ISTITUTO NAZIONALE DI FISICA NUCLEARE

Sezione di Trieste

INFN/AE-92/13

3 aprile 1992

C. Bortolotto, A. De Angelis, N. De Groot and J. Seixas

NEURAL NETWORKS IN EXPERIMENTAL HIGH-ENERGY PHYSICS

Servizio Documentazione
dei Laboratori Nazionali di Frascati

NEURAL NETWORKS IN EXPERIMENTAL HIGH-ENERGY PHYSICS

C. BORTOLOTTO, A. DE ANGELIS
Istituto di Fisica dell'Universita' di Udine and INFN Trieste
Via Fagagna 208, I-33100 Udine, Italy

N. DE GROOT
NIKHEF
Amsterdam NL-1009, The Netherlands

and

J. SEIXAS
CERN
Geneva CH-1211, Switzerland

ABSTRACT

During the last years, the possibility to use Artificial Neural Networks in experimental High Energy Physics has been widely studied. In particular, applications to pattern recognition and pattern classification problems have been investigated. The purpose of this article is to review the status of such investigations and the techniques established.

Keywords: Neural networks; Classification problems; Optimization problems; Taxonomy; Nonlinear separators.

To be published in International Journal of Modern Physics C

INTRODUCTION

During the last decade the interest on neural networks has been increasing steadily and their field of application is nowadays growing fast as their ability to solve intricate pattern recognition problems is shown to be more and more powerful. In High Energy Physics (HEP), however, the interest on these systems has been up to now somewhat restricted, mainly because conventional data aquisition and analysis methods work quite well. This situation is bound to change with the next generation of accelerators: higher energies mean higher multiplicities and thus the next generation of HEP experiments will have to deal with an unparalleled wealth of information both on-line an off-line. Treating vast amounts of data requires an increase in speed of data processing, which makes massive parallelism unavoidable and, clearly, neural networks a natural tool.

It is important to emphasize that the term "neural" is in this context somewhat misleading. Although neural nets are collections of basic units connected between themselves in a certain way, the resemblance to a real brain stops here: the basic units have very little to do with actual neurons and even in the best cases the rate of firing imposed to these basic units by far exceeds the one found in real neurons. The main point is that in neural nets information processing tends to be parallel (and in some models asynchronous and stochastic), rather than sequential, clocked and deterministic as in conventional computers systems. Also, information storage is distributed across the network rather than stored in specific memory location. Since there is no separation between CPU and memory units one avoids the von Neumann bottleneck: in this resides all the power of neural networks.

It is also worth mentioning that although presently existing models should be used as a guiding line to the network designer, practical applications can (and should) be attacked in general with a much more broad minded approach. Implementation in hardware of a real neural system becomes more and more difficult with the increasing number of units involved, and so one should always contemplate the possibility of trading architecture complexity for neuron complexity.

The paper is organized as follows: in Section 1 we shall introduce some basic facts about neural systems and in Sections 2 and 3 we shall describe two well known models: Hopfield nets and back-propagation nets. Together with the description of the techniques, some current applications are presented. Finally we conclude and present some prospects for the future development of neural nets in HEP.

1. Basics

1.1. Neural Networks as Computing Machines

Before embarking directly on the subject of neural networks (NN) let us first make a lightning detour on generalities of computing machines (Ref. 1). A computing machine is a black box which interacts with a given environment $\mathcal{E}$ receiving from time to time inputs $I(t)$ and delivering from time to time some outputs $O(t)$. The machine is, of course, composed of internal parts which, by the action of the inputs, will change their state during the history of the apparatus. Thus, to fully characterize a computing machine one needs to define two functions. First, an input/output function

$$O(t') = F(I(t), Q(t)) \tag{1.1}$$

which characterizes the output delivered by the machine at a later time t' as a function of the input received and of the internal state of the machine $Q(t)$. However, as the input is received the internal state can also change, and so we also need to know the change of state function

$$Q(t') = G(I(t), Q(t)) \tag{1.2}$$

If the possible number of states through which a machine can pass is finite we say it is a finite state machine; otherwise we say it is an infinite state machine. In all the cases we will consider in this paper, time is a discrete variable and so expressions (1.1-2) can be rewritten in the form

$$O(t+1) = F(I(t), Q(t)) \tag{1.3}$$

$$Q(t+1) = G(I(t), Q(t)) \tag{1.4}$$

Let us now introduce the formal neuron (Ref. 2), a very simple computing machine, represented as in Fig. 1.

