occur, traditional sets use probability theory, measuring the chance with which a given event is expected to occur. In situations such as the hip of a coin or death, probability theory plays a role. In contrast, *fuzzy* theory measures the degree to which an event occurs. The degree to which a person is bald is very different than the probability that a person is bald. Probability theory states unequivocally that a person either is, or is not, bald. Fuzzy theory simply states that a person is somewhat bald, or a little bald, or quite bald, or sort of bald, and so on. In traditional set theory, where crisp events are described, for any given set A that's part of a universe U, we can specify the elements of A simply by defining over the universe U the *characteristic function* of A, the output of which has values given by the binary set $\{0,1\}$ and is equal to one only in case the input is an element belonging to A.

$$m_A : U \to \{0,1\} \text{ t.c. } m_A(x) = 1 \Leftrightarrow x \in A \tag{2.46}$$



$$m_A(x) = \begin{cases} 1 \text{ , if } -1 \le x \le 1 \\ 0 \text{ , if } x < -1 \text{ or } x > 1 \end{cases}$$

Fig. 2.15. crisp characteristic function

For example, let A be the set of real numbers between -1 and 1, borders included. In this case, the *characteristic function* will be the one depicted in Fig. 2.15. In contrast, *fuzzy* theory takes into consideration *characteristic functions*, in this case called *membership functions* (*MF*), that can assume all the values comprised in the interval [0, 1], borders included.

So, let χ be a space of points (objects), with a generic element of χ denoted by x. [χ is often referred to as the universe of discourse]. A *fuzzy set* (*class*) A in χ is characterized by a *membership* (characteristic) *function* $m_A(x)$ which associates each point in χ a real number in the interval [0…1], with the value of $m_A(x)$ representing the *degree of membership* of x in A (this is the only way to define a set in *fuzzy logic*; a point of the universe not belonging to any given set of the same universe, in the aristotelic sense, doesn't exist). Thus, the nearer the value of $m_A(x)$ to unity, the higher the *degree of membership* of x in A. A more formal definition of *fuzzy sets* is the following: a *fuzzy set A* is a subset of the universe of discourses χ that admits *partial membership*. The *fuzzy set A* is defined as the ordered pair:

$$A = \{x, m_A(x)\} \tag{2.47}$$

where $x \in \chi$ and $0 \le m_A(x) \le 1$. The *MF* $m_A(x)$ describes the degree to which the object x belongs to the *set A*, where $m_A(x) = 0$ represents *no membership*, and $m_A(x) = 1$ represents *full membership*. As an example, let χ represent the ages of all people. The subset A of χ that represents those people who are young is a *fuzzy set* with the *MF* shown in Fig. 2.16.

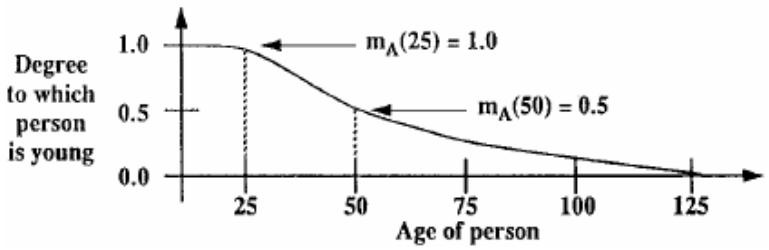


Fig. 2.16. This MF describes the relationship between a person's age and the degree to which a person is considered to be young. This MF determines that a 25 year old person belongs to A twice as much as a 50 year old person.

Assuming $A, B \in \chi$, the operations on *fuzzy sets* are defined as follows.

Comparison:

$$A = B \text{ iff } \quad m_A(x) = m_B(x) \qquad \forall x \in \chi \tag{2.48}$$

Containment:

$$A \subset B \text{ iff } \quad m_A(x) < m_B(x) \qquad \forall x \in \chi \tag{2.49}$$

Union: The union of two *fuzzy sets*, $A \cup B$, is found by combining the *MFs* of A and B. There have been several different union operations defined; the most common and by far the simplest union is defined as

$$m_{A \cup B}(x) = \max[m_A(x), m_B(x)] \qquad \forall x \in \chi \tag{2.50}$$

Intersection: like the *union*, the *intersection* of two *fuzzy sets A* and *B*, $A \cap B$, is found by combining the *MFs* of *A* and *B* and is defined as

$$m_{A \cap B}(x) = \min[m_A(x), m_B(x)] \qquad \forall x \in \chi \tag{2.51}$$

Complement: the *complement* of the *fuzzy set A*, $\overline{A}$, is defined as

$$m_{\overline{A}}(x) = 1 - m_A(x) \qquad \forall x \in \chi \tag{2.52}$$

In addition to these operations, De Morgan's laws, the distributive laws, algebraic operations such as addition and multiplication, and the notion of *convexity* have *fuzzy set* equivalents. By adopting the preceding definitions it is sometimes possible to fall in contradictions that couldn't even exist in case of "*crisp*" set theory. For example, given any set *A*, we cannot assume for sure anymore that $A \cup \overline{A} = U$, or $A \cap \overline{A} = \varnothing$.

2.5.2 Fuzzy Sets, Pattern Classes and Neural Nets

The combination of *fuzzy sets* and *pattern classification* has been examined by many people. Zadeh has stated the relationship between *pattern classes* and *fuzzy sets* and has shown how *hyperplanes* can be used to separate *fuzzy sets*. An evolution of these concepts has regarded the introduction of *fuzzy hyperplane decision boundaries* replacing the *crisp decision boundaries* of the *perceptron neural networks*. There has been a great amount of interest in the synergistic combination of *neural networks* and *fuzzy systems*. Many efforts have focused upon methods for implementing *fuzzy rules* in a *neural network* framework and techniques for parallelizing *fuzzy applications*. *Fuzzy sets* and *neural networks* can be effectively merged by utilizing *neural network neurons* as *fuzzy sets* and using *fuzzy set operations* during *learning* and *recall*.

2.5.3 Fuzzy min – max classification neural networks

In this chapter a *neural network classifier* that automatically creates *fuzzy classes* as aggregates of smaller *fuzzy sets* (*hyperboxes*) will be introduced. It deals with the so called *Fuzzy Min-Max Neural Network* (*FMMNN*) introduced by *Simpson*. *FMMNNs* are built around the so called *hyperbox fuzzy sets*. An *hyperbox* (parallelepiped with faces parallel to the coordinate planes of the chosen reference system defined in the input space) defines a region of the *n-dimensional pattern space* that contains *patterns* with *full class membership*. An *hyperbox fuzzy set* (*HB*) is then completely defined by its *min point*, its *max point* and a *MF* defined with respect to these points. *HBs* are aggregated to form a single *fuzzy set class*. We will see that the resulting structure fits naturally into a *neural network* framework; this is the reason why this *classification* system is called a *fuzzy min-max classification neural network*. *Learning* is performed by properly placing and

adjusting *HBs* (determining its *min-max points* and its *MF parameters*) in the *pattern space*, by means of an *expansion-contraction process* that can learn *non linear class boundaries* in a single pass through data and that provides the ability to incorporate new and refine existing *classes* without retraining. The *HB min-max points* are determined using a *fuzzy min-max learning algorithm*. *FMNN recall* operation consists of computing the *fuzzy union* of the *MF* values produced from each of the *fuzzy set hyperboxes*. The relevant properties that a *pattern classifier* should possess have motivated all the choices made in the development of the *fuzzy min – max classification neural network*. These properties are:

On-Line Adaptation

A *pattern classifier* should be able to learn new *classes* and refine existing *classes* without destroying old *class* information (*on-line learning* property). *off-line adaptation* should be avoided: each time new information is added to the *classification* system, a complete retraining of the system with both the old and the new information is required.

Nonlinear Separability

A *pattern classifier* should be able to build *decision regions* that separate *classes* of any shape and size.

Overlapping Classes

Pattern classes often tend to *overlap*. A *pattern classifier* should have the ability to form a *decision boundary* that minimizes the amount of *misclassification*, for all of the *overlapping classes*. The most prevalent method of *minimizing misclassification* is the construction of a *Bayes classifier*. Unfortunately, the building of a *Bayes classifier* requires knowledge of the underlying *probability density function* for each *class*, an information that is quite often unavailable. Moreover, *probability density functions* must be calculated *on-line*, in order to represent the current state of the data being received.

Training Time

A desirable property of a *pattern classification* approach capable to learn *nonlinear decision boundaries* is a short *training time*. This property, when combined with the *on-line adaptation* property, defines a formidable *classifier*.

Soft and Hard Decisions

A *pattern classifier* should be able to provide both *soft* and *hard classification decisions*. A *hard*, or *crisp*, *decision* is either 0 or 1. A *pattern* is either in a class or it is not. A *soft decision* provides a value that describes the degree to which a *pattern* fits within a *class*. As an example,

a *pattern classifier* for object recognition might not be trained to recognize colours. It is simply not possible to train a *classification* system with every possible combination of the three primary colours, but it is possible to train a system to recognize the *classes* of the primary colours and then use a *soft decision* process to determine the degree to which each of the three colours is present.

Verification and Validation

It is important that a *classifier*, *neural* or traditional, have a mechanism for verifying and validating its performance in some way. Contour plots, scatter plots, and closed-form solutions have all been used to perform this function.

Tuning Parameters

A *classifier* should have as few *parameters* to tune as possible.

2.5.4 Fuzzy Set Classes as Aggregates of Hyperbox Fuzzy Sets

Pattern classification can be described as in Fig. 2.17:

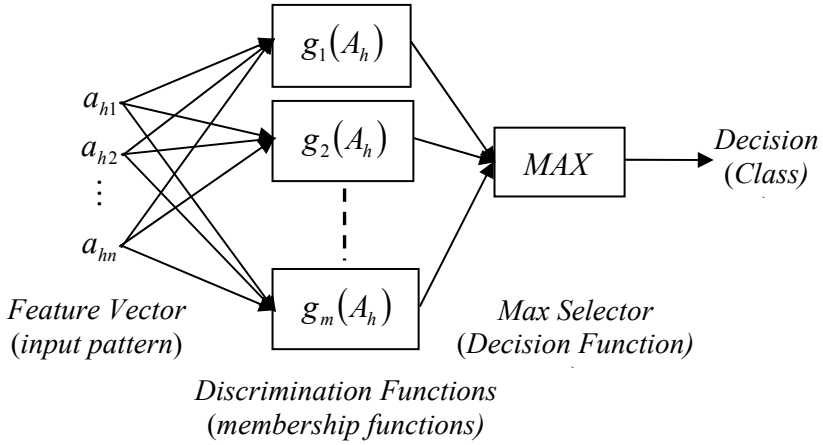


Fig. 2.17. Pattern classification system.

An *input pattern* A_h is passed through each of the *m discriminative*, or *characteristic*, *functions*, where each *discriminative function* represents a *pattern class*. The values of the *discriminative functions* are compared, and the one with the largest value is used to identify the *pattern class*. The *discriminative functions* have been implemented in many ways, including *probability density functions* (*Bayes classification*), *distance functions* (*SVM classification*), *fuzzy sets* and *neural networks*. As regards the *FMMNN classifier*, *HBs*, defined by pairs of *min-max points* and their corresponding *MFs*, are used to create *fuzzy subsets* in the *n-dimensional pattern space*. One of the most important advantages of this approach is that the majority of the processing is concerned with finding and fine-tuning the *boundaries* of the *classes*. By relegating these operations to simple

compare, add, and subtract operations, the resulting *learning algorithm* is extremely efficient. An illustration of the *min* and *max points* in a three-dimensional *HB* is shown in Fig. 2.18.

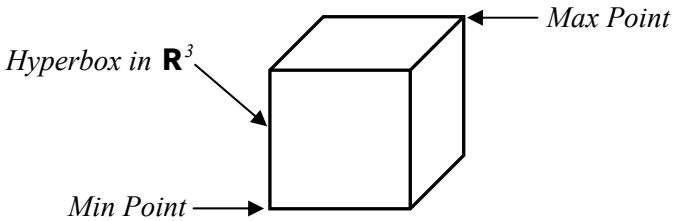


Fig. 2.18. The min-max hyperbox $B_j = \{V_j, W_j\}$ in $\mathbf{R}^3$.

Although it is possible to use *HBs* that have an arbitrary range of values in any dimension, we will consider the case of *normalized pattern features* (values that range from 0 to 1 along each dimension); hence the *pattern space* will be the n-dimensional unit cube I^n. A *MF* is associated with the *HB* that determines the degree to which any point (*pattern*) $\boldsymbol{x} \in \mathbf{R}^n$ (I^n) is contained within the box. In addition, it is typical to have the *membership* values range between 0 and 1, with no direct relationship to the range of *pattern feature* values.

For our purposes, a collection of *HBs* forms a *class*. The aggregation of several *HBs* in I^2 is illustrated for a *two-class problem* in Fig. 2.19:

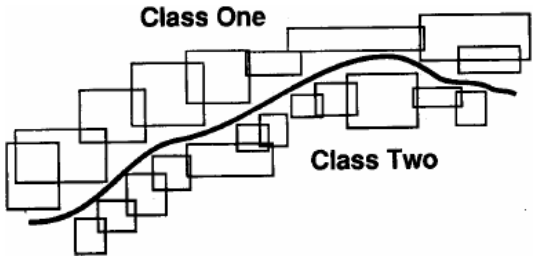


Fig. 2.19. An example of HBs placed along the boundary of two classes. The HBs are non-overlapping.

Let each *HB* fuzzy set, B_j, be defined by the ordered set:

$$B_j = \left\{\mathbf{X}, \mathbf{V}_j, \mathbf{W}_j, f\left(\mathbf{X}, \mathbf{V}_j, \mathbf{W}_j\right)\right\} \qquad \forall \mathbf{X} \in I^n \tag{2.53}$$

The aggregate of *fuzzy sets* that defines the k^{th} *pattern class* C_k is defined as:

$$C_k = \bigcup_{j \in K} Bj \tag{2.54}$$

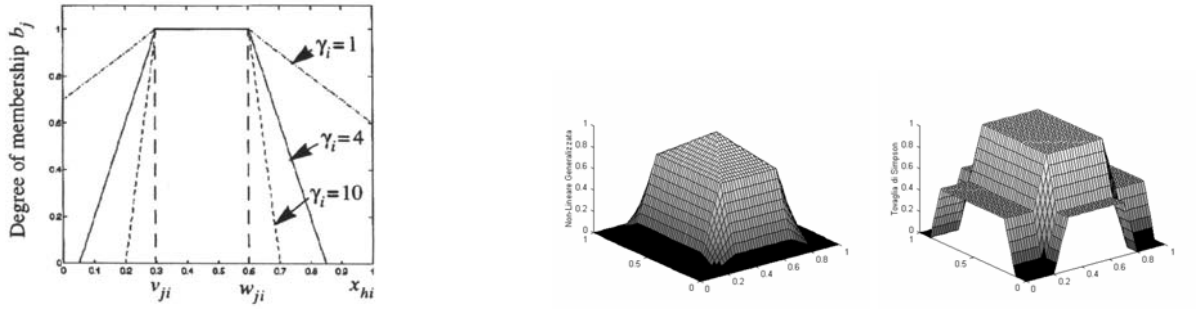
where K is the index set of the *HBs* associated with *class k*. Remember that the union operation in *fuzzy sets* is typically the max of all of the associated *fuzzy set MFs*. The *learning algorithm* developed for this family of *classifiers* allows *overlapping HBs* from the same *class* and eliminates the overlap between *HBs* from different *classes*. This does not mean that the *fuzzy sets* do not overlap, but only that those portions of the *fuzzy set* representing *full membership* are non-overlapping. Using this configuration, it is possible to define *crisp class boundaries* as a special case. These *class boundaries* are defined as those points where the *membership* values are equal.

2.5.5 Hypercube Membership Functions

The *MF* for the j^{th} *HB*, $b_j(A_h)$, where $0 \le b_j(A_h) \le 1$, must measure the degree to which the h^{th} *input pattern* A_h falls outside of the *HB* B_j. On every dimension, $b_j(A_h)$ can be considered a measure of how far each *feature* is greater (less) than the *max* (*min*) point value along each dimension that falls outside the *min-max bounds* of the *HB*. Moreover, as $b_j(A_h)$ approaches 1, the point should be more contained by the *HB*, with the value 1 representing complete *HB* containment. The function that meets all these criteria is the sum of two complements – the average amount of *max point* violations and the average amount of *min point* violations. The resulting *MF* is defined as

$$b_j(A_h) = \frac{1}{2n}\sum_{i=1}^{n}\left[\max\left(0,1-\max\left(0,\gamma\min\left(1,a_{hi}-w_{ji}\right)\right)\right)+\max\left(0,1-\max\left(0,\gamma\min\left(1,v_{ji}-a_{hi}\right)\right)\right)\right] \qquad (2.55)$$

where $A_h = (a_{h1}, a_{h2}, \ldots, a_{hn}) \in I^n$ is the h^{th} *input pattern*, $V_j = (v_{j1}, v_{j2}, \ldots, v_{jn})$ is the *min point* for B_j, $W_j = (w_{j1}, w_{j2}, \ldots, w_{jn})$ is the max point for B_j, and γ is the *sensitivity parameter* that regulates how fast the *membership* values decreases as the distance between A_h and B_j increases. Examples of *MFs* for one and two dimensions are shown in Fig. 2.20:



(a) (b)

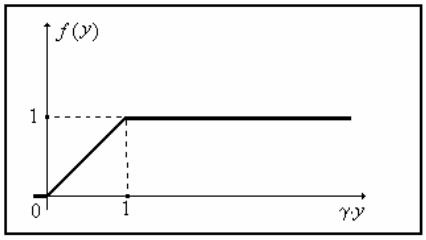
Fig. 2.20. (a) mono dimensional MF ; (b) The MF in two dimensions (the Simpson's "tablecloth") . The min point is at V_j = (0.2, 0,2) and the max point is at W_j = (0.3,0,4). The sensitivity parameter γ = 4 is set to produce a moderately quick decrease from full membership to no membership.

A different definition of the *MF* can be the following (*degree of membership* of a *pattern* to an *HB*):

$$b_j(A_h) = \frac{1}{n}\sum_{i=1}^{n}[1 - f(x_{hi} - w_{ji}) - f(v_{ji} - x_{hi})] \qquad (2.56)$$

Where *f* is a function $f : \mathbf{R} \rightarrow [0\ldots1]$ defined as follows:

$$f(y) = \begin{cases} 1 & , \text{ if } \gamma \cdot y > 1 \\ \gamma \cdot y & , \text{ if } 0 \le \gamma \cdot y \le 1 \\ 0 & , \text{ if } \gamma \cdot y < 0 \end{cases} \qquad (2.57)$$



2.5.6 FMMNN Classifier implementation

Considering the *fuzzy min-max classifier* as a *neural network*, we can appreciate its parallel nature

and provide a mechanism for a fast and efficient implementation. The *three-layer neural network* that implements the *fuzzy min-max classifier* is shown in Fig. 2.21:

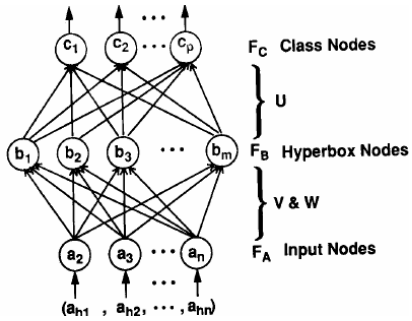


Fig. 2.21. The neural network that implements the fuzzy min-max classifier.

The topology of this *neural network* grows to meet the demands of the problem. The *input layer* $F_A = (a_1, a_2, \ldots, a_n)$ has *n* processing elements, one for each of the *n* dimensions (*features*) of the *input pattern* A_h. Each *node* in the $F_B = (b_1, b_2, \ldots, b_m)$ *layer* represents a *HB*. There are two sets of *connections* between each *input node* from *layer* F_A and each *node* in *layer* F_B and they represent the *min* and the *max points* (the *min vector* V_j and the *max vector* W_j). The *min points* are stored in the matrix **V** and the max points are stored in the matrix **W**. These dual *connections* are adjusted using the *fuzzy min-max classification learning algorithm* that will be introduced later. The transfer function for each *node* of F_B is the *HB MF* defined in (2.55). Each $F_C = (c_1, c_2, \ldots, c_p)$ *node* represents a *pattern class*. The connections between the *layer* F_B *nodes* and the *p output nodes* in *layer* F_C are binary valued and are determined as each F_B node is added during *learning*. They are stored in the matrix **U**. The equation for assigning the values to the F_B to F_C *connections* is

$$u_{jk} = \begin{cases} 1 & \text{if } b_j \text{ is a hyperbox for class } c_k \\ 0 & \text{otherwise} \end{cases} \tag{2.58}$$

where c_k is the k_{th} F_C *node*. A detailed view of the j^{th} F_B *node* b_j is shown in Fig. 2.22.

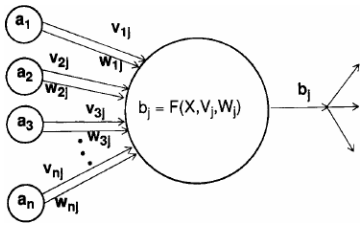


Fig. 2.22. The implementation of a hyperbox and its associated membership function as a neural network assembly.

The output of the F_C *node* is the degree to which the input pattern A_h fits within the *class k*. The transfer function for each of the F_C *node* performs the *fuzzy union* of the appropriate *HB values*:

$$c_k = \max_{j=1}^{m}\left(b_j u_{jk}\right) \tag{2.59}$$

for each of the p F_C *nodes*. The outputs of the F_C *class nodes* can be exploited in two main ways:

Soft decision

the outputs are utilized directly

Hard decision

the F_C *node* with the highest value is selected and its value is set to 1, to indicate that it is the closest *pattern class* (the remaining F_C *node* values are set to 0).

This last operation is commonly referred to as a *winner-take-all* response.

2.5.7 Learning Algorithm

It consists of an *expansion/contraction* process. The *training set D* consists of a set of *M* ordered pairs $\{\boldsymbol{X}_h, d_h\}$, where $\mathbf{X}_h = \left(x_{h1}, x_{h2}, \ldots, x_{hn}\right) \in I^n$ is the *input pattern* and $d_h \in \{1, 2, \ldots, m\}$ is the index of one of the *m classes*. Note that **X** and **A** are both used to represent *input patterns*. The *learning process* begins by selecting an ordered pair from *D* and finding a *HB* for the same *class* that can *expand* (if necessary) to include the input. If a *HB* cannot be found that meets the *expansion* criteria, a new *HB* is formed and added to the *NN*. This growth process allows *classes* to be formed that are *non-linearly separable*, it allows existing *classes* to be refined over time, and it allows new *classes* to be added without retraining. During *expansion*, some *HBs* can *overlap*. This *overlap* is not a problem when it occurs between *HBs* representing the same *class*. On the contrary, when *overlap* occurs between *HBs* that represent different *classes*, it must be eliminated using a *contraction process*. Note that the *contraction process* eliminates only the *overlap* between portions of *HBs* from separate *classes* that have *full membership*. There is still *overlap* between non unit valued members of each of the *HBs*. Thanks to the "*on-line*" nature of the algorithm, *patterns* are processed in their order of presentation. In the beginning, the part of the *hidden layer* of the *network* that's relative to *class k* is empty. When the first *pattern* $\boldsymbol{X}_1$ is presented to the *network*, the algorithm creates an *HB* with its *min – max points* that coincide (*degenerate HB*). In summary, the *fuzzy min-max classification learning algorithm* is a three-step process, in one *training epoch*:

1) Expansion

Identify the *HB* that can expand and expand it. If an expandable *HB* cannot be found, add a new *HB* for that class.

2) Overlap Test

Determine if any *overlap* exists between *HBs* from different *classes*.

3) Contraction

If so, eliminate the *overlap* by minimally adjusting each of the *HBs* involved.

2.5.7.1 Hyperbox Expansion

Given an ordered pair $\{\boldsymbol{X}_h, d_h\} \in D$, find the *HB* B_j that provides the highest *degree of membership*, that allows *expansion* (if needed) and that represents the same *class* as d_h. The *degree of membership* is measured using (2.55). The maximum average size of a *HB* is upper bounded by a user-defined value $0 \leq \theta \leq 1$. We can consider the advisability of including or not a new *pattern* of the *training set* into an existing *HB*, by adopting one of the following three criteria:

THETA RULE

Given the h^{th} *pattern* $\boldsymbol{X}_h$ of the *training set* and the j^{th} *HB* B_j, the latter will be expanded till it can contain the new *pattern*, if and only if:

$$n\theta \geq \sum_{i=1}^{n} \left(\max\left(w_{ji}, x_{hi}\right) - \min\left(v_{ji}, x_{hi}\right) \right) \tag{2.60}$$

or, in other words, if and only if the average length of the edges of B_j along all the dimensions is, after the *expansion*, less or equal a given value of a *parameter* $\theta \in [0,1]$ that rules the *training process*, is fixed before it begins and is the same for all the *hyperboxes* of a *class*. This is the rule proposed by *Simpson*.

DELTA RULE

Given the h^{th} *pattern* $\boldsymbol{X}_h$ of the *training set* and the j^{th} *HB* B_j and defined as $\underline{c}_j$ the *centroid* of B_j (average of the *patterns* already included into it), B_j will be expanded till it can contain the new *pattern*, if and only if:

$$\left\| \underline{c}_j - \underline{\boldsymbol{X}}_h \right\| \leq \delta \tag{2.61}$$

or, in other words, if and only if the h^{th} *pattern* lies inside an *hypersphere* with radius δ and centre in $\underline{c}_j$. *Parameter* δ must be fixed before the *training* process begins and rules it as regards *class k*. The range of values that can be tolerated for this *parameter*, in case of an *n*-dimensional problem, is $[0;\sqrt{n}]$. As a matter of fact, $\sqrt{n}$ is the greatest possible distance between two points, inside a unit *n*-dimensional *HB*. The behaviour of the algorithm varies with the increase of both *parameters* θ and δ: high values lead to few very large *HBs*, each containing a lot of *patterns*, while low values cause the forming of a generally higher number of less dense *HBs*. In both cases, the *HB* with the

smaller distance between the its *centroid* and its *patterns* is chosen. Unlike *parameter* θ, *parameter* δ affects the expansion along each dimension in a way that's independent from all the other dimensions.

BETA RULE

Given the h^{th} *pattern* $\mathbf{X}_h$ of the *training set* and the j^{th} *hyperbox* B_j, the latter will be expanded till it can contain the new *pattern*, if and only if:

$$b_j(X_h) \geq \beta \tag{2.62}$$

or, in other words, if and only if the *fuzzy membership* of $\mathbf{X}_h$ in B_j, is grater then a given value $\beta \in [0;1]$, that must be fixed before the *training* process begins; it rules *training* as regards *class k*. While in the previous rules the inclusion was affected by essentially geometric quantities (the extension of the *HBs*), in this case, inclusion is subject to the *fuzzy* proximity of a *pattern* to an *HB*. Therefore, the meaning of this rule is immediate, since it's easier to a priori fix a *fuzzy degree of membership*, rather then a value representing the maximum average expansion capability of an *HB*. The main drawback of this approach is revealed in case of uniform *pattern* distributions, since, during the *training* phase, the behaviour is always the same, regardless the new *pattern* involved and the current dimensions of an *HB*. This is due to the fact that the decay slope of the *MF* (affected by the value of γ only) does not depend on the dimensions of the *HB*; therefore, in case of uniform distributions, the processing of the first *pattern* gives the same results of the processing of any other following *pattern*. On the contrary, "geometric" rules show some sort of memory, thanks to which the inclusion of a new *pattern* into an *HB* doesn't depend on the position of that *pattern* with respect to the *min - max points* of that *HB* only, but even on the conditions of the *HB* at inclusion time.

If the *expansion criterion* has been met for *HB* B_j, the *min point* and the *max point* of the *HB* are adjusted using respectively the equations:

$$\begin{aligned} v_{ji}^{new} &= \min\left(v_{ji}^{old}, x_{hi}\right) \qquad \forall i = 1,2,\ldots,n \\ w_{ji}^{new} &= \max\left(w_{ji}^{old}, x_{hi}\right) \qquad \forall i = 1,2,\ldots,n \end{aligned} \tag{2.63}$$

2.5.7.2 Hyperbox Overlap Test

As previously stated, it is necessary to eliminate *overlap* between *HBs* that represent different *classes*. To determine if a previous *expansion* created any *overlap*, a dimension by dimension comparison between *HBs* is performed. If, for each dimension, at least one of the following four cases is satisfied, then *overlap* exists between two *HBs*. Assume that the *HB* B_j was expanded in the previous step and that the *HB* B_k represents another *class*; let's test for possible *overlap* between B_j

and B_k. While testing for the *overlap*, the smallest *overlap* along any dimension, δ, and the index of the dimension involved, Δ, is saved to be used during the *contraction* portion of the *learning process*. Assuming initially $\delta^{old} = 1$, the four test cases and the corresponding minimum *overlap* value for the i^{th} dimension are as follows:

$$\textit{Case 1}: \quad v_{ji} < v_{ki} < w_{ji} < w_{ki} \quad \Rightarrow \quad \delta^{new} = \min\left(w_{ji} - v_{ki}, \delta^{old}\right) \tag{2.64}$$

$$\textit{Case 2}: \quad v_{ki} < v_{ji} < w_{ki} < w_{ji} \quad \Rightarrow \quad \delta^{new} = \min\left(w_{ki} - v_{ji}, \delta^{old}\right) \tag{2.65}$$

$$\textit{Case 3}: \quad v_{ji} < v_{ki} < w_{ki} < w_{ji} \quad \Rightarrow \quad \delta^{new} = \min\left(\min\left(w_{ki} - v_{ji}, w_{ji} - v_{ki}\right), \delta^{old}\right) \tag{2.66}$$

$$\textit{Case 4}: \quad v_{ki} < v_{ji} < w_{ji} < w_{ki} \quad \Rightarrow \quad \delta^{new} = \min\left(\min\left(w_{ji} - v_{ki}, w_{ki} - v_{ji}\right), \delta^{old}\right) \tag{2.67}$$

If $\delta^{new} < \delta^{old}$, then $\Delta = i$ and $\delta^{old} = \delta^{new}$ signifying that there was *overlap* for the Δ^{th} dimension and *overlap* testing will proceed with the next dimension. If not, the testing stops and the *minimum overlap* index variable is set to indicate that the next *contraction step* is not necessary, i.e., $\Delta = -1$.

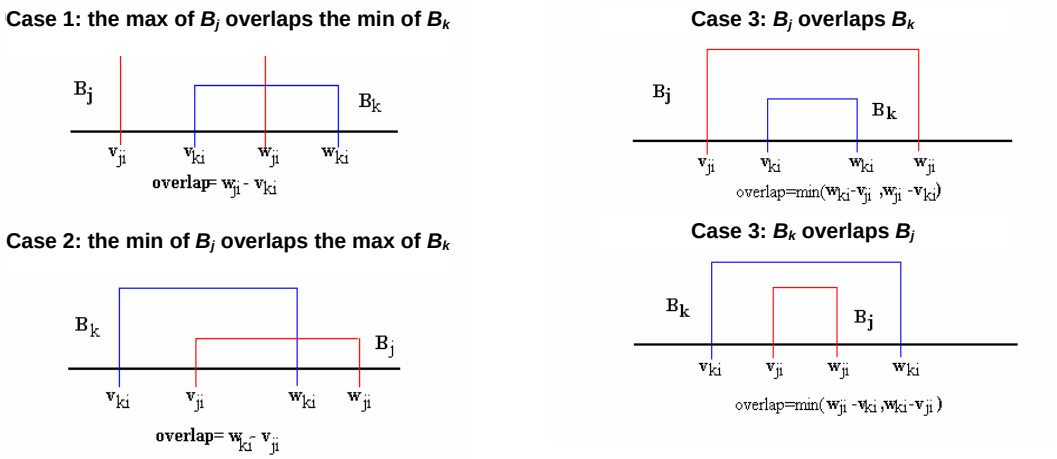


Fig. 2.23. The four cases of overlap for the i^{th} dimension.

2.5.7.3 Hyperbox Contraction

If $\Delta > 0$, then only the Δ^{th} dimensions of the two overlapping *HBs* are adjusted. Only one of the *n* dimensions is adjusted in each of the *HBs*, in order to keep the *HB* size as large as possible and to minimally impact the shape of the *HBs* being modified. It is in fact plausible that this approach can be relied on to provide more robust *pattern classification*. Even in case of *contraction* operation we can have four cases:

$$\textit{Case 1}: \quad v_{j\Delta} < v_{k\Delta} < w_{j\Delta} < w_{k\Delta} \quad \Rightarrow \quad w_{j\Delta}^{new} = v_{k\Delta}^{new} = \frac{w_{j\Delta}^{old} + v_{k\Delta}^{old}}{2} \tag{2.68}$$

$$\textit{Case 2}: \quad v_{k\Delta} < v_{j\Delta} < w_{k\Delta} < w_{j\Delta} \quad \Rightarrow \quad w_{k\Delta}^{new} = v_{j\Delta}^{new} = \frac{w_{k\Delta}^{old} + v_{j\Delta}^{old}}{2} \tag{2.69}$$

Case 3a:
$$\begin{matrix} v_{j\Delta} < v_{k\Delta} < w_{k\Delta} < w_{j\Delta} \\ \left(w_{k\Delta} - v_{j\Delta}\right) < \left(w_{j\Delta} - v_{k\Delta}\right) \end{matrix} \Rightarrow \quad v_{j\Delta}^{new} = w_{k\Delta}^{old} \tag{2.70}$$

Case 3b:
$$\begin{matrix} v_{j\Delta} < v_{k\Delta} < w_{k\Delta} < w_{j\Delta} \\ \left(w_{k\Delta} - v_{j\Delta}\right) > \left(w_{j\Delta} - v_{k\Delta}\right) \end{matrix} \Rightarrow \quad w_{j\Delta}^{new} = v_{k\Delta}^{old} \tag{2.71}$$

Case 4a:
$$\begin{matrix} v_{k\Delta} < v_{j\Delta} < w_{j\Delta} < w_{k\Delta} \\ \left(w_{k\Delta} - v_{j\Delta}\right) < \left(w_{j\Delta} - v_{k\Delta}\right) \end{matrix} \Rightarrow \quad w_{k\Delta}^{new} = v_{j\Delta}^{old} \tag{2.72}$$

Case 4b:
$$\begin{matrix} v_{k\Delta} < v_{j\Delta} < w_{j\Delta} < w_{k\Delta} \\ \left(w_{k\Delta} - v_{j\Delta}\right) > \left(w_{j\Delta} - v_{k\Delta}\right) \end{matrix} \Rightarrow \quad v_{k\Delta}^{new} = w_{j\Delta}^{old} \tag{2.73}$$

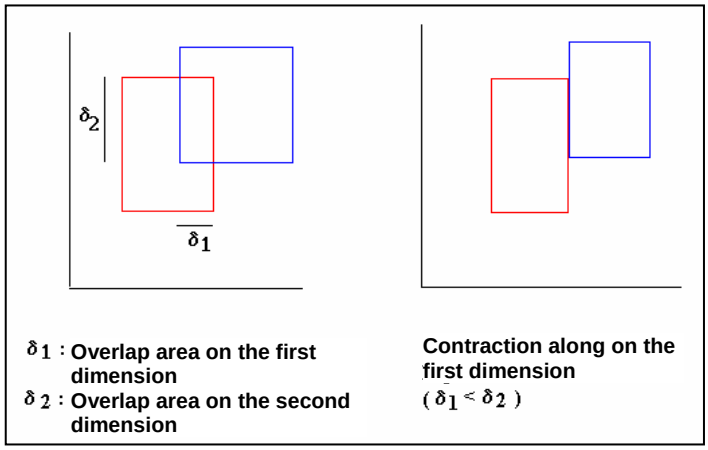


Fig. 2.24. contraction process in the bidimensional case (n = 2).

At the end of the *training* process, a point of the *training set* will be considered "strictly" belonging to an *HB*, if the *MF* of that *HB*, evaluated on that point, will give a unitary value; on the contrary, the points considered external to that *HB* will show as much *membership* as much the higher the distance from that *HB*, according to the parameter γ, responsible of the *degree of fuzziness* that characterizes the *MF*. The two *parameters* γ and θ rule the *training process* in different ways. γ has an influence on the definition of the *decision regions* only, being the creation of the *HBs*, their number and extension, totally independent of it. The *training* process is based solely on pure geometric concepts that have nothing to do with the *fuzzy* nature of the *classifier*, while it depends on the value of the parameter θ only. After the *training* has been accomplished, each *HB* is "covered" with a *MF*, in the same way of a "tablecloth", starting from 1, in correspondence of the points with *full membership*, and degrading up to 0 outside the *HB*, with a slope that depends on the parameter γ only. Once a *cost function* that quantifies the quality of the *partition* of the *input space* operated by the *training* process has been defined, it's possible to fix parameter γ to a reasonable default value and look for the *optimal partition*, according to that *cost function*, while letting *parameter* θ to vary inside the range [0…1].

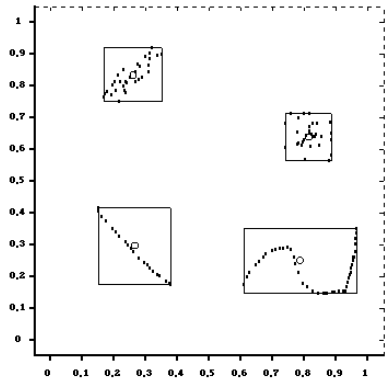


Fig. 2.25. Examples of some clustering attempts.

2.5.8 In search of the optimum

With the aim of *optimizing* the *generalization capabilities* of a *classifier*, we will base our argumentation on a well known principle: the *Occam's Razor*. According to it, the model characterized by the best *generalization capability*, under the same performances on the *training set*, is the one that shows the lowest *structural complexity*. In what follows, we aim to the development of an automatic *optimization* process of our *classifiers*, just based on this simple principle. The methods that face up to the solution of this problem all agree upon a common strategy, according to which the structure with the best *generalization capability* is the one that minimizes the following function:

$$F = E_s + \lambda E_c \tag{2.74}$$

where E_s is an estimation of the *global* error on the *training set*, E_c is a quantity that measures the complexity of the *network* and λ is a real positive number introduced to give a different weight to both the contributions. Since we are only interested in values of λ ranging from 0 to 1, we prefer to adopt the following *cost function*:

$$F = (1 - \lambda) E_s + \lambda E_c \tag{2.75}$$

For the success of this approach, a correct choice for the above mentioned measures is essential. In *classification* problems, a measure of the performances can be suitably given starting from the percentage of errors made over the *training set*. Moreover, we are not so far from the truth if we consider the *complexity* of the system as a function of the *parameters* that influence its behaviour. The correct balancing between the measure of the *complexity* and that of the *error* is of fundamental importance, so that an increase in the *complexity*, ΔE_c, of the *network* entails a decrease of the *error* that's proportional to ΔE_c. As a matter of fact, the *Min-Max algorithm* fortunately lends itself to a simple and efficient definition of the measure of *complexity*. It is obvious that the *complexity* of the *network* depends essentially on the number of *HBs* formed and in general it is greater when the

points in the *training set* are particularly hard to be covered. Besides, this *complexity* depends on the *parameter* θ, that influences the advancement of the *training* process. The value of this parameter spans the range [0…1] and is in charge of the maximum extension of the *HBs* along all the dimensions. *Trainings* with small values for θ lead to a huge number of *HBs*, while higher values for this parameter guarantee the forming of a little number of *HBs*, each of a greater extension with respect to the previous situation.

Let's consider a *FMMNN* with *n* inputs, *p* outputs and *m neurons* in the *hidden layer*. The number of *connections* between the first and the second layer is equal to $2 \cdot n \cdot m$, while the number of *connections* between the second and the output layer is equal to $m \cdot p$. Therefore, given a *training set* (that is, *n* and *p*), the number of *connections* is proportional to the number of *neurons* in the *hidden layer*. This is why we define E_c as the percentage of *HBs* with respect to the total number of *patterns* in the *training set* (this choice imposes both the addends of function F to vary in the range[0…100]. Therefore, in what follows we will exploit the function F in order to find the optimum value for the *parameter* θ of the *Simpson's classifier*. So, let's take into account a set $S_\theta = \{\theta_1, \theta_2, \ldots, \theta_q\}$ of samples of θ, generated by any sampling algorithm of the interval [0, 1]. We call $MM(\theta_j)$ the *Min-Max model* generated by sample θ_j, characterized by a value for the *complexity* of $E_c(\theta_j)$ and by a measure for the *error* of $E_s(\theta_j)$, given the *training set* S_{tr}. We define as the optimum value θ_{opt} for *parameter* θ the one that minimize the function F, that is:

$$\theta_{opt} = \theta_j \in S_\theta : F(\theta_{opt}) = \min_{1 \le j \le q}(F(\theta_j)) \tag{2.76}$$

where:

$$F(\theta) = \lambda \cdot E_C(\theta) + (1 - \lambda) \cdot E_S(\theta) \tag{2.77}$$

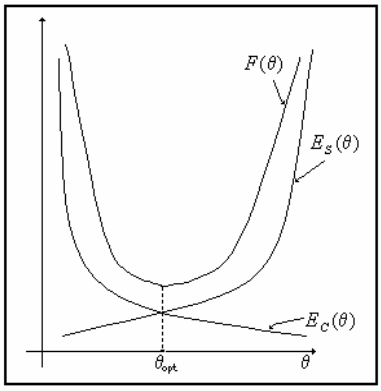


Fig. 2.26. qualitative shape of a cost function.

This way, in what follows we define a new *learning algorithm*, that we'll call *Optimized Min-Max* (*OMM*). We have basically replaced parameter θ with the new parameter λ, the numerical values of which have a more explicit meaning, as it allows us to rule the *learning* process as regards the quality of the *classifier* we need. If we need minimum complexity networks, we must choose values

for λ that are close to unity, while if we need the lower error on the given training set, λ must be close to zero. Unfortunately, the *OMM* algorithm shares some of the drawbacks that were already present in the original algorithm:

- *learning depends too much on the order of arrival of the patterns*;
- the tie imposed by the value of θ on the maximum dimension allowed to any *HB* to reach is constant all over the input space. This restriction reduces the flexibility in the "covering" of the input *patterns* by means of the *HBs*, since the value given to *parameter* θ is the same for any *class*, whatever the distribution of patterns inside of it, with a consequent degradation of the *generalization capability* of the generated models.

These drawbacks are due just to the *expansion*/*contraction* mechanism of the *HBs*, inherited by the original *learning* algorithm. As an example, let's assume that we have two *classes*, one with a high concentration of *patterns* in various compact regions of the *input space*, while the other is characterized by a lower concentration, with *patterns* distributed in regions of grater dimensions. Any optimum value for θ should be the one that represents the best compromise between the value that best "covers" the first *class* and a values that's best suited to the second *class*. The former should be as little as possible, maybe close to zero, while the latter should be almost equal to one, with object of forming larger *HBs* that those formed in case of the first *class*. So, a compromise could be really inadequate, both for the first and the second *class*. This is why we concentrated our efforts in the development of some sort of algorithms that could allow us to define the best value of *parameter* θ for every single *class*, by "covering" it apart from the "covering" of the other classes. This matter will be dealt with in the following chapter when we will introduce the so called *parallel clustering*. Actually, it's not so convenient to go too far with the parallelization of the single problems of *clustering*. It is in fact convenient to take into account the cross information contained in the given *training set* that effectively make a connection among the outputs of all the *clustering* operations. In the following example, we have two *classes A* e *B* with *patterns* distributed in two concentric regions with two almost coincident barycentre.

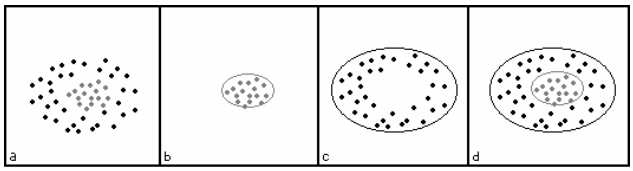


Fig. 2.27. two concentric classes.

Every clustering system would "cover" its class in the way depicted in Fig. 2.27, leading to a situation in which the inner *cluster*, the one regarding *class B*, turns out to be completed absorbed

by the outer one, the one regarding *class A*, till it disappears. This means that, in the *decision region*, all the points inside the larger *cluster*, the points of *class B* included, will be considered as part of *class A*, even if they belong to *class B* only. This happens because, during the *clustering* of *class A*, the "hole" at the centre of the *class* is not taken in any consideration, because the *clustering* process does not have any information about it. To solve this problem, one can think to enrich the *training* procedure by adding a further step during which the output of all the *clustering* processes are reorganized on the basis of the mutual relations that exist among the *classes*. Some coefficients are introduced, in order to opportunely weight those outputs; the weights are then iteratively updated, starting from the information about all the *misclassifications* encountered during tests.

2.5.9 Output Interface

The coding of the output of the *classifier* depends on the particular application. In case "*crisp*" values for the *class labels* are required, the outputs of the *classifier* need to be *defuzzified*, that is they must supply a {0,1} value instead of a *fuzzy* value. This can be easily accomplished resorting to a *WTA* mechanism. Obviously, a *defuzzification* provokes a loss of information to the *classifier*, in order of its knowledge about the *input space* in terms of *degree of membership* of a *pattern* to a given *class*. In case the actual application requires the complete *fuzzy* information given by the outputs of the *classifier*, the only operation to accomplish is the adaptation of those outputs to the range of values required as inputs by the system below the *classifier* itself. Therefore, given the continuous nature of the problem, the binary outputs supplied by the *WTA* approach would be clearly unsuitable.

2.6 The Parallel Clustering Classifier

The *parallel clustering classifier* represents *clusters* by means of the *HBs* introduced for *Simpson's classifier*: *clusters* are *hyperboxes*. We know the great advantages of making this choice, due to the low number of parameters needed to describe an *hyperbox* (*2n*, if *n* is the number of dimensions of the input space) and to the relatively trivial method that is required to verify the *membership* of a given *pattern* to a given *class* (simple comparison operations). As stated above, by means of *parallel clustering*, we are looking for the *optimal partition* (or "covering") of a single *class*, where this operation is carried out in a totally independent way of the *clustering* of other *classes*. This allows us to *train* the *network*, that is to single out the "*covering*" *HBs* of a given *class*, taking values for the parameter θ that best suite to that *class*. The partition of the *input space*, as regards the k^{th} *class* of the *training set*, is up to a single dedicated *clusterizer*. *Training* is carried out by means of a *constructive unsupervised* procedure that builds up that portion of the final *network* that contains all and only the *clusters* pertaining to that *class* and that's inspired by the *training* procedure originally proposed by Simpson. Generally, a *clustering* problem can be faced by breaking it up in two steps. The first one consists in the association to every *pattern* in the *training set* of information about the *cluster* (*HB*) to which that *pattern* belongs to (we will call this information "*address"* of the *pattern*; it's substantially a relevant point inside the *HB*, for example the *centroid*). This operation yields a list where every *pattern* matches the address of its *cluster*. Nevertheless, this table doesn't contain nor any information about the points of the *input space* that don't belong to the *training set*, neither any information about the geometry of the *clusters* involved. The input space is generally unknown or locally known only (in discrete form). The second step requires the assignment of a physical structure to the *clusters*, in order to evaluate their effectiveness in managing the points of the whole *input space*, making continuous the discrete information we have. In case of a *fuzzy system*, this second step consists in two further sub-steps: the choice of an opportune geometric shape for the *clusters* and afterwards the "covering" of those *clusters* with an opportune *MF*.

2.6.1 The Partition of the Input Space: Decision Boundaries

Once, for every *class* in the *training set*, we have "*covered"* all the points of a *class* with the opportune *clusters*, the problem arises about how a *decision boundary* between two *classes* can be located. Let's suppose to have two *clusters* A and B (Fig. 2.28). It's not needed that they belong to different *classes*, since, in some situations, the focus of the discussion could be the influence of *clusters* on the input space, instead of *decision boundaries*. Besides, let's suppose that *cluster A* be

more dense (high number of points, for example $n_A = 100$ with respect to $n_B = 20$, and little variance) than *cluster B*.

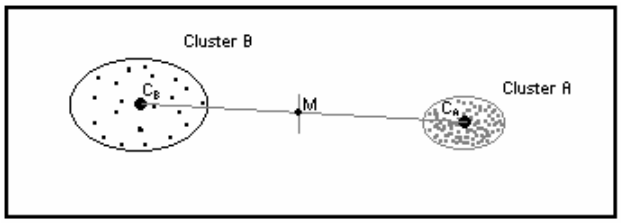


Fig. 2.28. Two generic Clusters.

We have previously seen that the original *Simpson's algorithm* partitions the *input space* thru geometrical entities called *HBs*. The single *clusterizer* carries out a similar task with the aim of infering geometric information about those object that will afterward be the *clusters* of the *partition*. The *learning algorithm* applied to the single k^{th} *class clusterizer* consists in a sequence of *HB creation/expansion* processes following the presentation to the *network* of the *patterns* A_h belonging to the k^{th} *class* itself. Due to the "*on-line*" nature of the algorithm, *patterns* are processed in the order of presentation. In Fig. 2.28, besides the two *clusters*, their *centroids* (C_A and C_B), the link and the mid point *M* between them are depicted. Now, we ask ourselves where can we put the boundary marker dividing the zone of influence of the two *clusters*, along the link between the two *centroids*, and why. We can define at least three decision criteria, each of which assigns a different meaning to the above mentioned zone of influence. Agreed that the internal regions of each *cluster* (the ellipses in Fig. 2.28) have to necessarily belong to the *cluster* itself, what we are asked to decide is how we must *partition* the space out of those *clusters*. In other words, we must decide how to locate the intersection point between a *decision boundary* and the line that connects the two *centroids*. The three approaches are the followings:

- *"Indifferent" Approach*: all the *clusters* are equally important, whatever their dimension and the density of *patterns* inside of them. Here, the intersection point coincides with the mid point *M* and each *cluster* imposes its influence over the space surrounding it as equally as all the other *clusters*.

- *"Geometric" Approach*: the importance of a *cluster* is proportional to its extension. In the example of Fig. 2.28, *B* is larger then *A*; consequently the zone of influence assigned to *B* will be bigger and the intersection point will be moved towards *A*.

- *"Gravitational" Approach*: the importance of a *cluster* is proportional to its "weigth", that is to the number of *patterns* inside of it, a number that we can think as the "mass" of the *cluster*. Then, we can rely on Newton's Universal Gravitation Low: given a celestial body,

the surrounding gravity field it generates depends on its mass only and not on its dimensions. Therefore, in the example of Fig. 2.28, the *boundary* will intersect the link between C_A and C_B, in a point that's closer to A then to B. This means that the influence of A on the outer space will be higher than the one imposed by B.

Which of the three approaches is better to adopt depends on the actual *classification* problem we are trying to solve and on the particular application the *clusterizer* is implicated in. Within this work, for generalization purposes, we have chosen to follow both the *geometric* and *gravitational* approaches.

2.6.2 Indices of Cluster Validity

Let's suppose to have k *classes*, and so k *clusterizers*. The strength of the approach we are describing consists in the intrinsic parallelism of the architecture adopted, where a certain number of basic processors (the *neurons*) is connected to solve the *clusterization* of one and only *class*. Besides the strict gain in terms of computational speed, once the value of the relative parameter θ has been fixed, this solution allows us to construct a *clusterizer* that is capable to carry out, as best as it can (as regards the best choice of parameter θ), the partition of the *class* it has been assigned to. Once k of these values of θ have been fixed, at the end of the *clustering* process, the *decision region*, based on the given *training set*, will be formed. It's clear that the *validity* of each *clusterization* will heavily depend on the values of θ assigned to each *class*. Consequently, these values have to be precisely chosen with regard to the real distribution of the *patterns* inside each *class*. Once we have collected a certain number of *partitions* produced by different values for θ, or different *clustering* strategies, we must find a method that allows us to decide which is the structure that's best suited to the *training set* for the *class* involved (*cluster validity*). Nevertheless, the *cluster validation method* cannot be based on the *misclassification errors*. In fact, each *classifier* must supply an *optimized partition* before a cost function could be estimated. As a matter of fact, we are able to take into consideration the *classification* in its entirety and to calculate the *errors* committed only after all the *classifiers* have performed their tasks and the information they supply have been integrated. Essentially, the partition of a single *class* is of an *unsupervised* kind. So we must find a way to optimize the single *clusters*, by adopting some *excellence criteria* that take into account some important characteristics that the *cluster* should a priori enjoy. Any approach can be adopted, as long as it's *unsupervised*. For this purpose, *cluster validation methods* make use of some *indices* for the quantitative estimation of the results given by the *clustering* operation, by measuring in some way the "quality" of the *partition* obtained. There are three approaches for

validating the *clusters*. The first one (*external criterion*) validates the performances of a *clustering* methodology on the basis of a predefined structure given to the *data set* that reflects the intuitions of the user about the real *clustering* structure of data. The second approach (*internal criterion*) validates the *clustering* in terms of quantities extracted from the *data set* itself. The third approach (*relative criterion*) makes a comparison among the partitions given by a single *clustering* method, but varying the values of the *parameter*. For the *indices* described later, the choice of the *best partition* is accomplished taking into account two aspects of the geometry of a *partition*:

- *compactness* : the objects inside a *cluster* should be as close to each other as possible.
- *separability* : *clusters* should be as far from each other as possible.

A *classification* will be as much as better as the *classes* it identifies are compact and separate. Some of the most used and appreciated *validity indices* are the following:

A. Davies-Bouldin Index (DI)

Given a *partition*, it is evaluated starting solely from the information given by the *pattern address table* (so, it doesn't depend on the geometric shape of the *clusters*). The geometric information, connected to the structure of the *partition*, comes into the calculation of the *index* through the distances existing among the *patterns* and among these and the *cluster centroids* (points that coincide with the barycentre of the "clouds" of *patterns* belonging to a *cluster*, where each *pattern* is considered as having unitary mass). The quantities used in the calculus of the *DI* are the following:

cluster k compactness:
$$e_k = \sqrt{\sum_{j=1}^{n_k} \left\| \underline{x}_j - \underline{c}_k \right\|^2} \tag{2.78}$$

separation between clusters k and j:
$$R_{jk} = \frac{e_j + e_k}{\left\| \underline{c}_j - \underline{c}_k \right\|} \tag{2.79}$$

index for cluster k:
$$R_k = \max_{j \neq k} \{R_{jk}\} \tag{2.80}$$

Finally, the *DI index* of a *partition*, as a function of the above characteristics, has the form:

DI index:
$$DI(N_c) = \frac{1}{N_c} \sum_{k=1}^{N_c} R_k \tag{2.81}$$

In the last equations, n_k is the number of *patterns* inside *cluster k*, while $\underline{c}_k$ is its *centroid*. N_c is the number of *clusters* globally produced by the *partition*. The more the *clusters* are compact (low values for e_k) and far from each other (high values for $\left\| \underline{c}_k - \underline{c}_j \right\|$), the more this *index* will assume a small value. Therefore, the optimal *partition* is reached by minimizing the *DI index*. A problem

arises with this *index*: *clusters* with one *pattern* only have a *compactness* $e_k = 0$. So, when the *clusterization* produces as many *clusters* as the number of *patterns* in the *training set*, the *index* is equal to 0. Therefore, the stop condition for the algorithm is lost. Moreover, the result would be a network with no *generalization capability*. This is why some modifications are needed.