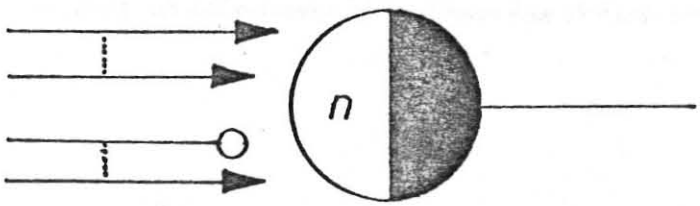

Fig. 1.

It has a certain number of input lines $I_m (m = 1, 2, \ldots, M)$ and a simple output line. These lines can be either be excitatory ($I_m = 1$, represented as $\longrightarrow$) or inhibitory ($I_m = 0$, represented as $\multimap$). The change of state function is constant and the input/output function is a theta function

$$O(t+1) = \Theta\left(\sum_{m=1}^{M} I_m(t) - n\right)$$

that is, a formal neuron only gives an output ("fires") when the sum over all the lines I_m is greater or equal than a given threshold n. Some specific examples of formal neurons are given in Fig. 2.

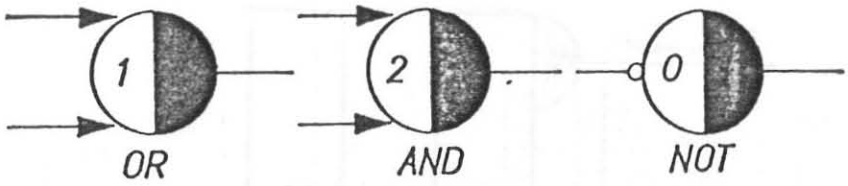

Fig. 2.

Formal neurons can be connected between themselves to produce more elaborated computing machines, in the sense that the outputs from some units can be considered as inputs to other units. Furthermore it is clear that any finite collection of formal neurons is a finite state machine.

The converse is also true (Ref. 2): given a finite state machine it is possible to construct a net of formal neurons that is equivalent to that machine. This means that given the (boolean) functions F and G it is possible to construct a net of formal neurons that will represent those functions. So, designing a net is equivalent to finding a function representation.

In practical situations, however, the representation of the functions F and G is in general not known. Therefore one has to devise methods to construct it or, at least, give the best approximation to the desired representation. Adaptive neural networks (ANN) as we will see in subsequent sections provide a solution to this problem in the form of an iterative process that converges to the right representation of the F and G functions. This iterative process – also called learning – corresponds thus to the search of fixed points in the space of all possible functions.

One should notice that it is not guaranteed off-hand that the methods provided by ANN give always the more economical or even the best solutions to the problem being studied. Very often the design of the net for a practical problem is largely a matter of trial and error. On the other hand, as we will show in some examples, non adaptable nets (NAN), like the nets of formal neurons we have described above, can also give quite good solutions in some well defined problems, with the additional advantage that they are much easier to implement in hardware.

1.2. Architecture

As we saw in the last section a neural net is an ensemble of very simple computing machines (neurons) connected among themselves with some definite architecture. The connections between the neurons can have some resistivity. This resistivity specifies how much the signal sent from unit j to unit i will influence the response of unit i, that is, it specifies the *weight* of unit j in the response of unit i. Therefore we associate with each link between two neurons i and j a weight w_{ij}. For the same inputs I_j on a given neuron i different sets of weights w_{ij} will produce different outputs. This means that the input/output relation for the *whole* network is coded in the particular set of weights chosen.

In the nets of formal neurons we introduced in the last section the w_{ij} could only take the values +1 (excitatory), -1 (inhibitory) or 0 (no connection) and these weights did not change during the history of the net. However, as we said above, we will be also interested in cases in which the values of the weights can change dynamically during the execution time of the machine. The nets for which this holds are called ANN and the prescription as how to change the value of the weights is called a learning rule. In the case the weights are kept fixed the net is said to be non-adaptive.

In adaptive networks we will not be interested in neurons as simple as the formal neurons. In Fig. 3 we depict a typical neuron for those nets. Its environment $\mathcal{E}$ is the ensemble of the neurons of

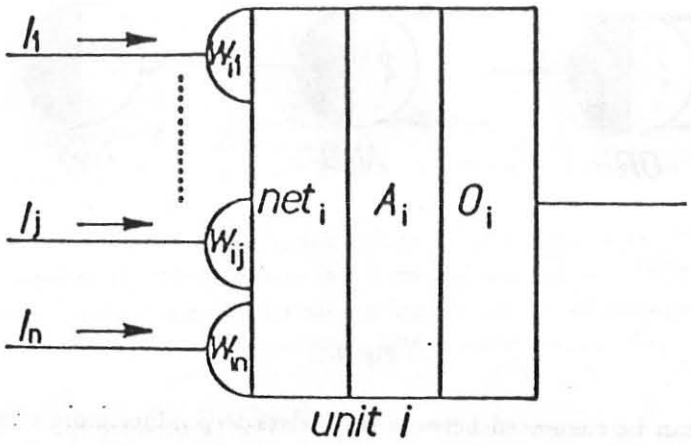

Fig. 3. A schematic representation of a neuron.

the network which might be connected with it. We call "fan-in" the number of connections which excite or inhibit a given unit. Similarly we call "fan-out" the number of units directly affected by a given unit. As a first step of processing in neuron i, the net input $net_i = \sum_{j \in Input} w_{ij} I_j$ is calculated, where w_{ij} represents the weight of the connection between unit j and unit i, and I_j is the magnitude of the input coming from unit j. In general, the input I_j is equal to the magnitude of he output O_j of unit j and so, unless otherwise stated, we will make no distinction between them.

As for the (input/)output function O_i and change of state (activation) function A_i, several recipes can be found in the literature. However, the analogy to the physiology of the actual biological neuron tends to enforce units whose output follows closely the biological behaviour (also, for instance, linear units often give somewhat trivial results). It is known that neuronal response tends to saturate when the input level is sufficiently high or sufficiently low. As a result the activation values (which, are sometimes equal to the output of the unit) are taken to follow the sigmoid shape depicted in Fig. 4, which is known empirically to describe the behaviour of the actual neuron.

Finally we will introduce some basic concepts for the architecture of the network, that is, the pattern of connectivity for the neurons involved in the particular system under study. Here we have essentially two different approaches, corresponding to two essentially distinct models:

(i) Layered networks, in which we have the units organized in layers (see Fig. 5) the simplest (and most limited) example being the *perceptron* introduced by Rosenblatt (Ref. 3,4). In this case there are only two layers, the input layer and the output layer, with each neuron of the input layer connected to the neuron of the output layer. However, as has been shown by Minsky and

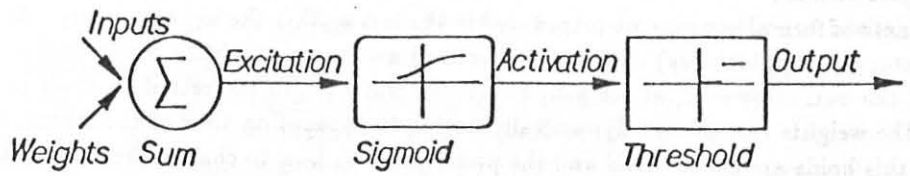

Fig. 4. Operations within a neuron.