B. Single-Weighted Davies Index (SWDI)

The *DI index* tends to decrease as the number of *clusters* in the *partition* increases; therefore, it generally supplies us with highly complex *networks*. Here, a multiplying term that penalizes these kind of *partitions* is introduced.

cluster k compactness:

$$e_k = \sqrt{\sum_{j=1}^{n_k} \left\| \underline{x}_j - \underline{c}_k \right\|^2} \tag{2.82}$$

separation between clusters k and j:

$$R_{jk} = \frac{e_j + e_k}{\left\| \underline{c}_j - \underline{c}_k \right\| \cdot (1 - \frac{N_c}{N})^2} \tag{2.83}$$

index for cluster k:

$$R_k = \max_{j \neq k} \{R_{jk}\} \tag{2.84}$$

Finally, the *SWDI index* of a *partition*, as a function of the above characteristics, has the form:

SWDI index:

$$SWDI(N_c) = \frac{1}{N_c} \sum_{k=1}^{N_c} R_k \tag{2.85}$$

With the addition of *N*, that is the total number of *patterns* in the *class* in question

$$N = \sum_{k=1}^{N_c} n_k \tag{2.86}$$

the notation adopted is identical to the one of the previous *index*. In the expression of *separability*, the multiplying term added in the denominator approaches zero as N_c approaches *N*. This causes the growth of the *index* with the increasing of *network complexity*.

C. Euclidean Distance Based (EDB)

It's like the *SWDI index*, but with different formulation for *cluster compactness* and for the *index* of the generic *cluster*.

cluster k compactness:

$$e_k = \frac{1}{n_k} \sum_{j=1}^{n_k} \left\| \underline{x}_j - \underline{c}_k \right\| \tag{2.87}$$

separation between clusters k and j:

$$R_{jk} = \frac{e_j + e_k}{\left\| \underline{c}_j - \underline{c}_k \right\| \cdot (1 - \frac{N_c}{N})^2} \tag{2.88}$$

index for cluster k:

$$R_k = \sum_{\substack{j=1 \\ j \neq k}}^{N_c} R_{jk} \tag{2.89}$$

Cluster compactness equals the average of the distances between each *pattern* and the *cluster centroid*, while the *cluster index* is obtained by summing the $R_{jk}s$, and not from the maximum of these values. Finally, the *EDB index*, as a function of the above quantities, is given by:

EDB index:

$$EDB(N_c) = \frac{1}{N_c} \sum_{k=1}^{N_c} R_k \tag{2.90}$$

D. Double Weighted Davies Index (DWDI)

Besides the correction factor introduced in the *SWDI index* in order to penalize highly complex networks, in the denominator of the *separation index* we introduce another correction factor, this time with the aim of increasing a *cluster index* whenever this *cluster* contains too many *patterns*.

cluster k compactness:

$$e_k = \sqrt{\sum_{j=1}^{n_k} \left\| \underline{x}_j - \underline{c}_k \right\|^2} \tag{2.91}$$

separation between clusters k and j:

$$R_{jk} = \frac{e_j + e_k}{\left\| \underline{c}_j - \underline{c}_k \right\| \cdot (1 - \frac{N_c}{N})(1 - \frac{n_k}{N})} \tag{2.92}$$

index for cluster k:

$$R_k = \max_{j \neq k} \{R_{jk}\} \tag{2.93}$$

Finally, the *DWDI index*, as a function of the above quantities, is given by:

DWDI index:

$$DWDI(N_c) = \sum_{k=1}^{N_c} R_k \tag{2.94}$$

In this case, we have a loss in symmetry as regards the expression of the index, that is $R_{jk} \neq R_{kj}$, since the new correction factor introduced appears in the calculation of R_k for *cluster k* only. We have even implemented a symmetric version (*DWDI2*) of the *DWDI index*, where a further correction factor has been introduced in the expression of the *separation*:

$$R_{jk} = \frac{e_j + e_k}{\left\| \underline{c}_j - \underline{c}_k \right\| \cdot (1 - \frac{N_c}{N})(1 - \frac{n_k}{N})(1 - \frac{n_j}{N})} \tag{2.95}$$

All these *indices* don't provide for the case of a *partition* with a single *cluster*, since the *separation* quantity doesn't obviously make sense. To overcome this problem we introduce two dummy *degenerate clusters* located at vertices (0,....,0) and (1,....,1) of the *unity hypercube*. This solution has been necessary (especially for *data sets* with high cardinality) in those cases where *patterns* show a uniform density distribution (distances among them nearly constant), where an explosion of the number of *clusters*, with the increase of the *training parameter*, is noticed, with a consequent

too complex *network* as a result (low *generalization capability*. This solution is acceptable because the introduction of the two *dummy clusters* doesn't alter the final results, since it does equally influence all the obtained *partitions*. Moreover, it shows itself suitable for all the *Davies' indices*. It must be noticed that, in the *validation* phase, all the above mentioned *indices* consider the obtained partitions strictly "*crisp*", regardless their *fuzzy* nature. It means that, in the estimation of the *indices*, each *pattern* is considered completely belonging to one and only *cluster*.

2.6.3 In search of the optimum

In general, during the *training process*, the evolution of the considered *indices* is not regular, since it locally shows some pronounced oscillations. Therefore, in order to avoid to fall into a *local minimum*, a uniform sampling of the interval of variation of the *training parameter* is required. The algorithm that we are going to introduce as been optimized in order to reduce the processing time as best as we can. The *clustering* process is an iterative algorithm, during every cycle of which the search interval of the *parameter* is split into s equally measured sub-intervals (s = 10). Be Δ the width of every sub-interval and s + 1 the number of points defined by the splitting operation (the two limits of the interval, plus the internal splitting points). At the end of each of the s + 1 iterations that form the *clustering* process, the *validity indices* for the partition obtained are evaluated. The value of the *training parameter*, amongst the s + 1 values at our disposal (one for each sub-interval), that leads to the minimum value for the *index* is then selected. The algorithm proceeds by making a zoom on the sub-interval associated to the best value for the *training parameter*: the interval of width equal to 2Δ and centred around the optimum value for the *training parameter* is further split into s + 2 equally measured sub-intervals, so as to individuate s + 3 points. The new search interval is given by the union of the sub-intervals starting from the second up to the $(s + 1)^{th}$; the splitting points are still s + 1 (all the s + 3 points individuated, the first and the last excluded). The algorithm ends when $\Delta = 0.001$.

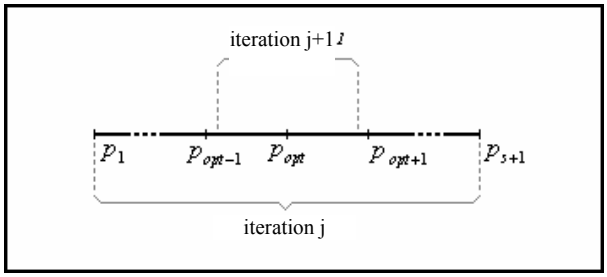


Fig. 2.29. the zoom on a sub-interval.

The j^{th} step of the algorithm (Fig. 2.29) can be formalized as follows:

$$\begin{cases} \Delta^{(j)} = \dfrac{2\Delta^{(j-1)}}{s+2} \\ p_1^{(j)} = p_{opt-1}^{(j-1)} + \Delta^{(j)} \\ p_{s+1}^{(j)} = p_{opt+1}^{(j-1)} - \Delta^{(j)} \end{cases} \tag{2.96}$$

where, in the first iteration:

$$\Delta^{(1)} = \frac{p_{stop} - p_{start}}{s} \tag{2.97}$$

In general, however, if we supply a *clusterizer* with any *optimization criterion*, we make that clusterizer more "intelligent" in some sense and we give us the opportunity to exploit a really automatic *clustering* process, since we free it from the really critic problem of choosing the correct value for the parameter that rules *training*. The architecture of the "*intelligent*" *clusterizer* is the following:

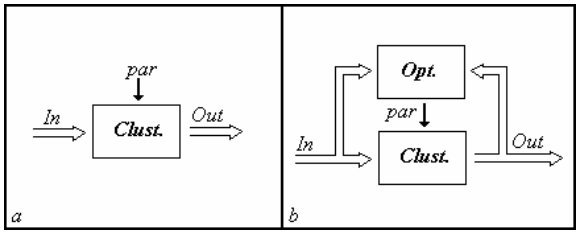


Fig. 2.30. Optimization of a Clusterizer

Based on information about the *partitions* supplied by the *clusterizer*, as well as about the *input data set* that gave rise to the *partitions*, the *optimization block* should produce a value of the *parameter* that better guaranties the optimal *partition,* given that *input data set*.

2.6.4 Network Architecture

It is identical to that of the *Simpson's classifier* (see Fig. 2.31): three *layers*, where the first (*input layer*) ha *n neurons*, one for each *feature* of the input *pattern*, each connected to all the *neurons* of the *forward layer* (*connections* with unary *weights*). The second *layer* is made up of as many *clusters* as those that have been singled out by the *clusterizer* during the *training* phase. This group of *neurons* can be partitioned into *k* subsets, each uniquely containing the *neurons* that are associated to the *clusters* of a single *class* (all the *neurons* of a *clusterizer* are surrounded by a dotted rectangle). Substantially, each subset is nothing more than the *output layer* of the single *clusterizer*. Once a new *pattern* is given as input to the *network*, the output of those *neurons* supplies the *degree of membership* of the *pattern* to its *cluster*. The third *layer* (*output layer*) has *k neurons*, one for each *class* to be discriminated, each receiving as inputs the outputs of all the *neurons* that are part of the relative *clusterizer* and supplying as output the *fuzzy degree of membership* of the *pattern* that's been supplied to the *network*. The *degree of membership* of a

pattern to a given *class* is the *fuzzy union* of the outputs of the *clusterizer* for that *class*. This result can be exploited as it is, or it can be normalized to binary values {0, 1} in case of "*crisp*" applications. The *connections* between the second and the third *layer* are characterized by couples of *weights*, each equal to the *min-max* points of the *HB* that include all the *patterns* of a given *cluster*.

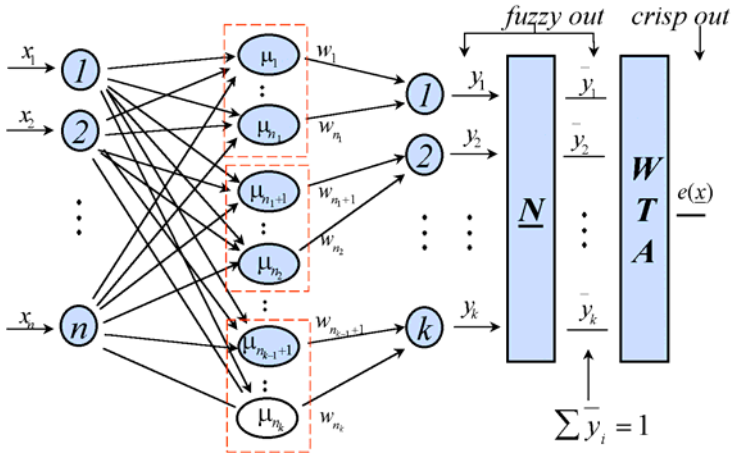


Fig. 2.31. The Structure of the Network

The $\overline{\sum}$ block gathers the outputs of the *k neurons* of the third *layer* being part of a *clusterizer* and normalizes them for the purpose that their sum always gives one. The last block (the *defuzzifier*) implements the *WTA* strategy that supplies the outer world with the estimated ("*crisp*") *label* of a *pattern* $\underline{x}$ presented to the *network*, where that *label* represents the *class* for which we have the maximum *fuzzy degree of membership* for $\underline{x}$.

2.6.5 The Generalized Bell membership function

Once an acceptable partition of the input space has been defined, as regards the *k classes* involved, the next step consists in the substitution of the *HBs* with geometrical entities that best describe the *clusters* in the *input space*, above all as regards the covering of those *clusters* with an opportune *MF*. The mentioned geometrical entities are *hyperellissoids* (*HE*) and the *MF* for what they have been introduced is the *Generalized Bell* (Fig. 2.32) proposed by Jang and Sun, the formulation of which, in case of a *pattern* $\underline{x}_i$ belonging to the h^{th} cluster, is the following:

$$\mu_h(\underline{x}_i) = \frac{1}{1 + \sum_{j=1}^{n} \left(\frac{(x_{ij} - c_{hj})}{a_{hj}} \right)^{2b_{hj}}} \tag{2.98}$$

The main characteristic of this function is that it's equal to unity in one and only point of the *input space*, that coincides with the *centroid* $\underline{c}_h$ of the relative *cluster h*; moreover, given a *degree of membership* value $\beta' \in (0,1]$, the locus of points of the *input space* that share the same *fuzzy degree*

of membership that do equals *β'* is an *HE* centred on $\underline{c}_h$ and with semi axes that are parallel to the coordinate axes of the reference system defined in the *input space*.

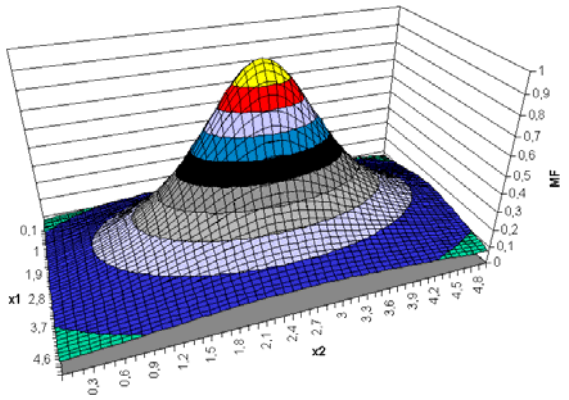


Fig. 2.32. The bidimensional Generalized Bell

The value of parameter a_{hj} equals the length of the *HE* semi axis that's oriented along the j^{th} dimension, when the *MF* has a value that equals *β'* = 0.5 on all the points of that *HE*. Parameter b_{hj} influences the decay slope of the *MF* along the j^{th} dimension, departing from the *centroid*. Fig. 2.33 depicts the shape of the *MF* in the one-dimensional case that has the form:

$$\mu_h(x) = \frac{1}{1+\left(\frac{(x-c)}{a}\right)^{2b}} \tag{2.99}$$

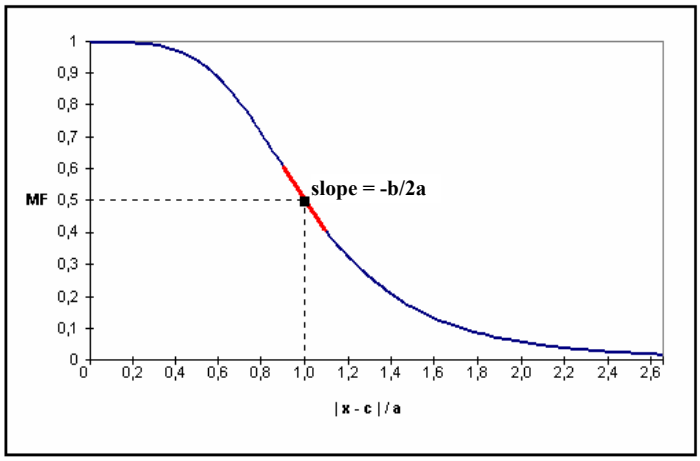


Fig. 2.33. one-dimensional generalized bell.

The *Generalized Bell MF* introduces a computational cost that's a little higher than that of the *Simpson's MF* and requires some more parameters (3*n* instead of 2*n*). Nevertheless, it much better manages the *overlap* problem among *clusters* belonging to different *classes*. This problem was solved by *Sympson's Min-Max algorithm* thru a *contraction* of the *HBs*, avoiding that some regions of the *input space* with unit *degree of membership* could belong to different *classes*. Here, that solution would have been impracticable, due to the intrinsic parallel mechanism of the present

approach. Anyway, in this case, regions characterized by unitary *degree of membership*, whichever the *class*, are zero dimensioned sets of points (each containing only the *centroids* $\underline{c}_h$ of the *clusters* associated to that *class*). Another typical problem of the *Simpson's classifier* that has been solved by means of the *Generalized Bell* is the unacceptable existence of *indecision regions* of the *input space* whose points have a zero *degree of membership* (points that cannot be labelled). The introduction of the *Generalized Bell MFs* eliminates this problem, since this *MF* assume values that are always greater then zero, no matter the distance from the *centroid* $\underline{c}_h$. The link between the geometrical information acquired during the *parallel clustering training* and the *HE* dimensions resides in the way we define the values for the 2*n parameters* $\underline{a}$ and $\underline{b}$ as functions of the values given by *training*, namely the *min-max* points, $\underline{v}$ and $\underline{w}$, of a given *HB*. The exchange between the *HBs* and the *HEs* is made by imposing, along a given dimension, the values the *MF* has to assume in two different points at two different distances from the *centroid*, along the semi axis parallel to that dimension. In details: given two values β_1 and β_2 ($\beta_2 < \beta_1$) and given two distances from the *centroid*, r_{hj}^1 and r_{hj}^2 ($r_{hj}^1 < r_{hj}^2$), *parameters* a_{hj} and b_{hj} of the h^{th} *cluster* along the j^{th} dimension can be evaluated by solving the following system:

$$\begin{cases} \mu_h(\underline{x}_{hj}^1) = \beta_1 \\ \mu_h(\underline{x}_{hj}^2) = \beta_2 \end{cases} \tag{2.100}$$

where $\underline{x}_{hj}^m$ (m = 1,2) is an array whose j^{th} component is equal to $c_{hj} + r_{hj}^m$ and all the others are equal to zero. The above system, once solved, returns the values of the two *parameters* a_{hj} and b_{hj}:

$$\begin{cases} b_{hj} = \dfrac{\ln(\frac{1}{\beta_1} - 1) - \ln(\frac{1}{\beta_2} - 1)}{2\ln(\frac{r_{hj}^1}{r_{hj}^2})} \\ a_{hj} = \dfrac{r_{hj}^2}{(\frac{1}{\beta_2} - 1)^{\frac{1}{2b_{hj}}}} \end{cases} \tag{2.101}$$

By defining an opportune link between the values of the two distances r_{hj}^1, r_{hj}^2 and the dimensions of the h^{th} *HB*, we also define the link between this *HB* and its *HE*. We have only to choose between two opportunities, because an *HE* can be inscribed or circumscribed to its *HB*. We opted for first choice even if, in both cases, the value for *parameter* r_{hj}^1 can be easily evaluated:

$$r_{hj}^1 = \begin{cases} \dfrac{w_{hj} - v_{hj}}{2}, & \text{inscribed ellissoid} \\ \sqrt{n}\left(\dfrac{w_{hj} - v_{hj}}{2}\right), & \text{circumscribed ellissoid} \end{cases} \tag{2.102}$$

In case of the *degenerate cluster* (*min* and *max points* coincide), r_{hj}^1 is equal to zero; so, a division by zero can arise in the calculus of a_{hj} and b_{hj}. To bypass this eventuality, we have imposed $a_{hj} = 10^{-1}$ and $b_{hj} = 8\times10^{-1}$. As regards the positioning, before *training* begins, of the *centroid* $\underline{c}_h$ with respect to the original *HB*, we have two choices:

A. the *centroid* coincides with the barycentre of the points (*patterns*) belonging to the *HB*:

$$\underline{c}_h = \frac{1}{n_h}\sum_{i=1}^{n_h}\underline{x}_i \tag{2.103}$$

where n_h equals the number of *patterns* inside the h^{th} *HB*.

B. the *centroid* coincides with the geometric centre of the *HB*:

$$c_{hj} = \frac{w_{hj} + v_{hj}}{2} \quad ,\forall\ \mathrm{j} = 1,..,\mathrm{n} \tag{2.104}$$

We opted for *centroid* A. As regards r_{hj}^2, the choice $r_{hj}^2 = r_{hj}^1 + \lambda$ has proved correct, where λ is a positive value that's the same for all the *clusters* and all the dimensions. This value can be chosen at run time, before the training process starts. A value of $\lambda = 0.4$ has worked in most of the situations. As regards β_1 and β_2, β_1 has been fixed to a value, common to all the *HBs* and to all the dimensions, equal to 0.9. In case of β_2, two strategies have been adopted. According to the first one, as in the case of β_1, even β_2 has been fixed to a predefined value (in particular: 0.1) universally effective. The second approach (geometrical) is inspired by the "gravitational" theory, since we desire that the importance of given *cluster* be more relevant when the number of *patterns* it contains is larger:

$$\beta_2 = \beta_1 \cdot (\frac{n_h}{N} - \varepsilon) \tag{2.105}$$

where n_h is the number of *patterns* in the h^{th} *cluster*, N is the number of *patterns* in the *training set* and ε is an infinitesimal positive number (equal to 10^{-6}) that we had to introduce with the aim of providing for the case of a mono *class clustering*, where only one *cluster* is produced. What we had to avoid is that, when $n_1/N = 1$, consequently $\beta_2 = \beta_1$. On the basis of the above choices, by departing from the *centroid* $\underline{c}_h$ of the *cluster* in parallel with the j^{th} dimension, μ_h assumes values starting from one, passing thru β_1 (= 0.9), after a shift along that dimension, equal to half of the length of the original *HB* edge (case of inscribed *HE*), and passing thru β_2, after a further shift this time equal to λ.

2.6.6 The elimination of the Contraction Phase (a study in depth)

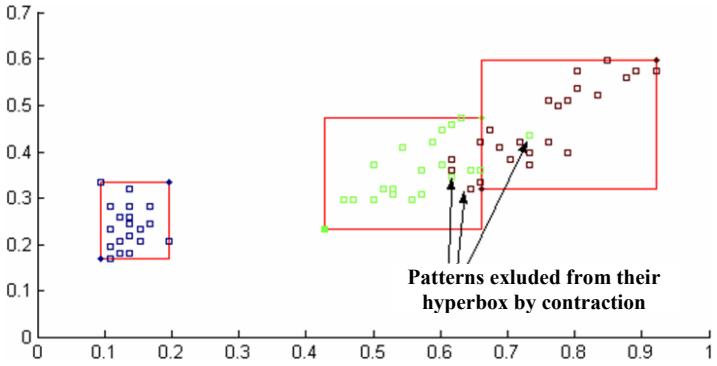


Fig. 2.34. Example of the effects of the contraction phase.

In the *training algorithm* proposed by *Simpson* for his *classifier*, the *contraction phase*, required by the need to eliminate, or at least reduce, any ambiguities in the localization of the *decision regions*, destroys some of the knowledge acquired at the moment during the *training phase*. In fact, if during the *learning* of a new *pattern*,the *HB expansion* causes the *overlap* of that *HB* with the *HB* of a *class* to which the given *pattern* doesn't belong to, a *contraction* is necessary in order to eliminate that *overlap*. In a lot of cased, this will provoke a loss of "*covering*" as regards *patterns* previously supplied to the *network*. We know that when a point inside an *HB* completely belongs to the relative *class*, it has a *degree of membership* equal to one. Obviously, if a point is let to completely belong to two distinct *classes*, an ambiguity in the decision would rise; this is why a contraction phase is necessary.

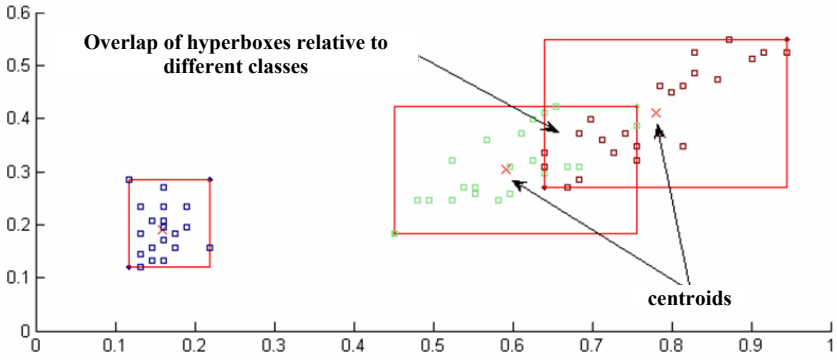


Fig. 2.35. HBs created in consequence of the elimination of the contraction phase. The centroids of the HBs are clearly visible. The MF is equal to one in these points only.

This is why we introduced an enhancement in the *FMMNN classifier* consisting in the elimination of the *contraction phase* (Fig. 2.35) and in the successive imposition of a new "covering" *MF*, namely the *Generalized Bell* previously introduced. In this case, the *MF* value is equal to one only on the *centroid* of an *HB*. The *centroid* $\underline{c}_h$ of an *HB* has been chosen according to two approaches:

1. it is the barycentre of the points of the *input space* inside the *HB*

$$\underline{c}_h = \frac{1}{n_h}\sum_{i=1}^{n_h}\underline{x}_i$$

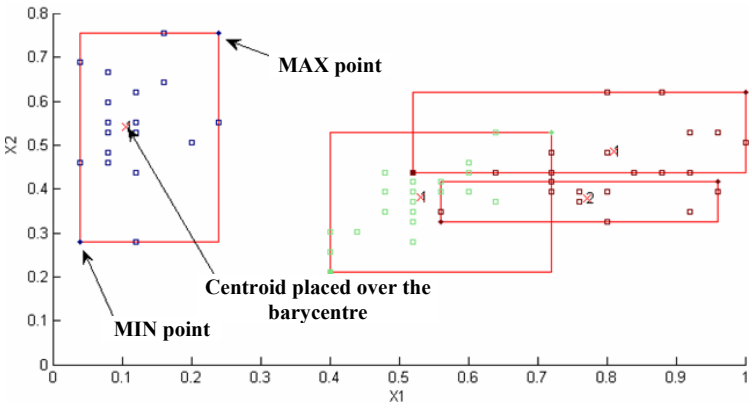


Fig. 2.36. The centroid as the barycentre of the HB.

2. it is the geometric centre of the *HB*

$$c_{hj} = \frac{w_{hj} + v_{hj}}{2} \quad ,\forall\, j = 1,..,1$$

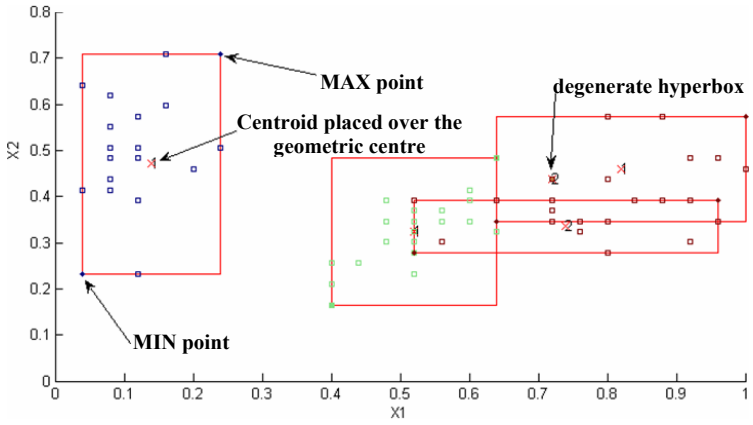


Fig. 2.37. The centroid as the geometric centre of the HB.

2.6.7 The Asymmetric Generalized Bell membership function

With the intention of shaping the *decision regions* given by the *training* phase, so that they could better reflect the gravitational characteristics of the formed *HBs*, we introduced what we have called the *Asymmetric Generalized Bell MF*. We proceed by discriminating the position of all the points inside an *HB* with respect to the *centroid*: points that fall on the right of the *centroid* will lead to some values of *parameters* a_{hj} and b_{hj}, while points that lie on the left of it will give different values of the same *parameters*. *Parameters* a_{hj} and b_{hj} will be different, according to whether we are on the left or on the right of the *centroid*, because the definition of parameters r_1 e r_2 will change accordingly. The left distance from the centroid where the membership function is equal to β_1 will be called r_{1L}, while the right distance from the centroid where the membership function is still equal to β_1 will be called r_{1R}. These last quantities have the following expressions:

$$r_{1R} = W_{hi} - C_{hi} \tag{2.106}$$

$$r_{1L} = C_{hi} - V_{hi} \tag{2.107}$$

where V_{hi} , W_{hi} , C_{hi} are respectively the *max*, *min points* and the *centroid* of the h^{th} *HB*, along the i^{th} dimension. If the point is on the right of the *centroid*, then the expressions for r_1 and r_2 are:

$$r_1 = r_{1R} \tag{2.107}$$

$$r_2 = r_1 + \lambda \tag{2.108}$$

otherwise,

$$r_1 = r_{1L} \tag{2.109}$$

$$r_2 = r_1 + \lambda(\frac{r_{1L}}{r_{1R}}) \tag{2.110}$$

An example of an *Asymmetric Generalized Bell* in one dimension is depicted in Fig. 2.38:

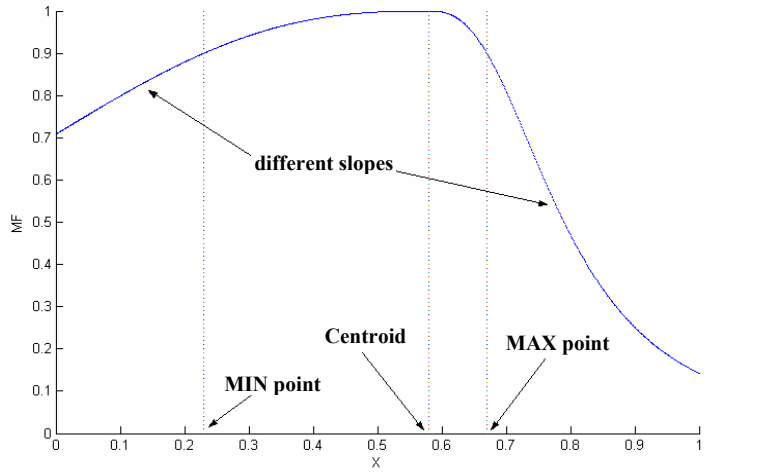


Fig. 2.38. Asymmetric Generalized Bell Membership Function.

To increase the accuracy of the *partition/covering* that characterizes the *Simpson's classifier* boosted by the introduction of the *Generalized Bell MF*, we can even think to choose the correct values for β_1 and β_2, r_1 and r_2, by taking into account the real distributions of *patterns* around the *centroid*, but this is a topic to be explored in future works. The identification of the best value for *parameter* θ is achieved by means of the procedure shown in chapter 2.5.8. First of all, the range of possible values for θ, [0,1], is splitted with a step of 0.01. At the end of every iteration, the total number of formed *HBs* (*network complexity*) is evaluated, as well as the number of *errors* made during the *test* phase (*misclassification*). These vales are then supplied to the usual *objective* (or *cost*) *function*:

$$F(\theta) = \lambda \cdot E_C(\theta) + (1-\lambda) \cdot E_S(\theta) \tag{2.111}$$

where $E_s(\theta)$ is the percentage of *misclassifications*, $E_c(\theta)$ is the percentage of *HBs* created during the *training* phase and λ quantifies the importance that we desire to give to one or to the other aspect. As stated in chapter 2.5.8, a lower *complexity* corresponds with a higher degree of *misclassification*. The greater is the value for *parameter* λ and the greater is the importance we give to the optimum value for *parameter* θ in correspondence to a lower complexity; on the contrary, a

small value for *parameter* λ means that we have the optimum when the lowest number of *misclassifications* has been achieved. Obviously, a good compromise could be λ = 0.5, such that both aspects have the same importance. The optimum will be given in correspondence of the minimum of the *cost function*. In Fig. 2.39 an example of the graph of the functions $F(\theta)$, $E_s(\theta)$ and $E_c(\theta)$ is shown; it allows us to locate best value of *parameter* θ for *training*.

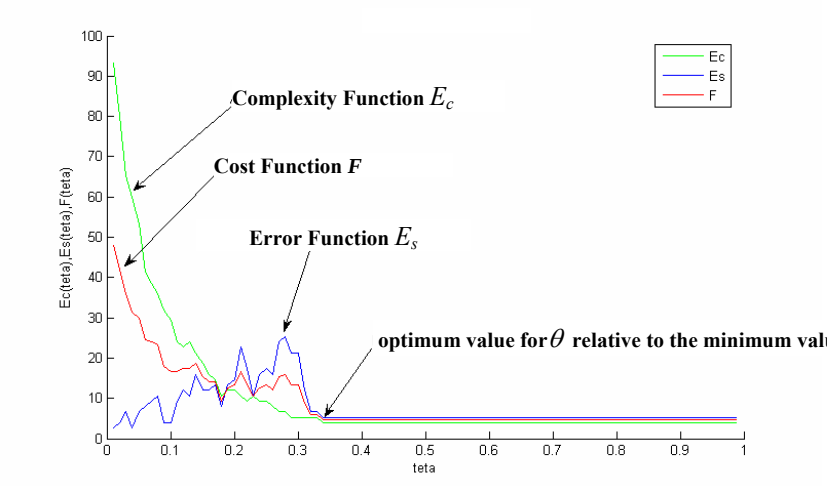


Fig. 2.39. Graph of the search for the optimum for parameter θ in relation with functions Ec, Es, e F.

2.7 The Discriminative Learning

2.7.1 Discrimination and Minimum Error Classification

Many people talk about *classification* and *discrimination*, but without stressing the main difference between them. Though they are strictly related, we have to make a distinction: in *classification*, the aim is to predict the true *class m* for an *observation x*, while in *discrimination* the aim is to split the *sample space X* into disjoint regions, each one related to one of *M classes*.

Let's consider a *set of observations* $\mathcal{L} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_K\}$, where each $\mathbf{x}_h$ is an array of N_0 elements that belongs to a *class* C_d, $d = 1,2,\ldots,N_M$. A *classifier* can be defined by giving:

- a *set of parameters* $\boldsymbol{\Lambda}$
- a *decision rule*

The design of a *minimum error classifier* (*optimum classifier*, *OC*, from now on) is the same as finding the *set of parameters* and the *decision rule* that together minimize the *non correct classification* (*misclassification*, from now on), on the basis of the given set of observations $\mathcal{L}$. For the purpose, let's define a set of *discriminative functions* (*DF*):

$$g_d(\mathbf{x}, \Lambda) \qquad d = 1,2,\ldots,N_M \tag{2.112}$$

characterized by a *set of parameters* $\boldsymbol{\Lambda}$. An *OC* is characterized by an *optimum set of parameters* of the *discriminative functions*, as regards the *minimization* of a *classification cost* over the given set of observations $\mathcal{L}$. As a rule, the *cost* is defined by a couple of indices (*n*, *k*), where *n* is the index of the correct *class* and *k* is the index of the *class* actually coupled to the given *input pattern* by the *classifier*. The value of the *cost* is related to the penalty we are subjected to, when the observation of *class* C_k is mistaken for that of *class* C_n. For the sake of simplicity, we will adopt *linear DF s* only. Let's consider a N_0–dimensional *feature vector* $\boldsymbol{x}$. A *linear DF* in $\boldsymbol{x}$ can be defined as follows:

$$g(\boldsymbol{x}) = \boldsymbol{w}^T \cdot \boldsymbol{x} + w_0 \tag{2.113}$$

where $\boldsymbol{w}$ and w_0 are the *weight vector* and the *bias*, respectively. For each *class*, we have a couple $(\boldsymbol{w}, w_0)$. Moreover, the number of *DFs* we need is equal to N_M. As a consequence, we can define the *parameter set* of the *classifier* as follows:

$$\boldsymbol{\Lambda} = \{\boldsymbol{\lambda}_1,\ \boldsymbol{\lambda}_2,\ \ldots,\ \boldsymbol{\lambda}_{N_M}\} = \{(\mathbf{w}_1, w_{10}),\ (\mathbf{w}_2, w_{20}),\ \ldots,\ (\mathbf{w}_{N_M}, w_{N_M 0})\} \tag{2.114}$$

where $\boldsymbol{\lambda}_n = \{\mathbf{w}_n, w_{n0}\}$. Therefore, each *DF* can be rewritten as:

$$g_n(\mathbf{x}, \boldsymbol{\lambda}_n) = \mathbf{w}_n^T \cdot \mathbf{x} + w_{n0} = \boldsymbol{\lambda}_n^T \cdot \mathbf{y} \tag{2.115}$$

where $\boldsymbol{\lambda}_n^T = \left[\mathbf{w}_n^T \quad w_{0i}\right]$ and $\boldsymbol{y}^T = \left[\boldsymbol{x}^T \; 1\right]$.

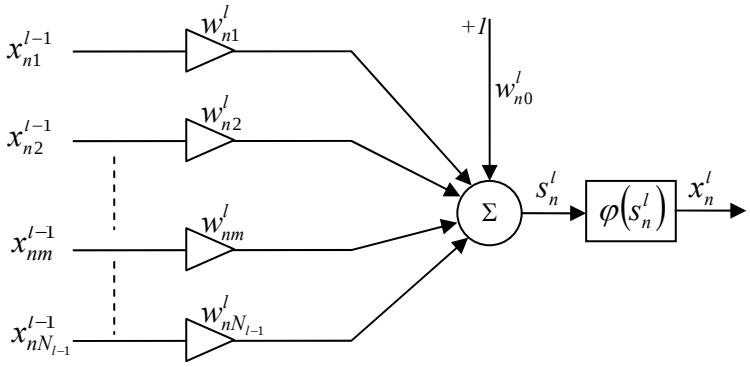


Fig. 2.40. The artificial neuron for an MLP architecture

In Fig. 2.40, the *n^{th} neuron*, in the *l^{th} layer*, of an *MLP* architecture is depicted. We will show later that, if we are involved in a *M class classification* problem, an *MLP* architecture with *M output neurons* will be required. The values returned by the *output neurons* represent the *discrimination* degree in the *classification* problem. So, according to the nature of the chosen *DF*, the *activation function* of the *neurons* in the *output layer* will be strictly *linear*. The *classifier* is based on the following *decision rule*:

$$C(\mathbf{x}_h) \in C_n \quad \text{if} \quad g_n(\mathbf{x}, \boldsymbol{\lambda}_n) = \max_{\mathrm{k}} g_k(\mathbf{x}, \boldsymbol{\lambda}_k) \tag{2.116}$$

that is to say, a feature pattern $\boldsymbol{x}$ belongs to the class C_n for which the value returned by the corresponding *DF* is the maximum among those returned by all the *DF* of the *classifier*. The use of *linear DFs* yields to the definition of *hyperplane decision boundaries*. The *linear DF* (2.115) can be generalized by taking into account non linear terms in $\boldsymbol{x}$ (polynomials). The *parameters* of the *classifier* can be calculated starting from the set of *K observations* $\boldsymbol{\mathcal{L}} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_K\}$. We suppose that the correspondence between *observations* and the *classes* they belong to be a priori known. We say that the given *set of observations* is *linearly separable*, if it's possible to find a set of *parameters* $\boldsymbol{\Lambda}$ such that the *classification* process, based on the *DFs* and the *decision rule* above defined, doesn't produce any *error*. The standard formulation of the *discriminative* operation is based on the definition of a *DF* as well as of a scalar criterion that's suited to a *gradient* based research procedure. In order to derive the objective criterion, the traditional *discriminative* formulation, have to be replaced with the following three-step procedure. Firstly, given an *input pattern* $\boldsymbol{x}$, the *classifier* makes its decision by choosing the higher of the values given by the *DFs*, when applied to $\boldsymbol{x}$. This *decision* operation must be formulated in order to allow some kind of *optimization* process. In the second step, a *measure of misclassification* that can be used to include the *decision* operation in the formulation of the *minimum classification error* (*MCE*) is introduced.

Many continue and differentiable, with respect to the *parameters* $\boldsymbol{\Lambda}$, definitions of the *measure of misclassification* have been given. One possibility is the following:

$$\xi_n(\mathbf{x}) \;=\; -g_n(\mathbf{x},\boldsymbol{\lambda}_n)+\left[\frac{1}{N_M-1}\sum_{k\neq n} g_k(\mathbf{x},\boldsymbol{\lambda}_k)^{\eta}\right]^{\frac{1}{\eta}} \tag{2.117}$$

where η is a large positive number (in most of the application, the functions g_k are taken positive). By opportunely choosing the value of this *parameter*, we can take into account, at various levels, all the *classes* involved. If $\eta \to \infty$, the *measure of misclassification* becomes:

$$\xi_n(\mathbf{x}) \;=\; -g_n(\mathbf{x},\boldsymbol{\lambda}_n)+g_j(x,\boldsymbol{\lambda}_j) \tag{2.118}$$

where C_j is the *class* with the higher *discrimination value*, among all the *classes* but C_n. Obviously, $\xi_n > 0$ means *misclassification*, while $\xi_n \le 0$ means correct *classification* (no *cost*). In the third step, the *cost* as a function of the *measure of misclassification* is defined ($E \equiv l$):

$$l_n(\mathbf{x},\boldsymbol{\lambda}_n) = l_n[\xi_n(\mathbf{x})] \tag{2.119}$$

It is worth noting that, for the sake of generality, the *cost function* l_n and the *measure of misclassification* ξ_n can be individually defined, one for each *class n*. In literature, two expressions of the *cost function* can be found:

a) ***exponential*** :

$$l_n(\xi_n) = \begin{cases} (\xi_n)^{\beta} & \text{if } \xi_n > 0 \\ 0 & \text{if } \xi_n \le 0 \end{cases} \qquad \text{where } \beta > 0 \quad \text{e} \quad \beta \to 0 \tag{2.120}$$

b) ***translated sigmoid*** :

$$l_n(\xi_n) = \frac{1}{1+e^{-\beta(\xi_n+\alpha)}} \qquad \text{where } \beta > 0 \tag{2.121}$$

both are *cost functions* with range [0…1] and both can be used in *gradient descent* algorithms. Obviously, a *correct classification* ($\xi_n \le 0$) has no costs, while a *misclassification* ($\xi_n > 0$) leads to a loss that quantifies the *classification error*. Finally, given any unknown *observation* $\boldsymbol{x}$, it's possible to define an *average empirical cost* (it measures the performances of the *classifier*) as

$$l(\mathbf{x},\boldsymbol{\Lambda}) = \sum_{d=1}^{N_M} l_d(\mathbf{x},\boldsymbol{\lambda}_d)\,\delta(\mathbf{x}\in C_d) \tag{2.122}$$

where

$$\delta(a) = \begin{cases} 1 & a \text{ is true} \\ 0 & a \text{ is false} \end{cases} \tag{2.123}$$

This *cost function* can be defined for every *input pattern* $\boldsymbol{x}$ and is the foundation of the *target* to be reached thru the usual *gradient descent* algorithms. Given a set of *labelled training patterns*

$\mathcal{L} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_K\}$, the *target* in terms of the *average empirical cost* is defined as follows:

$$L_0(\boldsymbol{\Lambda}) = \frac{1}{K}\sum_{k=1}^{K}\sum_{d=1}^{N_M} l_d(\mathbf{x}_k;\boldsymbol{\lambda}_d)\,\delta(\mathbf{x}_k \in C_d) \tag{2.124}$$

As usual, this *cost function* can be minimized by means of the following rule (*gradient descent*):

$$\boldsymbol{\Lambda}_{t+1} = \boldsymbol{\Lambda}_t - \mu\nabla L_0(\boldsymbol{\Lambda}_t) \tag{2.124}$$

where $\boldsymbol{\Lambda}_t$ is the set of *parameters* at the t^{th} iteration. The time at which the parameters have to be updated can be chosen in two ways. We can decide to adjust the *parameters* of the *classifier* after the presentation of each *training pattern* $\mathbf{x}_t \in C_d$ and taking $\nabla l_d(\boldsymbol{x}_t;\boldsymbol{\Lambda})$ as the *gradient* in (2.124). On the other hand, we can adjust the *parameters* at the end of the presentation to the *classifier* of the whole *training set* $\mathcal{L}$. In this case, in (2.124)we are dealing with an *average gradient*.

Let's now suppose to have at our disposal an *M layer MLP classifier*, each *layer* being composed by N_l *neurons*. In particular, we assume to deal with a $M-1$ *layer neural network*, the output of which is the input to a *layer* of *DFs* (the real *output layer* of the *classifier*). Therefore, the structure of the *classifier* is as follows:

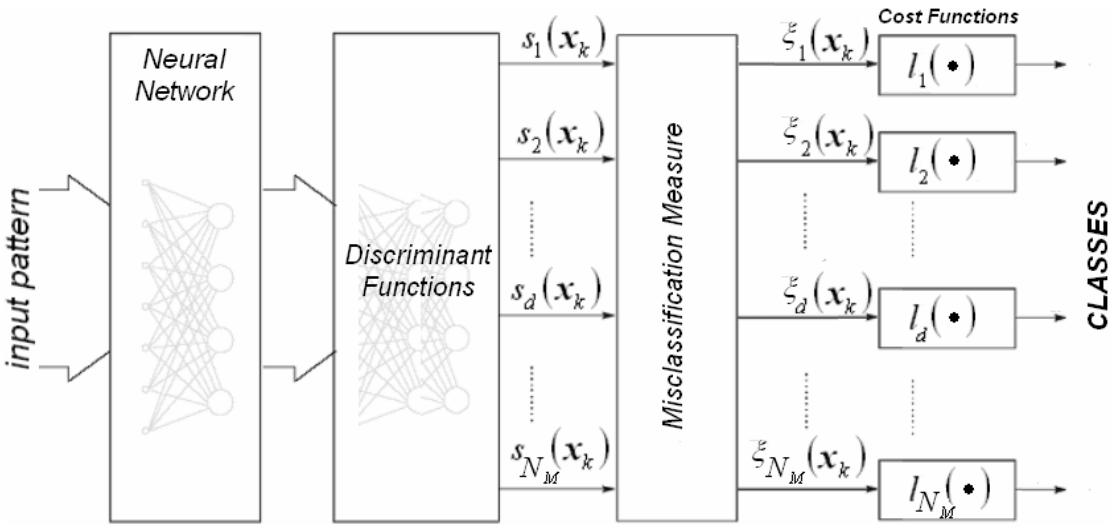


Fig. 2.41. Multilayer structure with output discriminative functions

The *DFs* can be defined in many ways; if these functions are simple linear combiners we are dealing with the above introduced *linear DFs*. More generally, we can think to *DFs* composed by a *linear combiner* followed by a non linear, continuous and differentiable function φ_d. In this case, the *output layer* of a *multilayer classifier* is as in Fig. 2.42 (see also Fig. 2.40). The *output layer* of the *classifier* will contain N_M *neurons*, one for each *class* taken into consideration. As usual, the *activation* of the n^{th} *neuron* in the l^{th} *layer* is defined as:

$$s_n^{(l)} = \sum_{m=0}^{N_{l-1}} w_{nm}^{(l)} x_m^{(l-1)} \tag{2.125}$$

while its output is as follows:

$$x_n^{(l)} = \varphi\left(s_n^{(l)}\right) \tag{2.126}$$

where φ is the *activation function*.

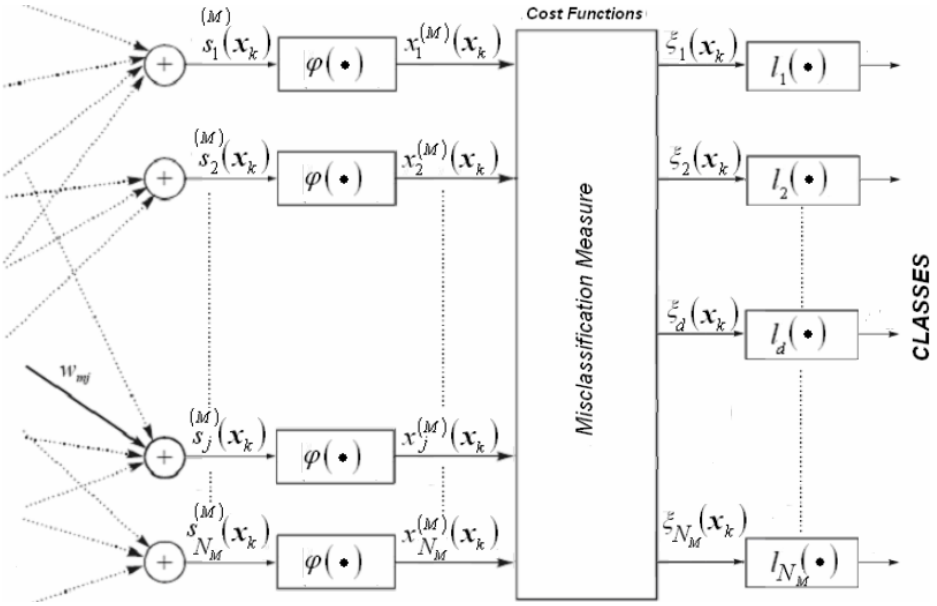


Fig. 2.42. discriminative output layer with misclassification measure block

This *classifier* will be trained by means of the well known *backpropagation*, a *supervised learning algorithm* based on a *training vector* $\boldsymbol{x}$ and the relative *target vector* $\boldsymbol{d}$:

$$\begin{aligned} \left[\mathbf{x}^{(0)}\right]^T &= x_1^{(0)}, x_2^{(0)}, \ldots, x_{N_0}^{(0)} \\ \left[\mathbf{d}\right]^T &= d_1, d_2, \ldots, d_{N_M} \end{aligned} \tag{2.127}$$

The *target vector* $\mathbf{d}$, associated to the *input training vector* $\mathbf{x} \in C_d$ of a N_M *classes classifier*, is typically binary valued:

$$d_i = \begin{cases} 1 & i = d \\ 0 & otherwise \end{cases} \tag{2.128}$$

Let's now assume that the *input pattern* $\boldsymbol{x}_k$ belongs to the d^{th} *class*: $\boldsymbol{x}_k \in C_d$. The *classifier* is trained with the aim of reducing the differences existing between the real output $\left[\mathbf{x}^{(M)}\right]^T = x_1^{(M)}, x_2^{(M)}, \ldots, x_{N_M}^{(M)}$ of the *classifier* and the *target vector* $\boldsymbol{d}$. In the standard *backpropagation* algorithm, the *cost function* to be minimized at time *t*, by means of *gradient descent* ($\mathbf{W}[t+1] = \mathbf{W}[t] - \mu \nabla_{\mathbf{W}[t]} E^2$, where $\boldsymbol{W}$ is the *weight matrix* and μ controls the magnitude of the adjustments applied) is the classical *mean-square error function*:

$$E^2 = \sum_{d=1}^{N_M} \left(d_d - x_d^{(M)}\right)^2 \tag{2.129}$$

Nevertheless, a *minimum square error* (*MSE*) criterion doesn't necessarily minimize the *classification error* or, in other words, the solution $\mathbf{W}^*$ that minimizes E^2 doesn't necessarily minimizes the probability of *misclassification*. Therefore, we have to define a *BP* algorithm

consistent with the aim of *minimizing* the *classification error*, by replacing the *squared error* E^2 with the *cost function* L_0. Particularly, given a *training pattern* $\mathbf{x}_k \in C_d$, let's consider the j^{th} *DF* and define its *local gradient* $\delta_j^{(M)}$ (derivative of the *cost function* with respect to the *activation* of the j^{th} *DF*):

$$\delta_j^{(M)} = -\frac{\partial l_d(\xi_d)}{\partial s_j^{(M)}} \tag{2.130}$$

which, according to the structure of the *output layer* of the *classifier* (see Fig. 2.42) can be specified as follows:

$$\delta_j^{(M)} = -\frac{\partial l_d}{\partial \xi_d}\frac{\partial \xi_d}{\partial x_j^{(M)}}\frac{\partial x_j^{(M)}}{\partial s_j^{(M)}} \tag{2.131}$$

where:

$$\frac{\partial x_j^{(M)}}{\partial s_j^{(M)}} = \varphi'\left(s_j^{(M)}\right) \qquad \frac{\partial l_d}{\partial \xi_d} = l_d'(\xi_d) \tag{2.132}$$

$$\Downarrow$$

$$\delta_j^{(M)} = -l_d'(\xi_d)\varphi'\left(s_j^{(M)}\right)\frac{\partial \xi_d}{\partial x_j^{(M)}} \tag{2.133}$$

By combining the definition (2.117) of the *measure of misclassification* and the structure of the *output layer* of the *classifier* (Fig. 2.42), we can say that:

$$\xi_d(\mathbf{x}_k) = -x_d^{(M)}(\mathbf{x}_k, \mathbf{w}) + \left[\frac{1}{N_M - 1}\sum_{h\neq d} x_h^{(M)}(\mathbf{x}_k, \mathbf{w})^{\eta}\right]^{\frac{1}{\eta}} \tag{2.134}$$

where $\boldsymbol{w}$ is the *set* of *parameters* of the *classifier*, for what concerns its *discriminative output layer*. In a first step, let's assume $j = d$. In this case, it holds:

$$\frac{\partial \xi_d}{\partial x_d^{(M)}} = \frac{\partial}{\partial x_d^{(M)}}\left(-x_d^{(M)} + \left[\frac{1}{N_M - 1}\sum_{h\neq d}\left(x_h^{(M)}\right)^{\eta}\right]^{\frac{1}{\eta}}\right) = -\frac{\partial x_d^{(M)}}{\partial x_d^{(M)}} + \left(\frac{1}{N_M - 1}\right)^{\frac{1}{\eta}}\frac{\partial}{\partial x_d^{(M)}}\left[\sum_{h\neq d}\left(x_h^{(M)}\right)^{\eta}\right]^{\frac{1}{\eta}} =$$

$$= -1 + \left(\frac{1}{N_M - 1}\right)^{\frac{1}{\eta}}\frac{1}{\eta}\left[\sum_{h\neq d}\left(x_h^{(M)}\right)^{\eta}\right]^{\frac{1}{\eta}-1}\frac{\partial}{\partial x_d^{(M)}}\sum_{h\neq d}\left(x_h^{(M)}\right)^{\eta} =$$

$$= -1 + \left(\frac{1}{N_M - 1}\right)^{\frac{1}{\eta}}\frac{1}{\eta}\left[\sum_{h\neq d}\left(x_h^{(M)}\right)^{\eta}\right]^{\frac{1}{\eta}-1}\sum_{h\neq d}\frac{\partial\left(x_h^{(M)}\right)^{\eta}}{\partial x_d^{(M)}} =$$

$$= -1 + \left(\frac{1}{N_M - 1}\right)^{\frac{1}{\eta}}\frac{1}{\eta}\left[\sum_{h\neq d}\left(x_h^{(M)}\right)^{\eta}\right]^{\frac{1}{\eta}-1}\eta\sum_{h\neq d}\left[\left(x_h^{(M)}\right)^{\eta-1}\frac{\partial x_h^{(M)}}{\partial x_d^{(M)}}\right] \tag{2.135}$$

Nevertheless:

$$\frac{\partial x_h^{(M)}}{\partial x_d^{(M)}} = \begin{cases} 1 & \text{if } h = d \\ 0 & \text{if } h \neq d \end{cases} \tag{2.136}$$

and in the last summation he have excluded the case $h = d$; therefore, we can conclude that

$$\frac{\partial \xi_d}{\partial x_j^{(M)}} = -1 \qquad \text{for} \qquad j = d \tag{2.137}$$

Let's now suppose to consider the generic *DF* ($j \neq d$). In this case, it holds:

$$\frac{\partial \xi_d}{\partial x_j^{(M)}} = \frac{\partial}{\partial x_j^{(M)}}\left(-x_d^{(M)} + \left[\frac{1}{N_M - 1} \sum_{h \neq d} \left(x_h^{(M)} \right)^{\eta} \right]^{\frac{1}{\eta}} \right) = -\frac{\partial x_d^{(M)}}{\partial x_j^{(M)}} + \left(\frac{1}{N_M - 1} \right)^{\frac{1}{\eta}} \frac{\partial}{\partial x_j^{(M)}} \left[\sum_{h \neq d} \left(x_h^{(M)} \right)^{\eta} \right]^{\frac{1}{\eta}} =$$

$$= \left(\frac{1}{N_M - 1} \right)^{\frac{1}{\eta}} \frac{1}{\eta} \left[\sum_{h \neq d} \left(x_h^{(M)} \right)^{\eta} \right]^{\frac{1}{\eta} - 1} \eta \sum_{h \neq d} \left(x_h^{(M)} \right)^{\eta - 1} \frac{\partial x_h^{(M)}}{\partial x_j^{(M)}}$$

where:

$$\frac{\partial x_h^{(M)}}{\partial x_j^{(M)}} = \begin{cases} 1 & \text{if } h = j \\ 0 & \text{if } h \neq j \end{cases} \tag{2.138}$$

As a consequence:

$$\frac{\partial \xi_d}{\partial x_j^{(M)}} = \left(\frac{1}{N_M - 1} \right)^{\frac{1}{\eta}} \left[\sum_{h \neq d} \left(x_h^{(M)} \right)^{\eta} \right]^{\frac{1}{\eta} - 1} \left(x_j^{(M)} \right)^{\eta - 1} = \frac{\left(x_j^{(M)} \right)^{\eta - 1}}{N_M - 1} \left[\frac{1}{N_M - 1} \sum_{h \neq d} \left(x_h^{(M)} \right)^{\eta} \right]^{\frac{1}{\eta} - 1} \tag{2.139}$$

Finally, we can summarize as follows:

$$\frac{\partial \xi_d}{\partial x_j^{(M)}} = \begin{cases} -1 & j = d \\ \dfrac{\left(x_j^{(M)} \right)^{\eta - 1}}{N_M - 1} \left[\dfrac{1}{N_M - 1} \displaystyle\sum_{h \neq d} \left(x_h^{(M)} \right)^{\eta} \right]^{\frac{1}{\eta} - 1} & j \neq d \end{cases} \tag{2.140}$$

Therefore, the generic *local gradient* for the generic *DF* is equal to :

$$\delta_j^{(M)} = \begin{cases} l_d'\left(\xi_d \right) \varphi'\left(s_j^{(M)} \right) & j = d \\ -l_d'\left(\xi_d \right) \varphi'\left(s_j^{(M)} \right) \dfrac{\left(x_j^{(M)} \right)^{\eta - 1}}{N_M - 1} \left[\dfrac{1}{N_M - 1} \displaystyle\sum_{h \neq d} \left(x_h^{(M)} \right)^{\eta} \right]^{\frac{1}{\eta} - 1} & j \neq d \end{cases} \tag{2.141}$$

We know that the adjustment to be applied to the m^{th} *weight* of the j^{th} *DF*, in order to minimize the *cost* due a *misclassification* relative to the d^{th} class is equal to:

$$\Delta w_{jm}^{(M)} = -\mu \frac{\partial l_d}{\partial w_{jm}^{(M)}} = -\mu \frac{\partial l_d}{\partial s_j^{(M)}} \frac{\partial s_j^{(M)}}{\partial w_{jm}^{(M)}} = \mu \, \delta_j^{(M)} x_m^{(M-1)} \tag{2.142}$$

where $x_m^{(M-1)}$ is the m^{th} input to the *discriminative layer*. The *backpropagation* proceeds with the adjustment of the *parameters* of the $M - 1$ *layers* that precede the *discriminative layer*. The *local gradient* concerning the j^{th} neuron in the l^{th} *layer* is equal to ($l = 1, 2, \ldots, M - 1, \quad j = 1, 2, \ldots, N_l$):

$$\delta_j^{(l)} = -\frac{\partial l_d}{\partial s_j^{(l)}} = -\frac{\partial l_d}{\partial x_j^{(l)}} \frac{\partial x_j^{(l)}}{\partial s_j^{(l)}} = -\frac{\partial l_d}{\partial x_j^{(l)}} \varphi'\left(s_j^{(l)} \right) \tag{2.143}$$

So, we need to compute the quantity:

$$\frac{\partial l_d}{\partial x_j^{(l)}} \tag{2.144}$$

The *cost function* l_d depends on the quantities $s_h^{(l+1)} \quad h = 1, \ldots N_{l+1}$, that in their turn all depend on

the same quantity $x_j^{(l)}$. By applying the rule of the derivative of a compound function of more than one variable, we can find that (we suppose the *measure of misclassification* e the *cost function* being continuous an differentiable):

$$\frac{\partial l_d}{\partial x_j^{(l)}} = \sum_{h=1}^{N_{l+1}} \frac{\partial l_d}{\partial s_h^{(l+1)}} \frac{\partial s_h^{(l+1)}}{\partial x_j^{(l)}} = -\sum_{h=1}^{N_{l+1}} \delta_h^{(l+1)} \frac{\partial s_h^{(l+1)}}{\partial x_j^{(l)}} = -\sum_{h=1}^{N_{l+1}} \delta_h^{(l+1)} w_{hj}^{(l+1)} \tag{2.145}$$

As a consequence, the gradient concerning the j^{th} neuron in the l^{th} *layer* is equal to:

$$\delta_j^{(l)} = -\frac{\partial l_d}{\partial s_j^{(l)}} = -\frac{\partial l_d}{\partial x_j^{(l)}} \frac{\partial x_j^{(l)}}{\partial s_j^{(l)}} = \varphi'\left(s_j^{(l)}\right) \sum_{h=1}^{N_{l+1}} \delta_h^{(l+1)} w_{hj}^{(l+1)} \qquad l = 1, \ldots M-1 \tag{2.146}$$

therefore, the computation of the *local gradient*, required by the adjustment of the *weights* ($\Delta w_{jm}^{(l)} = \mu\, \delta_j^{(l)} x_m^{(l-1)}$), can be accomplished starting from the *local gradients* of the *layer* that follows.

As stated above, the *optimization algorithm* works on the *parameters* of the system, trying to minimize an *error function*. Starting from the analysis of the structure of the *classifier* and from some experimental results we have found that the choice of the correct *parameters* deeply influences the performances of the *optimization algorithm*. For example, if we choose the *sigmoidal cost function* $l_n(\xi_n) = \frac{1}{1+e^{-\beta(\xi_n+\alpha)}}$ and bring out the so called *form factor* ζ:

$$\begin{aligned} \frac{\partial l_n(\xi_n)}{\partial \xi_n} &= \frac{\partial}{\partial \xi_n}\left[\frac{1}{1+e^{-\beta(\xi_n+\alpha)}}\right] = -\frac{1}{\left(1+e^{-\beta(\xi_n+\alpha)}\right)^2} e^{-\beta(\xi_n+\alpha)}(-\beta) = \\ &= \beta \frac{1}{1+e^{-\beta(\xi_n+\alpha)}} \frac{1+e^{-\beta(\xi_n+\alpha)}-1}{1+e^{-\beta(\xi_n+\alpha)}} = \beta\, l_n(\xi_n)\left[1-l_n(\xi_n)\right] \end{aligned} \tag{2.147}$$

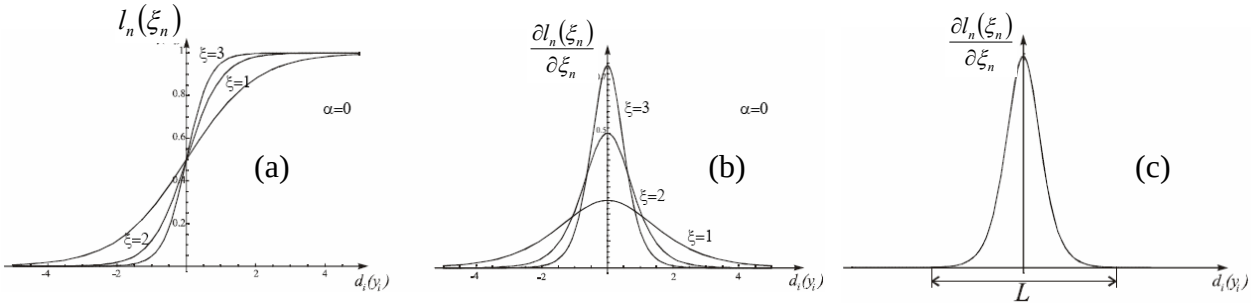


Fig. 2.43. Cost function and its derivative as a function of the form factor ζ

The *form factor* ζ influences the slope of the *cost function*. Fig. 2.43 (a) shows us that, once a value for ζ has been fixed, a corresponding range of values *L* for ξ_n is fixed too, such that $\partial l_n / \partial \xi_n >> 0$. Inside range *L* (Fig. 2.43 (c)), the algorithm can be considered dynamically active, while outside *L* the *weight adjustment* is substantially negligible. Consequently, in the states for which $\xi_n << 0$ e $\xi_n >> 0$ the system doesn't evolve; so, these states can be considered states of stability. Therefore, to increase the efficiency of the algorithm, it's necessary to avoid that $\xi_n >> 0$ for at least the first

learning epochs. According to the operational definition of *discriminative learning* (*DL*), when the *correct classification* is achieved, the *cost* at the output relative to the correct *class* shouldn't influence the computation of the *gradient*. Practically, this situation can be implemented by zeroing the *cost function* when $\xi_n < 0$ (condition of *correct classification*). Nevertheless, this approach can be dangerous, since when the *classification* leads to $\xi_n < 0$ for different *samples* belonging to different *classes*. To avoid this case, we have introduced for the *cost function* a *truncation parameter* γ:

$$l_n(\xi_n) = \begin{cases} \dfrac{1}{1+e^{-\beta(\xi_n+\alpha)}} & \xi_n > -|\gamma| \\ 0 & \xi_n < -|\gamma| \end{cases} \tag{2.148}$$

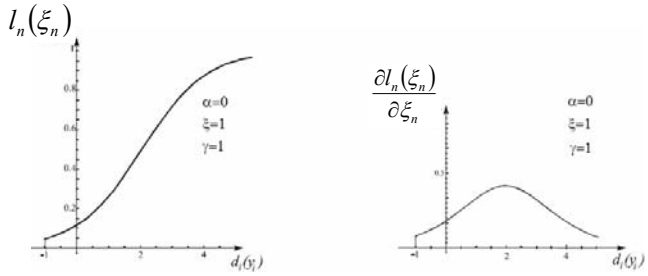


Fig. 2.44. truncation of the cost function.

If we don't take into account this *truncation parameter*, the algorithm would indefinitely continue in minimizing the *cost function* even in the presence of a *correct classification*.

2.7.2 The DL in the Classification of Time Series

A *classifier* (or a *predictor*) for *Time Series Processing* is a dynamic system, the architecture of which is composed by two blocks (Fig. 2.45):

- a *short term memory* (*STM*, to store past relevant events)
- an *associator* (*classifies* or *predicts* based on the *STM*)

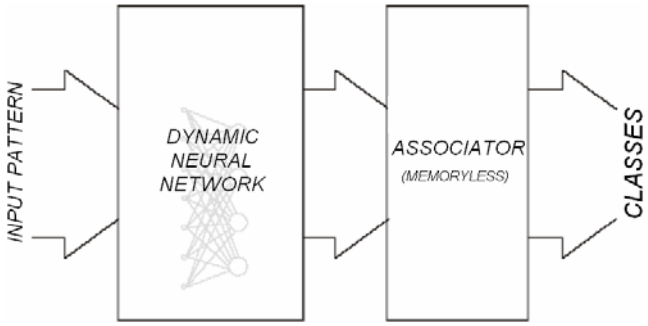


Fig. 2.45. a Dynamic Classifier with an output Static Associator.

Time series are *patterns* that evolve during time. Therefore, the output of the *classifier* has to depend not only on the actual input, but also on input received in the preceding times. *RNNs* are powerful tools to be used to solve the problem of *non linear time series classification*. Their

training can be accomplished in *batch* mode, as well as in *on-line* mode.

Batch mode

the *parameters* of the *classifier* are adjusted at the end of the presentation of a whole *training input sequence*. This mode is restricted to applications where only *finite length sequences* are considered; moreover, *run phase* and *training phase* are strictly distinguished in this mode.

On-line

the *parameters* of the *classifier* are adjusted during the presentation of the *training input sequence*; therefore, we have non restriction on the length of that *sequence*; moreover, it's possible to adjust the *parameters* of the *classifier* while it is in use.

The *associator*, based on the output of the preceding *dynamic network*, discriminates the class that the *training input sequence* belongs to.

Let's consider a *training input sequence* that belongs to one of the *classes* considered by the *classifier* : $\mathbf{x}_k[t] \in C_d$, and train the *classifier* according to the relative *training target sequence* $\boldsymbol{d}[t]$ (in case of an N_M *classes classifier*, each sample of this *sequence* is an N_M – dimensional array). When the *training input pattern* $\mathbf{x}_k[t] \in C_d$ is presented to the *classifier*, after a proper decision delay, the d^{th} component of the *training target sequence* $\mathbf{d}_k[t]$ will be equal to 1, for the whole length of $\boldsymbol{x}_k$. $d_d[t]$ is a *sequence* that forms the d^{th} component of $\mathbf{d}_k[t]$. It's a "squared" sequence, that is it's equal to 1 if and only if $\boldsymbol{x}_k[t] \in C_d$, otherwise it's always equal to 0. A *backpropagation* algorithm is based on the computation of the *error sequence* at the input of the *associator*. Since the *associator* is memory-less, the *error sequences* to be propagated thru the multilayer structure will be binary valued. Experience has shown that significantly better results can be obtained if we introduce memory also in the *associator*:

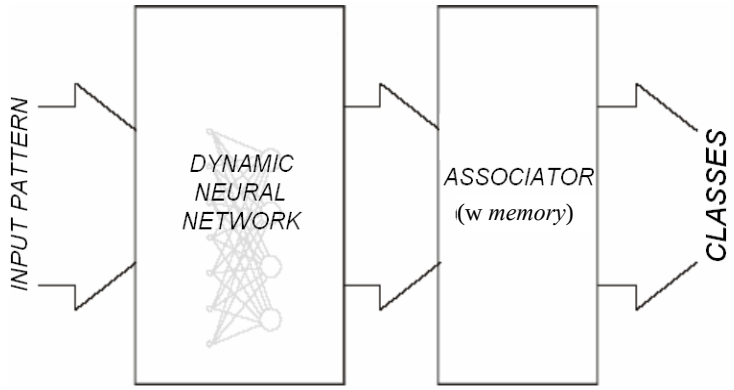


Fig. 2.46. a Dynamic Classifier with an output Dynamic Associator.

2.7.2.1 Minimum Classification Error in Time Series Classification

Let's define some quantities:

Λ = set of *parameters* of the *classifier* (2.149)

Γ = set of all the considered *classes* (2.150)

C_d = d^{th} *class* ($C_d \in \Gamma$) (2.151)

D = *cardinality* of Γ (2.152)

K = *cardinality* of the *training sequence set* (2.153)

input training sequence

$$\begin{aligned}&\mathbf{x}_k : \{\mathbf{x}_k[T],\ \mathbf{x}_k[T-1],\ \ldots,\ \mathbf{x}_k[t],\ \ldots,\ \mathbf{x}_k[1]\}\\&\mathbf{x}_k[t] = \{x_{k1}[t],\ \ldots,\ x_{kn}[t],\ \ldots,\ x_{kN_0}[t]\}\end{aligned} \tag{2.154}$$

belonging class of $\mathbf{x}_k$

$$C_d \qquad d = 1, \ldots, N_{\mathrm{M}} \tag{2.155}$$

cost function

qualified to quantify the *degree of membership* of $\boldsymbol{x}_k \in C_d$

$$l_d(\boldsymbol{x}_k[t], \boldsymbol{x}_k[t-1], \ldots, \boldsymbol{x}_k[t-p]; \boldsymbol{\Lambda}) \rightarrow \{0,1\} \tag{2.156}$$

and such that (it's 1 in presence of an error in the *classification*) :

$$l_d(\boldsymbol{x}_k ; \boldsymbol{\Lambda}) = \begin{cases} 1 & \boldsymbol{x}_k \notin C_d \\ 0 & \boldsymbol{x}_k \in C_d \end{cases} \tag{2.157}$$

The *cost function* at time *t* depends on the ordered set of the *input patterns* presented to the *classifier* starting from time *n - p*, as well as on the *parameters* of the *classifier* $\boldsymbol{\Lambda}$ always at time *t*. Parameter *p* defines the length of the time window in which the *classifier* can work; therefore, it can be seen as the *temporal depth* of the *classifier* itself. We define the *cumulative membership cost* relatively to the *sequence* $\boldsymbol{x}_k : \{\boldsymbol{x}_k[t],\ \boldsymbol{x}_k[t-1],\ \ldots,\ \boldsymbol{x}_k[1]\}$ as:

$$E_d[\mathbf{x}_k(t,1)] = \frac{1}{t}\sum_{\tau=t_0}^{t} l_d(\mathbf{x}_k[\tau]; \boldsymbol{\Lambda}[\tau]) \tag{2.158}$$

that is by accumulating all the contributions of the *cost function* $l_d(x[t], x[t-1], \ldots, x[t-p]; \boldsymbol{\Lambda})$ from time 1 to time *t*. We define the *generalized cumulative cost* produced by the *classification* applied to the *sequence* $\boldsymbol{x}_k(t,1)$ as:

$$E[\mathbf{x}_k(t,1)] = \frac{1}{t}\sum_{d=1}^{N_M}\sum_{\tau=1}^{t} l_d(\mathbf{x}_k[\tau]; \boldsymbol{\Lambda}[\tau])\,\delta(\mathbf{x}_k \in C_d) \tag{2.159}$$

Expression (2.159) takes into account the *cost* produced by the wrong association of the *sequence* $\boldsymbol{x}_k$ (or parts of it) to *classes* different from the *correct class* C_d. At the end of the *presentation* of

the whole *input sequence*, expression (2.159) can be rewritten as:

$$E[\mathbf{x}_k] = \frac{1}{T}\sum_{d=1}^{N_M}\sum_{t=1}^{T} l_d(\mathbf{x}_k[t]; \boldsymbol{\Lambda}[t])\,\delta(\mathbf{x}_k \in C_d) \tag{2.160}$$

Given a *training set* containing *K training sequences*, we define, for a *training epoch*, the *average empirical cost* as follows:

$$\tilde{E}(\boldsymbol{\Lambda}) = \frac{1}{K}\sum_{k=1}^{K} E[\mathbf{x}_k] \tag{2.161}$$

The *optimal classifier* can be found by computing that set of *parameters* that minimizes the *classification error* (*misclassification*) for every *training input sequence*. The *cumulative cost* given in (2.159) is a piecewise continuous function, being it generated starting from binary contributions {0,1} of the *cost function*. Therefore, if we want to adopt the usual *gradient based optimization algorithm*s, we have to define a *cost function* that is continuous and differentiable in its *parameters*. This problem can be solved by associating the *cost function* to the statistical interpretation of the membership relation $\mathbf{x}_k \in C_d$:

$$l_d(\boldsymbol{x}_k ; \boldsymbol{\Lambda}) \rightarrow [0,1] \tag{2.162}$$

such that:

$$l_d(\boldsymbol{x}_k ; \boldsymbol{\Lambda}) \rightarrow \begin{cases} 1 & P(\boldsymbol{x}_k \notin C_d) > P(\boldsymbol{x}_k \in C_d) \\ 0 & P(\boldsymbol{x}_k \in C_d) > P(\boldsymbol{x}_k \notin C_d) \end{cases} \tag{2.163}$$

where $P(\boldsymbol{x}_k \in C_d)$ is the probability of the sequence $\boldsymbol{x}_k$ to belong to the *class* C_d. The new *cost function* returns a measure (in terms of a scalar value between 0 and 1) of the non-belonging of a *sequence* to a *class*. Therefore, according to all we have stated above, we can assert the *optimum classifier* is the one that's characterized by an array of *parameters* $\boldsymbol{\Lambda}_{opt}$, such that the *null sequence* is produced by the *cost function* $l_d(\boldsymbol{x}_k ; \boldsymbol{\Lambda}_{opt})$ only, when $\mathbf{x}_k \in C_d$. Given a *generic sequence* $\boldsymbol{x}_k(t,1)$: $\{\boldsymbol{x}_k[t], \boldsymbol{x}_k[t-1], \ldots, \boldsymbol{x}_k[1]\}$ belonging to class C_d, if the *cost function* $l_d(\boldsymbol{x}_k ; \boldsymbol{\Lambda})$ produces a *null sequence* $l_d[\tau] = \{0,0,\ldots,0\}$ $\forall \tau \in [t,1]$, than the *membership cumulative cost* will be equal to zero: $E_d[\boldsymbol{x}_k(t,1)] = 0$. This condition, however, doesn't guarantee that even the *cumulative cost* (2.159) be equal to zero. To achieve this result, it's necessary that the *null sequence* be a consequence of the presentation of $\mathbf{x}_k$ only, or of *sequences* belonging to C_d. In other situations, a *cumulative cost* that's different form zero have to be always generated. The zeroing of the *cumulative cost*, for every presented *sequence*, is sufficient for the zeroing of the *average empirical cost* (2.161), and, as a consequence, for the *classifier* no to make mistakes. Let's remember that a *training epoch* ends with the presentation of all the *sequences* in the *training set*.

2.7.3 Locally Recurrent Multilayer Classifier

Four functional blocks form this *classifier*:

- *Cost Functions*
- *Measure of misclassification*
- *Dynamic discriminative Functions*
- *Recurrent Neural Network*

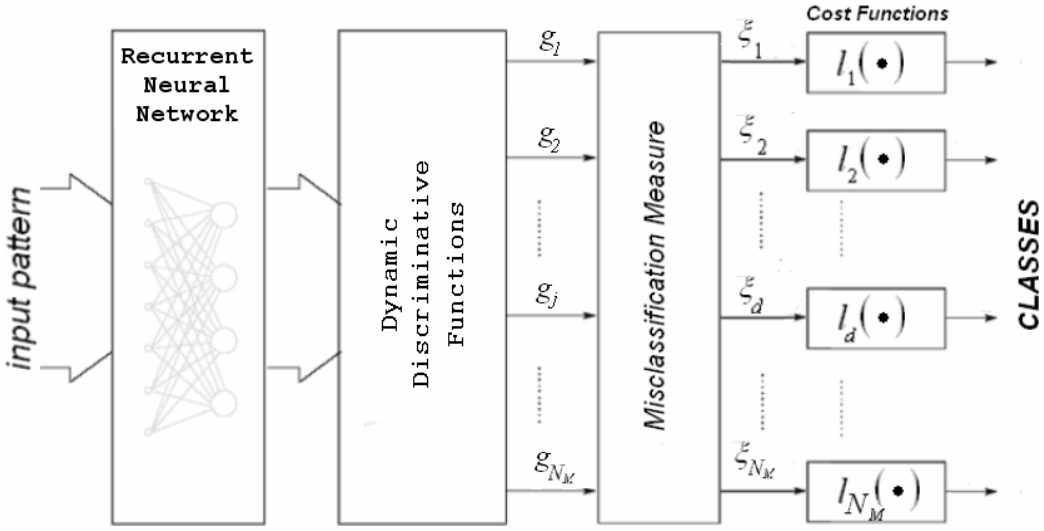


Fig. 2.47. N_M classes dynamic classifier

With the aim of processing *time dependent patterns*, we introduce a dynamic model that will be optimized by means of *gradient descent algorithms*, in order to reach the *MCE*. Therefore, in this chapter we will apply the *DL* paradigm even to the training of *dynamic multilayer architectures*, and in particular to *LRNNs*. The input block of the *classifier* is in fact an *RNN*, followed by an *associator* built around opportune *time dependent DFs*, a block devoted to the *measure of misclassification* and a block implementing the *cost function* (the latter two blocks are strictly static). The memory inside the *classifier* is distributed between the input *RNN* and the *DFs*.

2.7.3.1 Dynamic Discriminative Functions

The *associator* of the *classifier* is formed by a *layer* of *dynamic DFs*. It is the block that evaluates the *degree of membership* of a given *input sequence* to one of the considered *classes*. In a multilayer architecture, the *discrimination* is not carried out directly on the *input pattern*, but on a so called *intermediate representation*. *Dynamic DFs* can be implemented by simply adding *linear filters* to the *connections* of the *linear combiner* that's part of the *static DF*. In the static case, the *discriminative functions* implements *hyperplanes* that partition the space of all the considered *classes*. The introduction of a dynamic realizes a *sequence of hyperplanes* that depends on the actual input, as well as on the past history of the *classifier*. As a consequence, the generic *cost*

function at time *t* depends on the input, on the *parameters* of the *classifier* at time *t*, as well as on the past inputs, depending on the particular memory introduced in the *classifier*.

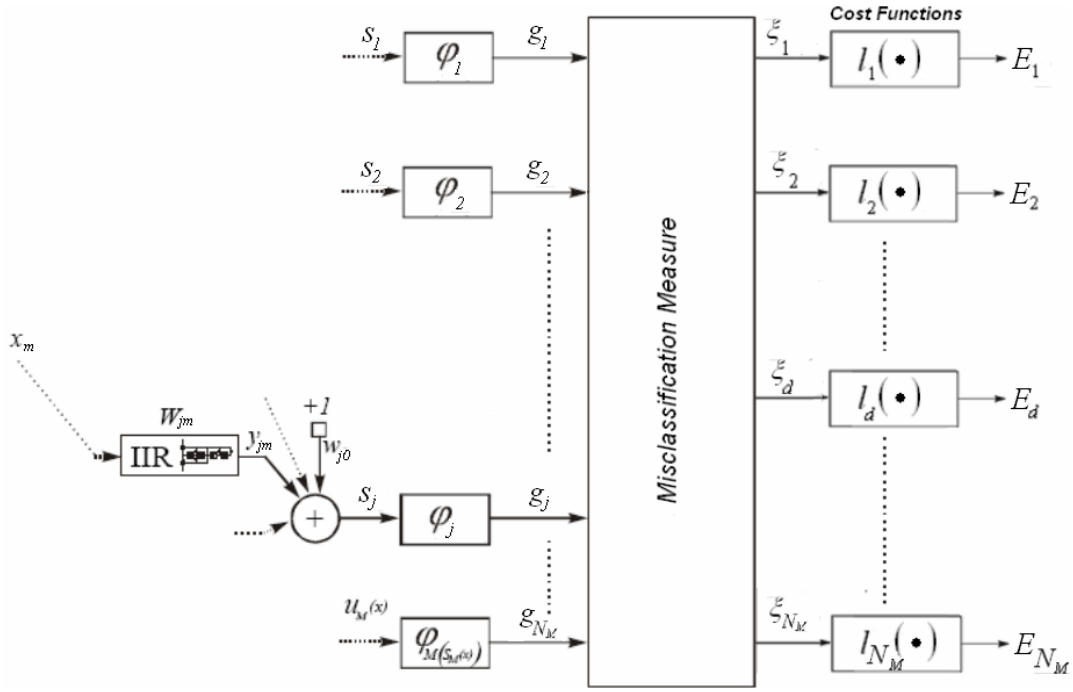


Fig. 2.48. Output layer of an N_M classes dynamic classifier

The presence of *IIR filters* guarantees to *classification* a huge temporal depth with a low computational overhead; it's only drawback is that some efforts are required in order to guarantee the stability of the system. Moreover, some problems of non causality arise in case of the *on-line* version of the *training* algorithm. The *DFs* can be defined as follows:

$$g_d\left(\mathbf{x}_k[t], \boldsymbol{\Lambda}[t]\right) = \varphi\left(s_d^{(M)}[t]\right) \qquad d = 1,2,\ldots,N_M \tag{2.164}$$

where the *activation function* is the well known *hyperbolic tangent*:

$$\varphi\left(s_d^{(M)}[t]\right) = a\,\tanh\left(b\,s_d^{(M)}[t]\right) \qquad a, b > 0 \tag{2.165}$$

and:

$$s_d^{(M)}[t] = \sum_{m=0}^{N_{M-1}} y_{dm}^{(M)}[t] \tag{2.166}$$

$$y_{dm}^{(M)}[t] = \sum_{p=0}^{L_{dm}^{(M)}-1} w_{dm(p)}^{(M)}\, x_m^{(M-1)}[t-p] + \sum_{p=1}^{I_{dm}^{(M)}} v_{dm(p)}^{(M)}\, y_{dm}^{(M)}[t-p] \tag{2.167}$$

where $L_{dm}^{(M)} - 1$ is the order of the *MA* of the *IIR filter* and $I_{dm}^{(M)}$ is the order of the *AR* of the same *filter*; moreover, $w_{dm(p)}^{(M)}$ (p = 0, 1, ..., $L_{dm}^{(M)} - 1$) are the *coefficients* of the *MA* and $v_{dm(p)}^{(M)}$ (p = 1, ..., $I_{dm}^{(M)}$) are the *coefficients* of the *AR*. The *measure of misclassification* is a continuous and differentiable function of the *parameters* $\boldsymbol{\Lambda}$ of the *classifier* (thanks to the nature of the *IIR filter* and to the definition of the chosen *activation function*):

$$\xi_d(\mathbf{x}_k[t],\boldsymbol{\Lambda}[t]) = -g_d(\mathbf{x}_k[t],\boldsymbol{\Lambda}[t]) + \left[\frac{1}{N_M - 1}\sum_{h\neq d} g_h(\mathbf{x}_k[t],\boldsymbol{\Lambda}[t])^{\eta}\right]^{\frac{1}{\eta}} \qquad d = 1,2,\ldots,N_M \tag{2.168}$$

The *cost function* is a function of the *misclassification*; amongst the possibilities at our disposal we have opted for the *translated sigmoid*:

$$E_d = l_d(\xi_d) = \frac{1}{1+e^{-\beta(\xi_d+\alpha)}} \tag{2.169}$$

where $\beta > 0$. According to the usual *gradient descent* algorithm, the variation of the *filter parameters* on the m^{th} *connection* of the j^{th} *DF* at time *t* is as follows:

$$\Delta w_{jm(p)}^{(M)}[t] = -\mu \left.\frac{\partial E_d}{\partial w_{jm(p)}^{(M)}}\right|_t \tag{2.170}$$

$$\Delta v_{jm(p)}^{(M)}[t] = -\mu \left.\frac{\partial E_d}{\partial v_{jm(p)}^{(M)}}\right|_t \tag{2.171}$$

where μ is the *learning rate*. Let's define the *local gradient* δ_j at time *t* for the j^{th} *DF* (*layer M* ≡ *discriminative layer*):

$$\delta_j^{(M)} = -\frac{\partial E_d}{\partial s_j^{(M)}} \tag{2.172}$$

As a consequence, it holds (ignoring the time index):

$$\Delta w_{jm(p)}^{(M)} = -\mu \frac{\partial E_d}{\partial s_j^{(M)}} \frac{\partial s_j^{(M)}}{\partial w_{jm(p)}^{(M)}} = \mu \delta_j^{(M)} \frac{\partial s_j^{(M)}}{\partial w_{jm(p)}^{(M)}} \tag{2.173}$$

$$\Delta v_{jm(p)}^{(M)} = -\mu \frac{\partial E_d}{\partial s_j^{(M)}} \frac{\partial s_j^{(M)}}{\partial v_{jm(p)}^{(M)}} = \mu \delta_j^{(M)} \frac{\partial s_j^{(M)}}{\partial v_{jm(p)}^{(M)}} \tag{2.174}$$

In particular:

$$\frac{\partial E_d}{\partial s_j^{(M)}} = \frac{\partial E_d}{\partial \xi_d} \frac{\partial \xi_d}{\partial g_j^{(M)}} \frac{\partial g_j^{(M)}}{\partial s_j^{(M)}} = l_d'(\xi_d)\varphi'(s_j^{(M)}) \frac{\partial \xi_d}{\partial g_j^{(M)}} \tag{2.175}$$

Moreover, as previously shown:

$$\frac{\partial \xi_d}{\partial g_j^{(M)}} = \begin{cases} -1 & j = d \\ \dfrac{(g_j^{(M)})^{\eta-1}}{N_M - 1}\left[\dfrac{1}{N_M - 1}\displaystyle\sum_{h\neq d}(g_h^{(M)})^{\eta}\right]^{\frac{1}{\eta}-1} & j \neq d \end{cases} \tag{2.176}$$

or:

$$\delta_j^{(M)} = \begin{cases} l_d'(\xi_d)\varphi'(s_j^{(M)}) & j = d \\ -l_d'(\xi_d)\varphi'(s_j^{(M)})\dfrac{(g_j^{(M)})^{\eta-1}}{N_M - 1}\left[\dfrac{1}{N_M - 1}\displaystyle\sum_{h\neq d}(g_h^{(M)})^{\eta}\right]^{\frac{1}{\eta}-1} & j \neq d \end{cases} \tag{2.177}$$

Besides, starting from the *adaptive filtering* theory, we have shown that :

$$\begin{aligned}\frac{\partial s_j^{(M)}[t]}{\partial w_{jm(p)}^{(M)}} &= \frac{z^{-p}}{1 - A_{jm}(t,z)} x_m^{(M-1)}[t] \\ \frac{\partial s_j^{(M)}[t]}{\partial v_{jm(p)}^{(M)}} &= H_{jm}(t,z)\frac{z^{-p}}{1 - A_{jm}(t,z)} x_m^{(M-1)}[t]\end{aligned} \tag{2.178}$$

where:

$$A(t,z) = \sum_{p=1}^{I} v_p[t]\, z^{-p} \qquad \text{and} \qquad B(t,z) = \sum_{p=0}^{L-1} w_p[t]\, z^{-p} \tag{2.179}$$

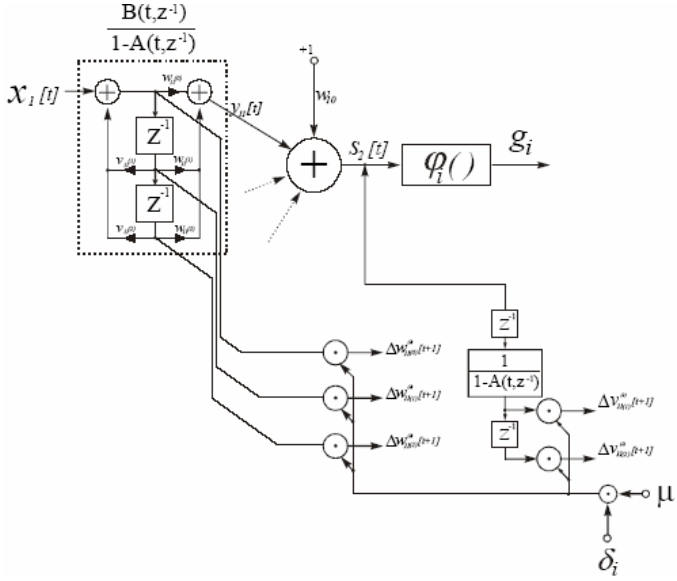


Fig. 2.49. generation of the IIR coefficient adjustements

If during the presentation of $\boldsymbol{x}_k(t,1)$ the *cost function* l_d generates a sequence of vanishing values (correct *classification*), at the output of the d^{th} *DF*, we will necessarily have to find the so called *dominant sequence*, that is that prevails against all the other *DFs*. The prevailing of the sequence $g_d = \{g_d[t], g_d[t-1], \ldots, g_d[1]\}$ let us to presume that the *non linear activation function* φ_d has generated g_d in saturation conditions, where $\varphi_d' \to 0$ and therefore where the *local gradient* is inhibiting unwanted *parameter* variations:

$$\delta_d^{(M)}[t] \to 0, \quad \tau \in [1 \ldots t] \qquad \Rightarrow \qquad \begin{cases} \Delta w_{dm(p)}^{(M)} \to 0 \\ \Delta v_{dm(p)}^{(M)} \to 0 \end{cases} \tag{2.180}$$

The presence of one or more *hidden recurrent layers* (containing *locally recurrent neurons* of the *IIR-MLP* kind) preceding the *discriminative layer* improves the performances of the *classifier*. Besides, so far, we have always considered the *discriminative layer* as the last layer of a *RNN* with *IIR synapses* and *hyperbolic tangent activation function*. The whole multilayer structure of the *classifier*, inclusive of the *DFs*, is essentially a *multilayer LRNN* with *IIR synapses*. The *training* algorithm for this structure is a variation of the previously introduced *RPB* (or its *on-line* version, *CRBP*), where instead of reducing the *global mean-quare error* for the whole *input sequence*, this time we reduce the *classification error*. Let's compute the *local gradient* for the generic *neuron* in

the generic *hidden layer* preceding the *discriminative layer*. The continuity and the differentiability of the *measure of misclassification* and of the *cost function* allow us to write:

$$\frac{\partial E_d}{\partial x_j^{(M-1)}} = \sum_{h=1}^{N_M} \frac{\partial E_d}{\partial s_h^{(M)}} \frac{\partial s_h^{(M)}}{\partial x_j^{(M-1)}} \tag{2.181}$$

Given a *sequence*:

$$\boldsymbol{x}_k(t_1, t_0) : \{\boldsymbol{x}_k[t_1],\ \boldsymbol{x}_k[t_1 - 1],\ \ldots,\ \boldsymbol{x}_k[t_0]\} \qquad , \qquad \boldsymbol{x}_k \in C_d \tag{2.182}$$

it holds:

$$\left.\frac{\partial E_d}{\partial x_j^{(M-1)}}\right|_t = \sum_{h=1}^{N_M} \sum_{\tau=t_0}^{t_1} \frac{\partial E_d}{\partial s_h^{(M)}[\tau]} \frac{\partial s_h^{(M)}[\tau]}{\partial x_j^{(M-1)}[t]} = -\sum_{h=1}^{N_M} \sum_{\tau=t_0}^{t_1} \delta_h^{(M)}[\tau] \frac{\partial s_h^{(M)}[\tau]}{\partial x_j^{(M-1)}[t]} \tag{2.183}$$

But we know that:

$$\left.\frac{\partial E_d}{\partial x_j^{(M-1)}}\right|_t = -\sum_{h=1}^{N_M} \sum_{r=t_0-t}^{t_1-t} \delta_h^{(M)}[t+r] \frac{\partial y_{jh}^{(M)}[t+r]}{\partial x_j^{(M-1)}[t]} \tag{2.184}$$

and that:

$$\frac{\partial y_{jh}^{(M)}[t+r]}{\partial x_j^{(M-1)}[t]} = \begin{cases} w_{jh(r)}^{(M)} & 0 \le r \le L_{jh}^{(M)} - 1 \\ 0 & otherwise \end{cases} + \sum_{p=1}^{\min\left(I_{jh}^{(M)}, r\right)} v_{jh(p)}^{(M)} \frac{\partial y_{jh}^{(M)}[(t+r)-p]}{\partial x_j^{(M-1)}[t]} \tag{2.185}$$

and so on.

2.7.4 Multi-Classification Discriminative-Learning

The *DL* model is attractive, but it can't be applied directly in all those cases where an *input sequence* can belong to more than one of the considered *classes*. Therefore, we need to re-consider the basic idea of *discrimination*: we need a procedure that takes into account the fact that there's the possibility of a *not complete separation* between *classes* and that some input could lie in that common territory. Let's suppose we have the usual *training input sequence* presented to the *classifier*:

$$\begin{aligned} &\boldsymbol{x}_k : \{\boldsymbol{x}_k[T],\ \boldsymbol{x}_k[T-1],\ \ldots,\ \boldsymbol{x}_k[t],\ \ldots,\ \boldsymbol{x}_k[1]\} \\ &\boldsymbol{x}_k[t] = \{x_{k1}[t],\ \ldots,\ x_{kn}[t],\ \ldots,\ x_{kN_0}[t]\} \end{aligned} \tag{2.186}$$

Be Γ the set of all the available *classes* and be $\Gamma_k[t]$ the subset of Γ that contains only the *classes* that at time *t* the sequence $\boldsymbol{x}_k$ belongs to:

$$\mathbf{x}_k[t] \in C_{d_1}, \quad \mathbf{x}_k[t] \in C_{d_2}, \quad \ldots, \quad \mathbf{x}_k[t] \in C_{d_{D_k}} \tag{2.187}$$

or:

$$\Gamma_k[t] = \{C_{d_1}, C_{d_2}, \ldots, C_{d_{D_k}}\}_t \subseteq \Gamma \quad 1 \le d_h \le N_M \quad 1 \le h \le D_k \tag{2.188}$$

where:

D = ***cardinality*** of Γ

D_k = ***cardinality*** di Γ_k $\qquad (1 \le D_k \le N_M)$

Accordingly, the *cost function* can be defined as follows:

$$l_{\Gamma_k}(\boldsymbol{x}_k[\mathrm{t}\ldots 1];\boldsymbol{\Lambda}[t]) \to [0,1] \tag{2.189}$$

continuous and differentiable function such that if $l_{\Gamma_k} = 1$ we are in presence of a *misclassification* and such that

$$l_{\Gamma_k} = \begin{cases} 1 & \boldsymbol{x}_k \notin \Gamma_k \\ 0 & \boldsymbol{x}_k \in \Gamma_k \end{cases} \tag{2.190}$$

quantifies the *degree of membership* $\mathbf{x}_k[t] \in \Gamma_k[t]$. By this improper formulation, we mean that, at a given time *t*, the sample of the *input sequence* $\mathbf{x}_k[t]$ belongs to all the classes that are part of the subset $\Gamma_k[t]$. In other words, the *cost function* gives a measure (scalar value between 0 and 1) of the *non membership* of a *sequence* to a *sequence of classes* and allow us to consider the *classifier* as a *fuzzy logic classifier*. The *cost function* at time *t*, $l_{\Gamma_k}(\boldsymbol{x}_k[\mathrm{t}\ldots 1];\boldsymbol{\Lambda}[t])$, depends on the ordered set of all the *input patterns* presented to the *classifier* from time 1 to time *t*, as well as on the *parameters* $\boldsymbol{\Lambda}$ of the *classifier* itself at time *t*. We define a *cumulative cost function*

$$E_{\Gamma_k}(\boldsymbol{x}_k[t\ldots 1]) = \frac{1}{t}\sum_{\tau=1}^{t} l_{\Gamma_k}(\boldsymbol{x}_k[\tau];\boldsymbol{\Lambda}[\tau]) \tag{2.191}$$

and a *global cumulative cost function*

$$E[\mathbf{x}_k] = \frac{1}{TN_M}\sum_{d=1}^{N_M}\sum_{t=1}^{T} l_d(\mathbf{x}_k[t];\boldsymbol{\Lambda}[t])\,\delta(\mathbf{x}_k \in C_d) \tag{2.192}$$

generated by the *classification* of *sequence* $\boldsymbol{x}_k$. Given a set of *K training input sequences*, we define, for each *training epoch*, the *average empirical cost* as follows:

$$\tilde{E}(\boldsymbol{\Lambda}) = \frac{1}{K}\sum_{k=1}^{K} E[\boldsymbol{x}_k] \tag{2.193}$$

As a consequence of the choice of continuous and differentiable *cost function*, we have a *global cumulative cost function* that's continuous and differentiable too as well as the chance of exploiting *gradient descent optimization methods.* The *optimal classifier* can be found by computing that set of *parameters* $\boldsymbol{\Lambda}_{opt}$ that minimizes the *average empirical cost* (*misclassification*) for every *training input sequence*, or, in other words, such that *null sequences* are produced by the *cost function* $l_{\Gamma_k}(\boldsymbol{x}_k;\boldsymbol{\Lambda}_{opt})$, only at that times at which $\mathbf{x}_k[t] \in \Gamma_k[t]$. Therefore, at those times, the *cumulative cost* will be null: $E_{\Gamma_k}(\boldsymbol{x}_k[t\ldots 1]) = 0$. This condition, however, doesn't guarantee that even the *global cumulative cost* be equal to zero, unless the *null sequence* be a consequence of the presentation of $\boldsymbol{x}_k$ only, or of *sequences* belonging to Γ_k. Other situations always generate a *global cumulative cost* that's different form zero. The zeroing of the *global cumulative cost*, for every presented *sequence*, is sufficient for the zeroing of the *average empirical cost*, and, as a consequence, for the *classifier* not to make mistakes. Let's remember that a *training epoch* ends with the presentation of

all the *sequences* in the *training set*. We can define the usual *DFs* :

$$g_{\Gamma_k[t]}\left(\mathbf{x}_k[t\ldots 1],\boldsymbol{\Lambda}[t]\right)=\varphi\left(s^{(M)}_{\Gamma_k[t]}[t]\right) \quad : \quad \begin{aligned} g_{d_1}\left(\mathbf{x}_k[t\ldots 1],\boldsymbol{\Lambda}[t]\right)&=\varphi\left(s^{(M)}_{d_1}[t]\right)\\ g_{d_2}\left(\mathbf{x}_k[t\ldots 1],\boldsymbol{\Lambda}[t]\right)&=\varphi\left(s^{(M)}_{d_2}[t]\right)\\ &\vdots\\ g_{d_{D_k}}\left(\mathbf{x}_k[t\ldots 1],\boldsymbol{\Lambda}[t]\right)&=\varphi\left(s^{(M)}_{d_{D_k}}[t]\right)\end{aligned} \tag{2.194}$$

where we adopt the usual *hyperbolic tangent activation function*:

$$\varphi\left(s^{(M)}_{d_h}[t]\right)=a\tanh\left(b\,s^{(M)}_{d_h}[t]\right) \qquad a,\mathrm{b}>0 \qquad h=1,\ldots,D_k[t] \tag{2.195}$$

To find a *training* process that takes into account the non complete separation between the *classes* that a given *input sequence* at a given time *t* belongs to, we could intuitively apply a sort of superimposition of effects in which we consider a *multiple classes input*:

$$\mathbf{x}_k\in\Gamma_k\subseteq\bigcup_{n=1}^{N_M}C_n \tag{2.196}$$

as belonging separately to each one of the right D_k *classes*, and then combine the results. We also need to define again the *decision rule* in case of *multi-classification*. This time, an input $\mathbf{x}_k$ belongs to a set of *classes* $\mathbf{x}_k\in\Gamma_k$, where $\Gamma_k=\left\{C_{d_1},C_{d_2},\ldots,C_{d_{D_k}}\right\}_t\subseteq\Gamma$. The values returned by the *DFs* associated to each one of these *classes* must satisfy a conditional rule that quantifies the *degree of membership*, as for example a threshold function. The extension to the *multi-class membership* can be as follows:

$$C\left(\mathbf{x}_k\right)=\left\{C_{d_1},C_{d_2},\ldots,C_{d_{D_k}}\right\} \tag{2.197}$$

where

$$g_{d_1}\left(\mathbf{x},\boldsymbol{\lambda}_{d1}\right)=\max_h g_h\left(\mathbf{x},\boldsymbol{\lambda}_h\right) \qquad h=1,\ldots,N_M \tag{2.198}$$

$$g_{d_1}\left(\mathbf{x},\boldsymbol{\lambda}_{d_1}\right)>\cdots>g_{d_j}\left(\mathbf{x},\boldsymbol{\lambda}_{d_j}\right)>\cdots>g_{d_{D_k}}\left(\mathbf{x},\boldsymbol{\lambda}_{d_{D_k}}\right) \tag{2.199}$$

$$g_{d_{D_k}}\left(\mathbf{x},\boldsymbol{\lambda}_{d_{D_k}}\right) \quad > \quad \min_h g_h\left(\mathbf{x},\boldsymbol{\lambda}_h\right) \qquad h=1,\ldots,N_M \tag{2.200}$$

that is $\mathbf{x}_k$ belongs to a set of *classes* $\in\Gamma_k$, where C_{d_1} is the *class* with the maximum *discriminative value* and $C_{d_{D_k}}$ is with the minimum *discriminative value* that ensures the *membership*. The problem with this approach consists in that we don't a priori know the number of *classes* that the generic input $\mathbf{x}_k$ belongs to. Therefore the definition of a rule is required that allows to quantify a *membership condition*, such as a *threshold function*, or an *integrity test*. A *measure of misclassification* for a *multi-classification learning* algorithm can be defined as follows. Taken into account one of the *classes* $\mathbf{x}_k$ belongs to, C_{d_n} $(n=1,\ldots,D_k)$, the relative *measure of misclassification* can be defined as:

$$\xi_{d_n}(\mathbf{x}) \;=\; -g_{d_n}(\mathbf{x},\boldsymbol{\lambda}_{d_n})+\left[\frac{1}{N_M-D_k}\sum_{k\notin\{d_1,\ldots,d_{D_k}\}} g_k(\mathbf{x},\boldsymbol{\lambda}_k)^{\eta}\right]^{\frac{1}{\eta}} \tag{2.201}$$

The *weight adjustment rule* could be the following:

$$\Delta w_{jm}^{(l)} = -\mu \sum_{h\in\{d_1,\ldots,d_{D_k}\}} \frac{\partial l_h}{\partial w_{jm}^{(l)}} = -\mu \sum_{h\in\{d_1,\ldots,d_{D_k}\}} \frac{\partial l_h}{\partial s_j^{(l)}} \frac{\partial s_j^{(l)}}{\partial w_{jm}^{(l)}} = \mu\, x_m^{(l-1)} \sum_{h\in\{d_1,\ldots,d_{D_k}\}} \delta_{jh}^{(l)} \tag{2.202}$$

where $\delta_{jh}^{(l)}$ is the *delta* for an input **x** that belongs to the h^{th} *right class* among the D_k, according to the *superimposition of effects* rule. Let's consider the *delta rule* for the last *layer*:

$$\delta_{jh}^{(M)} = -\frac{\partial l_h}{\partial s_j^{(M)}} = -\frac{\partial l_h}{\partial \xi_h}\frac{\partial \xi_h}{\partial g_j^{(M)}}\frac{\partial g_j^{(M)}}{\partial s_j^{(M)}} = -l_h'\left(\xi_h\right)\varphi'\left(s_j^{(M)}\right)\frac{\partial \xi_h}{\partial g_j^{(M)}} \qquad h\in\{d_1,\ldots,d_{D_k}\} \tag{2.203}$$

By repeating computation previously made, we can find that:

$$\frac{\partial \xi_h}{\partial g_j^{(M)}} = \begin{cases} -1 & j\in\{d_1,\ldots,d_{D_k}\} \\ \dfrac{\left(g_j^{(M)}\right)^{\eta-1}}{N_M-D_k}\left[\dfrac{1}{N_M-D_k}\displaystyle\sum_{j\notin\{d_1,\ldots,d_{D_k}\}}\left(g_j^{(M)}\right)^{\eta}\right]^{\frac{1}{\eta}-1} & j\notin\{d_1,\ldots,d_{D_k}\} \end{cases} \qquad h\in\{d_1,\ldots,d_{D_k}\} \tag{2.204}$$

Combining the results above:

$$\Delta w_{jm}^{(l)} = \begin{cases} \mu\, x_m^{(l-1)}\varphi'\left(s_j^{(M)}\right)\displaystyle\sum_{h\in\{d_1,\ldots,d_{D_k}\}} l_h'\left(\xi_h\right) & j\in\{d_1,\ldots,d_{D_k}\} \\ -\mu\, x_m^{(l-1)}\varphi'\left(s_j^{(M)}\right)\dfrac{\left(g_j^{(M)}\right)^{\eta-1}}{N_M-D_k}\left[\dfrac{1}{N_M-D_k}\displaystyle\sum_{j\notin\{d_1,\ldots,d_{D_k}\}}\left(g_j^{(M)}\right)^{\eta}\right]^{\frac{1}{\eta}-1}\displaystyle\sum_{h\in\{d_1,\ldots,d_{D_k}\}} l_h'\left(\xi_h\right) & j\notin\{d_1,\ldots,d_{D_k}\} \end{cases} \tag{2.205}$$

If an input **x** belongs to no *classes*, no *adjustment* is made ($\Delta w_{jm}^{(l)} = 0$).

CHAPTER 3

AN APPLICATION OF DYNAMIC CLASSIFICATION: AUTOMATIC TRANSCRIPTION OF POLYPHONIC PIANO MUSIC

3.1 Automatic Music Transcription

"*Music Transcription*" is the act through which a man writes down by listening the *score* of a musical piece, according a given music notation. "*Automatic Music Transcription*" (*AMT*) is the act through which the problem of finding the *score* is delegated to an artificial system (Fig. 3.1).

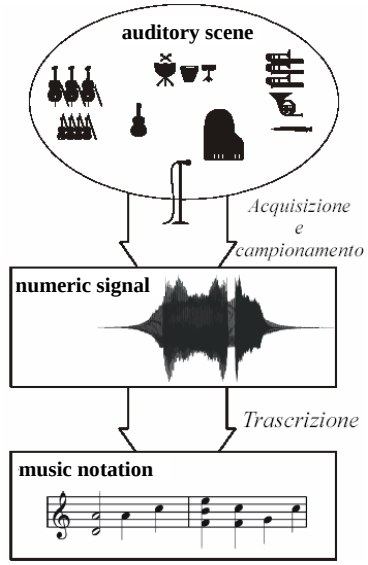


Fig. 3.1: Flux of information in a transcription process

In other words, we can say that the *transcription process* consists in moving from a *low level representation* (the *waveform*) to a *high level representation* (the *score* in *musical notation*). Substantially, the problem is broken up into several levels of abstraction, where the information is represented in a more and more abstract way. In practice, sometimes an *intermediate representation*, between the *waveform* and the *score*, is adopted and it depends on the particular approach to the problem to be solved. An *intermediate representation* allows us to reduce the dimensions of the input to the *transcription system*, without loosing any information that could be vital for the *transcription* itself. Finding the *score* means finding a parametric representation of the

acoustic waveform, that is: *notes*, *intensity*, *starting times*, *durations*, *instruments* and other sound features. Though with some difficulties, a man can learn techniques that enable him to transcribe music; the difficulties depend on the number of *instruments* (the *polyphony* of *instruments*), the genre, the clearness of sounds, the number of notes played at the same time for each *instrument* (the polyphony of the single *instrument*), the velocity of the piece and (last but not least) the ability (or the innate attitude) of the listener to find the score. The extraction of peculiar information from a mixture of sounds requires robust algorithms the performances of which deteriorate with the increasing of noise. Up to now, an automatic system that can transcribe without errors the score of a complex musical piece doesn't exist. Maybe, it is impossible to develop the perfect *transcriber*, but some approaches can lead to interesting results if applied with restrictions. For example, the *transcription* of a *monophonic* signal (a sound produced by a single *instrument* that plays a single note at time) is practically a solved problem. The adopted algorithms are based on time domain techniques, such as zero-crossing, autocorrelation, as well as on frequency domain techniques, such as *DFT* and derivates, *Cepstrum* et al.

3.1.1 Polyphonic transcription

Polyphonic transcription has turn out to be a problem hard to be solved. In the most general case, we have to deal with the analysis of an *instrument* inside a musical context where the sound produced by other *instrument* is present too. It can be seen as the most critical scenario (presence of high level noise that's correlated to the signal of interest), both for automatic and human transcription. A first simplification can be introduced if we limit ourselves to the transcription of the *polyphony* of one and only *instrument* (reduction of the noisy context). The methods adopted in the *monophonic* case are hardly suited to the *polyphonic* case, since the *harmonic series* of different sounds typically extend on common frequency ranges. As a consequence, it's harder to assign an *harmonic* to its real *fundamental*: if two *partials* overlap, their amplitude and phase envelopes cannot be singled out anymore. Moorer was the first to highlight the most incisive among the problems inherent to *transcription*: *octave errors*. They are produced when two *notes* simultaneously generated by the same *instrument* share one or more *partials*. The recognition of *octave intervals* is a critical problem, since the highest *note* can have all the *partials* in common with the lowest *note*. If we don't make any assumption on the amplitude of the *partials* (by giving, for example, the model of the *instrument*), the detection of the higher *note* could be impossible, as shown in Fig. 3.2: the spectra generated by a *piano* represent the worst case, since the higher *note* is completely masked by the lower one. To solve this problem, we go and see the knowledge we have about the model of the *instrument* (assumptions on the relative strength of the *partials* of a *piano*

note), or about the musical model adopted (i.e., counterpoint rules, allowed harmonic and melodic movements, etc.).