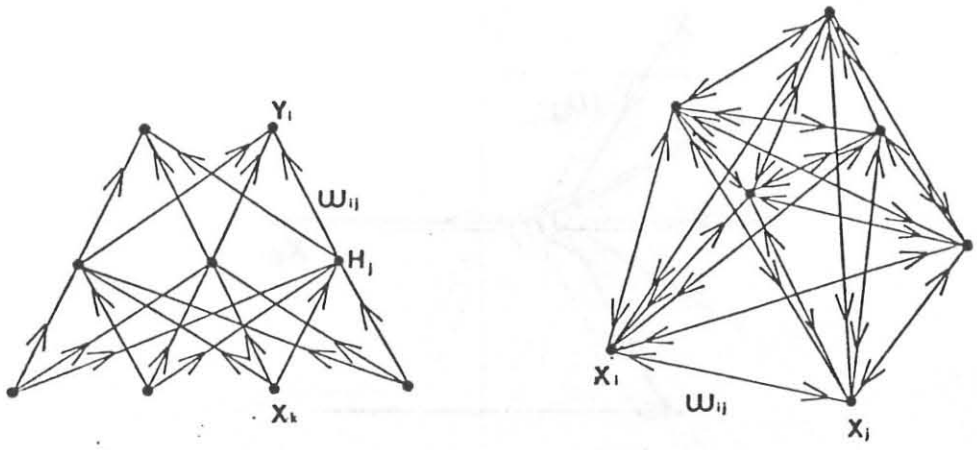

Fig. 5. Feed-forward and feed-back architecture.

Papert (Ref. 5) two layers only are not enough to encode all possible input/output functions. Thus, the layered networks presently used have at least three layers – input layer, middle or "hidden" layers and output layer. This type of architecture is often associated to a pattern of connectivity where the output of units on a given layer act only as inputs to units on layers above their own (the so-called feed-forward networks).

(ii) Hopfield-type networks in which every unit is connected to all other units without any particular hyerarchichal structure (see Fig. 5). So, this kind of networks is in principle a generalization of the ones previously introduced. In practice this type of architecture is quite often associated with a fixed (in the sense of non-adaptable) pattern of connectivity, and so the network does nothing more than to "recall" patterns which have been previously stored "by hand". The major advantage of the Hopfield-type networks stems from the fact that in some well defined conditions the whole machinery of Statistical Mechanics can be applied allowing for a careful study of problems like determining, for instance, the number of patterns which can be stored and retrived with a given error.

As we saw, a basic fact about any model of neural networks is that information is stored in the particular choice of weights w_{ij} which has been taken. So, during the history of the network not only the neurons can change their internal state, but also the weights can change by a learning procedure. In the next section we will look more closely at this dynamics.

1.3. Dynamics

So far we have analysed the basic components of a neural network. We will now turn our attention to the actual interplay of the various elements of the network during the history of the system. Here we must distinguish between two phases in this history:

(i) The learning phase during which the weights are modified according to a chosen learning rule.

(ii) When the learning phase is over, the network is ready to be used. Now we enter a phase in which we should be able to retrieve information which has been stored using the learning algorithm. Notice that the redundancy introduced in the network by the large number of connections present implies that even if some of these connections do not work, or some error is present on input, we still expect the network to give the right – or an approximation to the right – answer.

In the next two sections we will see how these different phases are realized on the two most

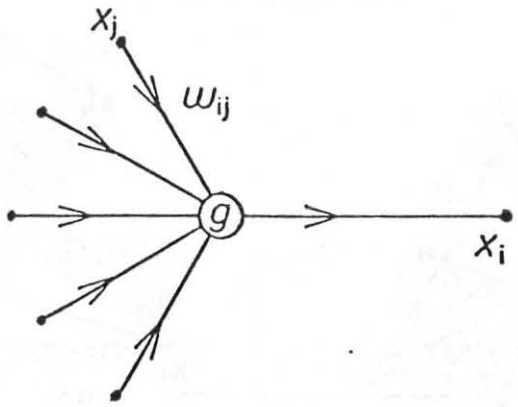

Fig. 6. A generic neural network updating.

widely used models, the Hopfield model and the back-propagation model.