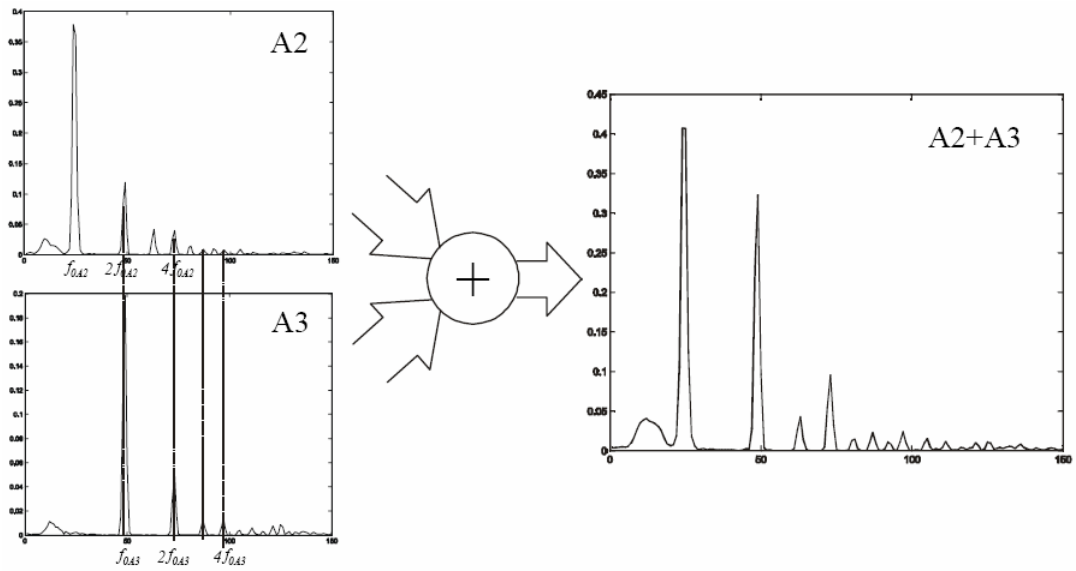


Fig. 3.2:Ambiguities in the spectra of A_2 and A_2+A_3 generated by a piano

Therefore, some good results can be achieved when the *transcription* is limited to music played on specific *instruments*. In particular, we are interested in the *Automatic Transcription of Polyphonic Piano Music*. We faced this problem as a typical *dynamic pattern recognition* topic, to be solved by exploiting the power of *LRNNs*. This model reveals its effectiveness only if completed with a *DL* strategy: it assures generalization, efficiency, coherency and less computational cost. The choice of the right *training set* becomes crucial in order to obtain a general scheme that can be adapted to every musical genre. This problem is crucial even when dealing with the human case. In fact, it is strictly related to the way we learn to recognize music: first, we have to understand what a *note* is, then what are its relationships with other *notes* (above all with near *notes*), what's *polyphony* and finally what are the peculiarities of a particular *genre*. The main applications in which *AMT* is involved are the following:

- *Format translation*

  Representation of music in a parametric, compact and standardized fashion.

- *Automatic Music Addressing*

- *Melody extraction, music segmentation and rhythmic tracking*

  Retrieving of data in musical databases

- *Music Analysis*

3.1.2 Types of Transcription Systems

All the existing *transcription systems* can by distributed into two categories, according to the kind of processing they adopt as regards the deduction of the musical information inside the *waveform* to analyze:

- *tonal mode*l based processing
- *auditory model* based processing

As regards the first approach, musical information is directly derived from the physical, or spectral, characteristics of the sound. First of all, a *time-frequency* (*t/f*) *representation* (*t/f*) of the musical signal is calculated; afterwards, the *harmonic* content is drawn, trying to locate and tracking the time evolution of the *partials*, by means of ad-hoc algorithms, such as *peak picking* and *peak connecting*. Once located, the *partial* form the base elements to start from in the *note* identification process. In order to group together the found *notes*, algorithms based on common *onset* or *harmony* are used. Sometimes, the *tonal model* of the *instrument* is used too.

The *auditory models* are inspired by the *human acoustic perception*, starting from which, structures that form the foundation of the science of *psychoacoustics* are derived. *Transcription* systems that follow this approach have been exploited with success in some *auditory scene analysis* applications; nevertheless, they lack efficacy in case of *transcription* problems, even if they show themselves applicable, when assisted by other techniques, as in case of the Marolt approach (see next chapter).

3.1.3 Previous works

The first attempts in the *transcription* of *polyphonic* music have been made by Moorer [1975 - Stanford University]. His system was based on an heuristic approach and operated on a *polyphony* of two instruments (*piano* and *guitar*). It had severe limitations on the content of the transcribed music piece: Moorer avoided *instruments* with similar *timbres* and overlapped frequency spectra (no crossing of *melody lines*, no coincidence of the *fundamental* of one voice with an *overtone* of the other voice), *vibratos*, *glissandos*, *notes* shorter than 80 ms, inharmonicity. His system achieved some surprisingly good results (even if he had a small test set at his disposal); however, the importance of Moorer's work is in the fact that some of the problems he identified (such as reverberation and octave confusion) persist to this day. Further attempts were made by Maher ['89 – '90]: *polyphony* still limited to two voices; fundamental frequency ranges non overlapped.

Kansei [Katayose and Inokuchi – Osaka, '89]: two systems for the *monophonic* transcription of traditional Japanese music and for the transcription of *polyphonic* compositions for *piano* and *guitar* o *shamisen*. They are based on information about the construction of musical *scales* in Japanese tunes, with the aim of reducing the ambiguities in human voice. The *polyphonic* system works on

five voices, but with some output errors.

Hawley [1993] : *auditory scene analysis* addressed to the *transcription* of *piano polyphonic* compositions. It was based on the Differential Spectral Analysis (differences in the distribution of two adjacent *FFTs*). The performances were good, since the set of input signal is limited and its peculiarities are taken into account.

Kashino [Tokyo University – '93] : it was based on the *human auditory* characteristics that manage the fusion or the segregation of simultaneous frequency components inside a signal. The processing was based on the *tonal model* (information about the sounds produced by the *instrument*) and included an algorithm for automatic tonal modelling (automatic extraction of the *tonal model* from a signal). In 1995, a *blackboard* architecture is introduced: flexible integration of information on the basis of different knowledge sources. A *Bayesian Network* is introduced with the aim of propagating the new information thru the system.

Keith Martin [MIT – '96] : the *blackboard* still has a central role, but high level musical knowledge is not exploited anymore as well as *neural networks*. Martin's approach, similar to that of Kashino, can be considered the state of the art in *music transcription*. The system was improved by adding a more perceptively motivated input stage the analysis of which was based on *correlograms*. In the first stage of the system, processing is accomplished by using a model derivated by the *human auditory system*.

3.1.3.1 Dixon's work

Dixon developed a system that takes *audio files* containing *polyphonic piano music* as input and produces *MIDI* output. In the first stage of processing, the signal is down-sampled to 12kHz. Then a *t/f representation* is created using a *Short Time Fourier Transform* (*STFT*); the *window*, by default, is 4096 samples (341 ms) long, containing 230 ms of signal shaped by *Hamming window* and zero-padded to fill the *window* itself. Then, the magnitude squared (power) spectrum is calculated and peaks are isolated by finding at each time point the local maxima in the frequency dimension which are above a minimum threshold and which contain at least 1% of the total power of the signal at that time. The result is a set of atoms of energy localised in time and frequency. The *t/f* atoms are then searched in the time dimension, in order to find the peaks corresponding to *note onsets*. These *onsets* are followed until the power drops below the minimum threshold, which determines the offset time of the *note*. Finally, the *velocity* is determined from the peak power that occurs at the *onset time* of the *note*. The frequency tracks so extracted represent *partials* of the musical *notes*. The final step consists in finding a set of *fundamental frequencies* which provides the best explanation for the observed frequency data, relative to an implicit generic model of *musical instruments tones*: *partials* that occur simultaneously, are harmonically related and do not fall

outside the spectral envelope for *piano tones* are combined.

3.1.3.2 Sonic

The most important related work is however *SONIC*, by Marolt. He developed a *transcription* system based on *neural networks*, remarking that they are particularly suited to the recognition of *notes* in a signal as a typical *pattern recognition* task. The system takes an acoustic *waveform* of a *piano* recording (44.100 kHz, 16 bit resolution, 1 channel) as its input; the output is a *MIDI file* containing the *transcription*. The *pre-processing* phase consists of two parts. At first, a *bank of filters* is used to split the signal into several *frequency channels*, modelling the movement of the *basilar membrane* of the *inner ear* (see later): it consists in an array of *band-pass IIR filters* called *gammatone filters*. Subsequently, the output of each *filter* is processed by the Meddis' model of the *hair cell* transduction. The *hair cell* model converts each *filter* output into a probabilistic representation of firing activity in the *auditory nerve*. The detection of periodicity in *frequency channels* (*tracking phase*) is obtained by means of *adaptive oscillators* that try to synchronize to signals inside of them. A *synchronized oscillator* means that the signal in that *channel* is periodic, or in other words, that a *partial* with frequency equal to that of the *oscillator* is present in the input signal. *SONIC* uses *adaptive oscillator networks*. Each *network* consists of up to ten interconnected *oscillators*, with initial frequencies set to integer multiplies of the frequency of the first *oscillator*. Thus each *network* tracks a group of up to ten harmonically related *partials*, eventually belonging to one *tone*. The output of a *network* represents the strength of the harmonically related *partials* tracked by the *oscillators* and provides a good indication of the presence of a *tone* in the input signal. The outputs of the *oscillator networks* are given in input to a set of *neural networks*, that operate as the *tracking* and *recognition* parts of the model. 76 *networks* are needed to recognize *notes* from *A1* to *C8*. Since each *network* is dedicated to one *note* only (the *target note*), it has just one output: a high output value indicates the presence of the *target note* in the input signal, while a low value indicates that the *note* is absent. Marolt obtained his best results with *TDNNs*. He made also use of an *onset detector* based on a model proposed by Smith: an *MLP neural network* capable of detecting repeated *notes*, as well as a tuning procedure to calculate the tuning of the *piano* and initialize frequencies of the *adaptive oscillators* accordingly.

3.2 Sound Signals generated by Musical Instrument

Sound waves have different characteristics. Perceived *pitch* is determined by *frequency*, or by how fast an object vibrates back and forth. The higher the *frequency* of an object, the higher the *pitch*. The normal *human ear* can hear sounds with *frequencies* between 20 and 20000 Hz. The amount of energy flowing in the *sound waves* is referred to as the *intensity* of sound. The *loudness* of the sound is based on the strength of the sensation received by the *eardrum* and sent to the *brain*. The same *intensity* of sound may produce different degrees of *loudness* for different people. *Intensity* and *loudness* of a sound depend on four factors: (1) how far the distance is from the source of the sound (especially in outdoor situations), (2) the *amplitude* of the *vibration*, (3) how dense the medium is through which the sound travels, and (4) the area of the vibrating object. Sound "quality" or *timbre* describes those characteristics of sound which allow the ear to distinguish sounds which have the same *pitch* and *loudness*, but are produced by different *sound sources*. *Timbre* is mainly determined by the *harmonic content* of a sound and the dynamic characteristics of the sound such as *vibrato* and the *attack-decay envelope* of the sound. A sound produced by a *musical instrument* can be described both in the *time-domain* and in the *frequency domain*. As regards the time evolution of a sound, we can distinguish three phases (Fig. 3.3).

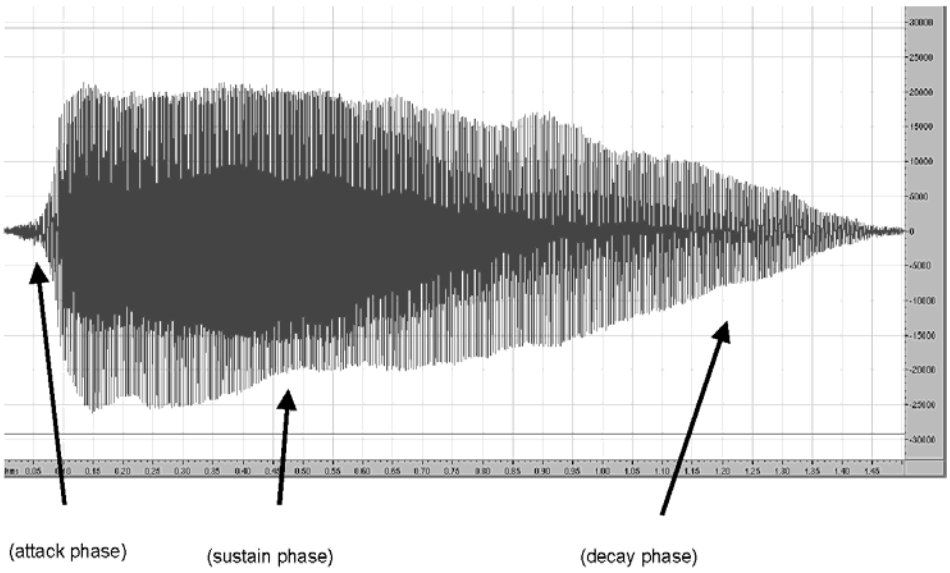


Fig. 3.3: attack, sustain and release phase of a note sound signal

- *attack phase*

  the sound is produced by all the mechanical transitory that precedes the stabilization of the stationary oscillation mechanism inside the vibrating body of the *instrument*. Therefore, the sound relative to this phase is non-periodic and almost noisy. Anyway, this phase is of great importance of our *auditory system* to recognize the *instrument* that's playing that sound.

- *sustain phase*

  the sound is due to the stationary oscillations set up in the body of the *instrument*; it is *periodic* and its spectrum is what we call the *timbre* of the *instrument* that's playing the sound. The *frequency* of the sound produced during this phase is somehow linked to the *pitch* of the played *note*.

- *release phase*

  the player stops acting on the instrument, but, during this phase, thanks to resonances typical of the body of the instrument, the sound signal still continues to persist and fades away for the damping effects introduced by friction.

If we limit ourselves to the observation of the *sustain phase* of the sound signal of a *note* played by any *musical instrument* (*pitched*, during this phase), and, by means of *FFT*, we calculate its frequency spectrum, we can see that every peak in the spectrum is associated to a note in the *natural musical scale*:

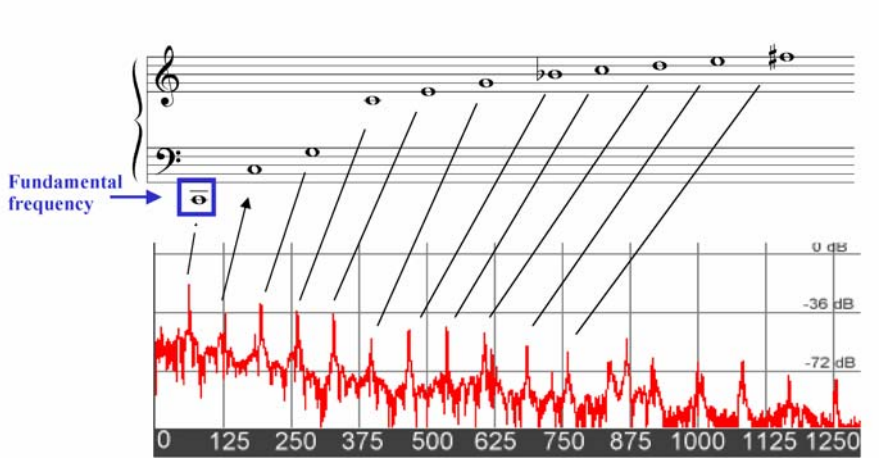


Fig. 3.4: correspondence between the peaks in the frequency spectrum and the harmonics of a note

If, on the contrary, we are interested in the whole evolution of the spectrum of the *note*, we have to go and see the *STFT*; what we can get is something like what's depicted in Fig. 3.5.

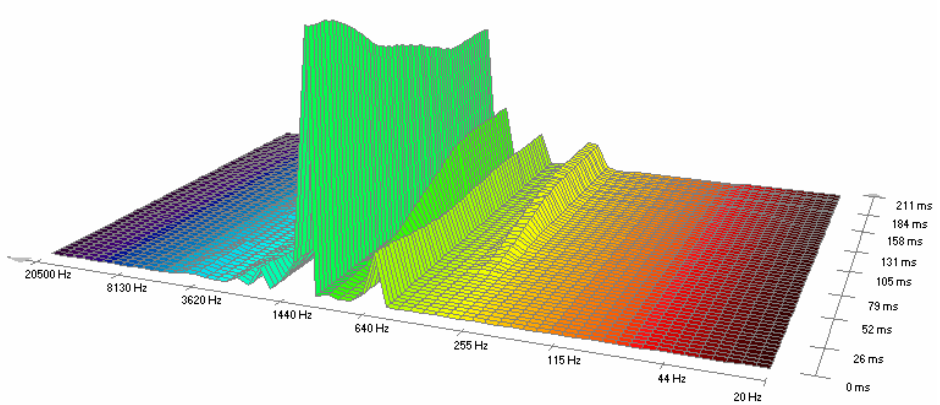


Fig. 3.5: graph of the spectrum in time

3.2.1 A taxonomy of Musical Instruments

In what follows, we will take into account only the *traditional musical instruments* used in western music. They can be shared out into classes or *families of instruments*. Organology (systematic study of *musical instruments,* from the point of view of acoustics and on the basis of homogenous criteria) starts with Mahillon (1841-1924) who defined 5 *classes*:

- *aerophones*
  make sounds as air vibrates through a tube. The vibration begins with either *lip* vibration as in a *trumpet*, a *reed* as in *clarinets*, *saxaphones* and *organs*, a ball as in a whistle or by striking a sharp edge as in a *flute*. The shape of the soundwave is determined by the width and length of the tube through which the air travels. The longer the tube, the longer the sound wave ans the deeper the sound. If a tube of one length produces the sound of *C*, then a tube twice a long will produce a sound one octave lower.

- *chordophones*
  make sound when a stretched *string* vibrates. There is usually something that makes the sound reverberate such as the body of a *guitar* or *violin*. The *strings* are set into motion by either *plucking* (like a *harp*), *strumming* (like a *guitar*) or by *rubbing* with a *bow* (like a *violin* or *cello*).

- *idiophones*
  instruments made from materials that have their own unique sounds – ceramics, glass, metal, or wood all create different vibrations. Sometimes they are hit (wood blocks, chimes), moved (bell and clapper) or shaken (rattles). They are generally considered part of the *percussion section* in a *band* or an *orchestra*.

- *membranophones*
  are instruments that make sounds when a stretched skin vibrates. Usually a *membranophone* is a *drum* which makes a sound when the membrane is hit. Hands or drumsticks are typically used to strike the skin. Some drums can be set to different pitches by tightening or loosing the tension on the skin.

- *electrophones*
  Musical instruments that create sound electronically. This does not include electronically amplified instruments (like the E-guitars) and effect devices.

The present work takes into account only some of the *instruments* used in an *orchestra*, *piano*

included. Nevertheless, we tried to choose a representative *instrument* for each one of the *families*. The considered *instruments* are:

- *Clarinet* (single reed instrument)
- *Flute* (wind instrument)
- *Oboe* (double reed instrument)
- *Piano* (hammered string instrument)
- *Saxophone* (single reed instrument)
- *Violin* (bowed string instrument)

3.2.1.1 The sound of brass instruments

The main representatives of the *brass family* of *instruments* are, among others, the *trumpet*, the *trombone*, the *french horn*, the *tuba*, and the *euphonium* (Fig. 3.6). In order to produce a sound the musician blows into the *mouthpiece* located at one of the ends of the *instrument*. As a consequence, a *stationary wave* is induced in the *tubular resonator*; its wavelength is determined by the speed of sound in the air and by the length of the *tube*. To change the *pitch* of the *note*, the musician modifies the tension on his *lips*, in order to excite *higher modes*, as well as to vary in some way the length of the acoustic channel (by means of further lateral channels that can be added acting on *valves*).

Fig. 3.6: some brass instruments

Before the perturbation has been entirely propagated from the mouth of the performer up to the point of reflection and the *stationary wave* has been generated, the pitch is unstable. Normally, the *pitch* of these *instruments* has an oscillatory behaviour around an average value; during the *attack phase* the *pitch* increases until it becomes stable around a nominal value. By means of a precise control of the lips, this effect can be reduced, but the introduction of air in the tube is never regular and as a consequence the *pitch* is never stable, expecially at *low frequencies*. Even during the *sustain phase*, when the *pitch* is almost identifiable, irregularities in the flux of air cause

irregularities in the upper modes. The spectrum varies with the pressure of the air. At low pressures, the *wave* is almost periodic, while it becomes more impulsive with the increasing of the pressure (as a consequence, the spectrum widens). Moreover, since *low frequencies* are reflected better than the higher ones, during the *attack phase*, the *low harmonics* grow more rapidly that the higher ones. Therefore, the spectrum evolves as if a high pass filtering action is occurring; the sound that reaches the listener differs from the sound generated inside the *instrument*. Therefore, the outer spectrum can be obtained by filtering the inner spectrum (Fig. 3.7).

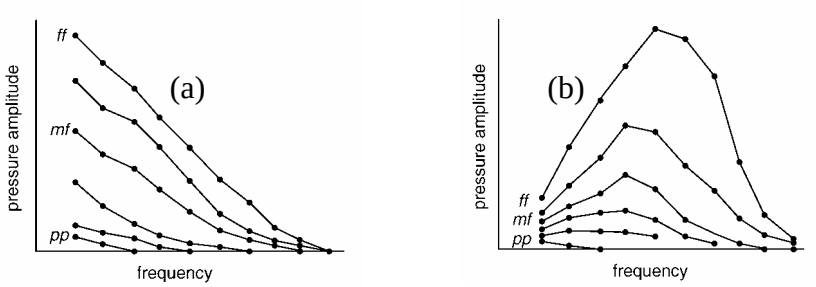


Fig. 3.7: spectrum of a brass instrument vs. different pressure levels; (a) inner spectrum; (b) outer spectrum

This filtering property is what makes the difference betweena *brass instrument* and another. The greater the *instrument*, the lower the *cut-off frequency*.

3.2.1.2 The sound of string instruments

The most common *string instruments* are: *violin*, *viola*, *cello* and *double bass* (Fig. 3.8).

Fig. 3.8: some string instruments

The *instrument* is in essence formed by a *soundboard*, a *ponticello* (connection between the soundboard and the end of the *string*s) and a *tuning system*. When a *bow* is drawn across a *string*, this start vibrating and puts the *soundboard* in resonance; the *soundboard* is responsible for the generation of the sound. Sometimes, the sound can be generated even by *plucking* the *string*. In the *sustain phase*, the *waveform* is nearly periodic (the *frequency* depends on the *string* length that the executor can vary with his finger starting from the *ponticello*). The shape of the spectrum strongly depends on the pressure applied on the *string*, let alone on the position of the *bow*. When played

close to the *nut*, the sound produced is more brilliant. In first approximation, the spectrum of a *note* resembles that of *saw-tooth waveform*. During the *attack phase*, the bow can "scratch" generating noise and the spectrum is generally not harmonic. The *low harmonics* grow very quickly when the string is excited abruptly. In case of *pizzicato notes*, the spectrum is never harmonic. The *waves* that travel at different speeds within the *string* disperse in irregular way; therefore, the *higher harmonics* are more emphasized than the lower ones. The spectrum depends on the position in which the *string* is *plucked*. The *ponticello* introduces wide resonances (*violin*: between 3 kHz and 6 kHz). The *body* of the *instrument* introduces the most important *modes of resonance*, due to the *soundboard* as well as to the resonance of the wood. The *low frequency resonances* are best balanced in the *instruments* of higher quality, while the *high frequency resonances* strongly depend on the wood and the shape of the *instrument*. Thru an analysis-synthesis procedure, it has been shown that the sound of a *violin* (strings tuned on G_3, D_4, A_4 and E_5: 196, 290, 440, 660 Hz) can be well reproduced by filtering the spectrum of a *saw-tooth* (with some zeroed *harmonics*) by means of a filter with specific *formants* (Fig. 3.9 – 3.10). The *resonance* peaks are located near *frequencies* that approximately coincide with those of the middle *strings*. A large maximum can be found around 3 kHz, due to the resonance of the *ponticello*.

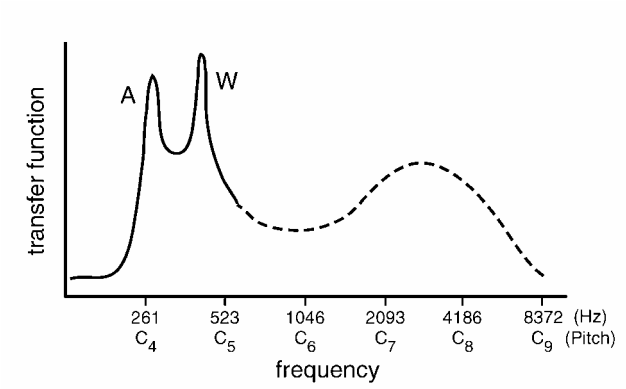


Fig. 3.9: natural frequencies of a violin. The first two modes, that of the air (A) and that of the wood (W) are normally tuned on particular frequencies. The upper modes are more complex and depend on the particular instrument. The maximum around 3 kHz is due to the resonance of the ponticello.

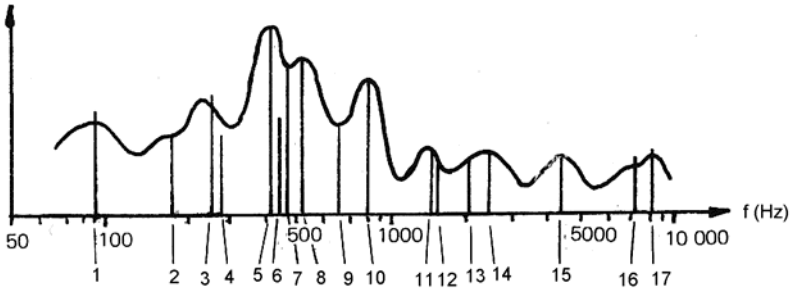


Fig. 3.10: details about the resonances of a violin: 1 ponticello, 2 G string, 3 cavity, 4 D string, 5 ponticello, left side, 6 A string, 7 neck, 8 low soundboard, 9 E string, 10 ponticello, right side, 11 ponticello, 12 G string – longitudinal oscillation, 13 ponticello, 14 D string – longitudinal oscillation, 15 A string, 16 low soundboard – longitudinal oscillation, 17 high soundboard.

Viola (C_3, G_3, D_4 and A_4: 130, 190, 290, 440 Hz): *resonances* due to air and *body* are at lower *frequencies* and less pronounced (around 230 Hz e 350 Hz); the upper *resonance* is around 2000 Hz. *Cello* (C_2, G_2, D_3 and A_3: 65, 98, 145, 220 Hz): *resonances* around 125 e 175 Hz; *upper resonance* at 1500 Hz. *Double bass* (E_1, A_1, D_2 and G_2: 41, 55, 73 e 98 Hz): *resonance frequencies* are still lower. The spectrum is further complicated by the presence of *vibrato* (the performer slightly moves the finger on the *ponticello*, modifying the *string* length and therefore the *pitch*), that is an amplitude modulation that varies in the same direction of the frequency modulation, but is not the same for all the *harmonics* (for resonance phenomena). The *sustain phase* is achieved after a long time. While for the *strings* this time can last also some fractions of a second, in the *brass instruments* this time is often lower than 100ms.

3.2.1.3 The sound of wind instruments

It's a family (Fig. 3.11) less homogeneous than the others, since it considers instruments of different kinds (*single reed*, *double reed*, *wood wind instruments*, etc.). The sound is generated by *blowing* air in an end of a *tube*, the length of which can be varied by opening and closing some *holes*. The performer can also use the technique of the *overblowing*, in order to change the *pitch* by exciting a more acute *resonance mode*.

Fig. 3.11: some wind instruments

The *open tube* structure of these *instruments* gives them a low-pass behaviour with a *cut off frequency* that is a characteristic of the quality of the *instrument* itself. Moreover, the *attack phase* of the *reed instruments* (therefore, *flute* excluded) is one of most immediate (obviously apart from that of a *piano* or an *arp*). *Double reed instruments* are: *oboe*, *english horn*, *bassoon* and *contrabassoon*. Even the spectrum of these *instrument* shows several *resonances*. Moreover, it can

be noticed that the time needed by the *waveform* and its *amplitude* to stabilize varies from instrument to instrument. It can be seen that in the *oboe* the fundamental is the first harmonic to appear, while in the *bassoon* the cut off frequency is approximately at 375 Hz. It's also worth noting that the spectrum of the *oboe* is not formed by *harmonics* that always decrease in *amplitude*, while the maximum *amplitude* is approximately obtained with the 3-5 *harmonic*, that it what confers to the sound of the *oboe* a nasal *timbre*. In the *clarinet* (*single reed instrument*), the *modes* that propagate are the even ones only; that's what causes a remarkable difference between the amplitudes of the even and the odd *harmonics*. Its spectrum is low limited by the *cut-off frequency* due to the effect of the *holes* at approximately 1200-1600 Hz and by the *cut-off frequency* of the *reed* (approximately 5 kHz). It is usually played with two *tonal modes*: once the *fundamental* of the *notes* is used and once the *higher harmonics* are exploited. The two *modes* differ by a twelfth (an eighth for the others; but the cylindrical tube of the clarinet is closed at one end). After the *cut-off frequency* there is no relevant difference between the *amplitudes* of the even and odd *modes*. The *attack times* are approximately 40 ms and 60 ms. The *fundamental* is the first to grow in *amplitude*; the *harmonics* grow more or less quickly, but after a longer time. The spectrum of the *saxophone* (*single reed instrument*) is formed by few *high harmonics*. Other details have not been defined yet. The family of the *flutes* shows characteristics remarkably different from those of the *reed instruments*. The maximum *resonance* can be found at 600 Hz and it decades with approximately 10-15 dB per *octave*. The *attack* is very slow and smoothed (approximately 160 ms) and is characterized by a periodic *amplitude modulation* (*tremolo*). For *frequencies* over 500 Hz, the spectrum is dominated by the *fundamental*; over 880 Hz the *waveform* is almost sinusoidal. As the *ottavino*, the characteristics are nearly identical: the *attack times* are respectively 25 ms and 100 ms. For *frequencies* over 1000 Hz the *waveform* is nearly sinusoidal.

3.2.1.4 The sound of the piano

The sound produced by a *piano* is originated by the *acoustic vibrations* of a stretched, fixed at both ends *string*, that's impulsively *struck* by a *hammer* and can by seen as a sequence of two acoustic phenomena:

- *attack sound*
  generated as soon as the *string* is *struck* by the *hammer*; it has short duration and is full of *non harmonic partials*.

- *residual sound*
  it has a longer duration and its frequency spectrum contains few, well defined *harmonics*.

3.2.1.4.1 Attack sound

It is an impulsive sound containing noisy components due to the vibrations of the mechanical parts of the piano.

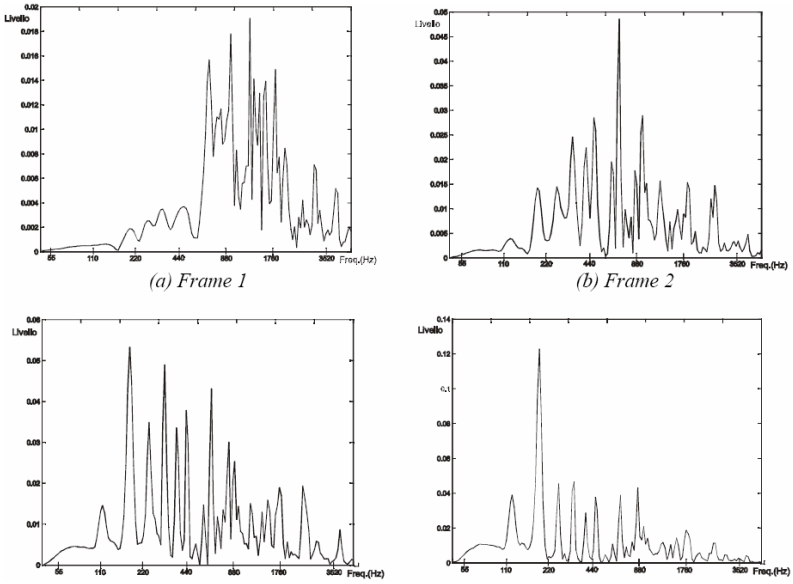


Fig. 3.12: Attack sound frames

In Fig. 3.12 a sequence of spectral analyses of the *attack* sound of C_2 is reported. We can observe the wide harmonic content in the incipient sound (frame 1 and 2). Afterwards, from frame 4 on, we can observe a transition to the spectral distribution that characterizes the *residual sound*. The *attack sound* contains spectral components the existence of which is due to couplings between the *longitudinal modes* of the low *strings*. For the lowest *notes*, the sound level relative to these modes is just at 10÷20 dB below the main sound, even if its decay rate is around 100 dB/sec.

3.2.1.4.2 Residual sound

It is fed mainly by the vibration of the *struck string*; therefore, it shows the typical harmonic nature of the *tied vibrating string* (ideal lossless string of length L tied at its ends: it tends in stationary conditions to vibrate according its *natural modes*, conditioned by system parameters such as *string tension*, *string length*, *linear density*. By exciting the *string*, we can observe that, far from the *transitory* phase, the vibrations follow the *natural modes*, with frequency nf_0, where f_0 is the first *mode*. In frequency domain, we note that the spectrum contains a *fundamental component* (first *mode*), plus the upper *harmonics*, whose *amplitude* diminishes with the increasing of *frequency*. In general, the spectrum provided by a *piano* follows the distribution shown in Fig. 3.13 (the *fundamental component* seems to prevail), with the exception of the lower notes, where the *fundamental component* shows a level that's lower than that of the first *harmonic*. As regards the *lower notes*, their *partials* can reach a *frequency* of about 3000 Hz, while the *higher notes partials* can arrive up to a *frequency* of 10000Hz.

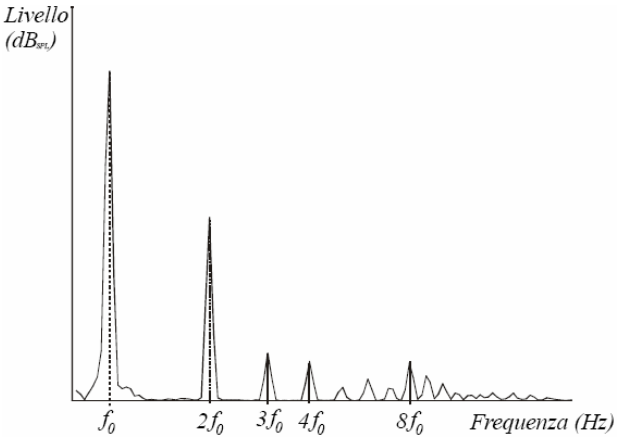


Fig. 3.13: spectrum of a piano note in stationary regime

3.2.1.4.3 The timbre

The *timbre* of a piano is ruled by the *attack phase*, where the partials grow and decay at different speeds and where the sounds produced by the mechanics and those produced by the *longitudinal vibrations* of the *string* almost prevail. A proof of the importance of the *attack phase* in the timbre of the piano can be found in the problems we run into when we try to recognize the *instrument*, while listening to a played backward recording. Even if the *attack* and *decay* curves are different for each *note* and for each *partial*, they anyway show similar characteristics in all the *octaves*. In Fig. 3.14, the *attack* and *decay* curves of the first four *partials* are shown, for notes C_2, A_4 and A_5.

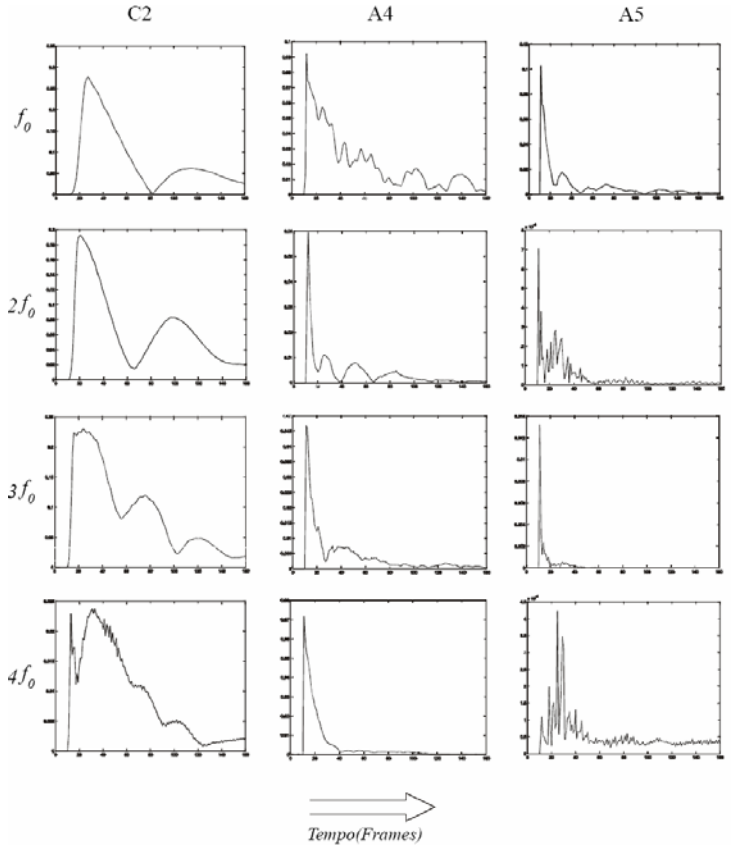


Fig. 3.14: attack and decay curves of the first four partials

In some cases, the transition from the incipient sound is not so sharp; sometimes, interference effects can be observed. In general, except for the *lower notes*, the *fundamental* decays very slowly, with respect to the other components. The spectrum is also characterized by irregularities in the *frequencies* of the *partials* that cannot be considered as *harmonics*. This is due to the extreme rigidity of the *strings*. The *sixteenth partial*, for example, is a *semitone* higher than the "true" *harmonic*. Another important characteristic of the spectrum is that the *partials* do not grow and decrease in a regular way. It has been shown that, in the *release phase*, some *partials* even increase in intensity.

3.2.1.4.4 Dynamics

The difference, in terms of sound level, between two *notes* played at *ff* o *pp* is between 30 and 35 dB. At a distance of about 10 meters from the *piano*, sound levels that go from 50 up to 85 dB, in the low range, and from 37 up to 70 dB, in the high range, can be measured. The difference between *ff* e *pp* is mostly due to the change in *timbre*, rather then to the intensity of sound (at *ff* tha high *partials* are emphasized).

3.3 Signal Pre-Processing

3.3.1 The mathematical model of human ear

In 1863, Hermann von Helmholz (1821-1874) published a book called "*On the Sensations of Tone as a Physiological Basis for the Theory of Music*", where he presented his *resonance theory*, according to which every sound causes the vibration of a well defined section of the *bank of resonators* formed by the fibers of the *basilar membrane* (*BM*). Another theory, proposed by William Rutherford, asserted that the *cochlear apparatus* works just like a microphone membrane, while the analysis role pertains to the brain (1886). But it's only in 1940 that Georg von Békésy succeeds in directly observing with the microscope the inner movements of the *cochlear apparatus*, so laying the foundations of the actual theories about the *human ear*, in which we can distinguish three main section: the *outer ear*, the *middle ear* and the *inner ear*.

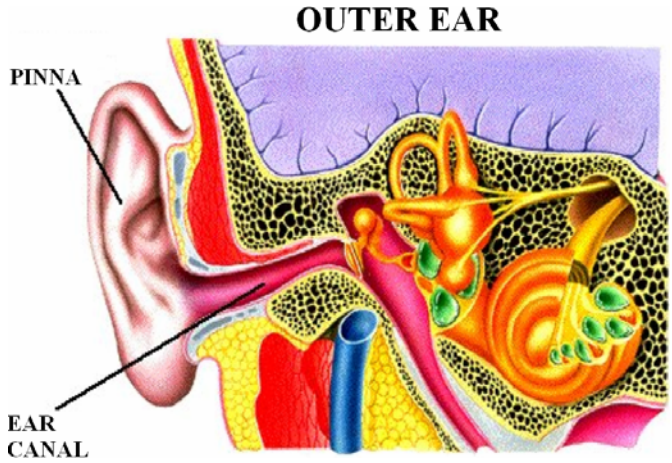


Fig. 3.15: the outer ear

The *outer ear* (Fig. 3.15) is provided with a *pinna* that helps direct sound through the *ear canal* to the *tympanic membrane* (*eardrum*). The *ear canal* (*external auditory meatus*, *external acoustic meatus*) is a tube running from the *outer ear* to the *middle ear*. It extends from the *pinna* to the *eardrum* and is about 26 mm in length and 7 mm in diameter. Its surface is coated with the *earwax*, also known by the medical term *cerumen*; it is a yellowish, waxy substance that plays an important role, assisting in cleaning and lubrication, and also provides some protection from bacteria, fungus, and insects. The *middle ear* (Fig. 3.16) is the portion of the ear internal to the *eardrum*, and external to the *oval window* of the *cochlea*. It contains three *ossicles* (*malleus*, *incus*, *stapes*), which amplify vibration of the *eardrum* into pressure waves, in the fluid in the *inner ear*, generated by the oscillations of the *fenestra ovalis*. The *tympanic membrane*, 1 cm in diameter, is hold up by a bony ring, from the upper edge of which the *ossicles* comes off. The hollow space of the *middle ear* has

also been called the *tympanic cavity*, or *cavum tympani*.

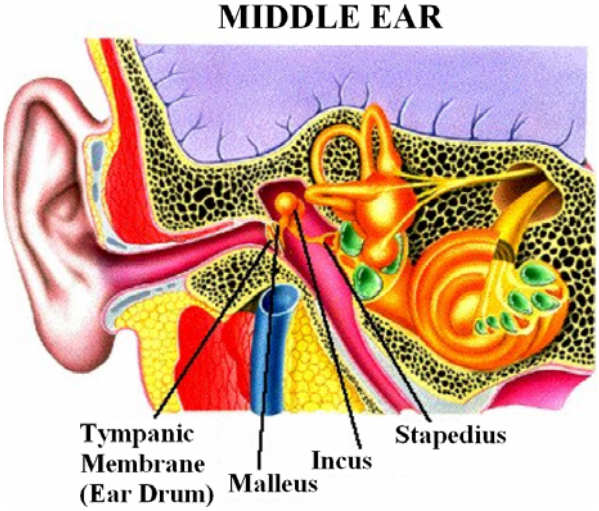


Fig. 3.16: the middle ear

The *eustachian tube* joins the *tympanic cavity* with the nasal cavity (*nasopharynx*), allowing pressure to equalize between the *inner ear* and throat. This allows the *tympanic membrane* to be stressed only by the variation of pressure with respect to the outer pressure. The function of the *middle ear* is to efficiently transfer sound energy from air to the liquid contained within the *cochlea*, in the *inner ear*. It also produces a gain of around 10-20 times (20-26 dB). The *ossicles* are connected by a cartilaginous tissue. The *malleus* receives the vibrations of the *eardrum*, while the *stapes* adheres to the *fenestra ovalis*, beyond which we find the *inner ear*. As regards the transmission of sound to the *ear*, the sound pressure entering the *pinna* arrives to the *eardrum*, which in its turn transfers mechanical energy to the *malleus*. The force applied to the *malleus* is transmitted thru the *incus* to the base of the *stapes*, that in its turn, in contact with the *fenestra ovalis*, applies a mechanical force over the *tympanic membrane*. This mechanism triples the transmitted force; moreover, the *fenestra ovalis* has an area that's 30 times that of the *tympanic membrane*. Therefore, thanks to the equilibrium of forces, we have a gain of about 90 times, with respect to the pressure that reaches the *tympanic membrane*. Two little muscles are connected to the two sides of the *ossicles* chain: the *tensor tympani* and the *stapedius* (that's the smallest muscle in human body). The former is responsible of the auditory heightening, while it's up to the latter to protect the *inner ear* against too loud sounds, by dampening the vibrations of the *stapes* (and so reducing the stimulus transmitted to the *fenestra ovalis*). Beyond the *fenestra ovalis*) the *inner ear* begins; it's formed by channels and small cavities that open inside the *temporal bone* (in fact, it's called *bony labyrinth*). In the *inner ear* (Fig. 3.17), we can see a *vestibular* apparatus, formed by the *semicircular canals*, and a *cochlear* apparatus, formed among other things, by the *cochlear helix*. The first apparatus forms the equilibrium organ, while the second is devoted to *auditory* function. The *cochlea* (Fig. 3.18) is a spiralled, hollow, conical chamber of bone, with two little openings,

closed by flexible membranes.

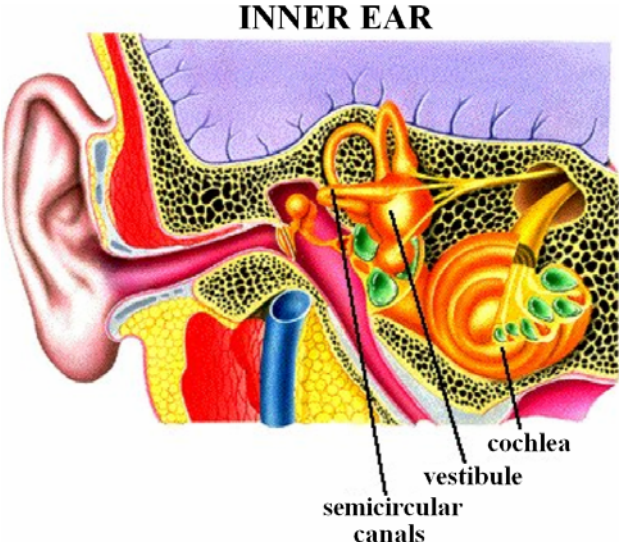


Fig. 3.17: the inner ear

It is filled with a watery liquid, which moves in response to the vibrations coming from the *middle ear* via the *oval window*. The *BM* within the *cochlea* separates two liquid filled tubes that run along its coil, the *scala vestibuli* and the *scala tympani*. The fluids in these two tubes, the *endolymph* and the *perilymph* are very different chemically, biochemically, and electrically. Therefore they have to be kept strictly separated. This separation is the main function of the *BM*. A leakage between the two tubes, due to an injury, causes a disruption or even a total breakdown of hearing in that ear. The vibrations travel along the *BM* and the high *frequencies* run over longer distances with respect to the low *frequencies*. Every section of the *BM* can be modeled as a *band pass filter*. Even if in the chain of the structure some non linearity can be found, these are negligible with respect to the filtering function and the consequent transmission of information to the brain. Therefore, the main function of the *cochlea* consists in subdividing the sound into several channels of acoustic frequencies.

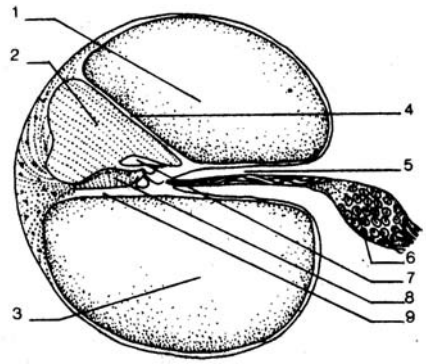


Fig. 3.18: transversal section of the cochlea: 1- scala vestibuli, 2 cochlear duct, 3 scala tympani, 4 Reissner's membrane, 5 bony spiral leaf, 6 acoustic nerve fibres, 7 tectoral membrane, 8 organ of Corti, 9 basilar membrane

The *BM*, along with the *scala vestibuli* and the *scala tympani*, works as a *transducer*, by converting the movement of the liquid inside the *cochlea* in electrical impulses, that in their turn are then sent to the *acoustic nerve*. The two *scalas*, intercommunicating thru a little opening (the *elicotrema*),

placed at the end of the *cochlear* helix are filled with the above mentioned fluids. The *pitch* of a sound is perceived thanks to the *elicotrema*. In fact, the brain is able to distinguish between the periods of the two vibrations that reach the *elicotrema* by traveling into the two *scalas*. Along and over the whole surface of the *BM*, lies the *organ of Corti* (Fig. 3.19).

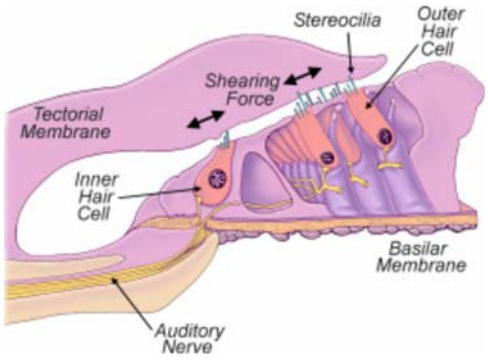


Fig. 3.19: The organ of Corti

For the most of its width, it's covered by the *tectorial membrane*. The *organ of Corti* is formed by a cellular structure made of a double rank of about 20.000 *acoustic hair cells* (*inner* and *outer*). These *cells* have *stereocilia* or "hairs" that stick out. The bottom of these cells are attached to the *BM*, while the *stereocilia* are in contact with the *tectorial membrane*. Inside the *cochlea*, sound waves cause the *BM* to vibrate up and down. This creates a shearing force between the *BM* and the *tectorial membrane*, causing the *hair cell stereocilia* to bend back and forth (Fig. 3.20).

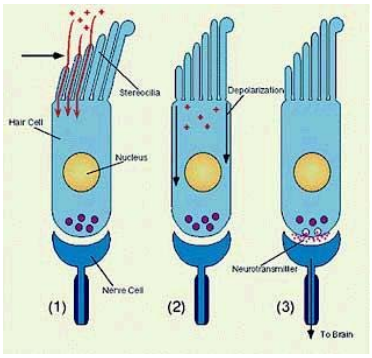


Fig. 3.20: How Hair Cells work

This leads to internal changes within the *hair cells* that creates electrical signals. *Auditory nerve fibers* rest below the *hair cells* and pass these signals on to the brain. So, the bending of the *stereocilia* is how *hair cells* sense sounds. The *auditory nerve system* is formed by a complex network of nerves that start from the *organ of Corti* and reach the *auditory cortex*. The role played by such a system is that of analyzing the simple sounds (*pure tones*), as well the complex sounds, and of localizing their origin, based on information coming from both the ears. These results are achieved by processing data related to *frequency*, *intensity*, as well as time and space variations of the sounds.

In order to define a mathematical model of *human ear*, two properties of the *inner hair cells* have to

be taken into account. These cells react to deflections of the *cilias* in one direction only: this introduces semi-wave rectification. Moreover, as *frequency* increases, phase locking starts to fail at 1.5 kHz and ends to work at about 5 kHz. When the *ear* doesn't manage anymore to lock the fine structure of the sound signal, it starts locking its envelope. This can be modeled by means of a convolution with a raised cosine function and a time constant of 0.25 ms. This function has a negligible effect at low *frequencies*, but works as a reasonable envelope function at the high ones. As regards the spectral properties of quasi-periodic signals, *human ear* can distinguish only up to about 7 *harmonics*. Higher *frequency* components are perceived only as a hole. Moreover, it can be taken into account the *low pass filtering* function, around 20 kHz, introduced by the mechanic inertia of *ossicles*; above this frequency, the vibrations are strongly attenuated and therefore hardly perceived. As *frequency* increases, the difference of pressure between the two *scalas* goes to zero; the *BM* starts not to vibrate anymore and than stops to excite the *cilias* of the *organ of Corti*. All these properties can be modeled by a sequence of mathematical operations, as shown in Fig. 3.21.

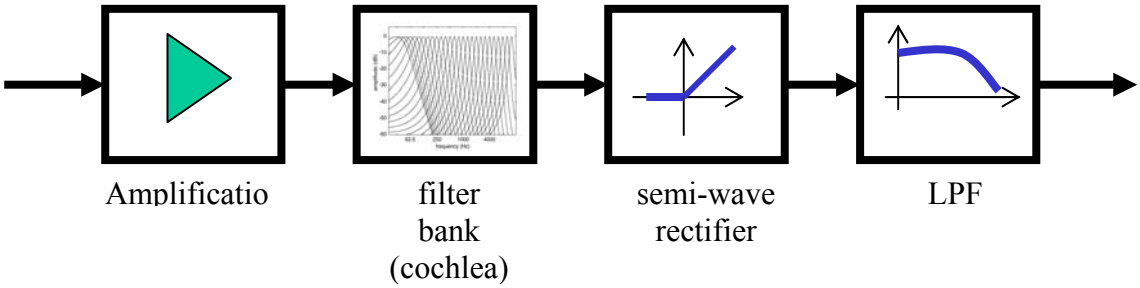


Fig. 3.21: the mathematical model of the ear

3.3.2 The intermediate representation

The digital signal acquired from the auditory context is non suited to a direct processing, firstly because it just a mixture and secondly because its dimensions are excessive, with respect to the conveyed information. Different kinds of *intermediate representation* have been proposed, being all of them based on two main approaches:

- *physiologic*
- *physic*

In the first case, a representation that's based on the musical perception model is desired, while the second approach tries to enucleate those elements that form musical information. In our work, we will follow the second approach, where the *pre-processing* phase aims to the computation of a *t/f representation* of musical signal. The simplest *t/f representation* of a signal is the well known *spectrogram*, derived from the time-invariant *Fourier analysis*, by applying an *analysis short time window* to subsequent overlapped *data frames*. Formally, let $s(t)$ be an *acoustic waveform* in the

time domain and $w(t)$ be the *analysis window*; the *spectrogram* is the squared module of the *Short Time Fourier Transform* $S(t, f, w)$. Nevertheless, the adoption of the *spectrogram* fails, because of its inconsistency with regard to the real functioning of *human ear*: the *spectrogram* is based on a linear segmentation of the *frequency* axis, while *human ear* perceives the *pitch* of a sound according to a *logarithmic scale*, as dictated by the physical structure of the *BM*, that works as a spectrum analyzer by dividing the signal into its *frequency components* on the basis of a spatial criterion. The *BM* has no uniform thickness: it becomes thinner and thinner going from one of its ends to the other; therefore, its resonance frequency changes along its axis with its thickness. As a consequence, sounds at different frequencies will lead different sections of the *BM* to vibrate. The distribution of the *resonance frequencies* along the *BM* is non linear in *frequency*. The scale that makes a connection between the *resonance frequencies* with the points along the *BM* is called *Bark scale*, or *critical band scale*. It's substantially a *logarithmic scale* and it's the foundation of the *tempered musical scale*. Inspired by these considerations, in 1988, *Judith C. Brown* proposed a novel *t/f representation* that could allow a *logarithmic frequency resolution*: the *Constant Q Transform* (*QFT*). Instead of *frequency resolution*, what remains constant in the computation of the *QFT* is the so called *Q* factor, that is the ratio between *frequency* and *frequency resolution*.

3.3.3 The Constant Q Transform (QFT)

The *Discrete Fourier Transform* (*DFT*, and its fast implementation *FFT*) produces equidistant *frequency components* (*bins*) according to a constant *frequency* interval. Therefore, it doesn't correctly maps musical *frequencies*. On the contrary, *QFT* is well suited to musical applications, thanks to its *logarithmic frequency resolution*. It produces *spectral components* geometrically distributed, or in other words, a *resolution* that's constant in the *logarithmic frequency scale*. Like *DFT*, the efficient computation of the *QFT* is based on the *FFT*. Like the *Fourier transform*, a *QFT* is a *bank of filters*, but, in contrast to the former, it has *geometrically spaced center frequencies*, according the following formulation (k^{th} *spectral component*):

$$f_k = f_{\min} 2^{\frac{k}{b}} \qquad (k = 0, \ldots) \tag{3.1}$$

where b dictates the number of *filters* per *octave* (number of *bins* per *octave*) and where $f_{\min}$ is the *minimal center frequency*. In order to make the *filter* domains adjacent, one chooses the *bandwidth* of the k^{th} filter as:

$$\Delta_k^{qft} = f_{k+1} - f_k = f_0 \left(2^{\frac{k+1}{b}} - 2^{\frac{k}{b}} \right) = f_0 2^{\frac{k}{b}} \left(2^{\frac{1}{b}} - 1 \right) \quad \Rightarrow$$

$$\Delta_k^{qft} = f_k \left(2^{\frac{1}{b}} - 1 \right) \tag{3.2}$$

This yields a constant ratio of *center frequency* and *filter bandwidth*, with *resolution*:

$$Q = \frac{f_k}{\Delta_k^{qft}} = \frac{1}{2^{\frac{1}{b}} - 1} \tag{3.3}$$

In what follows, each *frequency component* of the *QFT*, computed as the output of a *filter*, will be called *qft-bin*. What makes the *QFT* so useful is that, by an appropriate choice for $f_{\min}$, and b, the *center frequencies* directly correspond to *musical notes*. For instance, choosing $b = 12$ and $f_{\min}$ as the frequency of *MIDI note* 0 (16.35 Hz), makes the k^{th} *qft-bin* correspond to the *MIDI note* number k. Anyway, *center frequency* f_k will be let vary from $f_{\min}$ to an upper frequency f_{MAX} that's always below the *Nyquist frequency*. Another nice feature of the *QFT* is its increasing *time resolution* towards higher frequencies. This resembles the situation in our *auditory system*. It is not only the digital computer that needs more time to perceive the frequency of a low tone but also our auditory sense. This is related to music usually being less agitated in the lower registers. In order to derive the calculation of the *QFT*, let's consider a *sequence* of length L, $x[n]$, got by sampling a signal $x(t)$ (L samples of the signal) at *sampling frequency* f_s. The L points of the *DFT* of this *sequence* are given by:

$$X(\omega_k) = \frac{1}{L} \sum_{n=0}^{L-1} x[n] e^{-j\omega_k n}, \quad \omega_k = k\frac{2\pi}{L}, \quad f_k = k\frac{f_s}{L}, \quad k = 0,1, \ldots, L-1 \tag{3.4}$$

If we consider *DFT* as a *bank of adjacent filters*, the *bandwidth* of the generic filter, apart from ω_k, is given by:

$$\Delta_k^{dft} = f_{k+1} - f_k = (k+1)\frac{f_s}{L} - k\frac{f_s}{L} \qquad \Rightarrow \qquad \Delta_k^{dft} = \frac{f_s}{L} \tag{3.5}$$

If the *sequence* $x[n]$ is not length limited, we must introduce a *windowing* procedure and define a new transform, the *STFT*, where the computation is made by transforming $x[n]$ multiplied by a proper *window* function $w[n]$ of length $N << L$:

$$X(\omega_k) = \frac{1}{N} \sum_{n=0}^{N-1} w[n] x[n] e^{-j\omega_k n}, \quad \omega_k = k\frac{2\pi}{N}, \quad f_k = k\frac{f_s}{N}, \quad k = 0,1, \ldots, N-1 \tag{3.6}$$

Windowing has even the role to avoid, or at least reduce, the effects introduced by an abrupt cutting off of the *sequence* (*spectral leakage*). Nevertheless, in case of *QFT*, the length of the *window* cannot be fixed (N), since the *bins* cannot be equidistant. We have shown that, in case of *DFT*, all the *bins* are equidistant (*bandwidth*: $\Delta = f_s / N$); so, the *window* of length N always captures the same number of samples. Therefore, in case of *QFT*, for each the *bin* f_k, we have to compute a different length N_k for the *window*:

$$(bandwidth) \quad \Delta_k^{qft} = \frac{f_k}{Q} \quad \Rightarrow \quad N_k = \frac{f_s}{\Delta_k^{qft}} \quad \Rightarrow \quad N_k = Q\frac{f_s}{f_k} \tag{3.7}$$

In case of *DFT*, the *filter frequency* to *resolution* ratio is equal to:

$$Q = \frac{f_k}{\Delta_k^{dft}} = \frac{k\frac{f_s}{N}}{\frac{f_s}{N}} \qquad \Rightarrow \qquad Q = k \tag{3.8}$$

Therefore, for integer values of Q, the k^{th} element of *QFT* clashes with the Q^{th} element of *DFT* and the *window* length is equal to $Q\,f_s / f_k$. Summarizing, we get the following recipe for the calculation of a *QFT* of the *sequence* $x[n]$. First, we choose the *minimal frequency* $f_{\min}$ and the number of *bins* per *octave* b, according to the requirements of the application. The *maximal frequency* f_{MAX} affects only the number of *qft-bins* to be calculated. The steps to accomplish are the following:

number of *qft-bins* $$K = \left\lfloor b \log_2\left(\frac{f_{MAX}}{f_{\min}}\right)\right\rfloor \tag{3.9}$$

resonance coefficient $$Q = \frac{1}{2^{\frac{1}{b}} - 1} \tag{3.10}$$

window length $$N_k = \left\lfloor Q\frac{f_s}{f_k}\right\rfloor \quad \text{for} \quad k = 0, 1, \ldots, K \tag{3.11}$$

k^{th} sample of the QFT $$X^{qft}[k] = \frac{1}{N_k}\sum_{n=0}^{N_k-1} x[n]\, w_{N_k}[n]\, e^{-j\frac{2\pi Q}{N_k}n} \quad \text{for} \quad k = 0, 1, \ldots, K \tag{3.12}$$

K as been calculated as follows:

$$f_{\max} \geq f_{\min} 2^{\frac{K}{b}} \Rightarrow \frac{f_{\max}}{f_{\min}} \geq 2^{\frac{K}{b}} \Rightarrow \log_2\left(\frac{f_{\max}}{f_{\min}}\right) \geq \log_2 2^{\frac{K}{b}} \Rightarrow$$

$$\Rightarrow \log_2\left(\frac{f_{\max}}{f_{\min}}\right) \geq \frac{K}{b}\log_2 2 \Rightarrow b\log_2\left(\frac{f_{\max}}{f_{\min}}\right) \geq K \Rightarrow$$

$$K = \left\lfloor b \log_2\left(\frac{f_{\max}}{f_{\min}}\right)\right\rfloor \tag{3.13}$$

Since the computation of the *QFT* is very time consuming, an efficient algorithm is highly desirable. The method we are going to introduce is based on the computation of a *spectral kernel* that doesn't depend on the *sequence* $x[n]$, but on $f_{\min}$, f_{MAX} and b only, as well as on the exploitation of the famous *FFT* algorithm. Using matrix multiplication, the *QFT* of a row vector x of length N ($N \geq N_k$ for all $k < K$) is given by:

$$\mathbf{X}^{qft} = \mathbf{x} \cdot \mathcal{K}^* \tag{3.14}$$

where $\mathcal{K}^*$ is the complex conjugate of the *temporal kernel* (matrix) $\mathcal{K}$, defined as follows:

$$\mathcal{K}_{nk} = \begin{cases} \dfrac{1}{N_k} w_{N_k}[n] e^{jn\frac{2\pi Q}{N_k}} & n < N_k \\ 0 & n \geq N_k \end{cases} \qquad \text{for} \quad n < N,\ k < K \tag{3.15}$$

Since the *temporal kernel* is independent of the input *sequence* $x[n]$, one can speed up successive *QFT*s by pre-calculating $\mathcal{K}^*$. This is very memory consuming and since there are many non vanishing elements in $\mathcal{K}$, the calculation of the matrix product $\mathbf{x} \cdot \mathcal{K}^*$ still takes quite a lot. Luckily Judith Brown and Miller Puckette came up with a very clever idea for improving the efficiency of the calculation [Brown and Puckette, 1992]. The idea is to carry out the matrix multiplication in the spectral domain. Therefore, we define *spectral kernel* $\boldsymbol{K}$ as the *FFT* of the columns (one dimensional *DFT*s applied column wise) of $\mathcal{K}$: $\boldsymbol{K} = DFT(\mathcal{K})$.In (3.14) we center the *filter* domains on the left for the ease of notation. Right-centering is more appropriate for real-time applications. Middle-centering has the advantage of making the *spectral kernel* real. Since the *windowed complex exponentials* of the *temporal kernel* have a *DFT* that vanishes almost everywhere, except for the immediate vicinity of the corresponding *frequency component*, the *spectral kernel* is a sparse matrix (after eliminating all components below some *threshold* value). This fact can be exploited for the efficient calculation of $\mathbf{X}^{qft}$. In fact, we know that, given two generic discrete time functions $x[n]$ and $y[n]$, the following equality is true (*Parseval* equation) :

$$\sum_{n=0}^{N-1} x[n] y^*[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] Y^*[k] \tag{3.16}$$

where $X[k]$ and $Y[k]$ are the *FFT* of $x[n]$ and $y[n]$, respectively, and $Y^*[k]$ is the complex conjugate of $Y[k]$. Let's write the k^{th} element of the *QFT* again:

$$\begin{aligned} X^{qft}[k] &= \frac{1}{N_k} \sum_{n=0}^{N_k-1} x[n] w_{N_k}[n] e^{-jn\frac{2\pi Q}{N_k}} = \sum_{n=0}^{N_k-1} x[n] \mathcal{K}^*_{nk}[n] = \\ &= \frac{1}{N_k} \sum_{k=0}^{N_k-1} X^{fft}[k] K^{*fft}_{nk}[k] \end{aligned} \tag{3.17}$$

or

$$\mathbf{X}^{qft} = \frac{1}{N_k} \mathbf{X}^{fft} \cdot \boldsymbol{K}^{*fft} \tag{3.18}$$

As stated before, in order to reduce the computational overhead, we introduce a *threshold* in the computation of the *spectral kernel* $\boldsymbol{K}$, according to which all the elements of $\boldsymbol{K}$ under this *threshold* are put to zero; this way we have to do with a *sparse kernel*, that is a *kernel* with very few non null elements. Obviously, the introduction of this *threshold* will inevitably introduce an imprecision in the calculation. Due to the sparseness of $\boldsymbol{K}$, the calculation of the product $\mathbf{X}^{fft} \cdot \boldsymbol{K}^{*fft}$ involves

essentially less multiplications than $\mathbf{x} \cdot \mathcal{K}^*$. Since the calculation of $\boldsymbol{K}$ is exclusively based on the parameters of the problem ($f_{\min}$, f_{MAX}, f_s, b), for a given application, the calculation of the *QFT* is nothing else then a simple multiplication of an array by a sparse matrix with constant elements. If we ignore the calculation of the complex matrix $\boldsymbol{K}$, accomplished only once at the beginning of the process, the computational complexity of the *QFT* is formed uniquely by the calculation of the *FFT* of the *input sequence*, plus the calculation of the matrix product *spectral kernel* $\boldsymbol{K}$ by the complex array $\boldsymbol{X}^{fft}$. Therefore, even if the calculation of the *spectral kernel* is quite expensive, having done this once, all succeeding *QFT*s are performed much faster. The efficiency of the *QFT* calculation algorithm is influenced by the dimensions of the *spectral kernel* $\boldsymbol{K}$ and by the number of non null elements in it. The number of columns of $\boldsymbol{K}$ is equal to the number of *spectral components K*, that in its turn depends on the number of *bins* per *octave* and on the ratio $f_{\mathrm{MAX}} / f_{\min}$. The number of rows of $\boldsymbol{K}$ is fixed by the max allowed dimension N_{MAX} for a *window* (which is the maximum of all N_k). N_{MAX} is chosen as the lowest power of 2 greater than or equal to $Q\, f_s / f_{\min}$:

$$N_k = \left\lfloor Q \frac{f_s}{f_k} \right\rfloor \qquad \Rightarrow \qquad N_{MAX} \geq \left\lfloor Q \frac{f_s}{f_{\min}} \right\rfloor \tag{3.19}$$

We can say that the *time window* influences the performances of the *QFT* in terms of *time resolution* and *frequency resolution*. Particularly, a too wide *window* will mask too short phenomena (*temporal resolution* is penalized), but will guarantee more information to be used in the detection of periodicities (*frequency resolution* is helped). On the contrary, a too short *window* helps *temporal resolution*, to the disadvantage of *frequency resolution*. The need of a compromise between both *resolutions* can be effectively faced by means of *multi rate architectures*.

3.3.3.1 Audio Signal Pre-Processing

By means of the *QFT*, we will try to segregate the *harmonic components* that form the *musical signal*. Tanks to the *QFT*, we can get an exact correspondence between the *frequency components* of the *spectrum* of the musical signal and the notes of the *chromatic tempered musical scale*. We have previously seen why a *windowing* of the signal is needed. We have also seen that the *windowed signal* is firstly processed by means of the common *FFT* algorithm and then multiplied by the complex matrix of the *spectral kernel*, in order to get the complex vector of the *spectral distribution*. The name given to this *t/f representation*, that is the *spectral distribution*, is called *spectrogram* (Fig. 3.22). We will call *frame* (Fig. 3.23) a single *spectral distribution* produced by the *QFT* and relative to the input samples of the *windowed input signal*, spanning over a finite time interval. A *frame* is substantially an estimation of the harmonic content of the signal along a given

time interval.

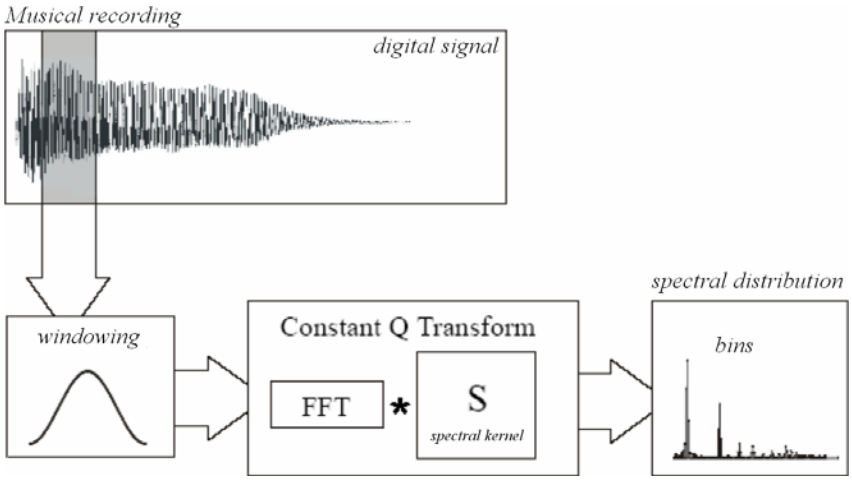


Fig. 3.22: block diagram of the calculation of a spectral distribution

It's possible to analyze the evolution of the musical signal, by studying the sequence of the *frames*, or in other words the *t/f representation* given by the *spectrogram* of the *musical signal* itself.

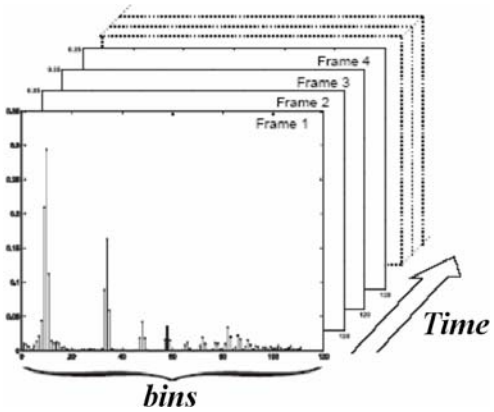


Fig. 3.23: frame sequence

Practically speaking, the *spectrogram* is computed by making the *window* sliding along the time axis and by computing at each step a new *frame*. In order to avoid excessive loss of information and to improve *time resolution*, at the cost of a little redundancy in the *spectrogram*, it is good manners to generate *frames* choosing time intervals that partially overlap. A *QFT spectrogram* can be considered as an *intermediate representation* of the musical signal, by means of which it's possible to segregate the various characteristics that form a sound, as its *timbre*, *pitch* and *level*. If we consider a sound signal produced by a *musical instrument*, *pitch* has immediate correspondence with the ordered components of *QFT*, while *timbre* can be represented by the ordered sequence of more frames. Fig. 3.24 shows the *spectrogram* of a *piano* note, where it's evident how the restrictions imposed on *time resolution* are responsible of the masking of the *attack spectrum*, while the *residual sound* is on the contrary well represented. The analysis of the *attack* of a sound requires a very high *time resolution*, that in any case mainly depends on the length of the *window* as well as the overlapping percentage of the *frames*.

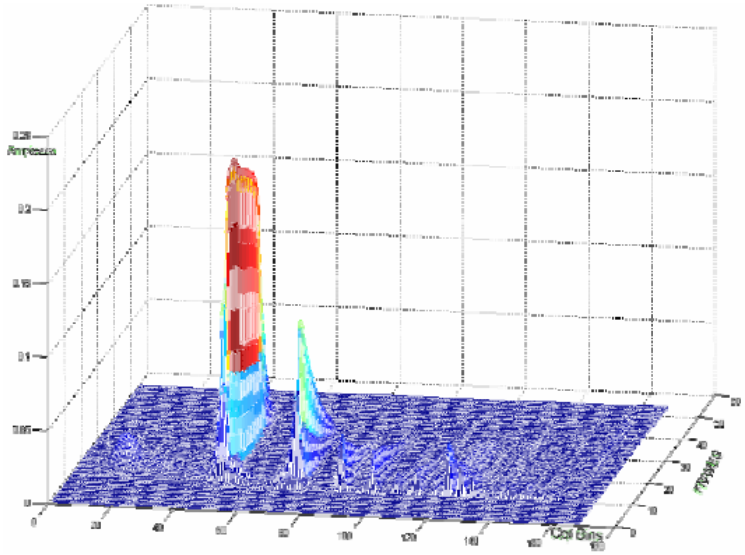


Fig. 3.24: spectrogram of a piano note

Each frame can be stored in an array; the *frame sequence* that forms the *spectrogram* can be stored in an ordered sequence of arrays (matrix ***X***, Fig. 3,25), each row of which contains in an orderly way a *frame* produced by *QFT*, that is the *harmonics* (average of *harmonics* in the time interval given by the length of the *window*) that at a given time form the input signal. On the contrary, the generic column of matrix ***X*** is in relation with the samples of the time evolution of the relative *bin*.

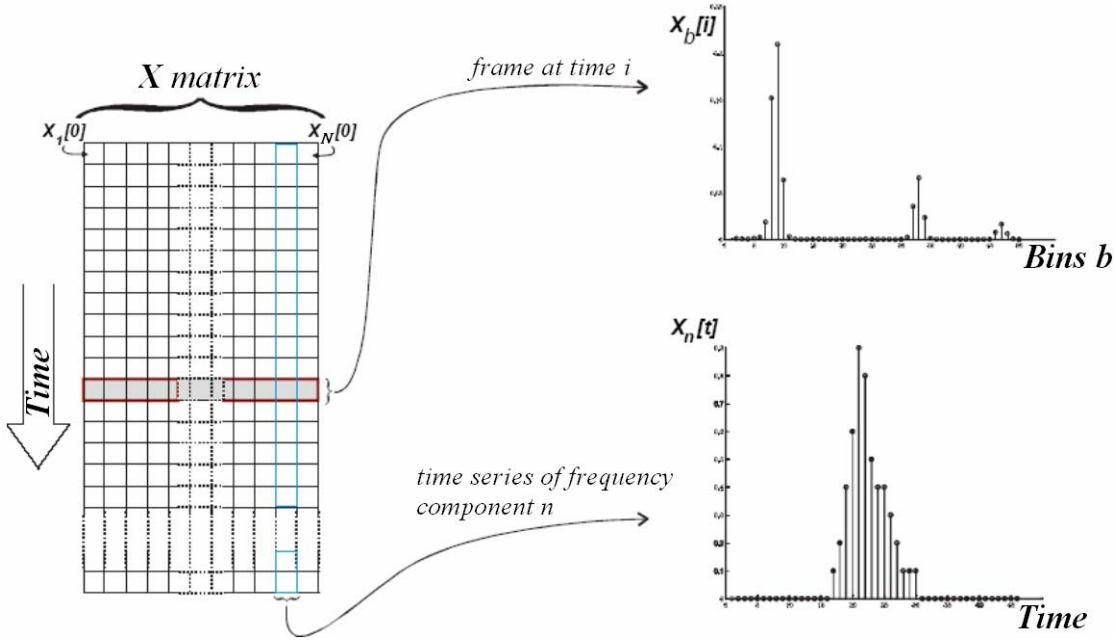


Fig. 3.25: spectrogram matrix and meaning of its rows and columns

Finally, the analysis of the time evolution of the *harmonic content* of a *musical signal* can be accomplished starting from the information in the *spectrogram*. Therefore, the problem consists essentially in the analysis of the time evolution of a vector. As a consequence, it's part of the theory of *Time Series Processing* and in particular regarding multivalued time series.

3.3.3.2 Operative considerations

The spectral contents of the sound of a *piano* are wide and span from the *low frequencies* of *note A0* (fundamental at 27.5Hz), up to *harmonics* with *frequencies* that are higher than 10 kHz (*note C8*, (fundamental at 4186 Hz). Therefore, the direct implementation of the *QFT* turns out to be unpractical, unless the *note range* isn't reduced. The aim of the our work, described in this chapter, has consisted in the exploitation of *QFT* for the extraction of the *intermediate representation* of an *acoustical signal*, together with a *Dynamic Neural Network* that processes the *time sequence* given by the *intermediate representation*, in order to automatically extract *musical information*. To calculate the *QFT* of the *musical signal*, we followed the following steps:

1. collection of *MIDI* files
2. recording of *WAVE* files
3. *onset detection* and *MIDI tracking*
4. calculation of the *QFT*

The aim of the *pre-processing* phase is to get the *QFT* of a sound signal to be proposed as the input to the *neural network*. This means that the *input-target couples* (*training set*) of the *neural network* will be respectively the *QFT* and the relative *tracking*, that is a text file containing information about the start and end times of the *target notes* that the *network* has to recognize. In other words, we need to take into account the *MIDI files* of the musical pieces that we'll use for *training* and their *WAV* counterparts. By processing the *WAV file*, we get information about the *QFT*, while by processing the *MIDI file* we get information about *tracking*. The *WAV file* was produced by recording the output of a *software synthesizer*, while playing the chosen *MIDI file*. For this purpose, we used a full-duplex *SoundBlaster* audio card for IBM PC, as A/D and D/A converters, and the *Roland Virtual Sound Canvas VSC-88*, as *software synthesizer*. In order to compensate the delays introduced by latencies between playing and recording, we had to time re-align the two files.

In order to compute the *QFT*, the *WAV file* is split into a sequence of *windowed sample buffers* (*SB*) and *QFT* is applied to each *SB*. So, we have to find the relation existing between the length of the *buffer*, l_{buf} , and the parameters of *QFT* ($f_{\min}$, f_{MAX}, f_s and b). In Fig. 3.16, the model used for the calculation of the *QFT* is shown. The *WAV file sampling frequency* f_s is in generally equal to 44.1 kHz and is derived from the one adopted at recording time. At each *sampling time* ($T_s = 1/f_s$), a *sample* is acquired and written into the *buffer*. We define f_s^{qft} as the *sampling frequency* of the *t/f representation*, or the calculation frequency of *QFT*. At each *QFT time* ($T_s^{qft} = 1/f_s^{qft}$), the content of the *buffer* in enveloped by means of an *Hamming window*, the length of which is equal to l_{buf} ;

then, *QFT* is computed.

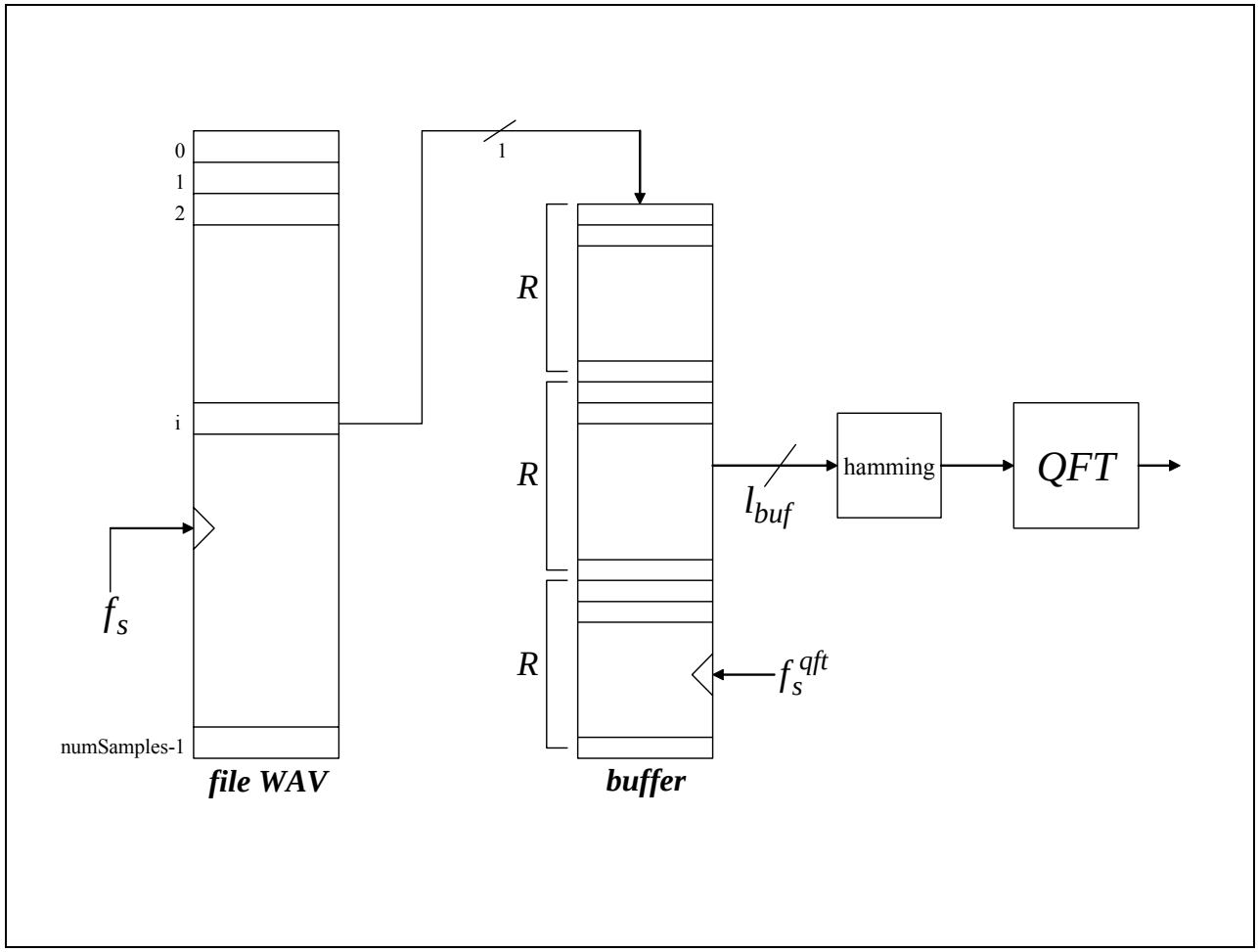


Fig. 3.26: block diagram describing the QFT algorithm

In general, $f_s^{qft} < f_s$, being useless to calculate the *QFT* at each T_s, since the *t/f representation* changes slowly; moreover, this choice is less computationally expensive. However we have to pay attention on how to choose the right T_s^{qft}; if we choose an high value for f_s^{qft}, we obviously obtain a good *resolution* (but high complexity and redundancy); on the contrary, if we choose a small value, we could loose *notes* with duration smaller than T_s^{qft}, or *notes* that have an in-between distance smaller than T_s^{qft}. We will use $f_s^{qft} = 20\,Hz$ or nearly; this acceptable since it is strictly connected with the human way of perceiving sounds (*human ear* can't appreciate variations in length that are smaller that 50 ms). The word "nearly" derives from the fact that it is convenient to always choose an integer sub-multiple of f_s: this will facilitate us in the next steps. The rate value *R* introduced in Fig. 3.26 (number of samples extracted from the *file WAV* and enqued in the *buffer*, before computing a new *QFT*) is equal to:

$$R = f_s / f_s^{qft} \tag{3.20}$$

Based on this value, we will find the right length for the *buffer*: $l_{buf} = 3R$. This choice comes from the following considerations: in order to avoid samples extracted from the *WAV file* that will never be evaluated in the *QFT* computation, it's important to assure a minimum length for the *buffer*, that is the rate value itself. In our tests, we have found improvements in analyzing *QFT* if we take into account the previous rate samples and the next rate samples, that is a length $3R$ of samples extracted from the *WAV file*. If we take into account the 8 *octaves* of a *piano*, we have to face the note range $[C_1 \ldots C_8]$, that's the same of the frequency range [32.703…4186.0] Hz (as concerns fundamental frequencies); at first, we'll ignore the lowest octave from C_0 to $A\flat_1$, since these *notes* are quite rare in *piano pieces*; moreover, results in the recognition of these notes are poor; so, we'll start from A_1, that is $f_{\min} = 55\,Hz$, up to frequency $f_{\max} = 5500\,Hz$ (that is nearly 11025/2, in case of *under-sampling*). As regards the number of bins *b*, a reasonable value seems to be 36; obviously, an higher value should assure a better *resolution*, but at the cost of an higher complexity.

3.3.3.3 Testing examples

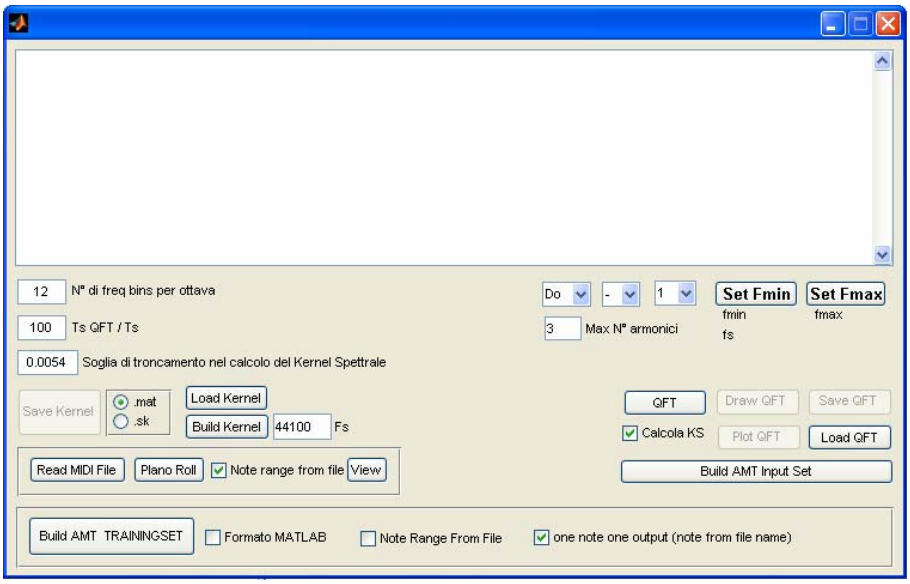


Fig. 3.27: user input form of the Matlab application used to compute QFT

Fig 3.27 reports the main form of a Matlab application written by us thru which the user can input all the parameters required by the calculation of *QFT*. Parameters are:

- the number of *bins* per *octave*
- the ratio between the *sampling period* of the signal and the *calculation period* of a *frame*
- the truncation *treshold* in the calculation of the *spectral kernel*
- the *frequency range* to be investigated, in terms of *min note* and *max note*

Moreover, this form can be used to read a *MIDI file* and to accordingly buid the proper *training set* to be used for a further *training* of an *LRNN*.

Example 1

Let's consider a pair of files *WAV*/*MIDI* of an A_4 *note*; the tracking of the *MIDI file* is very simple, since it represents an A_4 *note* starting after 1s and stopping after 1s. Fig. 3.28 shows the *WAV* representation of the A_4 *note* signal.

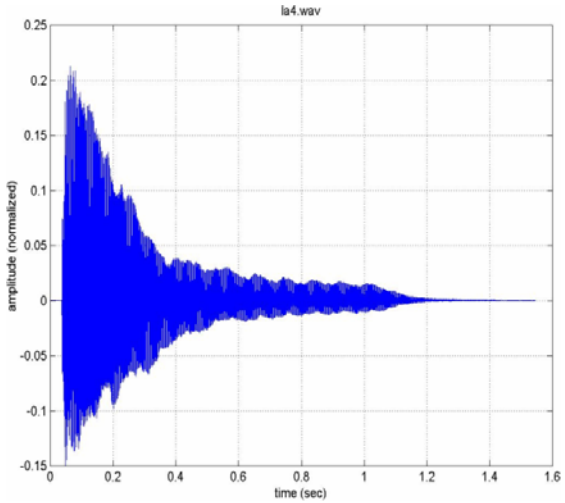


Fig. 3.28: waveform of the A_4 note signal

For the computation of the *QFT*, the values chosen for the parameters are the following:

- $f_s = 11025\ Hz$
- $f_s^{qft} = 21\ Hz$
- $R = 525$
- $f_{\min} = 55\ Hz$
- $f_{\max} = 5500\ Hz$
- $b = 36$
- $l_{qft} = 240\ samples$
- $pbl = 33\ f_s^{qft}$

In the *QFT* plot (Fig. 85), we can find *peaks* that correspond to *partials* of the A_4 *note*: the main *peak* correspond to the *fundamental frequency* of the *note*, that is 440 Hz; while the smooth *peaks* correspond to multiples of the *fundamental frequency*.

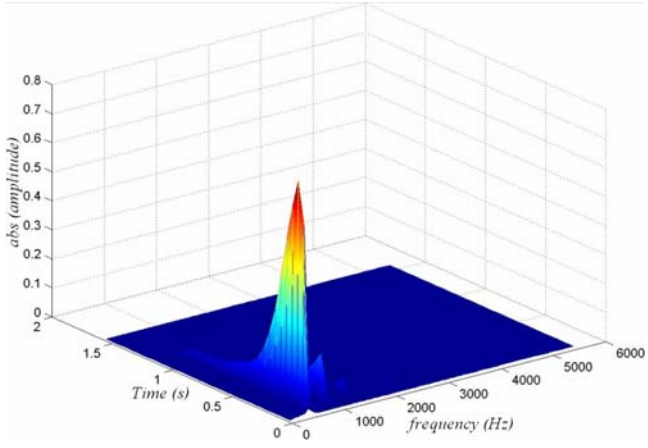


Fig. 3.29: QFT of the A_4 note signal

In Fig. 3.30, what happens after the end of the *attack* phase of the signal is shown:

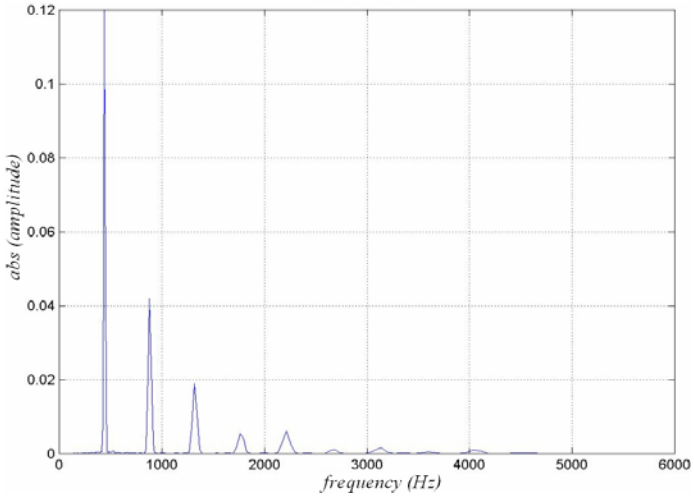


Fig. 3.30: QFT of the A_4 note signal after the end of the attack phase

Example 2

Let's now consider an experiment that's a little bit more complex. Fig. 3.31 and Tab. 3.1 show respectively the *WAV file* representation, and the *MIDI tracking* of an *arpeggio* played on a *piano*.

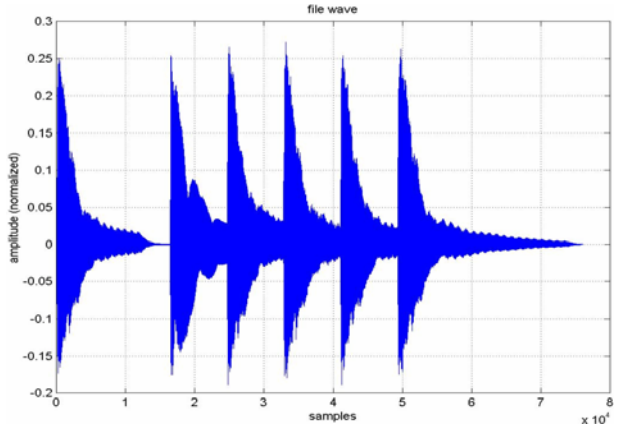


Fig. 3.31: waveform of an arpeggio played on a piano

INIZIO	NOTA	DURATA
0.998958	***La4***	1.10312
2.47917	***Fa5***	3.69167
3.21979	***Do5***	0.732292
3.95938	***La4***	0.732292
4.69896	***La#4***	0.732292
5.43854	***Sol4***	2.21354

Tab. 3.1 MIDI tracking of an arpeggio played on a piano

In Fig. 3.32 we can find the *QFT* plot:

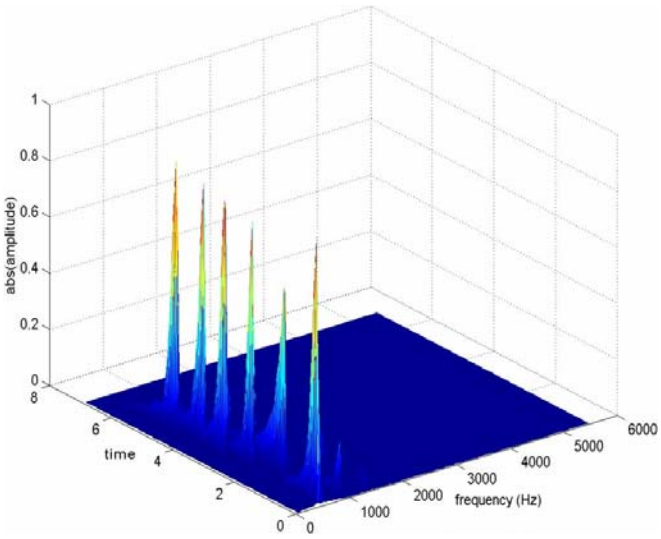


Fig. 3.32: QFT of an arpeggio played on a piano

For the computation of the *QFT*, the values chosen for the parameters are the following:

- $f_s = 11025\ Hz$
- $f_s^{qft} = 21\ Hz$
- $R = 525$
- $f_{\min} = 55\ Hz$
- $f_{\max} = 5500\ Hz$
- $b = 36$
- $l_{qft} = 240\ samples$

By observing this plot, it's not so easy to distinguish between the various *peaks*; however, it is clear when *notes* begin and that more than one *note* has been played. However let's see what happens at different times. If we analyze the *MIDI tracking*, we notice that from 0.99895 s to 2.10207 s (0.99895 +1.10312) there's only A_4 playing. At this point, we can consider an in between block that points the A_4 playing, for example at about 0.6 s:

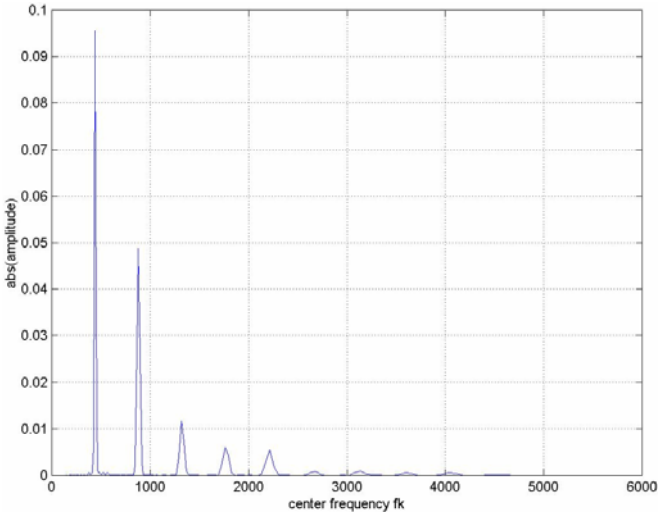


Fig. 3.33: QFT of the A_4 note signal playing alone

As we can see from Fig. 3.33, we find only *peaks* that refer to A_4, that is to its *fundamental frequency* of 440 Hz and its multiples. If we advance the time of observation, for example from 2.47917 s to 3.69167 s, we find only F_5 playing and until 3.21979 s it's the only *note* playing. So, considering an in between block from nearly 1.4 sec. to 2.7 sec. we can analyze its *peaks*:

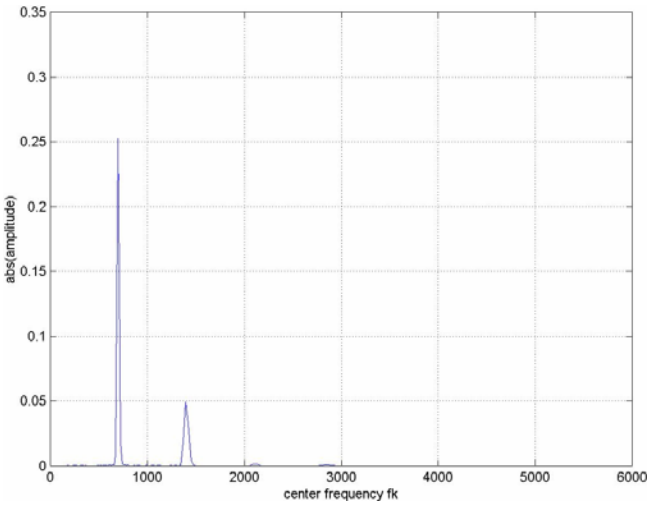


Fig. 3.34: QFT of the F_5 note signal, playing alone

Fig. 3.34 refers to nearly 2 s after the beginning of playing; we can clearly found only peaks relative to F_5, that is 698.46 Hz and its multiples. Tere's no trace of the past A_4. It is interesting to consider what happen at about 2.5 s (Fig. 3.35). In this case we find two notes playing, that is F_5 and C_5, thus we find peaks relative to 698.46 Hz (F_5), 523.25 Hz (C_5) and their multiples.

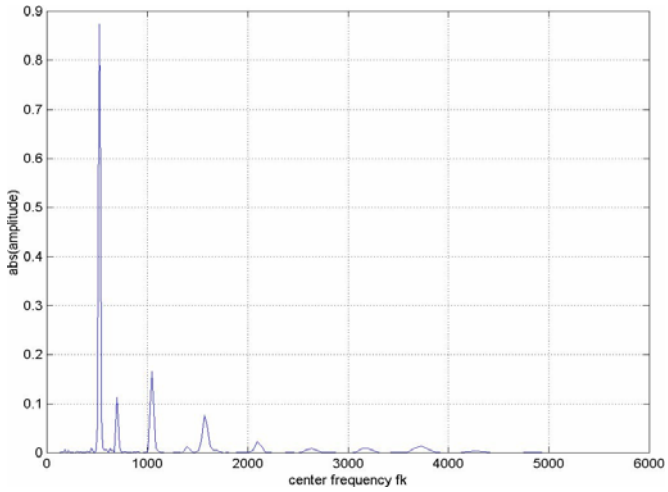


Fig. 3.35: QFT of the F_5 and C_5 note signals, playing together

The main *peak* refers to the *fundamental frequency* of C_5, while the *peak* relative to the *fundamental frequency* of F_5 (the 2nd *peak*) is hardly visible, since F_5 is slowly disappearing (it will disappear in nearly 3.6 sec.). Other *peaks* reflect the strength of the two main *peaks*, that is *peaks* relative to multiples of C_5 are higher, while *peaks* relative to multiples of F_5 are weaker.

3.4 AMT as a Pattern Recognition Problem

Pattern recognition is formally defined as the process whereby a received *pattern-signal* is associated to one of a predefined number of *classes* (*categories*). Since we have to associate sounds to *notes*, we can consider *AMT* as a typical problem of *pattern classification*. *Pattern recognition/classification* can be implemented by means of one of the following tasks:

1. *Supervised Classification* (*discriminative analysis*)
   the *input pattern* is identified as a member of a predefined *class* (the *classes* are defined by the system designer)

2. *Unsupervised Classification* (*clustering, categorization*)
   the *input pattern* is assigned to a hitherto unknown *class* (*classes* are learned on the basis of similarities existing among *patterns*)

The design of a *pattern recognition* system involves three aspects:

1. data acquisition and pre-processing
2. data representation
3. decision making

Obviously the problem domain dictates the choice of pre-processing techniques, representation scheme and decision model; moreover, it suggests different approaches for solutions. The four best known models are:

1. template matching
2. statistical classification
3. syntactic or structural matching
4. neural networks

It would be wrong to consider these approaches as necessarily independent: sometimes they are strictly connected and the same *pattern recognition* method exists with different interpretations. In our work, we have explored only the capabilities put at our disposal by the *neural* approach.

3.4.1 Neural Networks for Automatic Music Transcription

The approach followed for *AMT* is based on the segregation and a subsequent processing of the *harmonic components* of *musical signal* in digital format, that is converted in a *t/f representation*, called *spectrogram*, by means of *QFT*. The *harmonic analysis* in time of the *input signal* is

performed as a processing of the *time series* extracted from the *spectrogram*. Therefore, the *note detection* operation is a *dynamic pattern recognition* problem, or in other words, the *t/f representation* leads to a formulation of *AMT* as a problem of *multivalued time series processing*. This is why we have exploited architectures of *dynamic neural networks* for *time series classification* (Fig. 3.36).

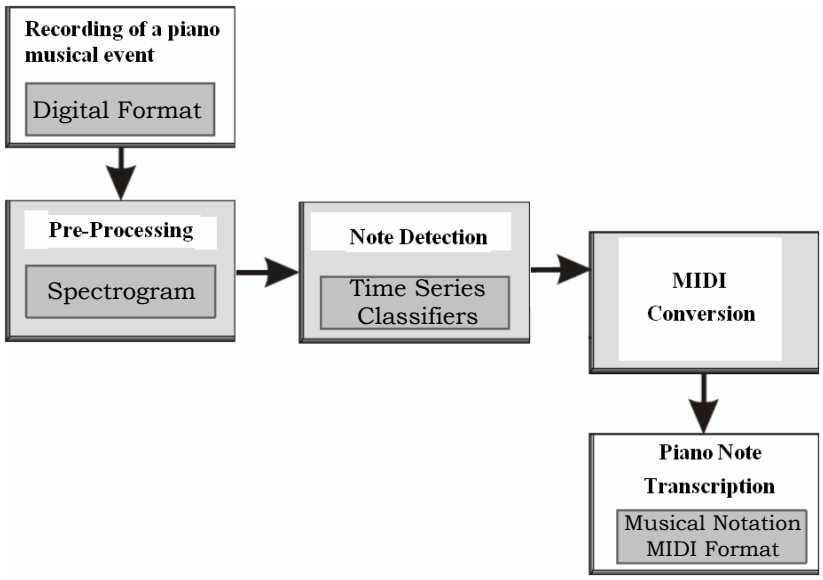


Fig. 3.36: Automatic Transcription System

The *neural network* model has the purpose to encode the empirical knowledge represented by the *training samples* into a corresponding set of *synaptic weight vectors* $\boldsymbol{W}$. The main characteristic of *neural networks* is that they can learn complex non-linear input-output relationships, use sequential training procedures, and adapt themselves to the data. The most commonly used *neural network* for *AMT* as a *pattern classification* problem is the *feed-forward multilayer perceptron network*, as previously described. We have seen that the *learning* process aims to adjusting the *network architecture* as well as the *connection weights* so that the *network* can efficiently perform a specific *classification* task. As regards the *AMT* application, we sometimes talk without distinction about *classification* and *discrimination*. Although strictly correlated, *classification* and *discrimination* have quite different meanings. The target of *classification* consists in predicting the *class m* that an observation *x* belongs to. *Discrimination* aims instead to subdivide the sample space *X* in two or more disjoined regions, to be associated to *M* given *classes*. Actually, in case of *AMT* it should be more appropriate to talk about *discrimination*. This is why, when we introduced the *learning* algorithm for *classification*, we implemented a *classification* system based on the so called *discriminative functions* $g_i(x)$. The *classifier* assigns a *pattern x* to the i^{th} *class*, if $g_i(x) > g_j(x)$, $\forall j \neq i$. Marolt's system for *classification* of *time sequence patterns*, SONIC, is based on the previously described *TDNNs*. He opted for this choice, that showed itself to be the best, at the end

of a long test session, where he tried to exploit several *neural network* models (MLP, RBF, time-delay, Elmann, Fuzzy ARTMAP). In our work we have instead privileged the *RNNs* and in particular the *Locally Recurrent Neural Networks* (*LRNN*) with *spline activation function neuron*, trained by means of the *on-line CRBP* algorithm, modified in order to introduce *discriminative learning*. Our complex system for *AMT* for *piano* music has been structured as shown in Fig. 3.37. In this scheme, each *network* is a *Minimum Classification Error Locally Recurrent Neural Network* (*MCE-LRNN*), that is, a *Recurrent Neural Network* trained with *discriminative learning*: each *network* is trained to recognize a *note* (its *target note*) and has two outputs that correspond to the classes *note ON* and *note OFF*. The *input target* is forward propagated thru each *network*; if the output of a network has a value that's above a given *threshold,* this means that the relative *note* is *ON,* or *OFF* otherwise. The *input pattern* is just the output of the *pre-processing* phase and in particular the *frequency bins* of a *frame* of the *QFT* of the input *training signal*.

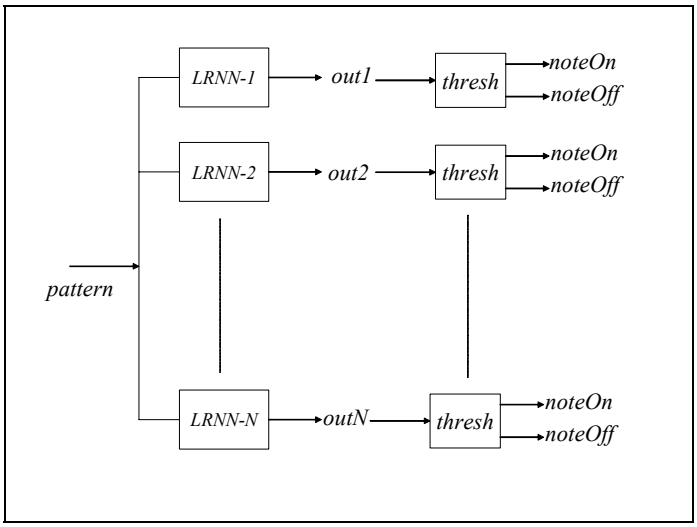


Fig. 3.37: Block diagram of a neural AMT system based on CRBP-LRNNs

After a first step dedicated to the *individual training* (based on *positive examples* only: we present to each *network* only examples formed by *waveforms* of the *note* to which it's dedicated), we turned to a further *training* of the same *networks*, by exploiting *discriminative learning*, in order to solve the following problem: each *network* can learn a lot about the *note* which it is assigned to, but has no idea on how this *note* differs from each other *note*. In order to reduce the *probability of misclassification*, we need to accomplish a *mutual training*, where the *error function* relative to the desired *note* is reduced, while the one relative to an unwanted *note* is simultaneously increased. Nevertheless, this approach cannot be directly adapted to *AMT*, since it implies that each *input pattern* (the *note*) belongs to one and only one *class*. Actually, the *AMT* application can require that an *input pattern* belongs to more than one *class* (*polyphony*). This is why we had to turn to *LRNNs trained* by means of *multi-classification discriminative learning*, based on the *Minimum Classification Error* (*MCE – LRNN*).

3.4.1.1 AMT Testing examples

Now it's time to collect all the subjects we have discussed before and define the testing procedure we set up for our purposes. The work accomplished can be summarized as follows:

1. *MIDI* files collecting
2. *WAVE* rendering
3. *input-target* tracking
4. *neural network* modelling
5. *training*
6. *testing*

In the following examples, our aim was to test *dynamic neural networks*, trained by means a base *CRBP* algorithm, where the *cost function* to minimize is the classic *mean-square error*.

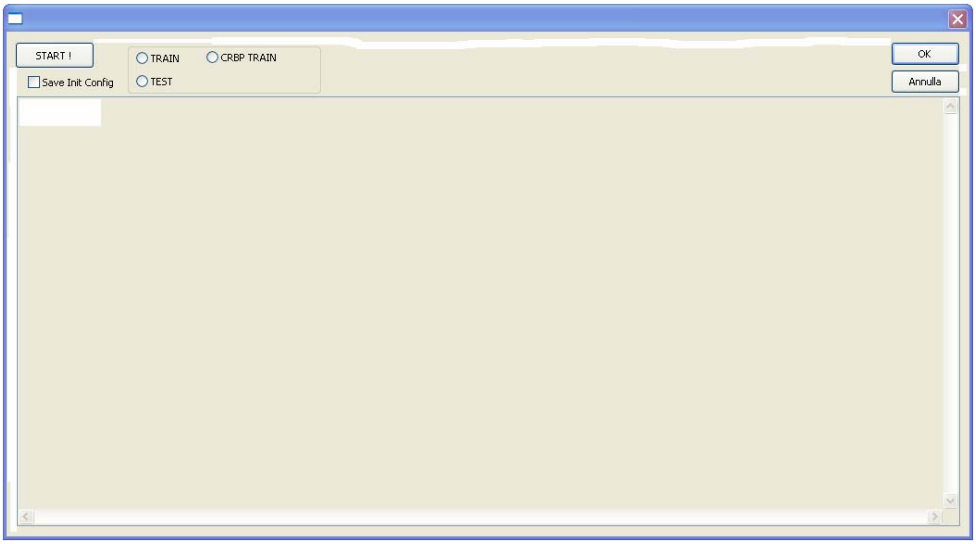


Fig. 3.38: user input form of the C++ application used to train and test LRNNs

Fig. 3.38 reports the main form of the home written application for training *LRNNs*, by means of the *RBP*, *CRBP* algorithms, and subsequently test them. The user can input the parameters that describe the network thru a script file that the application reads after the user presses the START! button. This script has a sintax similar to the one that follows:

```
TRAINING SET         : do3.trs
N° DI STRATI         : 3
ERRORE MINIMO        : 0.12
MINIMO DELTA ERRORE  : 0.001
N° MASSIMO DI EPOCHE : 10
LEARNING RATE        : 0.01
MOMENTUM             : 0.1
TRAINING TYPE        : causale
                     : non discriminativo

STRATO 1
              FUNZIONE DI ATTIVAZIONE   : spline
                                          DELTA         : 0.4
                                          L. RATE       : 0.0011
                                          TIPO          : SIGMOIDE
                                          A             : 2.1
                                          B             : 0.044
                                          C             : 3.7
```

```
              N° DI NEURONI             : 30
              N° TAPS MA                : 10
              N° TAPS AR                : 10
              Q                         : 4

STRATO 2
              FUNZIONE DI ATTIVAZIONE   : standard
                                          TIPO          : LINEARE
                                          A             : 2.1
                                          B             : 0.044

              N° DI NEURONI             : 30
              N° TAPS MA                : 10
              N° TAPS AR                : 10
              Q                         : 4

STRATO 3
              FUNZIONE DI ATTIVAZIONE   : standard
                                          TIPO          : SIGMOIDE
                                          A             : 1.0
                                          B             : 0.0
                                          C             : 1.0

              N° DI NEURONI             : ----
              N° TAPS MA                : 10
              N° TAPS AR                : 10
              Q                         : 4
```

Moreover, in order to input to the software the training set, or the test set, the user can exploit the files produced with the Matlab software described above.

A simple test

We started our analysis with a simple test (maybe the simplest test): we have considered an A_4 file in its two formats (*MIDI* and *WAV*). The *tracking* of the *MIDI file* is very simple, since it represents an A_4 starting after 1 s and stopping after 1 s. The parameters of the *QFT* were: $f_{\min} = 95$, $f_{\max} = 2100$, $b = 48$, $f_s^{qft} = 1/21 f_s$; we trained separately a *TDNN* (*FIR* = 1, *IIR* = 0) and a *RNN* (*FIR* = 1, *IIR* = 1) both of which had 215 inputs, one *hidden layer* with 3 *hidden neurons*, one *output neuron* and *learning rate* = 0.001. The *activation functions* were *hyperbolic splines* with *delta* = 0.4 and *learning rate* = 0.01 in both layers. The *training* results are depicted in Fig. 95.