2. Hopfield-type Neural Networks

A Hopfield-type (Ref. 6) neural network is a N-dimensional dynamic system that evolves at discrete time steps. Each neuron i at time t, has a binary state $x_i(t)$, which takes, for example, the value $+1$ when the neuron fires and -1 when it is quiescent. In such a way, the state of the whole network at time t can be described by a binary "state vector":

$$x(t) = (x_1(t), x_2(t), \ldots, x_N(t))$$

where N in the number of neurons in the network, and by a weight matrix $[w_{ij}]$ where $w_{ij} > 0$ represents an excitatory influence of neuron j on neuron i, while $w_{ij} < 0$ represents an inhibitory influence. As usual, if $w_{ij} = 0$ there is no connection between neuron j and neuron i. A *"local threshold"* Θ_i is associated with each neuron i. The neuron i fires at time $t' = t + \delta t$, according to a probability which is a function of the integrated input received, compared to the threshold Θ_i:

$$Prob(x_i(t') = +1) = g(\sum_{j=1}^{N} w_{ij} x_j(t) - \Theta_i)$$

(see Fig. 6) where g is a sigmoid function as shown in Fig. 7. In general the function g is also a function of a parameter T called *temperature*, which represents the width of the region in which g increases from near 0 to near 1. T is then a measure of the stochasticity of the process: $T = 0$ corresponds to the deterministic limit of the step function (bold line in Fig. 7).

If a discrete time dynamics and $T = 0$ are assumed, the time evolution of the system is described through the dynamic equation:

$$x_i(t+1) = Sign[\sum_{j=1}^{N} w_{ij} x_j(t) - \Theta_i] \tag{2.1}$$

where

$$Sign(y) = \begin{cases} +1, & \text{if } y > 0; \\ -1, & \text{if } y < 0; \\ undefined, & \text{if } y = 0. \end{cases}$$

If the weighted sum over all the neurons j connected to neuron i is greater than the threshold Θ_i, then i fires and its state becomes $+1$. If the weighted sum is less than Θ_i, i turns off and its state

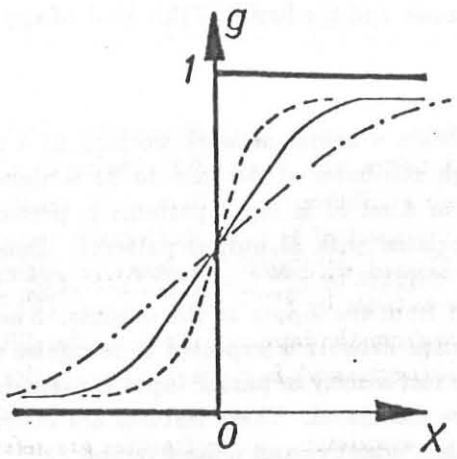

Fig. 7. The probability of firing as a function of integrated input for different temperatures T.

becomes -1. If the sum equals Θ_i, the convention is that the neuron i maintains its previous state. The weights w_{ij} and the thresholds Θ_i are the network parameters.

There are two major ways to implement the updating of the states: *parallel-synchronous* and *sequential-asynchronous*. In parallel updating all neurons are updated at the same time, then this kind of dynamics is described by the dynamic Eq. (2.1), where each new state $x_i(t+1)$ of a neuron i is computed as a function of all non-updated states $x_j(t)$ of the other neurons. In contrast, in an orderly sequential iteration, where neuron i updates its state immediately after neuron $i-1$, Eq. (2.1) should be replaced by

$$x_i(t+1) = Sign[\sum_{j=1}^{i-1} w_{ij}x_j(t+1) + \sum_{j=i}^{N} w_{ij}x_j(t) - \Theta_i]$$

Not all sequential iterations are ordered: the neurons states can be updated in random order. Sequential updating usually applies in statistical mechanics and is more natural for disordered systems (Ref. 7), where there is no clock to control the simultaneous updating. On the other hand delays in signal transmission are relatively large. Dynamic rules between these two extreme cases are possible. Since the information about the updating of a neuron might not have enough time to arrive at the next one, this effect could be modelled by parallel synchronous updating of randomly chosen blocks of neurons (*block-sequential* updating). Different models imply different dynamics, but parallel and sequential updatings present analogies (Ref. 8). Sometimes, the components of the state vector assume the values 0 and 1 instead of -1 and $+1$. Equation (2.1) is then substituted by

$$x_i(t+1) = H[\sum_{j=1}^{N} w_{ij}x_j(t) - \Theta_i]$$

where H is the *Heaviside function*

$$H(y) = \begin{cases} +1, & \text{if } y > 0; \\ 0, & \text{if } y < 0; \\ undefined, & \text{if } y = 0. \end{cases}$$

In the following sections, we will present two kinds of problems in which a feed-back neural network model works well: associative memories (Hopfield model) and the track finding prob-

lem (Ref. 9), showing the successes and the limits of this kind of approach.