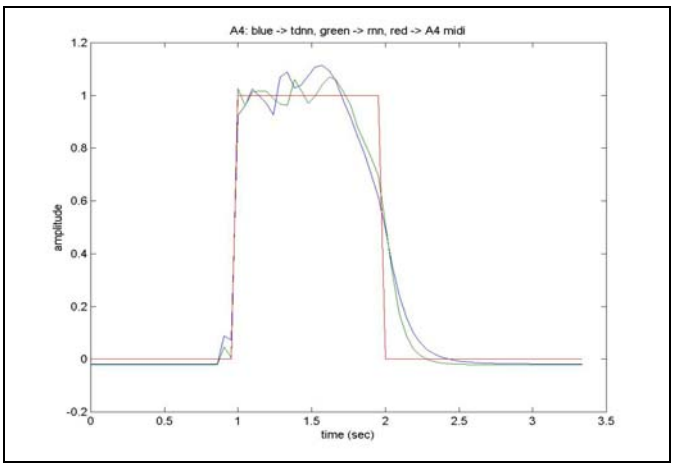


Fig. 3.39: training results of a simple A_4 as input to a TDNN and an LRNN

We have then considered an E_4 and tested it through our *networks* (previously trained with the A_4

pattern). The results we obtained are depicted in Fig. 3.40:

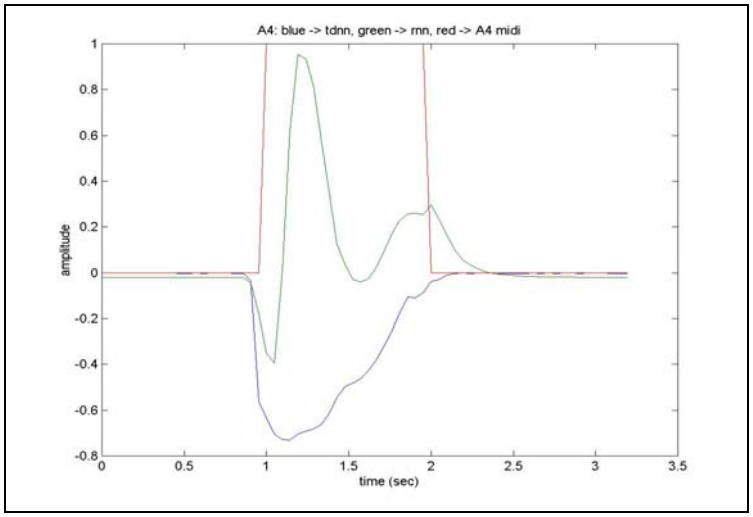


Fig. 3.40: training results of a simple E_4 as input to the same TDNN and LRNN of Fig. 3.39

As we can see, the *TDNN* shows better generalization with respect to the *RNN*. For example, if we choose a threshold of 0.5, it is clear that the *RNN classifies* E_4 as an A_4 for a short time (it *misclassifies* E_4). We have then taken into account an A_3; in Fig. 3.41 what happened is shown:

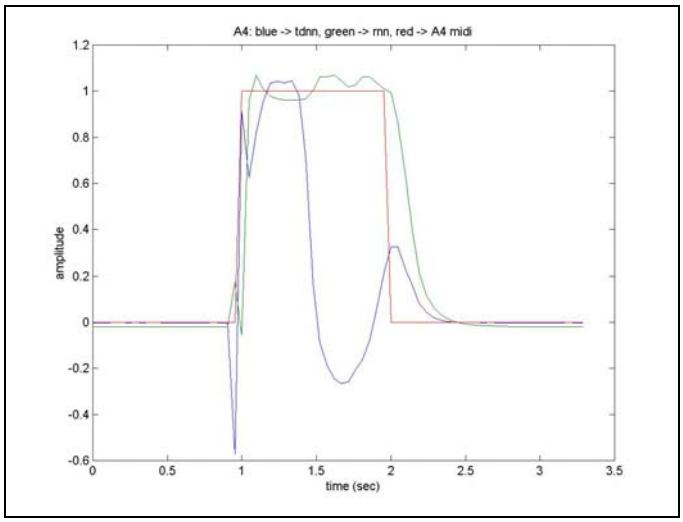


Fig. 3.41: training results of a simple A_3 as input to the same TDNN and LRNN of Fig. 3.39

It is clear that the *LRNN* exactly mistakes A_3 for A_4, while the *TDNN* reveals only a short *misclassification* at the beginning of the sound. These results are quite general: the *TDNN* trained on A_4 shows good generalization, except for *A notes* belonging to other *octaves*. We are not surprised at this, since the *QFT* of *notes* with the same name, but belonging to different *octaves*, are quite similar, except for the *fundamental peak* (that corresponds to the *fundamental frequency*). On the contrary, the *LRNN* shows a worst behavior: it has not learned what a *note* is, or in other words, when a *non-note* appears, it hasn't the possibility to distinguish it from the original *note*. However, it's important to emphasize the fact that the value of the *mean-square error* reached on the *training set* (in this case on A_4 only) is lower in *LRNN* than in *TDNN*. Taking the advantages of this last

aspect, we trained the *TDNN* and the *LRNN* on a *training set* made up of A_3, A_4 and E_4. Fig. 3.42 confirms what we stated before: the error reached by the *LRNN* is lower than the one reached by the *TDNN*.

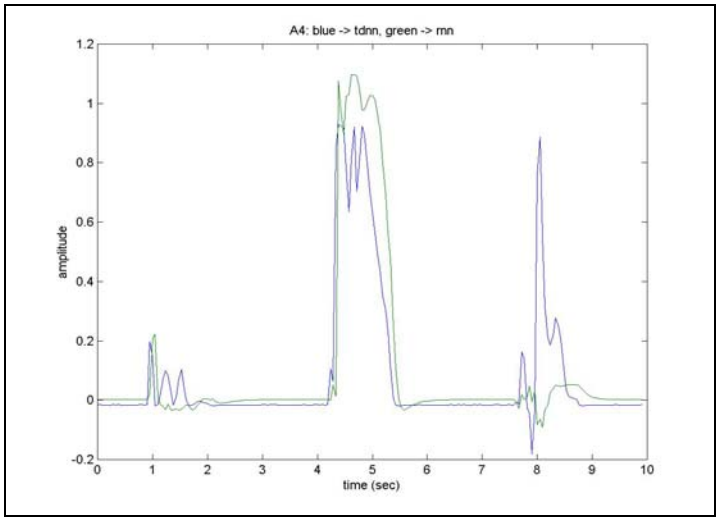


Fig. 3.42: training results of a sequence of A_3, A_4 and E_4 as input to a TDNN and an LRNN

These are the outputs of the *TDNN* and the *RNN*, when A_3, A_4 and E_4 are presented to them. As we can see, the *RNN* more or less correctly *classifies* all *notes*, while the *TDNN misclassifies* the E_4. This sounds quite strange, since in the previous test the *TDNN* showed good results on E_4. What happens in this last test is not a pure coincidence, since the *LRNN* reveals the possibility to be better *trained* than the *TDNN*, above all when the *probability of misclassification* is quite high. In our last test, in fact, what generates confusion is the presence of A_3 and A_4 together in the same *training set*, without the possibility to understand exactly (for the *TDNN*) what an A_4 is and what a non-A_4 is. What we are saying is confirmed by the fact that if we test the so trained *networks* with an A_5 and a C_5 we obtain the result depicted in Fig. 3.43.

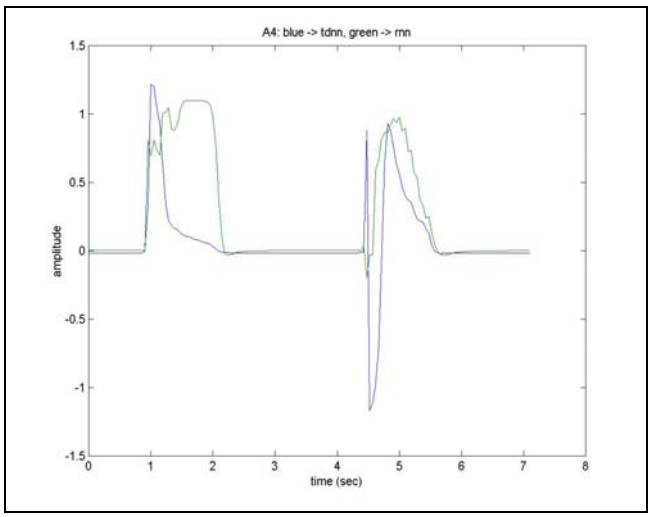


Fig. 3.43: training results of a sequence A_5 and C_5 as input to the same TDNN and LRNN of Fig. 98

Paradoxically, the *LRNN* gives a worst *classification* when A_5 is presented, with respect to the C_5.

That's because it has learned what an A_4 is with respect to A_3 (and so with respect to a similar note in terms of *QFT*), but it knows what a non- A_4 is, with respect to E_4 only. On the other side, the *TDNN* reveals a behavior that is coherent with what we have stated before. These simple experiments reveal if we work on an optimum *training set*, the *LRNN* has the capability to be more efficient than the *TDNN*.

3.4.1.2 General considerations

Working with different *training/testing sets*, we could understand what generates confusion in the *learning* of *notes*:

1. the distinction between *near notes*
2. the distinction between *notes* with the same name but belonging to different *octaves* (the famous *octaves errors*)
3. *velocity* of *notes*
4. *polyphony* (above all when we are in condition 1, 2 and 3)

We trained in paralle *TDNNs* and *LRNNs* on *training sets* built upon the conditions above (that is with *near notes*, strange *polyphony*…) and then tested them on similar *test sets*. *LRNNs* showed best behaviours both on *training* and *test sets*, while *TDNNs* errors were above all *octaves errors*. If we test the *trained networks* on general *musical pieces*, what happens is that *LRNNs* show a better behavior only if *notes passages* in the *musical pieces* contain structures previously seen in the *training sets*; if that's not true, *TDNNs* show better *generalization*, even if they are not always coherent with what they have learned: it can happen that the same *note passages* are interpreted into different ways, if they appear in different parts of the *musical piece*. When we tried to complete the *training set* with those not-seen *passages*, we found that the *LRNN* shows improvements with respect to the *TDNN*; however, the number of *note passages* that is possible to find in music is enormous (complicated by the presence of growing *polyphony*) and therefore it becomes impossible to include all of them in the *training set*. Moreover, a too large *training set* is hard to minimize (in terms of errors) and can be redundant for most of the *musical pieces*. The possibility to construct ad hoc *training sets* for particular *musical genres* can be a possible solution; this is, however, a defeat for us, since our intention was to find an *architecture* that could be as general as possible.

3.4.1.3 AMT with MCE-LRNNs Testing examples

In the previous sections, we have focused our attention on the comparison between *TDNNs* and *LRNNs*, by observing their different way of working. Even if *LRNNs* reveal the possibility to bring

improvements to the solution of the *AMT* problem, they still appear insufficient, because of their necessity to be *trained* on sets that assure generalization on one hand and that are specific for *musical genres* on the other. Moreover, this sort of a model (that is the one depicted in Fig. 3.37) has to be trained differently for each *network*, since if a "minimum note quantity" is not assured in the *learning* phase, the *network* hasn't the possibility to learn; on the other hand, if an "excessive note quantity" is brought, the *network*, then it will be *over-trained*.

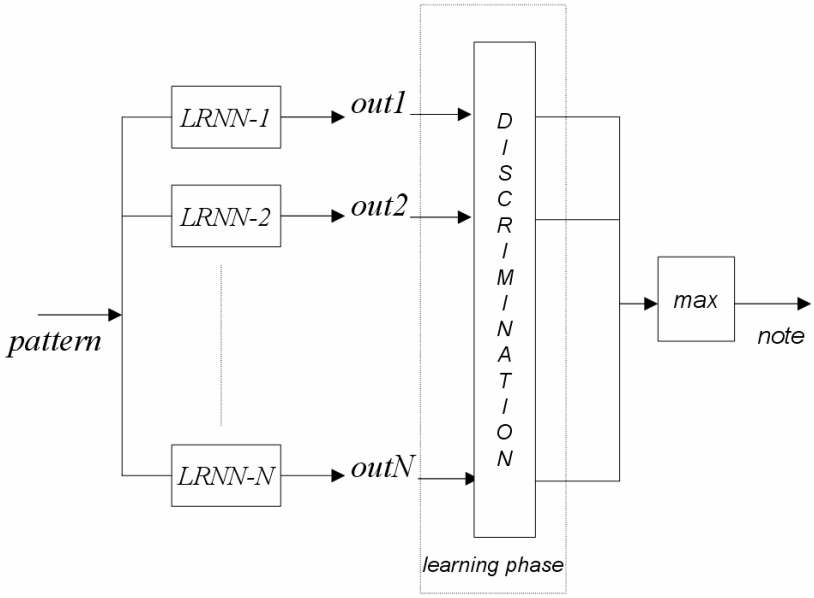


Fig. 3.44: Block diagram of a neural AMT system based on MCE-LRNNs

This is why we decided to turn to the *MCE-LRNN* model. An architecture for *AMT* that's based on this model is the one depicted in Fig. 3.44. During the *learning* phase, we operate *discriminative learning*, adjusting nets' *weights*, in accordance with the *target notes*. During the *test phase*, we simply calculate outputs and choose for higher values.

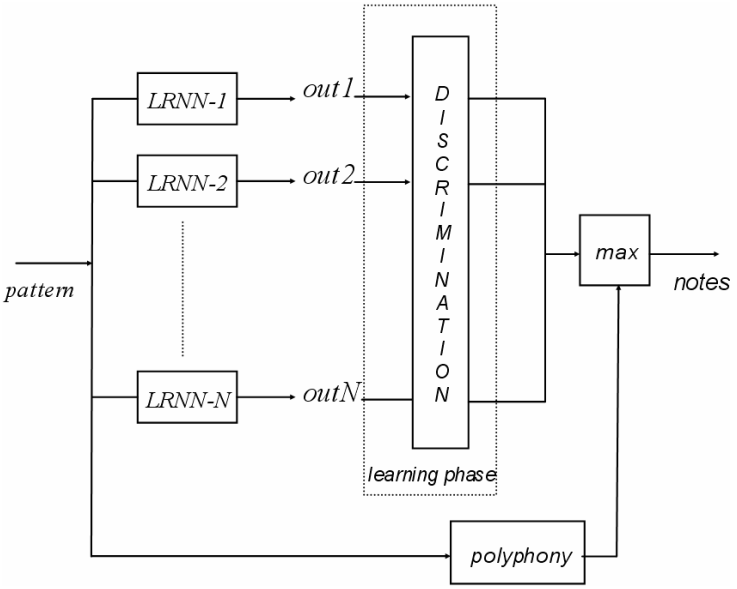


Fig. 3.45: Block diagram of a neural AMT system based on MCE-LRNNs and polyphony detector

There's however a problem: there's no general rule to use in order to decide how many outputs to take, so we needed an external block in order to perform this task. We used a *polyphony net*, that is trained to recognize how many notes are playing (are *ON*) at a particular time. The new model is depicted in Fig. 3.45.

3.4.1.3.1 Monophony

We have then considered the note range [G_3… $F\#_5$]. The parameters of the model are the same as before except for $f_{\min} = 190$, $f_{\max} = 1500$ and consequently for the number of inputs of each *network*, that is 144. We trained each *LRNN* by means of the *CRBP* algorithm, in order to recognize its target note, by simply using a couple of *MIDI*/*WAV* containing the *target note*: in other words, we only used a positive (or a set of positive) *example* for each *network*. After this phase of individual pre-training, we have performed the first phase of *discriminative over-training*, in which the whole *training set* (made up of the single *audio files* used to train the single *nets*) is presented to each *net* within a *discriminative learning* processing. By means of the *discriminative over-training*, we obtained a more coherent architecture, in which each *net* is related to the others thanks to the fact that *weight variations* depend also on what happens to the other *nets*, when the same *input pattern* is presented to all the *nets* in the system. The surprising result is that after this *over-training* procedure, the model is able to correctly recognize each *note* given as input, without the need to increase the *training set*. As we have seen before, this couldn't be achieved by means of a standard *LRNN* model.

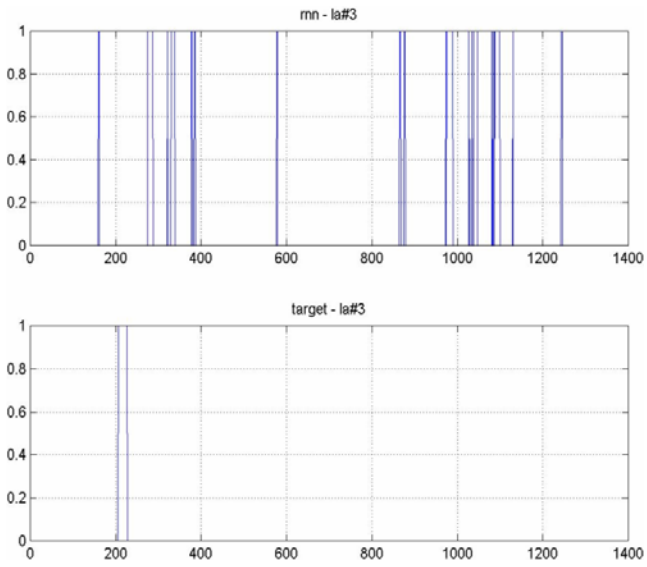


Fig. 3.46: Test results for all the nets without A_3 net (upper) and for A_3 net (lower)

In Fig. 3.46. we can find the response of the network $A\#_3$ when all notes in the range [G_3… $F\#_5$] are presented to it. Obviously, the *peak* in the target part of the figure is related to the presence of $A\#_3$ as input to the *network*; *peaks* in the upper part of the figure are related to the *network*'s behavior when

different *notes* are presented; in particular, when the $A\#_3$ is presented, no output is associated. After the *over-training* step, each *network* correctly recognizes its *target note* without *misclassifications*. As regards the *positive set* used to train each *network*, something more has to be said: at first, we used a set made up of *single notes* (each *network* was individually trained on its *target note*) of 1 s in length. The model was able to recognize *notes* with a duration no longer than 0.65 s; in order to obtain a more flexible model, it was necessary to use a *training set* made up of variable length *notes*. We also tested the so trained model with *monophonic sequences*: they were all successfully recognized. This is really interesting and reveals the strength of this architecture: no *train* was made on *sequences* and in spite of this we were able to obtain correct *classification*. As stated above, the general variable-length *monophonic recognizer* can be obtained only if a *variable-length training set* has been used. Now, we have to place emphasis on a particular situation: if we use a *training set* made up of 1 s length *notes*, with the *QFT* parameters previously specified, and try to test *sequences* with 0.65 s length *notes*, we find some distortions in case of $F\#_5$, while everything works in case of F_5 and D_5. This happens because the *QFT* frequency range is too short; in particular, the value of $f_{\max}$ is too low, considering that $F\#_5$ has the *fundamental peak* at 739.99 Hz and the second *peak* at 1479.98 Hz, that's too close the upper bound of the *frequency range*. In order to assure a better behavior at higher frequencies, it is necessary to use a wide enough frequency range, that could include at least three *peaks* of the highest *note* in the set. While working with *discriminative learning*, we have to pay attention to the right value to assign to parameter η (in *misclassification measure*). We found that a reasonable value for η is in the range [4…8]; while working with single notes, we could increase its value up to 16, while in case of *polyphony* better results can be get with the classic values.

3.4.1.3.2 Polyphony

In order to test the architecture depicted in Fig. 3.45, we considered the same note range [G_3… $F\#_5$] and the same *QFT* parameters adopted for the *monophonic* case. We started with a *polyphony* of two, that is with two *notes* played at the same time. Starting from the *pre-trained model* obtained before (each *LRNN trained* individually and then *over-training* of the model with *discriminative learning*) and taking into account, in order to form the *training set*, all the *notes pairs* that can be chosen in the above range, we *over-trained* the model always by means of a *discriminative* procedure. Even in this case, we could get correct *classification*. Moreover, this good result could be achieved even testing again the model on single sequences. This means that *over-training* for *polyphony* doesn't corrupt the results obtained for *monophony*. When we tried to increase *polyphony*, it was necessary to consider sub-classes of *polyphony*. In other words, instead of *over-*

training the model with all possible *sequences* of *n notes* (if polyphony is *n*), we found reasonable to consider *classes of chords*. For example, we considered *3-notes chords*, (*major chords*, *minor chords*, *diminished chords*, etc) and *over-trained* the model with the *training set* formed this way. After *over-training* the model with *major chords* only, we could get again complete *classification*. Unfortunately, we found that directly *over-training* a *single-note-trained* model with a *3-polyphony training set* generates distortions. We solved this problem by previously *over-training* the model with at least all pairs of *notes* contained in each *3-notes chord* (for example, if we have to *train* the model to recognize the *chord* C_4 - E_4 - G_4, we need to previously *over-train* the model with *bi-chords* C_4 - E_4, C_4 - G_4, E_4 - G_4) and only after to *over-train* with the *3-polyphony training set*. While working with high *polyphony* values, distortions arose when we tested the new model with previous *trainings sets* or *sequences*. Nevertheless, this was suspect, since, based on our previous experience, we found strange that *over-training* corrupts previous results. So, we reconsidered the previous *over-trainings* or the choices made about the model's parameters. We found that in most of the situations, the problem relies in the wrong choice of η, or in the short *frequency range* used to compute *QFT*, with respect to the *notes* in the *training sets*. Moreover, it was clear the usefulness of *variable-length training sets*. When *polyphony* is too high, the model has to be trained with *variable-length notes*, in order to better single out them when played in *sequences*. In fact, the passage from a *note* to another always generates confusion and, even if it is quite simple to obtain good results with *monophonic sequences*, everything becomes really hard when we face *polyphony*.

3.4.2 Conclusions

Finally, we can assert that good results have been obtained, even if many aspects of the *polyphony* problem still have to be explored in depth. Anyway, the choice of a valid *training set* seems to be the key problem. Relying on the potentialities of the *DL* in terms of *classification* (or better, in terms of *discrimination*), and on the strength of *LRNNs* in terms of *generalization*, there's the need to exactly understand how to train the model in order to improve its efficiency. The *MCE-LRNN* model has revealed its efficacy in most of the typical *AMT* problems, by combining the advantages of *LRNNs* (with respect to other *dynamic pattern recognition* techniques) and *DL* (with respect to other *classification* techniques). Because of the novelty of this approach, it was impossible to explore it in all of its parts. This is why we decided to consider at first the relationship with standard models, finding out the possibility to train the new model in a more suitable and simple way. We have supposed that the choice of the right *training set* becomes crucial in the development of a general scheme that can be adapted to every *musical genre*. On the other hand, this recalls the human way of learning *music*: a *pre-training* about what a *note* is, its relationship with other *notes*

(above all with *near notes*), *polyphony* and finally the focusing on a particular *genre* are required. Future developments of this study will consist in investigating how the model reacts to growing *polyphony* and how *over-training* modifies previous results. It will be necessary to *pre-train* again the whole model with a richer *variable length single note training set*, in order to assure the possibility to recognize *notes* even if they are really short. It will be necessary to reconsider the choice of f_s^{qft}, that is strictly connected to *note's* length and separation, making the *learning* phase longer but much more accurate. Moreover, other improvements could be assured by opting for a different kind of *target file*: instead of considering each target file as a series of 0 (*note OFF*) and 1 (*note ON*) , as found by the *tracking* of the *MIDI file* by searching for the *note-onsets* and the *note-offsets*, we could consider that only the *steady-state* of a note has property of being 1. Finally, still more effective improvements can be achieved, if typical *speech recognition*'s techniques are introduced in our *AMT* system: some structural, unavoidable errors typical of the *transcription* system could be corrected by a *post-processing block* (generally an *Hidden-Markov model* or a *neural network*), as in Fig. 3.47.

Fig. 3.47: Test results for all the nets without A3 net (upper) and for A3 net (lower)

The *post-processing* block receives in input the *MIDI file* supplied by the *MCE-LRNN Automatic Music Transcriber* and is trained with respect to the correct *MIDI file* (the one with no errors), that's the same file we used to synthesize the *WAV file*. It's a hard job, since the model should be based on a *multi-rate adjustment technique*, in order to allow the adjustments back propagate even to the previous levels. Anyway, for sake of simplicity, in a first step we could try working with *static networks*, dealing only with the reconstructed *MIDI sequence*, with no back propagation. Finally, we have to always bear in mind the key role of the *polyphony network*. In fact, an interesting investigation could be the one that aims to examine in deep the behavior of this *network*, especially as regards the definition of the best *training set*. Anyway, a still grater improvement could be introduced by an intelligent refinement of the *pre-processing* block, where the *QFT* could be further processed in order to define a higher level set of *input features* for the *network* (see next chapter).

CHAPTER 4

AN APPLICATION OF STATIC NON EXCLUSIVE CLASSIFICATION: IDENTIFICATION OF MUSICAL SOURCES

4.1 Introduction

The perception of sounds and the identification of the sources that generated them is one of the most important activities of human beings. The association of sound to sources is essential for survival: the alarm of a car, the voice of a friendly person, the roar of a lion must be recognized in order to make the best choices as regards our probability to stay alive. Even if this mechanism occurs in a semi automatic way, until now our knowledge about the working of our brain, as regards the processing of sound signals, especially those associated to spoken language, is still scanty. In any way, what seems clear is that the association sound/source can be considered as a problem of *classification* of sounds: we *classify* a sound by associating it to its source (the *class*). Nowadays, search engines capable of identifying words in a written text are widely operative, but so far the identification of a given piece of music inside a musical document is still a hard task. Some interesting results have been achieved in human voice recognition, by means of a simple and useful application that allows the real time transcription of a spoken phrase. The real time *transcription* of a musical piece is another application that requires the identification and processing of sound sources, that are task that sorely try even musicians with a solid experience. In our purposes, we aimed to test many of the opportunities that one can exploit in order to solve the problem of the identification of sound sources, as well as to develop a new methodology that could go beyond the limits imposed by traditional methods. In particular, we have concentrated our interest on particular sound sources: the *traditional musical instruments*. Moreover, we have set up the problem based on the way musical sounds are identified by the human subject, or in other words, on the procedures that characterize the interactions between *ear* and brain. As for the *AMT* application, the *ear* and the way it processes signals have been modeled by means of the right *pre-processing* algorithms, while the functioning of the brain has been simulated by means of a particular kind of *neural network*,

capable of *classifying* a given sound source starting from its characteristic parameters. Nevertheless, when we try to implement the *classification* procedure, we face the relevant problem that a *musical instrument* can be played in many ways, with different styles and different strengths. Moreover, sounds produced by different *instruments* may have similar properties too. Therefore, when we try to *classify musical instruments* based on their characteristic parameters, we will almost as sure identify *decision regions* that *overlap*. In other words, we'll never have clean borders that uniquely separate the regions associated to the considered *instruments*. This is why we chose as the *classifier* to adopt *non exclusive fuzzy neural networks* and in particular FMMNNs, in the various flavours listed in the dedicated chapters.

4.2 Features of a sound signal

Characteristics in frequency	Sustain phase	*pitch*
		instantaneous spectrum envelope
		time averaged spectrum envelope
		cut-off frequency, tangent of the spectrum at that frequency
		absolute spectral centroid
		relative spectral centroid
		envelope of each partial in the spectrum
		spectrum irregularities and non present partials
		Energy of partials
		phase envelope
	Attack phase	*growth of the single partial*
		growth of the spectrum
		growth of the spectral energy
Characteristics in time	Sustain phase	*spectral intensity*
		Amplitude of pitch variations in the note
		Variations of the amplitude of the note
		Amplitude of tremolo – vibrato – amplitude modulation
		Relation between tremolo and pitch variations
		time evolution of the spectral centroid
		time evolution of partials
		Relation between vibrato and partial variations
	Attack phase	*rise time*
		time envelope
		Tangent to the rise time
		time of growth of partials

Tab. 4.1 the main characteristics in time and frequency of the musical signal

The first problem that arises in the recognition of *musical instruments* consists in finding the opportune *characteristics*, from now on called *features*, that univocally define sounds emitted by a *musical instrument*. In the past, was common opinion that important information about a perceived sound can be derived only from its *spectral characteristics* and not from the time *waveform* of the same sound. Nowadays, has been accepted the fact that even *time characteristics* can have they value and importance. As we know, *musical signal*s can be analyzed in both *time* and *frequency* domains. From this analysis various *features* can be derived, as listed in Tab. 4.1. Such characteristics can be used in the recognition of *musical instrument* and can be derived from the *preprocessing* phase of the sound signal. They will be explained in the following.

4.2.1 Frequency analysis: spectrum derived features

4.2.1.1 Pitch

According to ANSI (American National Standards Institute), the definition of *pitch* is as follows: "*pitch is that auditory attribute of sound, according to which sounds can be ordered on a scale from low to high*". *Pitch* is in relation with the *frequency* of a *musical note*. The relation between *frequency* and *period* is $f = 1/T$, where f and T are the *frequency* and the *period* of the *note*, respectively. The *scale* of *pitches* has been fixed during the Conference of London (1939) based on the conventional assumption that fixes the frequency of the *reference note* A_4 to 440 Hz (also called *corista*). The Conference of London has also stated that we can attribute all the *notes* a *frequency*, based on the facts that the upper *octave* of a sound is characterized by a doubling in *frequency*, as well as that the *octave* is formed by 12 *semitones*. There exist various methods starting from which we can attribute a *frequency* to every *semitone* in the *octave* (*Pythagorean scale*, scale based on simple relationships, *tempered scale*). In case of the *tempered scale*, the "ideal" *pitch* of a *note* can be calculated by starting from the *frequency* of the *reference note* A_4 and multiplying or dividing it by $\sqrt[12]{2} = 1.059463094$, for every semitone that's between A_4 and the considered *note*. In Tab. 4.2, the *frequencies* of some *notes* of the *tempered scale*, as well as differences in *cents* (an *octave* contains 1200 *cents*) with respect to other *scales*, are reported. Obviously, the *instruments* do not have the same extension. Even starting from the evaluation of the *pitch* of a *note* we can argue whether it's been played on an *instrument* or another. During the emission of a sound by an *instrument*, the *pitch* of a *note* doesn't remain constant. Irregularities in the *blowing* of a *sax* performer, the way the *bow* rubs the *string* of a *violin*, the way the finger is hold on the *ponticello*, as well as many other parameters, all are at the basis of the *pitch modulation* of the *note* around a nominal value that characterizes its *pitch*.

Note	Tempered frequency	Tempered	Pythagorean	Harmonic	Mean tone	Werckmeister
do4	261.6	0	0	0	0	0
do #	277.1	100	113.7	70.7	76.1	90.2
re	293.6	200	203.9	203.9	193.2	192.2
re #	311.1	300	294.1	315.6	310.2	294.1
mi	329.6	400	407.8	386.3	386.3	390.2
fa	349.2	500	498.0	498.0	503.4	498.0
fa #	370.0	600	611.7	590.2	579.5	588.3
sol	392.0	700	702.0	702.0	696.6	696.1
sol #	415.3	800	815.6	772.6	772.6	792.2
la	440	900	905.9	884.4	889.8	888.2
la #	466.2	1000	996.1	1017.6	1006.9	996.1
si	493.9	1100	1109.8	1088.3	1082.9	1092.1
do 5	523.3	1200	1200	1200	1200	1200

Tab. 4.2 pitches of the notes in the 4th octave according the tempered scale and differences from other scales.

When the *pitch* of a *note* changes regularly around a nominal value, we talk about *vibrato*; when the variations are irregular, we talk about *jitter*. In Fig. 4.1, the evolution of the *pitch* of a note played on a *violin*, a *trumpet* and a *flute* are shown.

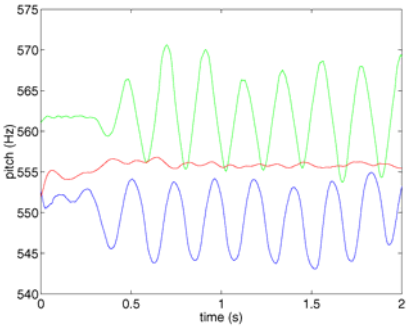


Fig. 4.1:pitch modulation of a note played on (from bottom to top) violin, trumpet and flute.

The three *instruments* emit the same *note* (for the sake of clarity, the *waveform* of the *violin* has been shifted by –5 Hz, while that of the *flute* by +5 Hz). In case of *flute* and *violin*, *vibrato* can be identified, while in case of the *trumpet*, the *pitch* variations are irregular (*jitter*).

4.2.1.2 Spectrum envelope

It has been noticed that the spectrum of some *musical instruments* has always an almost well defined shape. Inside a quarter of *octave*, *brass instruments* present only light displacements (in the order of some dB) from a *common spectrum envelope*, as depicted in Fig. 4.2, as regards the *French horn.*

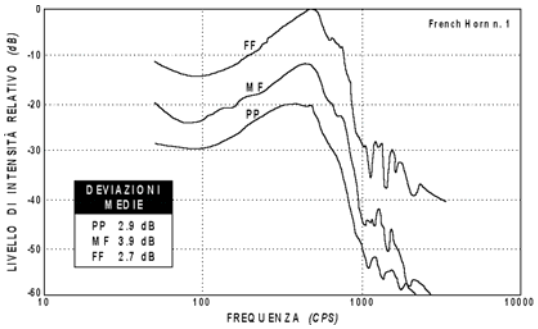


Fig. 4.2: envelope of the spectrum of a French horn. The three envelopes correspond to three different playing modes: pianissimo, mezzopiano and fortissimo.

As an example, in the spectrum of the *oboe*, the fundamental has an amplitude that's lower with respect to the higher harmonics, while in case of the *piano*, the spectral envelope seems to assume an exponential decreasing envelope ($y = e^{-x}$).

4.2.1.3 Spectrum cut-off frequency and slope of the tangent

We have seen that the slope of the spectrum of a sound emitted by a *musical instrument* can be seen as that of a *low-pass filter*, whose amplitude decreases with the increasing of frequency. Therefore, we can define a *cut-off frequency* below which the level of the spectrum never goes under a predefined threshold (-90 dB); moreover, in that point we can also calculate the *tangent* to the spectrum and take into account its slope (Fig. 4.3). This can be accomplished by calculating the tangent in correspondence of the last *peaks* (of the *harmonics*).

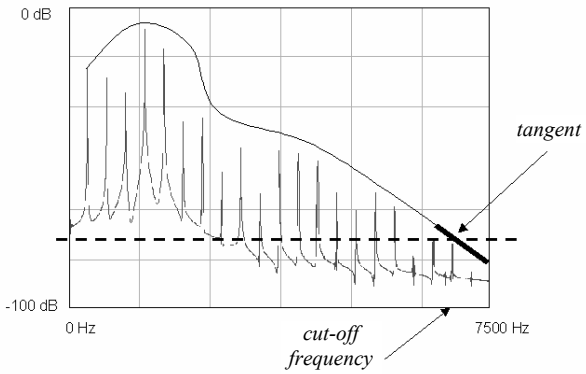


Fig. 4.3: envelope of the spectrum and tangent at the cut-off frequency.

4.2.1.4 Absolute Spectral Centroid

The *absolute spectral centroid* (or simply, the *spectral centroid*) is the first momentum of the *energy* of the spectrum and can be calculated as follows:

$$C = \frac{\int_{spectrum} f \cdot E(f)}{\int_{spectrum} E(f)} \tag{4.1}$$

therefore, it has the dimensions of a *frequency*. Since we are dealing with sampled signals, the discrete formula, that considers the spectrum as the output of many *channels*, is more useful. In our work, we have used 24 logarithmically distributed *channels* per *octave*, calculated by means of the *QFT*. The discrete formula of the centroid becomes:

$$C = \frac{\sum_k kE_k}{\sum_k E_k} \tag{4.2}$$

where E_k is the *energy* in the single *channel*. The corresponding *frequency* can be get as follows:

$$f = 1000 \times 2^{\frac{c-t}{o}} \tag{4.3}$$

where t is the channel relative to the 1000 Hz *frequency* and o is the number of *channels* in an *octave*. The *spectral centroid* is therefore the *frequency* of the point of equilibrium of the whole spectrum. Its value is often connected to the *brilliance* of the sound produced by a *musical instrument*. Fig. 4.4 depicts the slopes of the *spectral centroids* for *violin*, *trumpet*, *clarinet*, *flute*, *oboe* and *sax*:

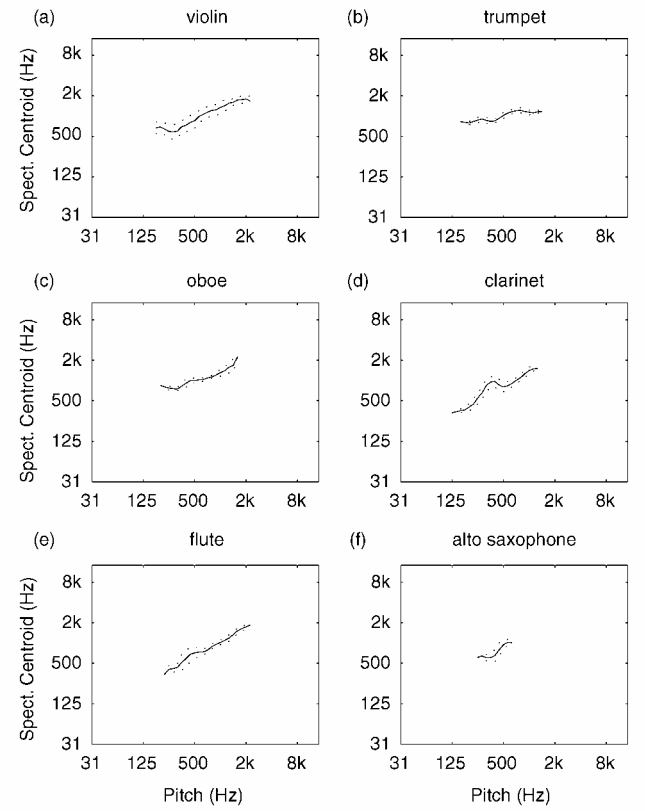


Fig. 4.4: slope of absolute centroid in relation with the fundamental frequency. The solid line represents the average value, while the dotted one represents the standard deviation.

4.2.1.5 Relative Spectral Centroid

It's the ratio between the *absolute spectral centroid* and the *pitch* of the *note*. Some slopes, in case of the same *instrument* of Fig. 4.4 are depicted in Fig. 4.5.

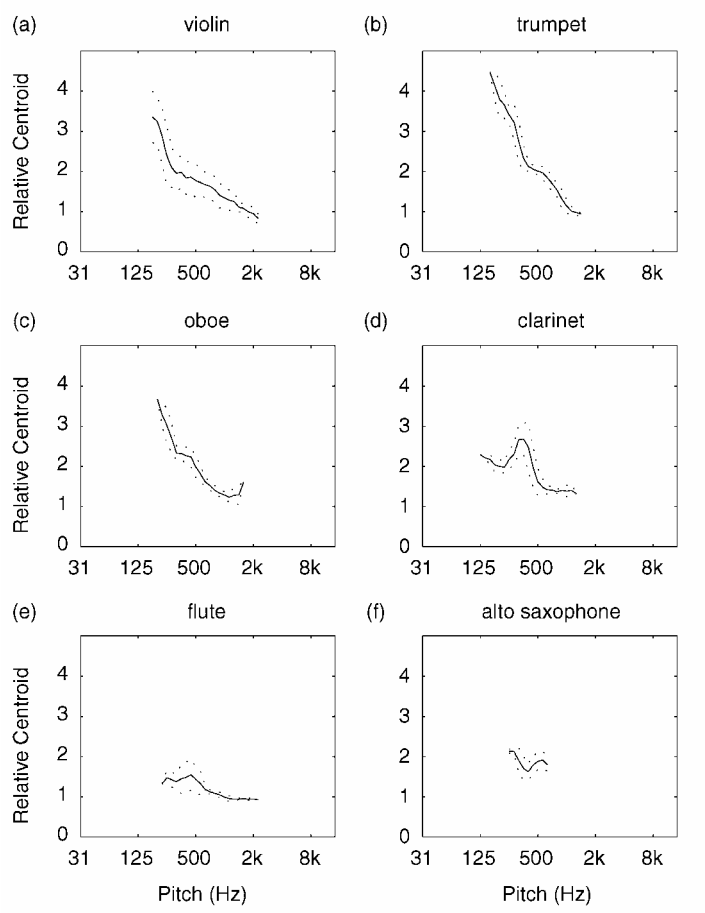


Fig. 4.5: slope of relative centroid in relation with the fundamental frequency. The solid line represents the average value, while the dotted one represents the standard deviation.

4.2.1.6 Maximum intensity of the single harmonic

An other way to analyze the spectrum consists in evaluating the *energy* contained in every single harmonic. In fact, through the analysis of the *harmonics* of the spectrum, one can accomplish the following energetic calculations (Fig. 4.6 to Fig. 4.8):

- the maximum value of every single *harmonic*
- the average value of every single *harmonic*
- the slope of the *harmonics*
- irregularities in the *harmonics* and *absent harmonics*
- *energy* in the odd *harmonics*
- *energy* in the even *harmonics*

It's worth noting that, while the first *harmonics* can fall in a single *channel* of a *QFT*, more than one *high harmonic* can fall in the same *channel*; therefore energetic value does not characterize anymore the *energy* of the single *harmonic*, but the *average energy* of more than one *harmonic* taken together. However this is how human auditory system works.

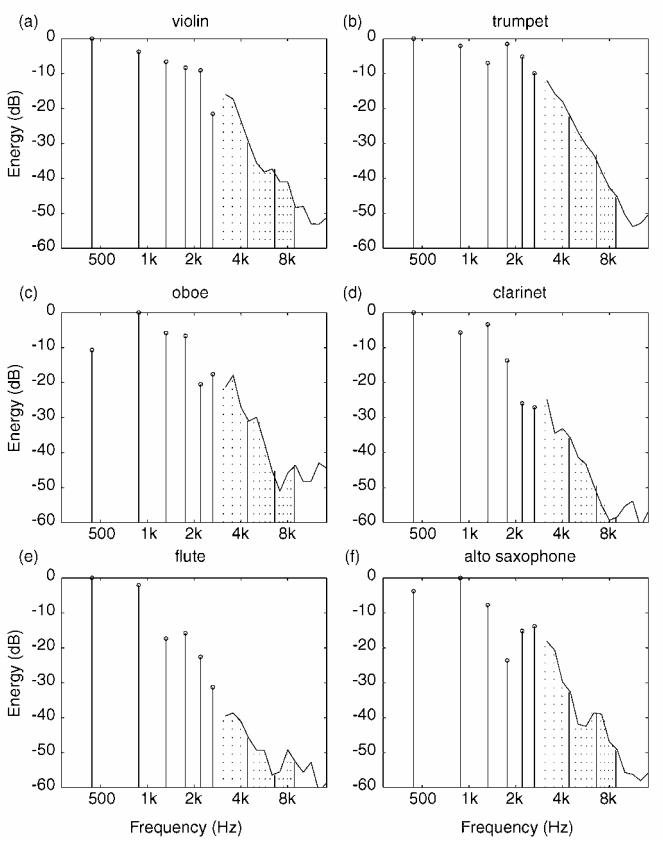


Fig. 4.6: maximum values of the harmonics. Above a given frequency (around the sixth harmonic), the channel cannot distinguish the single harmonics anymore.

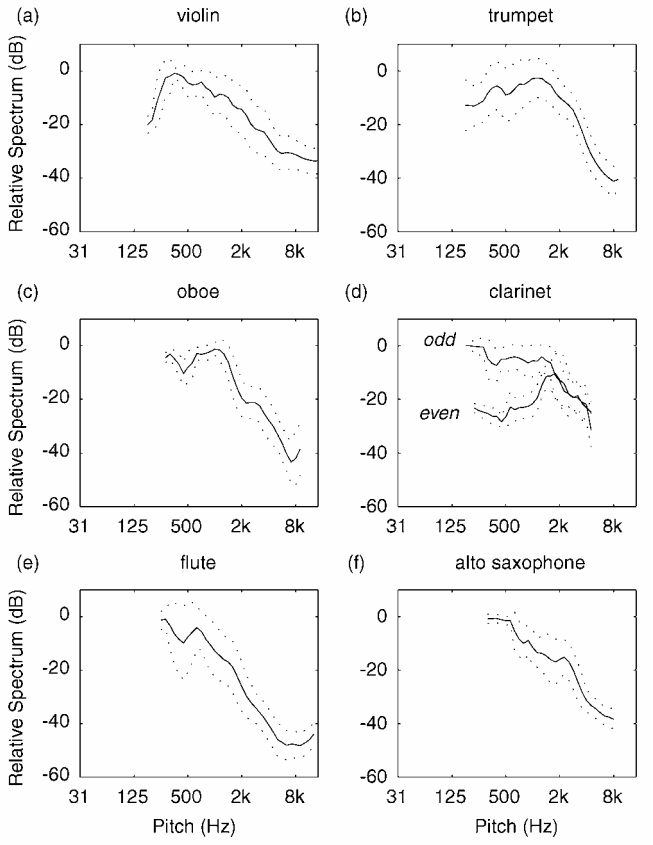


Fig. 4.7: envelope of the spectrum. Le solid lines represent the average value, while the dotted lines represent the standard deviation.

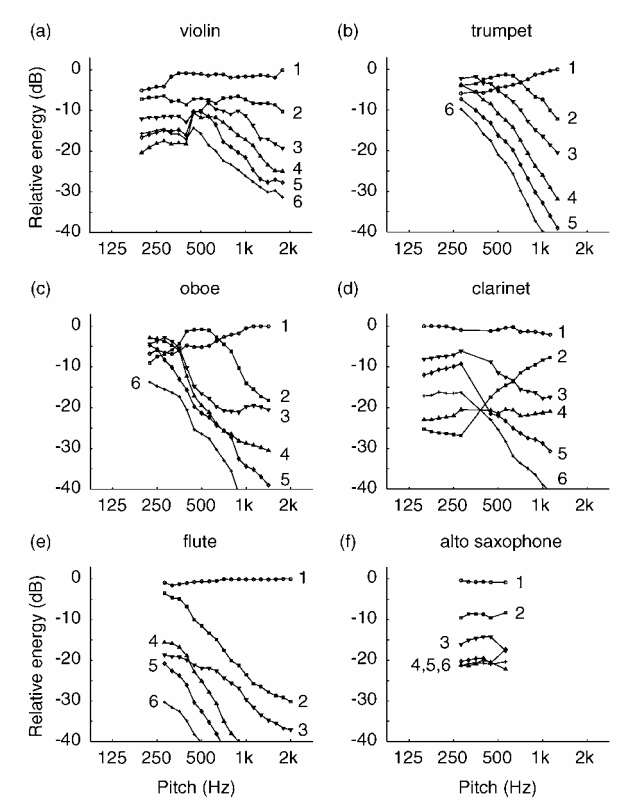


Fig. 4.8: amplitude of the first six harmonics with respect to the fundamental frequency.

4.2.1.7 The Phase

The slope of the *phase* can be exploited in order to locate more precisely the *peaks* of the *harmonics*. In our work, we have carried on experiments in which changes in the *tangent* of the *phase* around an *harmonic* were used with the aim of localizing the *frequency peaks*.

4.2.2 Time analysis

The first step in order to exclude from the beginning the belonging of a *note* to an *instrument* consists in finding if the *fundamental frequency* of the *note* is contained in the *tonal area* of the *instrument*. Indeed, an *instrument* cannot play all the possible *notes* and therefore it is possible to exclude some ambiguities. Moreover, advanced studies have shown that some *instruments* use some *notes* and *frequencies* more frequently. The histograms depicted in Fig. 4.9 show whether a *note* lies in the *tonal area* of an instrument and its frequency of occurrence.

4.2.2.1 Spectral intensity

The sum of the *energy* of the spectral envelope approximates the *power* of a sound. Its time graph gives an idea of the *amplitude modulation*. Therefore it is possible to single out *tremolos* (modulations of the total energy), *vibratos* and *resonances*.

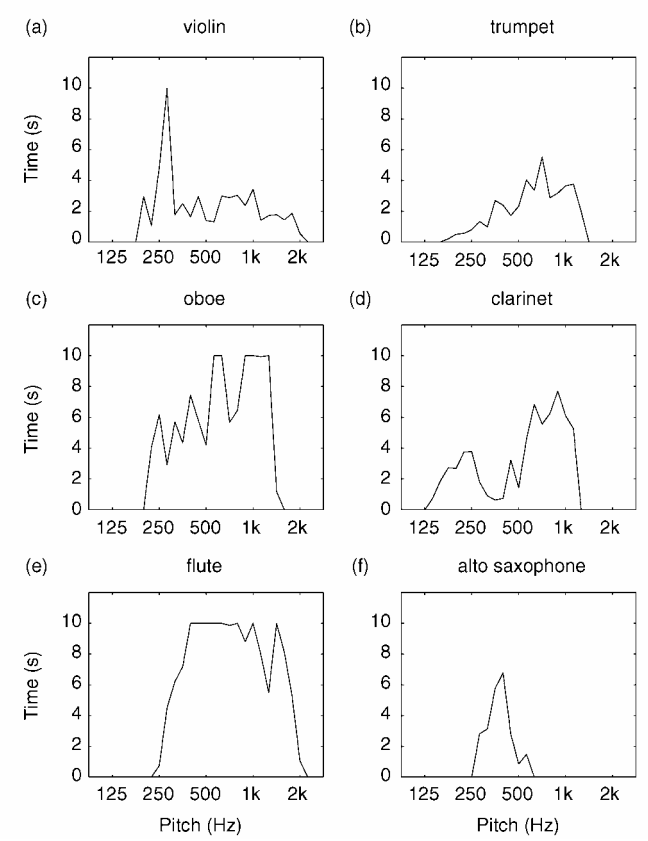


Fig. 4.9: Histograms of the fundamental frequency. On the abscissa the fundamental frequency is reported, while on the ordinate the time interval (10s max) during which the sound has been observed is reported.

According to Beauchamp this *feature*, along with the *spectrum centroid*, is of fundamental importance for the recognition of an *instrument*. The *modulation* of the *total energy* is also called *tremolo*. In figure they come shown some examples of instantaneous measures of intensity.

4.2.2.2 Vibrato

In case of some *instruments*, *vibrato* is intentionally introduced by the performer in order to obtain a regular variation of the *fundamental frequency* of the *note* within some Hz (usually about 6 Hz). However, in other cases, *vibrato* is involuntarily introduced by the performer or naturally produced by the *instrument* itself. This is the case of *oboe* and *flute*. Usually, *vibrato* is expressed in *cents*. By following the slope of the *fundamental*, we can get a tracking of the *pitch*. By *windowing* this tracking and calculating the *FFT*, we can get the spectrum of the *pitch*. Usually, the *peak* of the modulus of the spectrum is in the interval of 4-8 Hz. At the same time, along with the *fundamental frequency*, the *upper harmonic frequencies* vary too, as well as the *spectral centroid* and the *total energy*. Anyway, the *harmonics* don't vary in parallel with the *fundamental*. Fig. 4.10 show these *features* for a *note* of a *violin*.

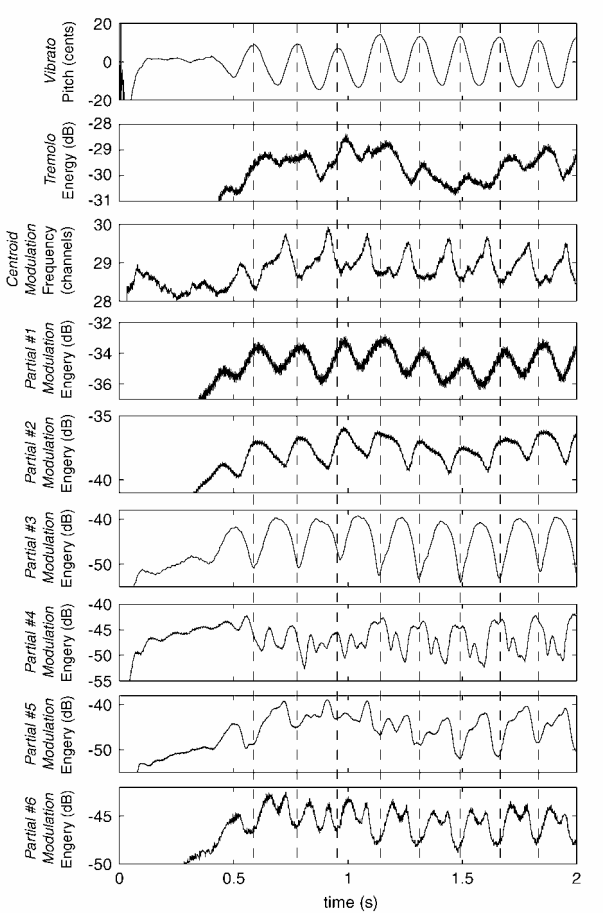


Fig. 4.10: effects of the vibrato on the sound of a violin.

4.2.2.3 The slope of the spectral centroid

In case of *instruments* played with *vibrato*, the *centroid* considerably changes. In Fig. 4.11, the slopes of the *centroid* of three different *instruments* is shown.

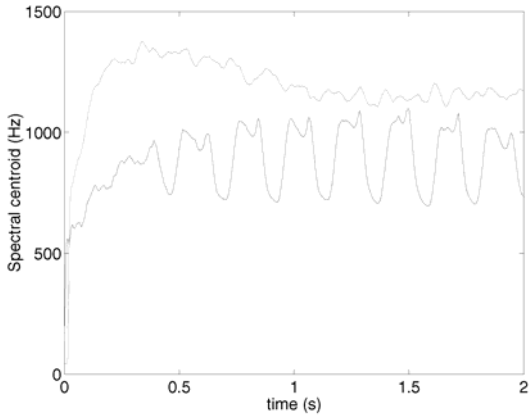


Fig. 4.11: slope of the spectral centroid for (from bottom to top) flute, violin, trumpet.

It can be noticed how the *trumpet* is more "brilliant" with respect to the other *instruments*, as well as the difference between the oscillating slope of the *centroid* of the *violin* (during *vibrato*) and that of the *flute*.

4.2.2.4 Second order characteristics

It's possible to analyze the magnitude of the *variations* of the *spectral intensity* and to calculate the average (in dB) and the variance of these *variations* in relation to the *pitch*. We can also calculate the *variation* of the *amplitude* in relation to the *vibrato* (in dB/cent). Finally, we can calculate the *phase* of the *amplitude modulation* (of the *spectral intensity*) and compare it to that of the *frequency modulation*. Starting from this information, we can calculate the probability that they are in phase.

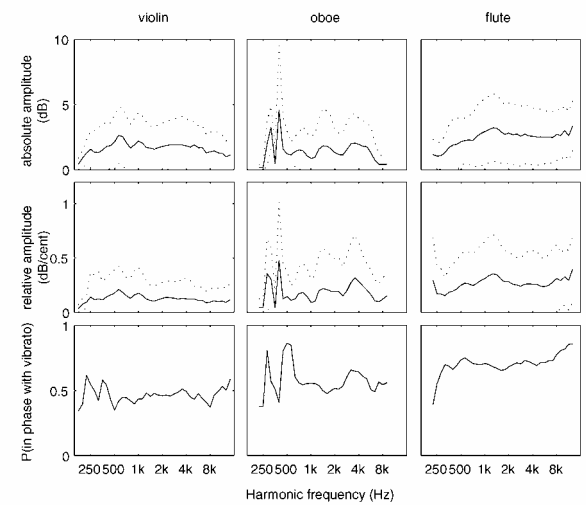


Fig. 4.12: second order characteristics.

Analogous calculations can be made as regards the *spectral centroid*: we can calculate its *variations* with respect to the *frequency*, its *modulation* with respect to the *vibrato* and the probability that they are in phase. Moreover, the same calculation can be made as regards *tremolo*.

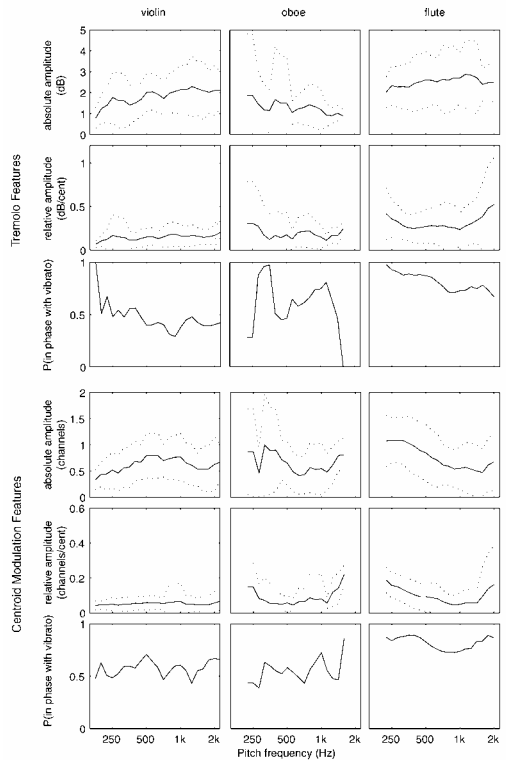


Fig. 4.13: Effect of vibrato and spectral centroid in relation to the pitch of the note.