2.1. Hopfield Model

Our goal will be now to obtain a neural network working as a memory in which the access to information is made through attributes of the item to be retrieved. As we mentioned in the last section in a learning session a set of M input patterns is presented to the network, and by some means they become associated with M output patterns. Denoting the input patterns by $\xi_p^{IN} = (\xi_{1p}^{IN}, \ldots, \xi_{Np}^{IN})$ and the outputs by $\xi_p^{OUT} = (\xi_{1p}^{OUT}, \ldots, \xi_{Np}^{OUT})$, during the learning phase a mapping must be established from the inputs to the outputs. The learning session is followed by a test session, during which the network is expected to recognize all the input patterns it was taught. Moreover, if during the test a noisy or partial input is presented, one would like an output "close" to the correct one to be associated. These features are referred to as the memory being associative or content-addressable, adaptive and noise-tolerant.

Since a Hopfield network has neither input nor output layers, the p-th input and the corresponding output pattern are "simulated" respectively by the initial state $x^{(p)}(0)$ and the final state $x^{(p)}(t)$ of the whole network. Then, if the network is initialized with an initial state $x^{(p)}(0) = \xi_p^{IN}$ at the dynamic Eq. (2.1) is demanded to produce the final state, for t large, $x^{(p)}(t) = \xi_p^{OUT}$, for $p = 1, \ldots, M$. This is done by making each $x^{(p)}(t)$ a fixed point (Ref. 10) of the dynamic Eq. (2.1), i.e., $x^{(p)}(t) = x^{(p)}(t+1)$.

The set of all initial configurations of neurons which lead to a given memory state is called *"basin of attraction"* (Ref. 10) of that memory state. It is clearly desirable to have a large enough basin of attraction around every pattern to be stored, to assure that all input patterns sufficiently close to it will be drawn to the associated fixed point by the network dynamics. This enables the reconstruction of stored information from a deteriorated or partial description. Different patterns correspond to different states of activation of the neurons. A partial pattern activates only some of the neurons. Interaction between the neurons then allows the set of active neurons to influence the others, completing thereby the state and generating the pattern that best fits the partial one.

The weight adaptability in the Hopfield model is simulated through Hebb's rule (Ref. 10,11) : one starts with no connections among the neurons and then increments the value of the connections according to the rule:

$$\Delta w_{ij} = \Delta w_{ji} = \eta \xi_i \xi_j$$

where η is a positive learning coefficient and ξ is the pattern to be stored. If there are M patterns to be stored $(\xi^{(1)}, \xi^{(2)}, \ldots, \xi^{(M)})$, the weights derived from this learning rule are

$$w_{ij} = \frac{1}{N} \sum_{p=1}^{M} \xi_i^{(p)} \xi_j^{(p)} \tag{2.2}$$

with $w_{ii} = 0$. By following Hebb's rule and by making a few extra assumption about the number of stored patterns and their statistical properties, the $\xi^{(p)}$ will be attracting fixed points of the Eq. (2.1). This will be shown in the next section.

2.1.1. Dynamics of the Hopfield model

To analyze the time evolution of a dynamical system and to prove the existence of stables states, a classical method is to associate to the system a time-varying *energy function* which is defined in terms of the current state of the system. In this case the minima of the energy function correspond to stable states for the dynamic system, i. e., fixed points for the dynamic Eq. (2.1). Hopfield (Ref. 6) observed that by adopting the dynamic rule (2.1) and assuming symmetric weights $w_{ij} = w_{ji}$ and $w_{ii} = 0$, like in Hebb's rule, one has precisely the equation describing a *Spin Glass* (Ref. 12) system at zero temperature relaxing towards equilibrium, when one flips one spin

at a time. There are in fact some common features between Spin Glasses and Hopfield Neural Networks: both are characterized by a great number of variables, each of them can assume one of two possible values, and interacting in a complex and not uniform way. Due to this analogy, the weights are often called *"bonds"* and denoted with J_{ij}, while the state of a neuron i at time t is often represented by an *"Ising Spin"* $S_i(t)$. From that, an energy function can be defined

$$E(t) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij}x_i(t)x_j(t) - \sum_{i=1}^{N}\Theta_i x_i(t) \tag{2.3}$$