4.2.3 Characteristics of the attack phase

It has been shown that information derived from the *attack phase* can improve the recognition of the *instrument* that's playing a *note*. In fact, as regards the spectrum, the *attack phase* contains as much information as the *sustain phase*. In spite of this, it's not easy to determine the times the *attack phase* ends and the *sustain phase* begins. The duration of the *attack phase* can be very different from *instrument* to *instrument* and even in different modalities of playing the same *instrument*. The note of a *violin* can have an attack phase of 5 ms, in case of *pizzicato*, and of 500 ms, if *bowed*. The easiest way to measure the *attack phase* consists in determining the *envelope* of the signal in a proper time interval and in detecting the time the envelope exceeds a given value (for example, a percentage of the *sustain value*).

4.2.3.1 Irregularities in the attack phase

By analyzing the spectrum during the *attack phase*, it can be seen that the *partials* don't grow in the same way. Usually, it is sufficient to singularly consider only the first *partials* and the *higher partials* in groups of thirds of *octave* (the sensibility of the *ear* to the *partials* works in this manner). The first *peak* that is found during the *attack* of the *note* is the peak of the *fundamental*. The secondary *peaks* grow only subsequently. So, the *fundamental* can be detected during the *attack phase*. Anyway, a good feature to be exploited is the *rise time* of the *partials* (Fig. 4.14).

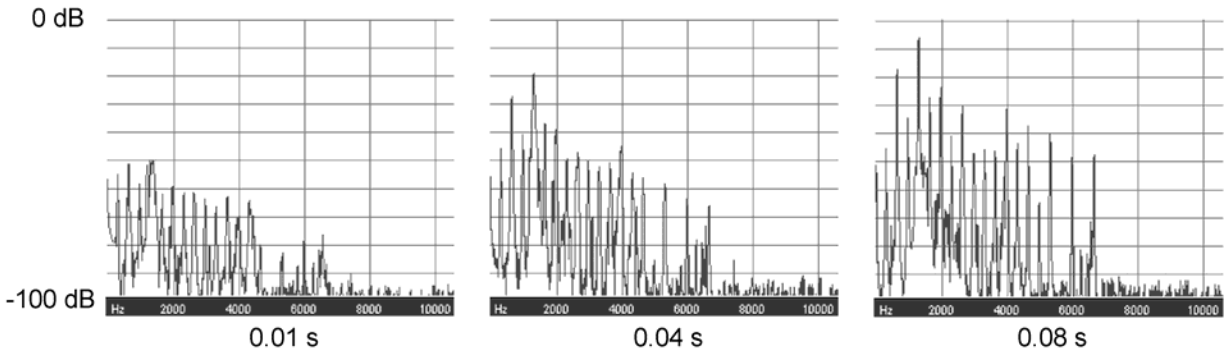


Fig. 4.14: time slope of the partials. We can see the spectrum after 0.1, 0.4 and 0.8 s.

4.3 Pre-processing of the sound signal

For our experiments, we have used *PCM* sound recordings (format "**.wav*", in a PC-Windows environment) at a *sampling frequency* of 44.100 Hz. Sound processing has been accomplished with reference to the way *human ear* works. In order to *classify* a sound signal, with the aim of associating it to a well defined *class* (the *instrument*), we have adopted a *neural network classifier*. The *input feature pattern* to be proposed to this network will be formed by the *partials* inside the spectrum of the *steady state* (*sustain phase*) section of the sound in a recording. Since, during the *sustain phase*, the musical sound signal is nearly *periodical*, it will be formed by a *fundamental* and some *higher harmonics*; so, the *partials* will coincide with the *harmonics*. An easy way to find the *harmonics* inside a spectrum consists in detecting the *peaks* of the spectrum itself. The calculation of the spectrum has been accomplished by means of the well known *FFT* algorithm.

4.3.1 The extraction of peaks

Based on a work of Terhard, Stoll and Seewann, we have defined an *harmonic extraction* procedure. By performing an *N-points FFT* of the *waveform*, in a proper time interval of the *sustain phase*, we get *N amplitudes* of the spectrum. Be L_i the *amplitude* in dB of the i^{th} component of the *FFT* and f_i its frequency; the *harmonic extraction* algorithm can be subdivided in two steps:

1. a component of the *FFT* of the signal is a good candidate for being an *harmonic* component, if it satisfies the following conditions

$$L_i \geq L_{i+1} \qquad \text{and} \qquad L_i > L_{i-1}$$

2. as well as, if it satisfies even the following conditions

$$L_i - L_{i+j} \geq 7\text{dB} \qquad j = -3, -2, +2, +3$$

In case both the above conditions are satisfied, the L_i component can be considered a *peak* in the spectrum of the signal and its *frequency* can be approximately calculated as follows:

$$f_c = f_i + 0.46\,(L_{i+1} - L_{i-1})$$

The value of 7dB adopted in point 2 is empirical; in general, values of 7-9 dB always work well. Practically speaking, it takes into account how much a *peak* is more or less acute. This preprocessing method has proved to be very fast and easy to implement. Anyway, it was not always able to correctly detect the real *harmonics* of the spectrum, since it could select a *peak* that doesn't correspond to an *harmonic*, even though the two points above were satisfied. Then, the algorithm has been improved by ignoring all the *bins* under a proper threshold (to avoid low *frequency* spurious *peaks*) and by defining a minimum value of distance between a *peak* and another (for example, the distance between the *fundamental frequency* and the *frequency* of the *harmonic*

immediately above). In order to improve the *peak extraction* algorithm, the information about the *phase* has been introduced. Both the *phase* and the *modulus* spectrum are discrete; therefore, the *phase* spectrum must be non zero only for *frequencies* where harmonics are present.

4.3.1.1 Improvements of the peak detection algorithm by means of the QFT

One of the drawbacks of the above algorithm is that it is always a bit imprecise and it works well enough only if we add some information, as the starting minimum *frequency* and the above mentioned minimum interval. The starting point for an improvement was based on the consideration that a uniform *frequency resolution* is too exaggerated at the high frequency, while it is insufficient for the low ones. Therefore, if the *fundamental* is at a very low *frequency*, the *FFT* just does not succeed in finding it. The error can be remarkable (approximately 10%). This is why we decided to abandon *FFT* and to bring into the algorithm the *QFT*, where the quantity that remains constant is the ratio between *central frequency* and *resolution*. With Q = 34 and concentrating on the fourth *octave*, the *QFT* algorithm could detect the *fundamental* with high precision (about 99.99%). Once the *fundamental* has been found, the search for the *harmonics* becomes easier and with less errors: the *harmonic peaks* are searched around multiples of the *fundamental*. Anyway, the simple search for the *harmonics* in the multiples of the *fundamental* still generated too many errors. We reduced these errors, by repeatedly updating, each time a new *upper harmonic* was found, the *fundamental frequency*, starting on the information about the frequency of the new *harmonic*. An example of the results supplied by the *harmonic extraction* algorithm is reported in Fig. 4.15.

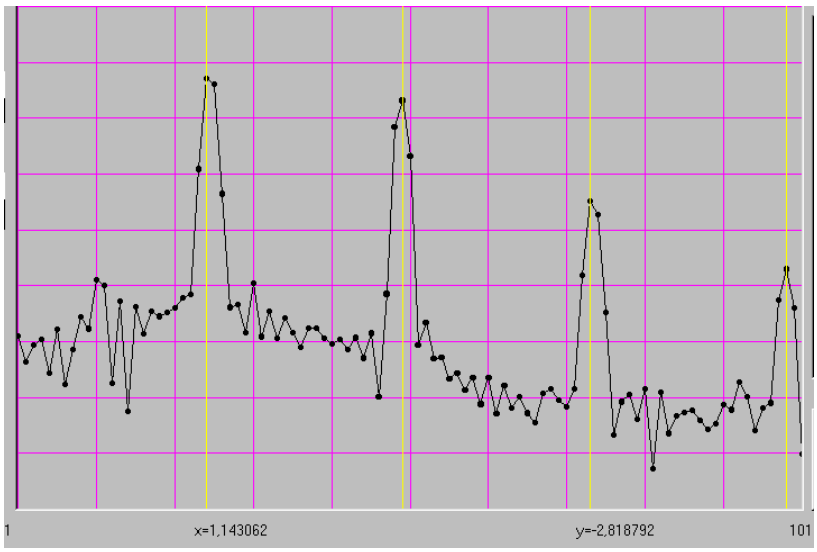


Fig. 4.15: QFT piano, C_4, 4096 samples.

4.3.1.2 Extraction of the harmonic amplitudes

Once the *frequencies* where the *harmonics* are situated have been found, we can go on calculating the *amplitude* of each *harmonic*. This *amplitude* can be expressed in linear form or in dB. It turned out useful to *normalize* all the *amplitudes* of the *harmonics* with respect to the higher *harmonic*. This was the last step of our *pre-processing* phase, since the array of *normalized harmonic amplitudes* has been taken as the *pattern* to give as input to the *classification* system. Fig. 4.16 shows the main form of the application, written in VisualBasic, written in order to extract the *peaks* of the *harmonics* of the *sustain phase* of a signal.

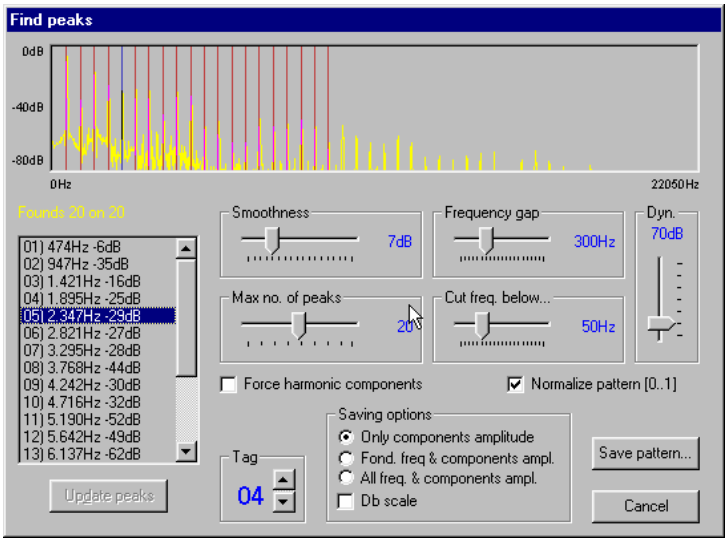


Fig. 4.16: main form for the extraction of the peaks of the harmonics.

4.3.2 The detection of the sustain phase

The end of the *attack phase* and therefore the beginning of the *sustain phase* have been detected by working on the *time envelope* of the sound signal. The *envelope detection* algorithm consisted in under-sampling the original signal and finding the *maximum sample* within each relative *under-sampling period*. This way, we got a piecewise linear function that more ore less accurately approximates the real *envelope* of the signal. Good results have been obtained by taking a sample every 440 samples, at a *sampling frequency* of 44100 Hz. Starting from the obtained *envelope*, the end of the *attack phase* could be selected by exploiting the derivative of the *envelope* itself: once the inclination of the *envelope* goes under a proper value the algorithm stops.

4.4 Experiments and results

We know that *instruments* belonging to different *families* can share similar *features*; so, it could be difficult to distinguish them based on these *features*. Anyway, some criteria can be defined that clearly distinguish some *families* of instruments from other *families*. As an example, the spectral characteristics of the *piano* are similar to those of the *violin*, but as regards the *time characteristics*, clean distinctions can be defined. A *musical instrument* may be played in different modes and styles, e.g. with a different intensity. On the contrary, *notes* played by different *instruments* may have similar properties and there may be therefore confusion between the spectra. Considering a subspace in which we represent *notes* by their *harmonics*, one can identify *decision regions* which are indicative for the considered *instruments*. This *regions* generally result *overlapped* and therefore sharp *boundaries* between them do not exist (Fig. 4.17). Traditional *classifiers* treat *overlaps* as a problem to solve and not as an information to exploit. Therefore, it is obvious that we are in presence of a typical typical *fuzzy classification problem* to be solved by means of a *neuro-fuzzy non-exclusive classifier*, for example a *FMMNN*.

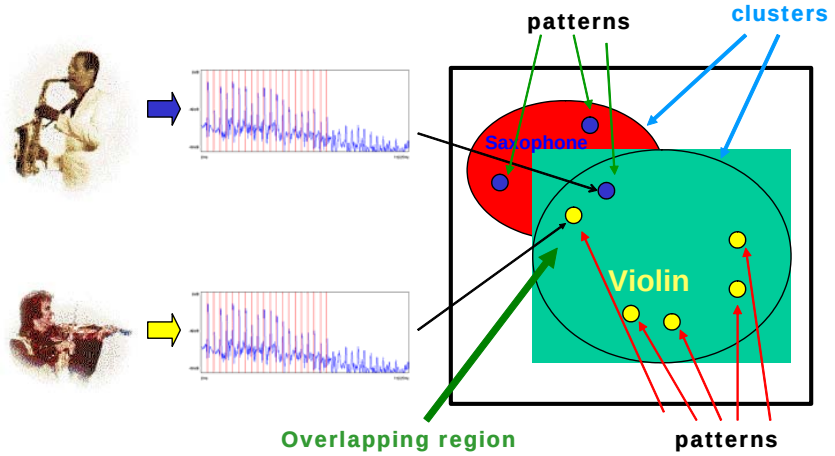


Fig. 4.17: overlapping decision regions in case of violin and saxophone.

The *musical sound sources* explored are depicted in Fig. 4.18:

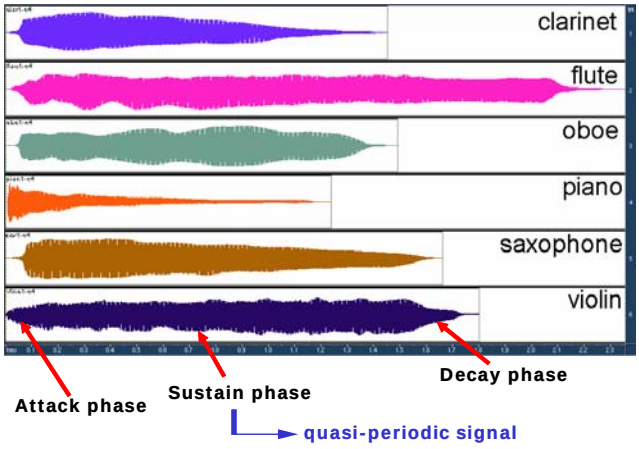


Fig. 4.18: musical sound sources explored.

The spectra of the *sustain phase* of the chosen *instruments* are shown in Fig. 4.19:

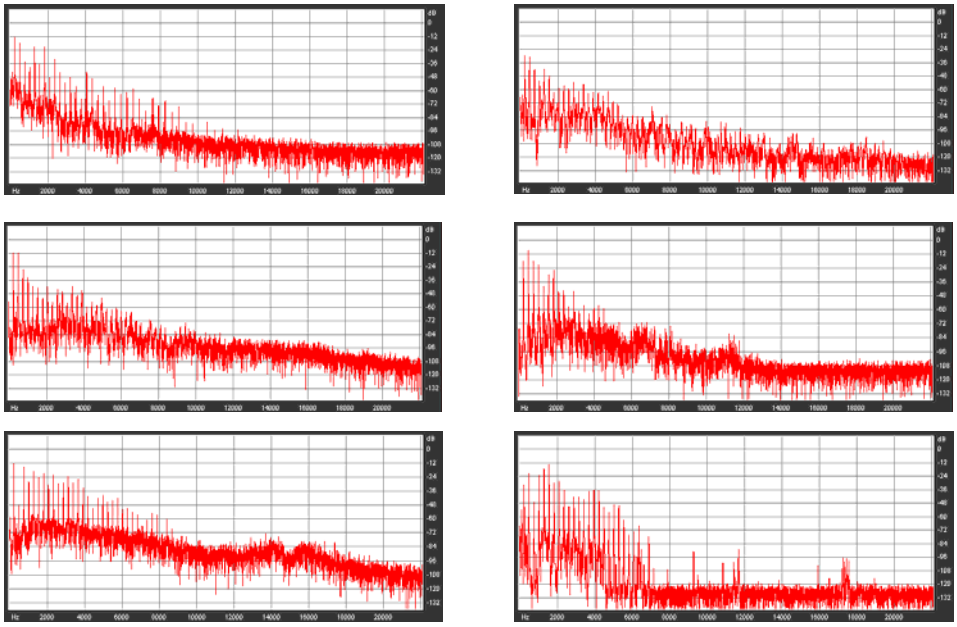


Fig. 4.19: the spectra of the sustain phase of the instruments in Fig. 4.16.

4.4.1 The Classiphy software

It has been developed in the Windows environment and written in the widely used Matlab language. The application receives as input the *normalized patterns* obtained as follows and *classifies* them by exploiting the *fuzzy classifiers* described in the previous chapters (all based on the *Simpson's classifier)*. Fig. 4.20 and Fig. 4.21 show the main forms the user can fill in order to define the parameters that rule the *classification*.

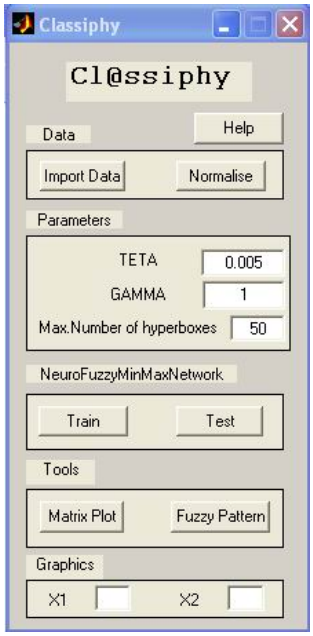


Fig. 4.20: the Classiphy user form

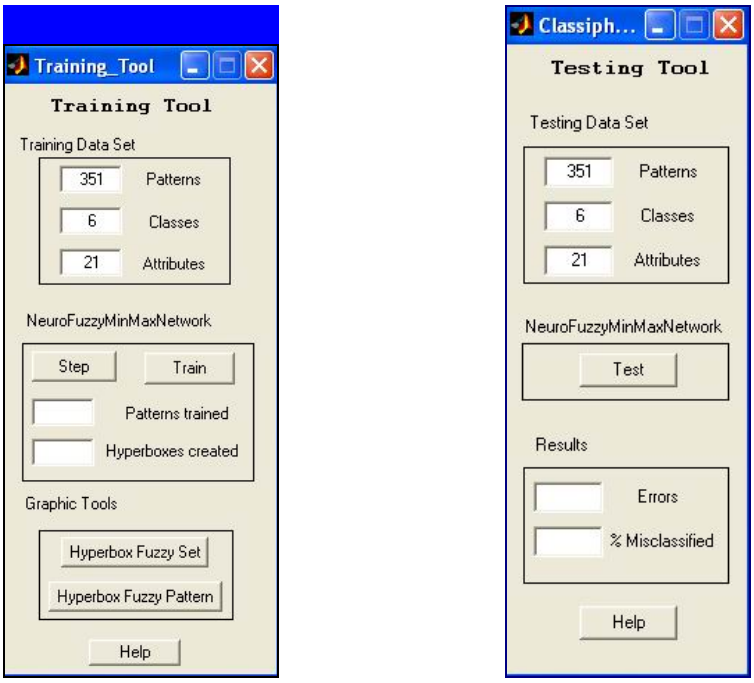


Fig. 4.21: the Classiphy user forms for training and testing

4.4.1.1 Experiment strategies

Although all the work aimed at the *classification* of *musical instruments*, it was necessary to put the *classifiers* implemented in the *Classify* software on a "test bench", in order to validate the software itself (its functioning and effectiveness) with respect to *data sets* universally used and considered as a reference in the comparison among different *classification* strategies. The chosen *data sets* can be downloaded free of charge from the following web sites:

- ftp://ftp.cs.cmu.edu/afs/cs/project/connect/bench
- http://www.ics.uci.edu/~mlearn/MLRepository.html

As already stated, the *classifiers* implemented are all based on the widely used *Simpson's Fuzzy Min-Max Neural Network*:

- *Optimized FMMNN*

  it's the classic *Simpson's classifier*, trained by means of the *optimization algorithm* previously introduced
- *Symmetric Generalized Bell (original)*

  it's the original *classifier* based on the *Generalized Bell* covering function previously introduced. The *centroid* coincides with the barycentre of the *HB*
- *Symmetric Generalized Bell (original)*

  it's the original *classifier* based on the *Generalized Bell* covering function previously introduced. The centroid coincides with the centre of the *HB* (point of crossing of the diagonals)

- *Asymmetric Generalized Bell (original)*
  it's the *classifier* based on the *Asymmetric Generalized Bell* covering function previously introduced.
- *Symmetric Generalized Bell (mean)*
  it's the *classifier* based on the *Generalized Bell* covering function previously introduced. In this case, the value returned by the covering function is an average of the value returned by the *mono-dimensional Generalized Bells*, dimension by dimension. The *centroid* coincides with the barycentre of the *HB*.
- *Symmetric Generalized Bell - CC (mean)*
  same as before, but in this case the *centroid* coincides with the centre of the *HB* (the point of crossing of the diagonals)
- *Asymmetric Generalized Bell (mean)*
  it's the *classifier* based on the *Asymmetric Generalized Bell* covering function previously introduced. In this case, the value returned by the covering function is an average of the value returned by the *mono-dimensional Generalized Bells*, dimension by dimension.
- *Parallel cluster (original)*
  it's the original *parallel classifier* previously introduced. All the *validity indices* have been tested.

4.4.1.2 Iris Data Set

This is perhaps the best known *database* to be found in the *pattern recognition* literature. It has been set up by R.Fisher and it's very interesting from a taxonomic point of view, thanks even to the incredible amount of available results that allow us to compare almost all the *classification* techniques. Practically, it is formed by 150 four dimensional *patterns* (each dimension is an *attribute*) pertaining to 3 *classes* that refer to three types of Iris plant (I.Setosa, I.Versicolor, I.Verginica); so, we have 50 *patterns* for each *class*. The *training set* has been set up by randomly selecting 25 *patterns* for each *class*, for a total of 75 *patterns*. The remaining 75 *patterns*, always 25 for each *class*, are then used for building the *test set*. One of the *classes* (I.Setosa) is *linearly separable* from the other two, but the other 2 are *non linearly separable* from each other. The *attributes* refer to the physical characteristics of the plant and are the following:

- sepal length in cm
- sepal width in cm
- petal length in cm
- petal width in cm

The results obtained on this *data set*, as regards all the *classifiers* tested, are shown in Fig. 4.22 and Fig. 4.23:

IRIS						
Architecture	***Teta Opt.***	***# of HB***	***Errors(Training)***	***%Misclassified(Training)***	***Errors(Test)***	***%Misclassified(Test)***
Optimized FMMNN	0,34	3	4/75	5,333%	9/75	12,000%
Sym.Gbell(original)	0,30	4	3/75	4,000%	15/75	20,000%
Sym.Gbell-CC(original)	0,28	5	4/75	5,333%	12/75	16,000%
Asym.Gbell(original)	0,31	4	4/75	5,333%	7/75	9,333%
Sym.Gbell(mean)	0,30	4	3/75	4,000%	16/75	21,333%
Sym.Gbell-CC(mean)	0,28	5	4/75	5,333%	14/75	18,667%
Asym.Gbell(mean)	0,34	3	4/75	5,333%	9/75	12,000%
	Index	***# of HB***	***Errors(Training)***	***%Misclassified(Training)***	***Errors(Test)***	***%Misclassified(Test)***
ParallelCluster(original)						
	DB	3	5/75	6,667%	10/75	13,333%
	SWDB	3	5/75	6,667%	10/75	13,333%
	EDB	3	5/75	6,667%	10/75	13,333%
	DWDB	9	5/75	6,667%	11/75	14,667%
	DWDB-2	10	7/75	9,333%	11/75	14,667%

Fig. 4.22: classification results on Iris Data Set

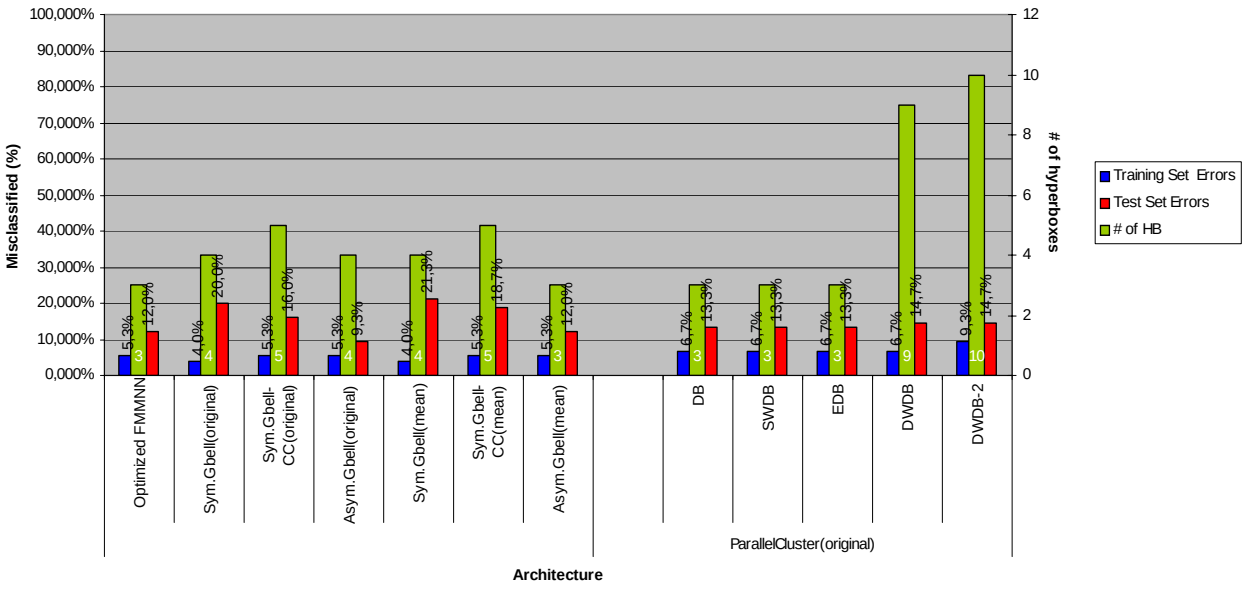


Fig. 4.23: graphic of the classification results obtained on the Iris data set.

As we can see, the best results have been obtained in the case of the *Asymmetric Generalized Bell (original)*. In fact, we have on average the best results on both the *training* and *test sets*. Moreover, the number of *HBs* created is not so high, with respect to the other *classifiers*. This aspect is strictly important, above all if *network complexity* is a characteristic that must be kept low.

4.4.1.3 Wine Data Set

This *data set* is a result of a set of chemical tests made on wines produced in the same Italian region, but that come from three different cultivations (*classes*). Tests have returned the values regarding 13 constitutive elements (*attributes*) that can be found in wines:

1) Alcohol
2) Malic acid
3) Ash
4) Alcalinity of ash
5) Magnesium
6) Total phenols
7) Flavanoids
8) Nonflavanoid phenols
9) Proanthocyanins
10) Color intensity
11) Hue
12) OD280/OD315 of diluted wines
13) Proline

The data set contains 178 *patterns*. In this work, we have considered 90 *patterns* for the *training set* and the remaining 88 *patterns* for the *test set*. Even in this case, a normalization of the data has been mandatory, in order to be compatible with the implemented software. The results obtained on this *data set*, as regards all the *classifiers* tested, are shown in Fig. 4.24 and Fig. 4.25:

WINES						
Architecture	***Teta Opt.***	***# of HB***	***Errors(Training)***	***% Misclassified(Training)***	***Errors(Test)***	***% Misclassified(Test)***
Optimized FMMNN	0,53	3	8/90	8,889%	7/88	7,955%
Sym.Gbell(original)	0,53	3	2/90	2,222%	1/88	1,136%
Sym.Gbell-CC(original)	0,49	4	3/90	3,333%	12/88	13,636%
Asym.Gbell(original)	0,49	4	2/90	2,222%	10/88	11,364%
Sym.Gbell(mean)	0,53	3	2/90	2,222%	4/88	4,545%
Sym.Gbell-CC(mean)	0,49	4	5/90	5,555%	11/88	12,500%
Asym.Gbell(mean)	0,49	4	4/90	4,444%	8/88	9,091%
	Index	***# of HB***	***Errors(Training)***	***% Misclassified(Training)***	***Errors(Test)***	***% Misclassified(Test)***
ParallelCluster						
	DB	3	2/90	2,222%	1/88	1,136%
	SWDB	3	2/90	2,222%	1/88	1,136%
	EDB	3	2/90	2,222%	1/88	1,136%
	DWDB	12	18/90	20,000%	33/88	37,500%
	DWDB-2	19	20/90	22,222%	49/88	55,682%

Fig. 4.24: classification results on Wine Data Set

In this case, the best results have been obtained with the *Symmetric Generalized Bell* (*original*).

This time, good results have been given also by the *parallel clusterizers*. This is the reason why we decided to contemporarily test different kinds of *fuzzy classifiers*. In fact, it's almost as obvious that every *data set* requires the proper *classifier*. On the same *data set*, some classifiers work well and others don't.

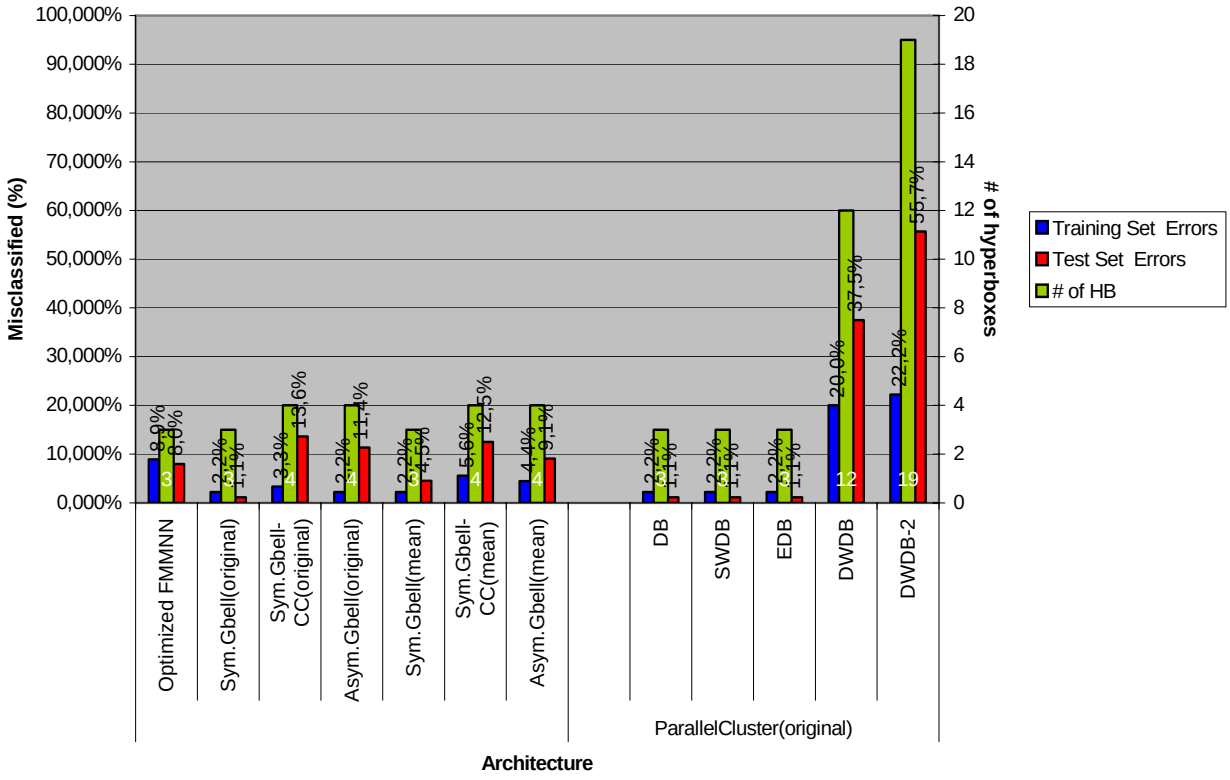


Fig. 4.25: graphic of the classification results obtained on the Wine data set.

4.4.1.4 Glass Data Set

The study and the *classification* of various types of glasses have been motivated by investigations and surveying that are carried out in criminology. Whichever fragment of glass found on the scene of a crime can be taken as evidence (if correctly identified). In this case, the *attributes* are 9:

- RI : refractive index
- Na : Sodium
- Mg : Magnesium
- Al : Aluminum
- Si : Silicon
- K : Potassium
- Ca : Calcium
- Ba : Barium

- Fe : Iron

The training set contains 108 *patterns*, while the test set contains 106 *patterns*. The results obtained on this *data set*, as regards all the *classifiers* tested, are shown in Fig. 4.26 and Fig. 4.27:

GLASS							
Architecture	*Teta Opt.*	*# of HB*	*Errors (Training)*	*%Misclassified (Training)*	*Errors (Test)*	*%Misclassified (Test)*	
Optimized FMMNN	0,13	21	11/108	10,185%	53/106	50,000%	
Sym Gbell (original)	0,27	10	25/108	23,148%	65/106	61,321%	
Sym Gbell-CC (original)	0,13	21	15/108	13,889%	54/106	50,943%	
Asym Gbell (original)	0,25	11	32/108	29,630%	58/106	54,717%	
Sym Gbell (mean)	0,27	10	25/108	23,148%	64/106	60,377%	
Sym Gbell-CC (mean)	0,13	21	15/108	13,889%	49/106	46,226%	
Asym Gbell (mean)	0,25	11	30/108	27,777%	56/106	52,830%	
	Index	*# of HB*	*Errors (Training)*	*%Misclassified (Training)*	*Errors (Test)*	*%Misclassified (Test)*	
Parallel Cluster (original)							
	DB	8	51/108	47,222%	55/106	51,887%	
	SWDB	8	51/108	47,222%	55/106	51,887%	
	EDB	6	52/108	48,148%	52/106	49,057%	
	DWDB	13	42/108	38,889%	54/106	50,943%	
	DWDB-2	31	42/108	38,889%	83/106	78,302%	

Fig. 4.26: classification results on Glass Data Set

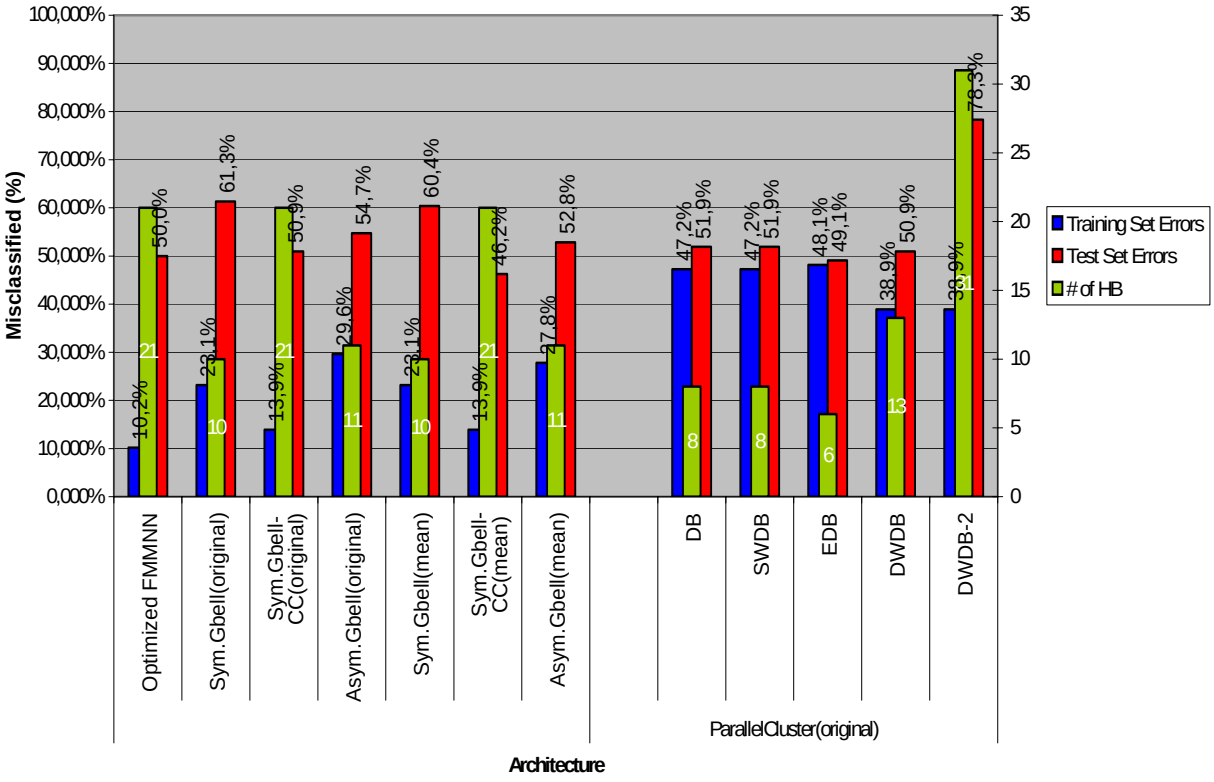


Fig. 4.27: graphic of the results obtained on the Wine data set.

This time, the results are really discouraging. This means that for this application some other approaches must be explored.

4.4.1.5 Musical Instrument Data Set

Finally, we have tested our *classifiers* on the data set for which all this work has been set up for. The *data set* contains 6 *classes*, representatives of 6 *traditional musical instruments* (*clarinet*, *flute*, *oboe*, *piano*, *saxophone* and *violin*). The 21 *attributes* are the first 20 *harmonics* plus the *fundamental*. The *training set* is formed by 351 *patterns*, while the *test set* is formed by 352 *patterns*. The results obtained on this *data set*, as regards all the *classifiers* tested, are shown in Fig. 4.28 and Fig. 4.29:

MUSICAL INSTRUMENTS						
Architecture	***Teta Opt.***	***# of HB***	***Errors(Training)***	***% Misclassified(Training)***	***Errors(Test)***	***% Misclassified(Test)***
Optimized FMMNN	0,09	28	18/351	5,128%	26/352	7,386%
Sym.Gbell(original)	0,09	28	9/351	2,564%	15/352	4,261%
Sym.Gbell-CC(original)	0,11	23	11/351	3,134%	17/352	4,830%
Asym.Gbell(original)	0,08	34	7/351	1,994%	14/352	3,977%
Sym.Gbell(mean)	0,09	28	12/351	3,419%	14/352	3,977%
Sym.Gbell-CC(mean)	0,11	23	11/351	3,134%	18/352	5,114%
Asym.Gbell(mean)	0,11	23	15/351	4,274%	17/352	4,830%
	Index	***# of HB***	***Errors(Training)***	***% Misclassified(Training)***	***Errors(Test)***	***% Misclassified(Test)***
ParallelCluster						
	DB	30	104/351	29,630%	129/352	36,648%
	SWDB	30	104/351	29,630%	129/352	36,648%
	EDB	6	70/351	19,943%	78/352	22,159%
	DWDB	36	35/351	9,972%	44/352	12,500%
	DWDB-2	57	16/351	4,558%	23/352	6,534%

Fig. 4.28: classification results on Musical Instrument Data Set

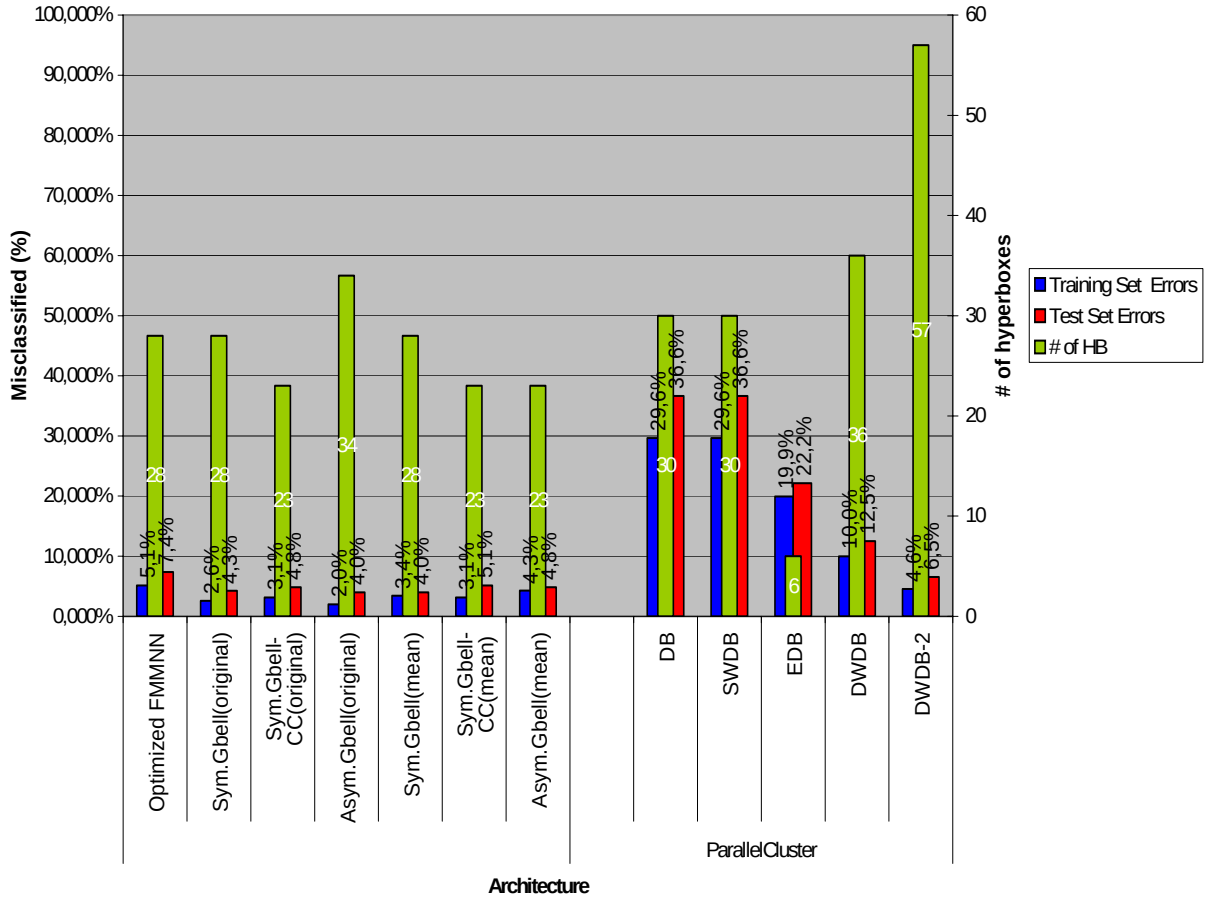


Fig. 4.29: graphic of the results obtained on the Musical Instrument data set.

And finally some comments on the results obtained as regards our core application. Even in this case, best results have been obtained with the *Asymmetric Generalized Bell (original)*. The higher number of generated *HBs* is mainly due to the higher number of *examples* to be learned by the *classifier*.

CHAPTER 5

A COMPARISON AMONG DIFFERENT STATISTICAL CLASSIFIERS: CLASSIFICATION OF THE SIT-TO-STAND HUMAN LOCOMOTION TASK

5.1 Introduction

In the last part of my doctorate work, we have introduced a new method to evaluate the ability to rise from a chair (a particular kind of postural transition) by means of the *sit-to-stand* (*STS*) locomotion task. It essentially was based on the analysis of the *vertical* component of the *acceleration* vector assessed by a home-made device. The aim of this preliminary investigation was to *discriminate* the rising from a chair, starting from different heights, and to *discriminate* between *pathological* and *non pathological subjects*. Human functional ability/disability is evaluated by carrying out some standard simple clinical tests, sometimes performed with only qualitative and/or partially quantitative observations. One of the most used tasks in the human ability/disability evaluation is exactly the *STS* task. In fact, it is of fundamental importance for the quality of life, being connected to the functional independence and commonly considered as the most mechanically demanding functional task in daily activities and essential for gait. One of the main problem that rises when studying human ability/disability is the lack of quantitative methods, despite of the simplicity of the used clinical tests. In order to exactly and easily assess this motor task, quantitative measurements should then be introduced in the evaluation process. Optoelectronic or ultrasound equipments are not suitable for the application we are interested in, because of their costs and encumbrance; they require a lot of markers, that restrict the investigated movement itself and suffer shadowing effect. Instead, *kinematic sensors* are a valid alternative: besides resolving the above-mentioned problems, they add the necessary quantitative measurements to the qualitative observation. Motion analysis performed by means of kinematic sensors is based essentially on the employment of *accelerometer sensors* that directly supply a measurement of motion *acceleration*. Also thanks to recent advances in miniature devices (e.g. MEMS technology), *accelerometers* (*ACs*) are small and light enough to be easily connected to a *body segment*, without hindering the

execution of the motor tasks. *ACs* can be successfully used in human continuous monitoring, such as gait analysis, sit-to-stand and stand-to-sit analysis, postural sway, fall risk. Furthermore, a growing interest for non-invasive patient monitoring has promoted a huge development of the employment of these sensors. *ACs* can be combined together into a single *accelerometric assembly* to be positioned in a single device or used individually as a single *motion sensor* to be affixed in different *body* positions, in order to provide an integrated and practical method for long-term ambulatory monitoring of *human motion*. In any case, little work has been reported on the use of *ACs* in the assessment of the *STS* motor task. Our aim was to investigate the *STS* task with a dedicated device to exploit different *classification* strategies. Therefore, we designed a specific protocol and performed a dedicated kinematic analysis.

5.2 The home made transducer

Padgaonkar introduced an analytical model that aimed at reconstructing the three-dimensional (3-D) position and orientation (*PO*) of a *body segment*, using the signal provided by nine *ACs* for the angle measurements of the *rigid body*. Morris introduced the use of *ACs* in the assessment of *human motion* by means of an analytical model aimed at reconstructing the 3-D *PO* of a *body segment* using the signals provided by six *ACs*. The *ACs* were small and light enough to construct 3-D markers that can be rigidly and easily attached to a *body segment* without interfering with the execution of the physical exercise. The major drawback of accelerometric systems is due to the *acceleration of gravity g*, a fraction of which is measured by *ACs* depending on their orientation. The intrinsic limit in the estimation of the orientation entails a significant limit in the correction of the sensor output to measure *acceleration* and, by applying appropriate integration algorithms, the relevant *PO* vector. It has been shown that errors in the estimation of *PO* are 10° and 0.3 m, respectively, for an observation interval of 1 s; it has been also shown that the predominant cause of error is due to the sensitivity of *ACs* to gravity. There are other *kinematic sensors*, such as *rate-gyroscopes* (*RG*), which have the advantages to be insensitive to the influence of *gravity*. Wun has adopted an assembly comprised of three *RGs*, three *ACs* and *optoelectronic markers* used in a clinical application with the purpose of reconstructing the *acceleration* of the center of mass of the *body*. The applicability of *kinematic sensors* to *motion analysis* depends widely on both the development and refinement of interesting clinical applications as well as on the evolution of microelectronic technology. Recently a growing interest for *non invasive patient monitoring* has promoted the development of portable/wearable *sensors* and systems. *ACs*, inclinometers, and *RG*, alone or combined, have been increasingly used for the realization of portable *sensors* for *acceleration* and *orientation monitoring*. Recently the *kinematic sensors* have also been introduced

in clinical applications where postural parameters are processed for biofeedback restitution to vestibular patients. All of the most commonly used *kinematic sensors* have a relative low cost, if compared to optolectronic/ultrasound solutions. All these solutions, found in literature, did not completely afford the problem of feasibility reconstruction range of the *PO* vector. In this chapter, we describe a device, based on an architecture of three *RG* and three *ACs*, designed and built at the "Istituto Superiore di Sanità" in Rome.

5.2.1 Algorithms

The *transducer* consists of three *mono-axial ACs* (3031-Euro Sensors, USA) and three *sensors* of *angular velocity* (Gyrostar ENC-03J-Murata, Japan), assembled together and relatively oriented according to an orthogonal reference system. Fig. 145 shows the relative orientation of the sensors.

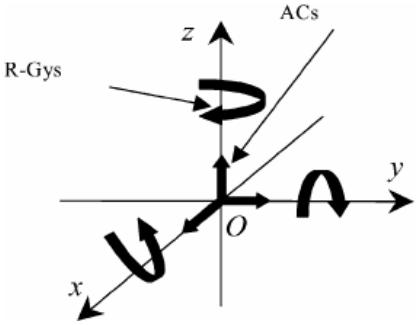


Fig. 5.1: Relative orientation of the sensors: 1, 2, 3 accelerometers; 4, 5, 6 gyrostars.

The actual *body segment angular velocity* vector $(\omega_x, \omega_y, \omega_z)$ is obtained by multiplying the relevant *calibration matrix* by the *gyrostar* tern output vector. The *orientation* of the *segment* is determined by the *orientation matrix* $[R]$, which is calculated by solving the following matrix differential equation:

$$R^{-1} * \frac{dR}{dt} = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \tag{5.1}$$

and where the three *orientation* angles (expressed in nautical angles) are obtained using:

$$R = \begin{bmatrix} \cos(\phi)\cos(\theta) & \cos(\phi)\sin(\theta)\sin(\psi) - \sin(\phi)\cos(\psi) & \cos(\phi)\sin(\theta)\cos(\psi) + \sin(\phi)\sin(\psi) \\ \sin(\phi)\sin(\theta) & \sin(\psi)\sin(\theta)\sin(\phi) - \cos(\phi)\cos(\psi) & \sin(\phi)\sin(\theta)\sin(\psi) - \cos(\phi)\sin(\psi) \\ \sin(\theta)\sin(\phi) & \cos(\theta)\sin(\psi) & \cos(\theta)\cos(\psi) \end{bmatrix} \tag{5.2}$$

It has been found convenient to express the *segment orientation* in nautical angles: θ is the *pitch* angle, ϕ is the *roll* angle and ψ is the *yaw* angle. The real instantaneous linear *acceleration* vector is obtained by:

$$\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = [R] \begin{bmatrix} a'_x \\ a'_y \\ a'_z \end{bmatrix} - \mathbf{g} \tag{5.3}$$

where a'_x, a'_y and a'_z are the *acceleration* vectors in the reference system solid with the device. The instantaneous motor *acceleration* vector is then double integrated in order to reconstruct the *trajectory*. All the algorithms were developed and tested in the Matlab environment. In particular, the algorithm used to solve (5.1) and (5.2) was developed with the Simulink Tool box; the differential equation system, for the calculation of the *orientation matrix*, was solved by means of the ordinary differential equation system solver (ODE). Another algorithm has been included, with the aim of minimizing the thermal drift of the *gyrostars*. It is a typical zeroing algorithm, based on a specific *Thermal sensor* (lm335-National Semiconductor, USA) and a tuning table compiled at different temperatures using a controlled oven to correct the offset-drift during calibration.

5.2.2 The architecture of the device

Fig. 5.2 shows the architecture of the *wearable device*. It is composed of two separate units: a *sensor* unit and a *powering*/*connecting* unit. The latter contains the battery and the circuitry for communication at a 433 MHz \FM radio frequency.

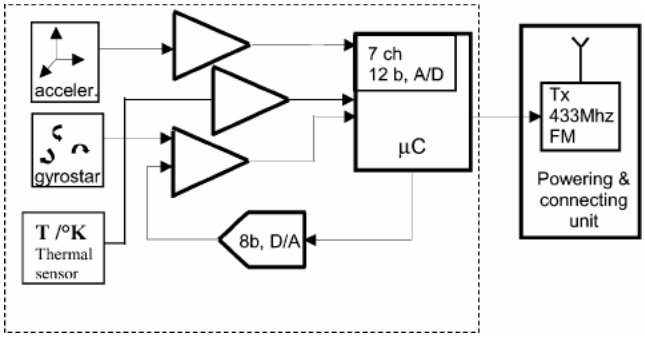


Fig. 5.2: Architecture of the wearable device..

The *sensor* unit is very small (4×5×2 cm) and light (250 g); this is why it has revealed itself to be a very practical tool as regards the applications of interest. Furthermore, the duration of the tests required by this *sensor* during specific clinical applications is short enough not to fatigue the *patient*. The circuitry of the wearable device is formed by 7 *signal conditioning chains* (3 for the *ACs*, 3 for the *gyrostars* and 1 for the *thermal sensor*), a *microcontroller* equipped with a 12 bit *A/D converter* and an 8 bit *D/A converter*. Each *conditioning chain* contains an *amplifier* (G = 10), a *Voltage Controlled Voltage Source* (Sallen-Key cells) and a *Butterworth second-order Low Pass Filter* with a cutoff frequency of 14 Hz; this *frequency* was fixed after some simulations made in P-Spice. Fig. 5.3 illustrates the placement and routing of the components that constitute the device.

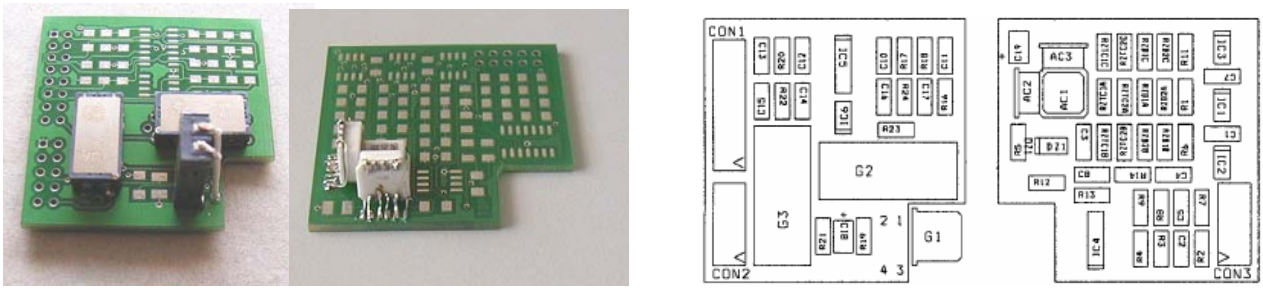


Fig. 5.3: Optimal components arrangement. AC1, AC2, AC3 are ACs, G1, G2, G3 are RGs.

The *sensor* circuitry has been mounted on two separate boards: one for the *gyrostar tern* and the other for the *accelerometer tern*. The circuitry is assembled with surface mounting technology, taking special care with the positioning of the *sensors* and their stability, so as to ensure the proper functioning and reliability of the device. Fig. 5.4 is a picture of the wearable device.

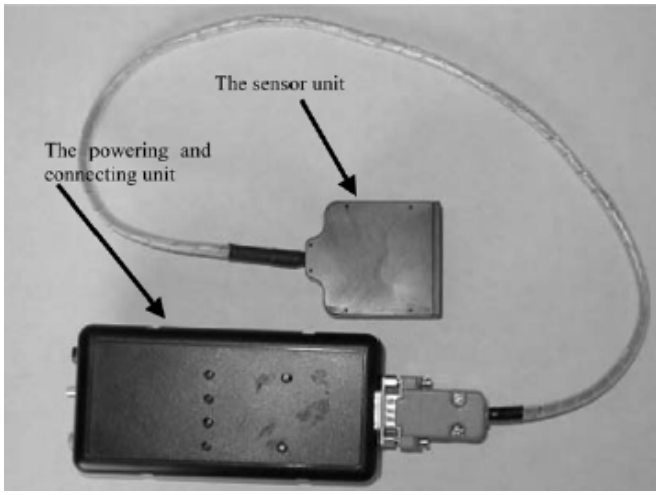


Fig. 5.4: The complete device where the sensor unit and the powering and connecting unit are highlighted.

5.2.3 Testing equipment, calibration and performance evaluation

A dedicated equipment was developed in order to impose opportune motion laws for calibration and for the bench test. The core of the system included a DMC-1410 controller and a step-by-step motor with encoder (Galil, Rocklin, CA). These elements were integrated together with

- a power supply ;
- a PC with ISA bus;
- a communication disk and Servo Design Software WSDK (Galil, Rocklin, CA);
- an Amplifier AMP-1460.

The complete testing architecture is shown in Fig. 5.5. The mechanical case and interface have been designed in the Laboratories at the “Istituto Superiore di Sanità” in Rome; the complete realized equipment is shown in Fig. 5.6.

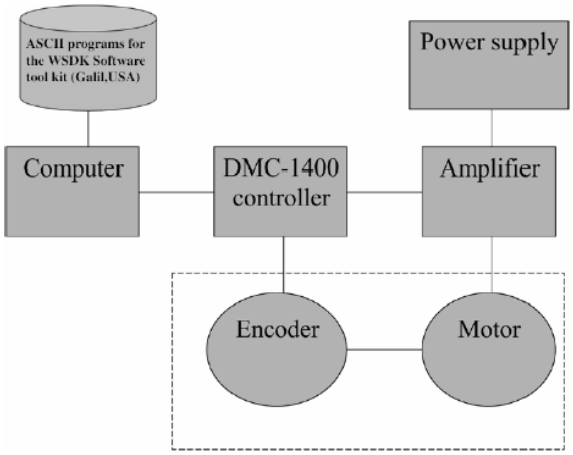


Fig. 5.5: The complete architecture of the equipment for the calibration and generation of motion laws based on the DMC-1410 component (Galil, Rocklin, CA).

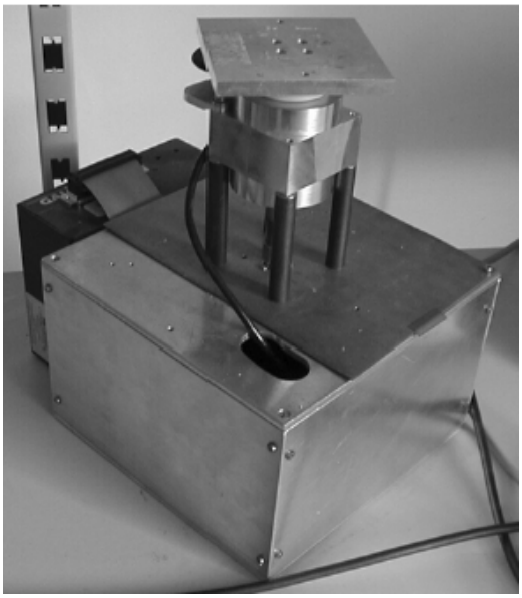


Fig. 5.6: The complete equipment with the mechanical case and interface developed in our laboratory.

The instrument was calibrated in two phases: one static, the other dynamic. The calibration of the *accelerometer tern* was carried out during the static phase. The *sensor unit* was subjected to different *g* vectors by positioning each of its six faces on a horizontal plane; the output signals were averaged over a 6 s time interval. The dynamic calibration was done with the Testing Equipment. A plate, on which the device was affixed, was rotated by means of the Testing equipment, in order to impose known rotational time laws with the necessary accuracy. Angular velocities ranging from 10°/s to 60°/s in both directions were imposed for each of the three orthogonal axes. In both cases the calibration matrixes were computed by the least squares method. $[C_A]$ was the calibration matrix for the *ACs* channels obtained by means of (5.4), where $[L_A]$ was the matrix of the imposed quantities. These were obtained when the sensor unit was subjected to different *g* vectors, by positioning each of its six faces on a horizontal plane. $[S_A]$ was the matrix of the quantities assessed

by the *sensor* unit (in the specific case, *g*).

$$[C_A] = [L_A][S_A]^T \left([S_A][S_A]^T\right)-1 \tag{5.4}$$

$[C_{RG}]$ was the matrix for the *RGs* channels, obtained by means of (5.5), where $[L_{RG}]$ was the matrix of the imposed quantities. These were the *angular velocities* imposed by means of the dedicated *testing equipment*, when the sensor unit was rotated with different *angular velocities* and with six different *orientations*. $[S_{RG}]$ was the matrix of the quantities assessed by the *sensor* unit (in the specific case, the different angular velocities).

$$[C_{RG}] = [L_{RG}][S_{RG}]^T \left([S_{RG}][S_{RG}][S_{RG}]^T\right)-1 \tag{5.5}$$

Rotational and translational laws were then imposed in a bench test, in the typical ranges relative to the locomotor tasks (0.1, 1 Hz frequency; 0°, 60° rotational amplitude; 0.1, 0.5 m translational amplitude). The *sensor* device was placed at position *L5* of the back of a volunteer, taking as a reference, for the estimation of the trunk flexion, the subject navel. The device was firmly fixed by means of a belt with a rigid pocket; the *powering/connecting* unit was fixed to the trousers at the right side. This position is very close to the *body* center of mass and it was chosen to test protocols, suitable for future comparison to data obtained from a force plate. The tasks chosen, meaningful as regarded our purposes, were the following:

- stand-to-sit;
- gait-initiation, asking the subject to perform only a step, starting and ending with the feet aligned;
- sit-to-stand.

The trajectory comparison was performed in a time interval of 4 seconds, in the sagitall plane, in order to test the feasibility in reconstructing the chosen *locomotory tasks* in the plane of interest. We also tested the system during the effective time of a duration of an *STS*. For the determination of the start and end positions of the trajectory of the *STS*, two different dedicated algorithms were used.

5.3 Signal processing

STS can be automatically divided into phases and *classified* by identifying the sequence of two different *postures*; the *sitting* one and the *standing* one.

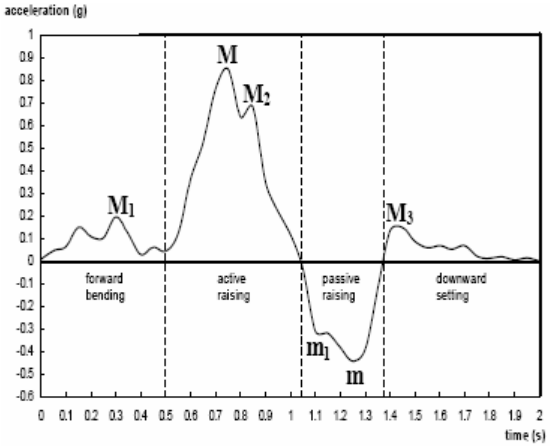


Fig. 5.7: The phases of the sit-to-stand a_z acceleration signal.

Fig. 5.7 shows the main phases of a *STS* investigation, as given by the vertical component of acceleration (a_z) supplied by the signal processing operated by the *microcontroller* inside the device. The block diagram of the pre-processing algorithm is depicted in Fig. 5.8. We can see on the upper-left side the sub-circuit that houses the three *RGs*, while on the upper-right side the sub-circuit that houses the three *ACs*. The signals provided by these transducers are, respectively, the *angular velocity* ω_S and the *linear acceleration* $\mathbf{a}_S$. For the purpose of our specific investigation, these signals are processed according to the flow chart in Fig. 5.8.

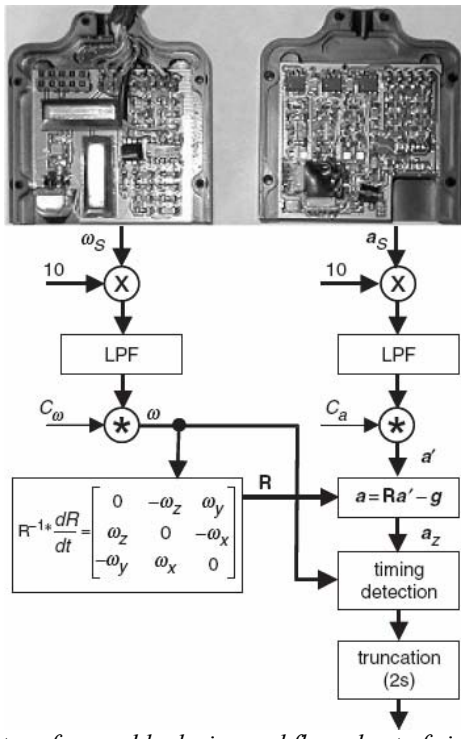


Fig. 5.8: Circuitry of wearable device and flow chart of signal processing.

The *filters* that appear in this flow chart are part of the *conditioning chain*. C_ω and C_a are the *calibration matrixes* for the sensors, while R is the *calibration matrix* obtained from the components of the *angular velocity* (see above). Using matrix R and the acceleration a' in the mobile reference system solid to the device, we obtain the absolute acceleration $\boldsymbol{a}$ by means of the formula (5.3), where $\boldsymbol{g}$ is the gravity acceleration. The vertical component a_z of the *acceleration* (same direction as g) is taken as the *waveform* to be investigated in *time* and *frequency domain*s, through of the following steps:

1. detection of onset and offset times (and so its time duration) of the waveform;
2. truncation of the waveform to fix it at 2s;
3. detection of the absolute maximum and minimum points
4. evaluation of the spectrum of the waveform by means of the well-known FFT algorithm.

5.4 Protocol of investigation

In our work, we have introduced a novel method for the *classification* of the *STS* task, based on analyses in the *time domain*, as well as in the *frequency domain*, of the *acceleration* assessed by the above described device. This method has been tested having in mind two main preliminary purposes:

1. the *discrimination* between different kind of *STS* heights
2. the *discrimination* between *pathological* and *non-pathological subjects*.

using as a starting point the quantities supplied by the measurement device. With these intents, the *STS* of five *healthy subjects* and three *Parkinsonian subjects* (at first stage of pathology) was recorded. The transducer was affixed at *L5* level of the trunk (standard clinical position). The *acceleration* waveform a_z was recorded with a sample period of 50 ms, during 4 s trials, in three different conditions:

A. chair height fixed to 90% of the feet-to-knee distance;
B. chair height fixed to 100% of the feet-to-knee distance;
C. chair height fixed to 110% of the feet-to-knee distance.

Three trials were performed for each *subject* and in each condition; the order of the trials was randomized. The *data set* to be exploited has been built by registering, for each trial, the value and temporal position (M_i, m_i) of the most meaningful points corresponding to the phases of a_z and by analyzing in the *frequency domain* the *waveform* of a_z. As a consequence, the *features* to be exploited in the *classification* phase were the coordinates of those meaningful points of a_z, together with the frequency components of a_z.

5.4.1 Investigation in the time domain

It is based on the value and the time position of the *absolute maximum* and *minimum* of the waveform in Fig. 5.7 (*M*, *m*).

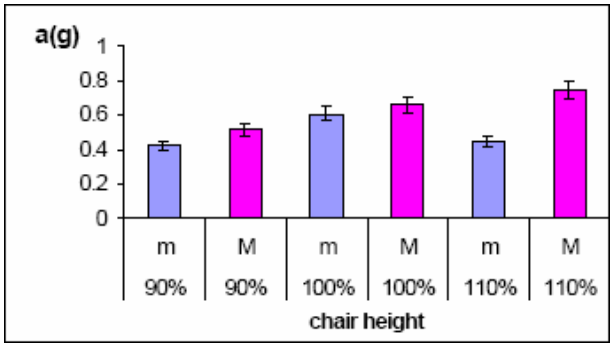


Fig. 5.9:Maximum and minimum acceleration peak (healthy subjects): mean value and standard deviation.

In Fig. 5.9, we report, for the three different chair heights, the minimum and maximum acceleration peak absolute mean values (in g), assessed over all the trials, while in Fig. 5.10 we report the minimum and maximum acceleration time mean values (in seconds).

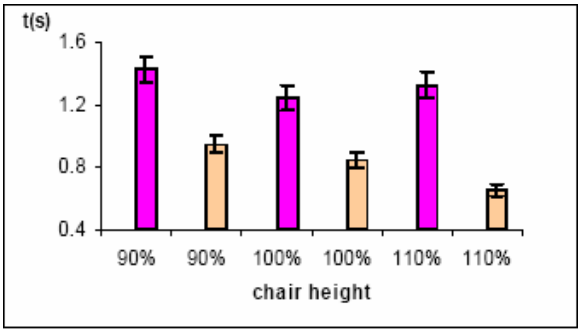


Fig. 5.10: Max and min acceleration peak time (healthy subjects): mean value and standard deviation.

Fig. 5.11 and 5.12 reports the same results for *pathological subjects*.

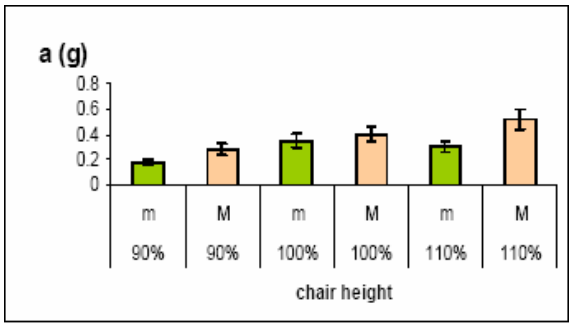


Fig. 5.11: Maximum and minimum acceleration peak (pathological): mean value and standard deviation.

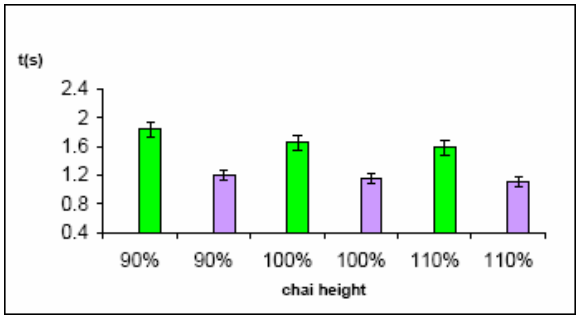


Fig. 5.12: Maximum and minimum acceleration peak time (pathological): mean value and standard deviation.

As we can observe in Fig. 5.9 and 5.10, both time and peak mean values are adequate to *discriminate* between the three different chair heights. Figures 5.11 and 5.12 report the same results for the *pathological subjects*, showing that the distributions are completely different. These preliminary results prove the power of our procedure as regards the determination of the principal characteristics of the *STS* locomotory task. They also prove the usefulness of the mean *acceleration* peak and mean timing values for *discriminating* between the three different chair heights for the *non pathological subjects*; furthermore, the same can be confirmed as regards the separation of the

space of investigation into *pathological* and *non pathological* subspaces. Therefore, this methodology has shown itself to be a promising starting point in the development of a *classification* procedure to be used in the *discrimination* of many locomotory acts as well as a powerful diagnostic tool for the identification of *kinematic pathologies* in their first stage. As previously foretold, the next step has been the widening of the set of parameters involved in the *classification*, with the inclusion of other important phases of the specific act (besides m_i, M_i). Obviously, a boost to the power of these investigations has be given by the derivation of the above mentioned parameters from *frequency domain*. Moreover, since the beginning of this study, it's been clear that the most interesting and promising improvement would have come from the exploitation of automatic *classification* methodologies based on adaptive systems, such as *Neural Networks*. Both *supervised* and *unsupervised* learning algorithms are well suited for this application. After all, *neural classifiers* can be exploited in order to process huge amounts of data and to develop knowledge bases for the identification of particular pathologies – such as Parkinson disease – in their early stage.

5.4.2 Investigation in the frequency domain

In Fig. 5.13, we report the mean values and the standard deviations, assessed over all the trials, of the *DC component*, the *fundamental* and the first three *harmonics* (*energy* is mostly distributed between these five frequency components) of a_z, both for *healthy* and *diseased subjects* rising from three different chair heights.

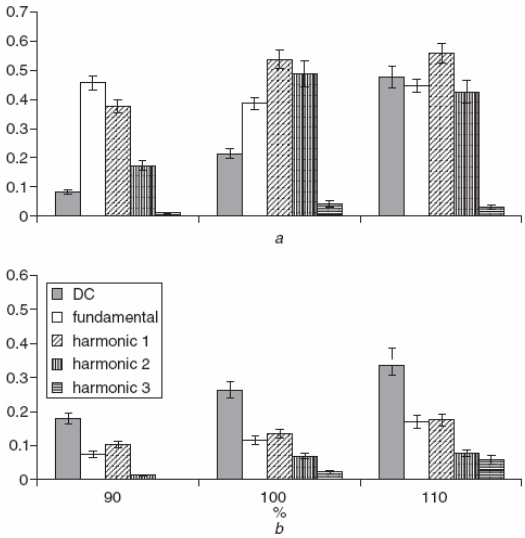


Fig. 5.13: Mean values and standard deviations of DC component, fundamental and first three harmonics for three different chair heights and for healthy and diseased subjects; a Healthy subjects; b Diseased subjects.

Our method is based on the fact that we can easily distinguish between different *STS* typologies, regardless of the subject under test, as well as between people with different states of health (bad,

good), by simply observing that every situation is labeled by a particular *energy* distribution over the *frequency components*. In particular, for *healthy subjects*, it can be noticed that:

A. rising from 90%
   - the *DC component* has a very low amplitude
   - most of the *energy* is concentrated on the *fundamental*
   - energy falls for the second and third *harmonic*

B. rising from 100%
   - the *DC component* has a generally more noticeable amplitude
   - most of the *energy* is concentrated on the first *harmonic*
   - little *energy* can be sometimes found on the third/fourth *harmonic*

C. rising from 110%
   - often, the *DC component* has a huge amplitude
   - the maximum *energy* is concentrated on the first *harmonic*
   - often, little *energy* can be found on the fourth *harmonic*

On the other hand, in case of a *subject* suffering from *Parkinson disease*, we can notice a deeply different *energy* distribution among the *frequency components*:

- the *DC component* is always surprisingly predominant
- with reference to the three cases A, B and C, the *energy* distribution among the *fundamental* and the *harmonics* noticeably changes with respect to the case of *healthy* people.

As previously told, the proposed methodology has been improved by exploiting automatic *classification* strategies based on adaptive systems, such as *neural networks*.

5.5 Discrimination between human functional ability/disability by means of Automatic Classification in the frequency domain

The last argument tackeld in my doctorate work, as regards the application of the *classification* approach to medical situations, has involved the development of a new method for discovering well defined *incipient pathologies* in human beings, by means of the analysis of the *STS locomotion task*. It is based on the *frequency analysis* of *acceleration* measurements supplied by the above described home made wearable device and on the exploitation of some of the most effective *classification* strategies at this time, starting from quantitative observations of the *STS* locomotion task. The quantitative measurements needed by the evaluation process have been carried out by means of the device. Even in this case, the signal to be investigated is the *acceleration* vector component a_z (same direction as g). The main *features* to be exploited in the *classification* of the *STS* are the *frequency components* of a_z (*DC* and the first four *harmonics*), in order to accomplish the *discrimination* between *pathological* and *non-pathological subjects* and between different chair heights. Unlike the previous attempts, this time we perform the *classification* process (exploiting the same *features*), by making use of state-of-the-art *classifiers*, having at our disposal even greatly larger *training sets* than the one previously used. Signal conditioning and calibration needed to obtain the actual *acceleration* are the same as before.

5.5.1 Pre-processing

The purpose of the *pre-processing* phase consisted in the evaluation of the *acceleration* component a_z and in the consequent investigation of the same *waveform* in the frequency domain. The spectrum of the *waveform* as been derived by means of the well known *FFT* algorithm. Even in this case, the device was affixed at *L5* level of the trunk and a_z recorded in the same conditions, starting from three different chair heights (*CH*): 90%, 100% and 110% of the feet-to-knee distance. The target of the investigation consisted in the *automatic discrimination* between *healthy* and *diseased subjects*, using the *frequency components* of a_z as a starting point. We analyzed the *STS* of 50 *healthy subjects* and 12 *diseased subjects* (third level in the Tinetti scale). So, for the *classification* phase, we could have at our disposal 786 examples (1x3 *CH* for each *diseased subject*, 5x3 *CH* for each *healthy subject*). In Fig. 5.7, the typical a_z *waveform* of an *healthy subject* - CH 100% - was reproduced. In Fig. 5.14, the a_z of three meaningful *diseased subjects* - *CH* 100% - are reproduced. An extreme variability in the shape of the *waveforms* can be noticed, compared with a substantial uniformity in case of the *healthy subjects*.

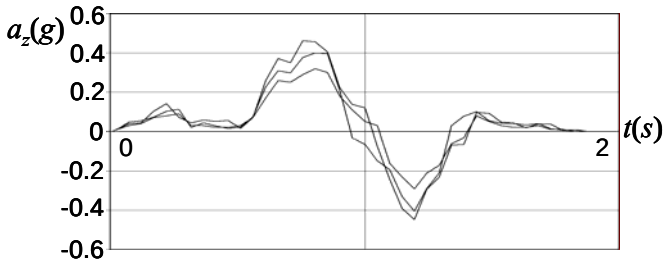


Fig. 5.14: a_z waveforms of three diseased subjects – CH 100%.

Anyway, the difference, in terms of *waveform*, between *healthy* and *diseased subject*, is clearly visible. The irregularities in the shape of the *waveforms* returned in case of the investigation on *diseased subjects* didn't allow us to extract time features (as *MIN*, *MAX*) that could be successfully meaningful in order to build proper *input patterns* for *automatic classifiers*. This is the reason why we turned to the *frequency domain* with the aim of deducing some information that could serve the purpose. In Fig. 5.15, the mean values of the first 21 *frequency components* of the spectrum - CH 100% - are reported.

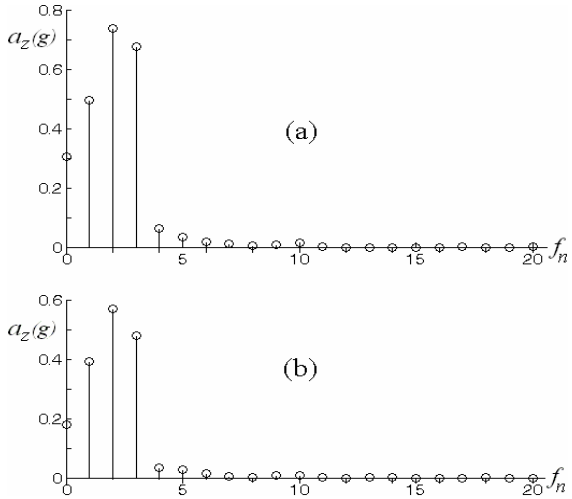


Fig. 5.15: mean values of a_z frequency components: (a) healthy subjects – CH 100%; (b) diseased subjects – CH 100%.

We can notice that the mean values given by *healthy subjects* notably differ from those given by *diseased subjects*. This is why, to *discriminate* between the two *classes* of *subjects*, we decided to exploit different *classification* methodologies.

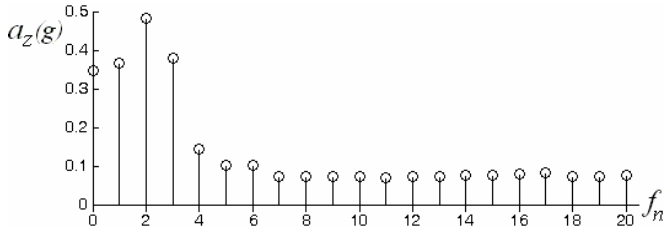


Fig. 5.16: mean values of a_z frequency components: diseased subjects – CH 110%.

Always from Fig. 5.15, it's evident that *energy* is in both cases spread mostly over the first six *frequency components*. Actually, this is not true for *diseased subjects* when they faced the CH

110% trial (Fig. 5.16), while what stated above is still valid for *healthy subjects*. This prompts us to exploit all the 21 *frequency components*, even because this boosts the *discrimination* power of the adopted *automatic classifier*.

5.5.2 Automatic Classification

The *data set* was a result of the analysis of the *acceleration waveform* a_z in the *frequency domain*, as previously described, that is, the *features* exploited in the *classification* phase were the *frequency components* of this *waveform*, but with the addition of the information about the *CH*. The *data set* has been subdivided into three sub-groups, each containing 33% of the given *examples*. The *training set* was formed by joining two of those sub-groups, in all the possible combinations, while the *test set* was each time just the remaining sub-group. Errors were mediated over all the combinations. *Classification* tests have been carried out with the help of *WEKA (Weikato university Environment for Knowledge Analysis)*, a software tool widely used in the academic world. The opening form of this useful and versatile application is depicted in Fig. 5.17.

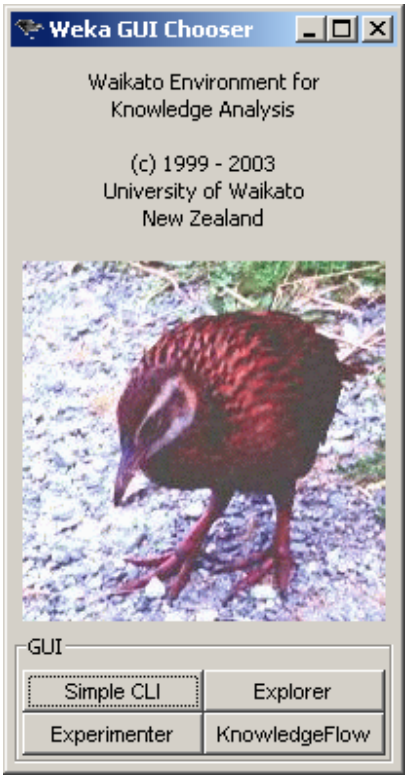


Fig. 5.17: WEKA opening form.

In Fig. 5.18 the *multlayer perceptron* generated by *WEKA* is depicted. On the left side of the diagram, the *input frequency features* are clearly visible. On the right side, instead, the three output of the *network*, relative to the three chair heights to *discriminate* among, are depicted.

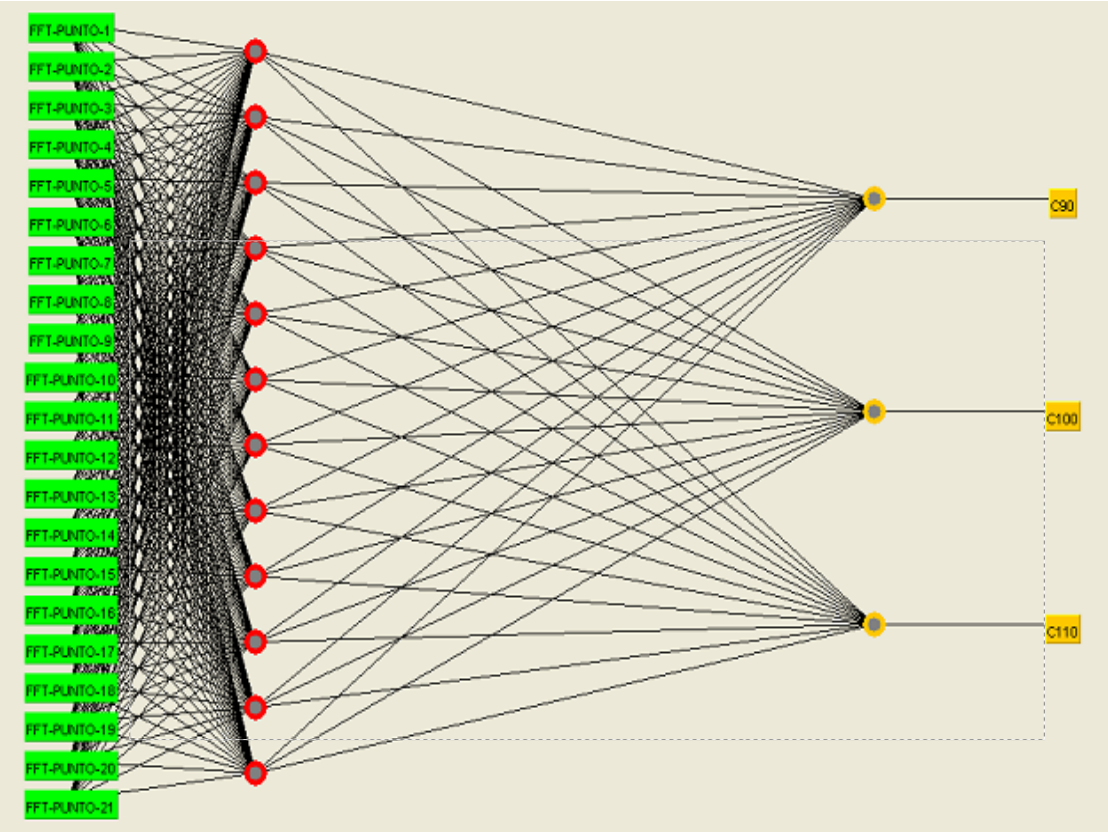


Fig. 5.18: the multlayer perceptron generated by WEKA.

In Fig. 5.19, we can see the typical view made available by *WEKA* for the entering of the chosen values for the parameters that influence the behaviour of an *SVM classifier*.

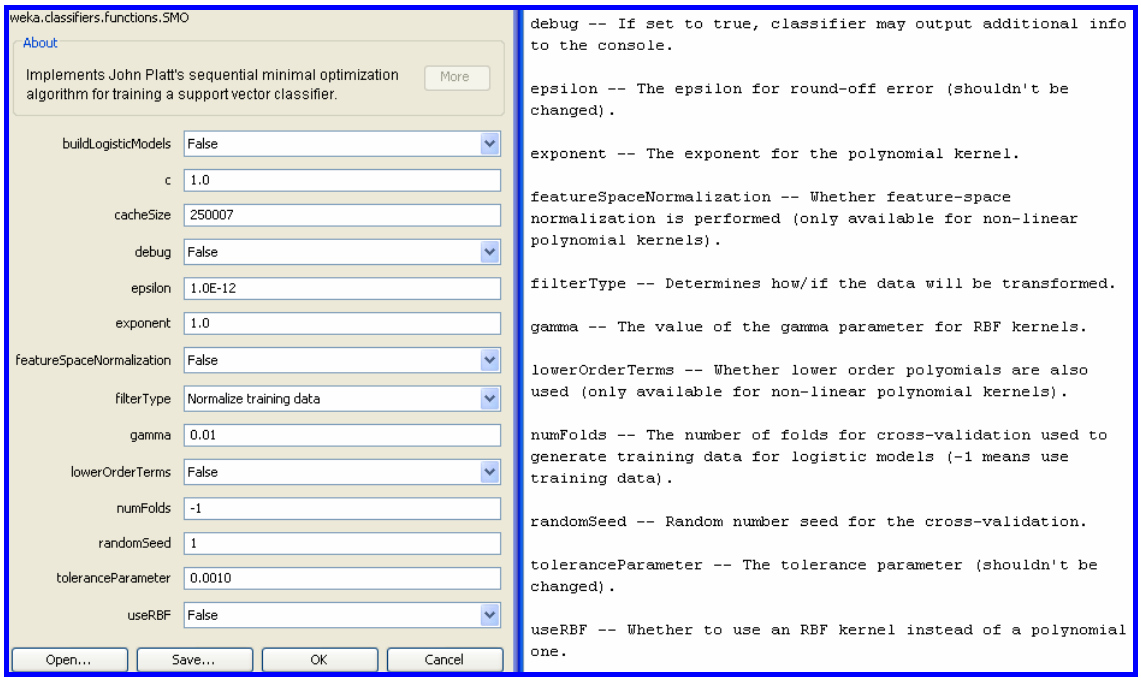


Fig. 5.19: WEKA form for the definirion of an SVM classifier.

We chose to exploit the following *classification* strategies: *Naïve Bayes*, *Bayes Net*, *Multilayer*

Perceptron and *Support Vector Machine* (SVM). The results of our tests are summarized in Tab. 2.2.

classifier	training	test
NaiveBayes	16.444%	29.933%
Bayes Net	0.241%	0.2667%
Multililayer P	0.222%	0.533%
SVM	1.333%	5.213%

Tab.2.2: error percentages on training set and test sets.

We tested four *classifiers* on the *data set* returned by the *pre-processing* phase. We were not surprised by these results. Infact, the poor robustness of the *Naïve Bayes classifier* is confirmed (in fact, it ignores the conditional dependence that may exist among the *input features*). On the contrary, we could appreciate the well known *discriminative* power of the *Bayes Net* and of the *Multilayer Perceptron*. Anyway, the poor results obtained with the *SVM* are substantially due to the choice of a *linear kernel* in *discriminating* between two *non linearly separable classes*. The above results tell us that we are on the right track. In future, we intend to exploit more complete *data sets* as well as to *discriminate* between different Tinetti levels. Moreover, we will make sure of refining the usage of the *classifiers* involved, above all as regards the *SVM*.

BIBLIOGRAPHY

Neural Network fundamentals

Kaykin S.

"Neural Networks, a comprensive foundation", second edition, Prentice Hall, (1991)

Hassoun, M. H.

"Fundamentals of Artificial Neural Networks". Cambridge, MA: MIT, (1996)

R. P. Lippmann

"An introduction to computing with neural nets", IEEE ASSP Mag., pp. 4-22, Apr. 1987.

G. Martinelli

"Fondamenti di Reti Neurali",Ed. Siderea, Roma, 1996

D. Rumelhart, E. Hinton, and J. Williams

"Learning internal representation by error propagation" , in Parallel Distributed Process-ing, vol. 1, Rumelhart and McClelland, Eds. Cambridge, MA: M.I.T. Press, 1986, pp. 318-364.

Pattern recognition

K. Fukunaga

"Introduction to Statistical Pattern Recognition", 2nd Ed. (Academic Press, New York, 1990.

R. Schalkoff

"Pattern Recognition: Statistical, Structural and Neural Approaches", (John Wiley & Sons, New York, 1992).

Friedman J. H.

"Adaptative techniques for machine learning and function approximation", in Cherkassky V., Friedman J. H., Wechsler H., editors, From Statistics to Neural Networks. Theory and Pattern Recognition Applications, ASI Proceedings Series F: Computer and Systems Sciences, pages 1-82, New York: Springer-Verlag

Watanabe S.

"Pattern Recognition: Human and Mechanical", New York: Wiley, (1985)

Jain A. K., Duin R. P. W., Mao J.

"Statistical Pattern Recognition : a review", IEEE Trans. on pattern analysis and machine intelligence, vol. 22, no. 1, (2000)

R. O. Duda and Peter E. Hart

"Pattern Classification and Scene Analysis". New York: Wiley, 1973.

T, Kohonen, G. Barna, and R. Chrisley

"Statistical pattern recognition with neural networks: Benchmarking studies", in IEEE Proc. ICNN, vol. 1, July 1988, pp, 61-68.

Clustering and Self Organizing Maps

Everitt, B.S.

"Cluster Analysis" John Wiley & Sons, Inc., (1974)

J. K. Hawkins

"Self-organizing systems – a review and commentary" , Proc. IRE, vol. 49, pp. 3l-48, Jan. 1961.

Human ear modeling

D. J. M. Robinson & M. J. Hawksford

"Time-Domain Auditory Model for the Assessment of High-Quality Coded Audio", preprint 5017, presented at the 107 th convention of the Audio Engineering Society in New York, (1999 Sept)

H. Traunmüller

"Analytical expressions for the tonotopic sensory scale", J. Acoust. Soc. Am., vol. 88, no. 1, pp. 97-100, (1990 July)

Slaney

"An efficient implementation of the Patterson-Holdsworth auditory filter bank", Apple Computer Technical Report 35, (1993)

Ellis

"Mid-level representations for Computational Auditory Scene Analysis", Proceedings of the International Joint Conference on AI, Workshop on Computational Auditory Scene Analysis, (1995 Aug)

Patterson R. D., Holdsworth J.

"A functional model of neural activity patterns and auditory images", in Advances in speech, hearing and auditory images, W.A. Ainsworth (ed.), London: JAI Press, (1990)

Meddis R.

"Simulation of mechanical to neural transduction in the auditory receptor", Journal of the Acoustic Society of America, vol.73, no.3, pp.702-711, (1986)

American National Standards Institute (1973).

"American national psychoacoustical terminology": New York: American Standards Association. (come citato da Houtsma,1997)

American Standards Association (1960)

American Standard Acoustical Terminology.Definition 12.9, Timbre, p. 45. New York.

Pickles, J. O.

"Introduction to the Physiology of Hearing." Academic Press, (1988)

Musical instruments

Beauchamp, J. W. (1974)

"Time-variant spectra of violin tones". J. Acoust. Soc.Am. 56(3), 995-1004.

Benade, A. H. (1990)

"Fundamentals of Musical Acoustics". New York: Dover.

Blackham, E.D. (1965)

"La fisica del pianoforte" Le Scienze – quaderni n. 87 Milano

Pintacuda, S.

"Acustica Musicale" Edizione Curci

Music Signal Analysis

Guillermain P., Kronland-Martinet R.

"Characterization of Acoustic Signal through Continuous Linear Time-Frequency Representations", Proc. IEEE, vol.84, no.4, pp.561-585, (1996)

W.J. Pielemeier and Wakefield, G.H.

"A high-resolution time-frequency representation for musical instrument signals", J. Acoust. Soc. Am., vol. 99, Pt. 1, pp. 23822396, Apr. 1996.

Moorer, J.A. (1975)

"On the Segmentation and Analysis of Continuous Sound by Digital Computer", Ph.D.Thesis, Department of Computer Science, Stanford University, Stanford

Brown J. C.

"Calculation of a constant Q spectral transform", J. Acoust. Soc. Am., vol. 89, pp. 425-434, Jan. 1991

Brown J. C., Puckette M. S.

"An efficient algorithm for the calculation of a constant Q transform", J. Acoust. Soc. Am.,

92(5):2698-2701, (1992)

Depalle, Garcia, Rodet

Tracking of partials for additive sound synthesis using hidden Markov models, IEEE Trans. On Acoustic, Speech and Signal Processing, (1993)

Serra

"Musical sound moddeling with sinuosoid plus noise", Road, Pope, Poli (cds.). Musical Signal Processing, Swets & Zeitlinger Publishers, (1997)

Smith L.S.

"Onset-based Sound Segmentation", in Advances in Neural Information Processing Systems 8, Touretzky, Mozer and Haselmo (ers.), Cambridge, MA: MIT Press, (1996)

Tristan Jehan

"Musical Signal Parameter Estimation", (1997)

Strong, W. J.

"Synthesis and Recognition Characteristics of Wind Instrument Tones." Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, (1963)

Terhardt, E., Stoll, G. and Seewann, M.

"Algorithm for extraction of pitch and pitch salience from complex tonal signals", J. Acoust. Soc. Am., 71(3), March 1982

Martin, K. D. & Kim, Y. E.

"Musical instrument identification: a pattern-recognition approach." Presented at the 136th meeting of the Acoustical Society of America, (1998)

Martin, K.D.

"Toward automatic sound source recognition: identifiying musical instruments" NATO Computational Hearing Advanced Study Institute, Il Ciocco, Italy 1998

Lichte, W. H.

"Attributes of complex tones". J. Experim. Psychol. 28, 455-481, (1941)

Spline Activation Functions

Amari, S.

"A universal theorem on learning curves". Neural Networks, 6, 161–166, (1993)

Chen, C. T., & Chang, W. D.

"A feedforward neural network with function shape autotuning. Neural Networks, 9, 627–641, (1996)

Piazza, F., Uncini, A., & Zenobi, M.

"Artificial neural networks with adaptive polynomial activation function. Proc. of the IJCNN, vol. 2, Bejing, China, pp. 343–349, (1992)

Piazza, F., Uncini, A., & Zenobi, M.

"Neural networks with digital LUT activation function", Proc. of the IJCNN, vol. 2, Nagoya, Japan, pp. 1401–1404, (1993)

S.Guarnieri, F.Piazza, A.Uncini,

"Multilayer Feedforward Networks with Adaptive Spline Activation Function", IEEE Transactions on Neural Network, Vol.10, N°3 May 1999.

Moody, J.E.

"The effective number of parameters: an analysis of generalization and regularization in nonlinear learning systems", in J.E. Moody, S.J. Hanson, & R.P. Lippmann, eds, Advances in Neural Information Processing Systems, 4, Morgan Kauffmann, pp. 847–854, (1992)

Recurrent Neural Networks

P.Campolucci, A.Uncini, F.Piazza, B.D.Rao

"On-Line Learning Algorithm for Locally Recurrent Neural Networks", IEEE Transactions on Neural Network, Vol.10, N°2 March 1999.

Waibel A.T., Hanazawa T., Hinton G., Shikano K., Lang K.J.

"Phoneme Recognition using TDNN", IEEE Trans. Acoustics., Speech and Signal Processing 37, (1989)

Parisi R., Di Claudio E. D., Orlandi G., Rao B. D.

"Fast adaptative digital equalization by recurrent neural networks", (1999)

Back A. D., Tsoi A. C.

"FIR and IIR synapses, a new neural network architecture for time series modelling", Neural Comput., vol. 3, pp. 375-385, (1991)

Back A. D., Tsoi A. C.

"Locally recurrent globally feedforward networks: a critical review of architectures", IEEE Trans. On Neural Networks, vol. 5, pp. 229-239, (1994)

Back, A.D., & Tsoi, A.C.

"A simplified gradient algorithm for IIR synapse Multilayer Perceptron". Neural Networks, 5, 456–462, (1993)

Campolucci P., Piazza F.

"Intrinsic Stability-Control Method for Recursive Filters and Neural Networks", IEEE Trans. On Circuits and Systems – II: Analog and Digital Signal Processing, vol.47, no.8, (2000)

Werbos P. J.

"Backpropagation Through Time: what it does and how to do it", IEEE, Special Iusses on Neural Networks, vol.78, pp.1550-1560, (1990)

Pearlmutter B. A.

"Gradient Calculations for dynamic recurrent neural networks: a survey", IEEE Trans. On Neural Networks, vol.6, (1995)

L. Vecci, F Piazza, A. Uncini

"Learning and Approximation Capabilities of Adaptive Spline Activation Function Neural Networks", Neural Networks, Vol. XI, No.2, pp. 259-279, (1998)

Discrimination and classification

Mette Langas

"Discrimination and classification"

S. Amari

"A theory of adaptive pattern classifiers", IEEE Trans. Elec. Comput., vol. EC-16, pp. 299 – 307, June 1967.

Day N. E., Kerridge D. F.

"A general maximum likelihood discriminant", Biometrics, 23, 313-323, (1967)

Duda R. O., Hart P. E.

"Pattern Classification and scene analysis", New York: Wiley, (1973)

W. Chou and B. H. Juang

"Adaptive discriminative learning in pattern recognition"

S. Katagiri, C. H. Lee, and B. H. Juang

"New discriminative training algorithm based on the generalized probabilistic descent method", in Proc. 1991 IEEE workshop Neural Networks for Signal Processing, Piscataway, NJ, Aug. 1991, pp. 299-308.

S. Katagiri, C. H. Lee, and B. H. Juang

"Discriminative multilayer feedforward networks", in Proc. 1991 IEEE Workshop Neural Networks for Signal Processing, Piscataway, NJ, Aug. 1991, pp. 11-20.

Support Vector Machines

A. Karatzoglou, D. Meyer, K. Hornik

"Support Vector Machines in R", Journal of Statistical Software, April 2006, Volume 15, Issue 9

Steve R. Gunn

"Support Vector Machines for Classification and Regression", UNIVERSITY OF SOUTHAMPTON Technical Report, 10 May 1998

MaxWelling

"Support Vector Machines", Department of Computer Science, University of Toronto

Edgar E. Osuna, R.Freund, F. Girosi

"Support Vector Machines: training and applications", A.I. memo N° 1602, C.B.C.L. paper N° 144, March 1977

Fuzzy Min-Max Neural Networks

L.Zadeh

"Fuzzy sets", Inform. And Control, vol. 8,pp 338-353,1965

A.Kandel

"Fuzzy Mathematical Techniques with Applications". Reading, MA.: Addison-Wesley, 1986.

Patrick K. Simpson

"Fuzzy Min-Max Neural Networks – Part 1: Classification", IEEE Transactions on Neural Networks, vol. 3, no. 5, September 1992.

Patrick K. Simpson

"Fuzzy Min-Max Neural Networks – Part 2: Clustering", IEEE Transactions on Fuzzy Systems, vol. 1, no. 1, February 1993.

Patrick K. Simpson

"Fuzzy Min-Max Neural Networks", work supported by General Dynamics Electronics Division Independant Research and Development Project No. 91015131, pp.1658-1669.

Patrick K. Simpson

"Fuzzy Min-Max Classification with Neural Networks", pp. 291-300.

P.Simpson

"Fuzzy Adaptive Theory", presented at the Southern Illinois University Neuroengineering Workshop, Sept. 6-7, 1990, Carbondale,IL. Also in Proc. GD AI 90 (Ft Worth, TX), Nov. 7-8, 1990, and published as General Dynamics Tech. Rep. GDE-ISG-PKS-11.

Bogdan Gabrys and Andrzej Bargiela

"General Fuzzy Min-Max Neural Network for Clustering and Classification", IEEE Transactions on Neural Networks, vol. 11, no. 3, May 2000.

Shigeo Abe and Ming-Shong Lan

"A Classifier Using Fuzzy Rules Extracted Directly from Numerical Data", IEEE 1993, pp. 1191-1198.

Shigeo Abe and Ming-Shong Lan

"A Method for Fuzzy Rules ExtractionDirectly from Numerical Data and Its Application to Pattern Classification", IEEE Transactions on Fuzzy Systems, vol. 3, no. 1, February 1995, pp. 18-28.

A.Rizzi, F.M. Frattale Mascioli, and G.Martinelli

"Generalised Min-Max Classifier", IEEE 2000, pp. 36-42.

F.M. Frattale Mascioli, A.Rizzi, M.Panella, and G.Martinelli

"Clustering with Unconstrained Hyperboxes", 1999 IEEE International Fuzzy Systems Conference Proceedings, August 22-25 1999, Seoul, Korea, pp. 1075-1080.

Audio musical source recognition

Berger, K. W.

"Some factors in the recognition of timbre". J. Acoust. Soc.Am. 36, 1888-1891, (1964)

Brown, J. C.

"Frequency ratios of spectral components of musical sounds." J. Acoust. Soc. Am. 99(2), 1210-1218, (1996)

Brown, J. C.

"Computer identification of musical instruments using pattern recognition." Presented at the 1997 Conference of the Society for Music Perception and Cognition, Cambridge, MA, (1997)

Brown, J. C.

"Musical instrument identification using pattern recognition with cepstral coefficients as features." J. Acoust. Soc. Am. 105(3) 1933-1941, (1999)

Brown, J.C.

"Frequency ratios of spectral components of musical sounds", J. Acoust. Soc. Am., vol. 99, pp. 1210-1218, Feb. 1996

Brown, J.C.

"Pitch center of stringed instrument vibrato tones", J. Acoust. Soc. Am., vol. 100, pp. 1728-

1735, Fep. 1996

Campbell, W. C. & Heller, J. J.

"The contribution of the legato transient to instrument identification." In E. P. A. Jr. (ed.) Proceedings of the research symposium on the psychology and acoustics of music (pp. 30-44). University of Kansas, (1978)

Crummer, G. C., Walton, J. P., Wayman, J. W., Hantz, E. C. & Frisina, R. D.

"Neural processing of musical timbre by musicians, nonmusicians, and musicians possessing absolute pitch". J. Acoust. Soc. Am. 95(5), 2720-2727, (1994)

Fletcher, H. & Sanders, L. C.

"Quality of violin vibrato tones." J. Acoust. Soc. Am. 41(6), 1534-1544, (1967)

Fletcher, H.

"Normal vibration frequencies of a stiff piano string." J. Acoust. Soc. Am. 36, 203-209, (1964)

Fletcher, H., Blackham, E. D. & Stratton, R.

"Quality of piano tones." J. Acoust. Soc. Am. 34(6), 749-761, (1962)

Risset, J. C.

"Computer study of trumpet tones." Bell Laboratories Technical Report, Murray Hill, New Jersey, (1966)

Discriminative Learning

Juang B. H., Katagiri S.

"Discriminative Learning for Minimum Error Classification", IEEE Trans. On Signal Processing, vol. 40, no. 12, (1992)

Katagiri S., Lee C. H., Juang B. H.

"New discriminative algorithm based on the generalized probabilistic descent method", Proc. 1991 IEEE Workshop Neural Networks for Signal Processing, Piscataway, NJ, pp. 299-308, (1991)

Lee T., Ching P.C., Chan L.W.

"An RNN based Speech Recognition System with Discriminative Training"

Automatic Transcription of Music

Moorer, J.

"On the Segmentation and Analysis of Continuous Musical Sound by Digital Computer". PhD

thesis, Stanford University, CCRMA. (1975).

Klapuri A.

"Automatic Transcription of Music", Master of Science Thesis, Tampere University of Technology, (1997)

Piszczalski, M. and Galler, B.

"Automatic music transcription". Computer Music Journal, 1(4):24–31, (1977)

Kashino, Tanaka

"A sound source separation system with the ability of automatic tone modelling", Proceedings of the International Computer Music Conference, (1993)

Katayose, Inokuchi

"The Kansei Music System", Computer Music Journal, 13(4) , (1989)

Kashino K., Nakadai K., Kinoshita T., Tanaka H.

"Application of Bayesian probability network to music scene analysis", Proceedings of International Joint Conference in AI, Workshop on Computational Auditory Scene Analysis, Montreal, Canada, (1995)

Kashino, K., Nakadai, K., Kinoshita, T., and Tanaka, H.

"Organization of hierarchical perceptual sounds: Music scene analysis with autonomous processing modules and a quantitative information integration mechanism", in Proceedings of the International Joint Conference on Artificial Intelligence, (1995)

Martin K.

"A Blackboard System for the Automatic Transcription of simple polyphonic music", MIT Media Laboratory, Perceptual Computing Section, Technical Report No.399, (1996)

Scheirer E.

"Extracting expressive performance information from recorded music", Master's thesys, Massachusset Institute of Technology, Media Laboratory, (1995)

Scheirer, E.

"Extracting expressive performance information from recorded music". Master's thesis, Massachusetts Institute of Technology, Media Laboratory, (1995)

Scheirer E.

"Using Musical knowledge to extract expressive performance information from audio recordings", In Okuno, H. and Rosenthal, D., editors, Readings in Computational Auditory Scene Analysis. Lawrence Erlbaum, (1997)

Dixon S.

"On the Computer Recognition of Solo Piano Music", Proceedings of Australian Computer

Music Conference, Brisbane, Australia, (2000)

Dixon, S.

"A dynamic modelling approach to music recognition", in Proceedings of the International Computer Music Conference, pages 83–86. Computer Music Association, San Francisco CA, (1996)

Dixon, S. and Cambouropoulos, E.

"Beat tracking with musical knowledge", in ECAI 2000: Proceedings of the 14th European Conference on Artificial Intelligence. IOS Press, (2000)

Marolt, M.

"Feedforward Neural Network for Piano Music Transcription", Proc. Of XII Colloquium on Musical Informatics, Gorizia, pp. 240-243, Sept. 1998.

Marolt M.

"SONIC: Transcription of Polyphonic Piano Music with Neural Networks", Ph.D. Thesis

Sterian A., Wakefield G. H.

"A model-base approach to partial tracking for musical transcription", (1998)

Chafe, C., Jaffe, D., Kashima, K., Mont-Reynaud, B., and Smith, J.

"Techniques for note identification in polyphonic music", in Proceedings of the International Computer Music Conference. Computer Music Association, San Francisco CA, (1985)

Schloss,W.

"On the Automatic Transcription of Percussive Music: From Acoustic Signal to High Level Analysis". PhD thesis, Stanford University, CCRMA, (1985)

Watson, C.

"The Computer Analysis of Polyphonic Music". PhD thesis, University of Sydney, Basser Department of Computer Science, (1985)

Human locomotion task analysis

Kerr K. M

"Analysis of the sit to stand movement cycle in normal subjects", Clinical Biomechanics 1997

Kralj A.

"Analysis of standing up and sitting down in humans: definitions and normative data presentation" J. Biomech. 23 1123-38

Munro B.J

"A kinematic and kinetic analysis of the sit-to-stand transfer using an ejector chair:

implications for rheumathoid arthritic patients" J. Biomech. 263-71

J..M. Winters and P. E. Crago

"Biomechanics and Neural Control of Posture and Movement" New York: Springer 2000

M.J Adrian and J.M Cooper

"Biomechanics of human movement" (Indianapolis, IN. Benchmark) 1989

Sensors for Human locomotion task analysis

M.J Adrian and J.M Cooper

"Biomechanics of human movement" (Indianapolis, IN: Benchmark)

Mathie et al.

"Accelerometry: providing an integrated, practical method for long-term, ambulatory monitoring of human movement" Physiological Measurement 25 (2004)

J..M. Mathie, P. E. Crago, B.G Celler

"Detection of daily physical activities using a triaxial accelerometer " Med Biol. Eng. Comp. 41 2003

K. Aminian, M. Depairon, D. Hayoz

" Physical activity monitoring based on accelerometry: validation and comparison with video observation. Med. Boil Eng. Comput 37 1999

J. R. W. Morris

"Accelerometry - a technique for the measurement of human body movements". Journal of Biomechanics 6, pp. 729-736, 1973.

A. J. Padgaonkar, K. W., King, A. I.

"Measurement of angular acceleration of a rigid body using linear accelerometers". ASME Journal of Applied Mechanics 42, pp. 552-556, 1975.

D. Giansanti, V. Macellari, Maccioni G., and Cappozzo A.

"Is it feasible to reconstruct body segment 3-D position and orientation using accelerometric data? " IEEE Trans. Biomedical Vol 50 N. 4, 2003.

D. Giansanti, V. Macellari, G. Maccioni, M. Paolizzi, A. Cappozzo

"Measurement of Kinematic Parametres using a wearable device" in ISPG 2001 book, Maastricht 2001.

D. Giansanti, V. Macellari, G. Maccioni, M. Paolizzi, S. Cesinaro

"A wearable device for measurement of kinematic parameters of sit to stand", proceeding of International Symposium on Biomechanics (Zurigo, 2001)

D . Giansanti, Maccioni G, V. Macellari

"The development and test of a device for the reconstruction of 3D position and orientation by means of a kinematic sensor assembly with rate gyroscopes and accelerometers " IEEE trans. Biomedical Engineering , Vol 52, Issue 7, pp. 1271- 1277 (2005)

D . Giansanti, , G. Maccioni

"Comparison of three different kinematic sensors assemblies", Physiol. Meas. 26 pp. 689-705, (2005)

D. Giansanti, G. Maccioni

"Physiological motion monitoring: wearable device and adaptive algorithm for sit-to-stand timing detection" 27, pp. 713-723 (2006)

D. Giansanti, G. Maccioni, G. Costantini, M. Carota

"The classification of the rising from a chair by means of an integrated kinematic sensor with accelerometers and rate-gyroscopes: preliminar results"

G. Costantini, M. Carota, G. Maccioni, D. Giansanti

"Classification of the sit-to-stand locomotion task based on spectral analysis of waveforms generated by accelerometric transducer" Electronic letters in press 2006

G. Costantini, M. Carota, G. Maccioni and D. Giansanti

"Discrimination between human functional ability/disability by means of different Classification Methodologies"

Chiari L., Dozza M., Horak F., Cappello A., Macellari V., Giansanti D.

"An Accelerometry-based System for Balance Improvement Using Audio-biofeedback" IEEE Transaction on Biomedical Engineering vol. 52 N. 12 pp. 2108-2111 (2005)

R. Moe-Nilssen

"Trunk accelerometry: A new method for assessing balance under various task and environmental constrains" book (2002) Bergen University ISBN 82-912332-20-2

R. Moe-Nilssen and J. L. Helbostad

"Trunk accelerometry as a measure of balance control during quiet standing," Gait Posture, vol. 16, pp. 60-68, 2002

P. H Veltink, H.B.J Bussmann, F. Koelma, H.M. Franken, W.L.J.Martens

"The feasibility of posture and movement detection by accelerometry" Proceeding of EMBS conference, pp. 1230-1, (1993)

M.E. Tinetti

"Performance-oriented assessment of mobility problems in elderly patients", J Am Geriatr Soc. 34(2), pp 119-26, (1986)