Because of the symmetric form of w_{ij} and the asynchronous updating of the neuron state (one at a time), Hopfield showed (Ref. 6) that E is a monotonically decreasing function: the dynamic process evolves until a stable state (fixed point) is reached, corresponding to a local minimum of the energy function (2.3). In such a way, each pattern to be stored corresponds to a minimum of E. Since the energy function resulting from the Spin Glass model has many local minima (Ref. 12) one can also obtain a large storage capacity. Each local minimum is surrounded by a *"basin of attraction"*: if the system state corresponds to an energy value in a local minimum basin, then the system evolves towards the associated pattern. In this way, each input $x^{(p)}(0)$ determines the choice of one minimum, and recalls the pattern $\xi^{(p)} = x^{(p)}(t)$ to which they are sufficiently closely associated. That is, if the input $x^{(p)}(0)$ has a large enough *"overlap"*

$$m^{(p)}(0) = \frac{1}{N}\sum_{i=1}^{N} x_i(0)\xi_i^{(p)}$$

with the network state $\xi^{(p)}$, then the dynamics should produce the final state $x^{(p)}(t) = \xi^{(p)}$. When the noise level is too high, i.e., the initial overlap $m^{(p)}(0)$ is too small, the network relaxes to some spurious states that are not near the right output pattern. Moreover, for the choice of the weights given by Eq. (2.2) and $\Theta_i = 0$ for $i = 1, \ldots, N$, the energy function (2.3) can be rewritten as

$$E(t) = -\frac{N}{2}\sum_{p=1}^{M}(m^{(p)}(t))^2 + \frac{1}{2}M$$

where $m^{(p)}(t)$ is the overlap of the network state $x(t)$ at time t with the pattern $\xi^{(p)}$. Up to an additive constant, E is now given as the negative sum of squares of the overlap. Then a minimum of the energy function corresponds to a maximum overlap.

Intuitively, it would seem that at some point the network would become saturated, i.e., that the number of coefficients would become insufficient to uniquely specify all the memory configurations. When the number of stored patterns M is limited to (Ref. 13)

$$M \leq \frac{N}{4\log N}$$

where N is the size of the pattern (number of neurons), one can statistically show (Ref. 13) that the stored patterns are just the local minima of the energy function. When M exceeds this upper limit, a lot of these minima turn into spurious states and the network ability to store memory patterns is drastically reduced. This is a rather small number in comparison to the maximum number of patterns the network can represent (2^N). The problem becomes worse if the patterns are more correlated. In fact, if the patterns are not orthogonal the memorization may be not perfect (Ref. 10). Furthermore, Hopfield model is based on the symmetry of the synapses (links). This restricts the model to *auto-associative* recalls only, i.e., an input pattern is associated with itself and the main goal is pattern completion. In order to perform hetero-associative task, Hopfield

introduced asymmetric synapses but did not provide a corresponding energy function and therefore could not retrieve hetero-associative memory pairs completely.

Hopfield dynamic model is essentially a $T = 0$ Monte Carlo (Ref. 7,14) dynamics. Starting from an arbitrary initial configuration, the system evolves by a sequence of single-spin flips, involving spins which are misaligned with their instantaneous molecular fields. This process monotonically decreases the energy function (2.3), and leads to steady states, which are the local minima of (2.3). A natural generalization of this model to a system with noise is to adopt single-spin dynamics at a finite temperature $T = \beta^{-1}$ (Ref. 15), using the formalism of *"simulated annealing"* (Ref. 12), so that the updating rule becomes probabilistic and Eq. (2.1) is substituted by

$$P(\{x_i(t+1)\}|\{x_j(t)\}) = \prod_{i=1}^{N} P(x_i(t+1)|\{x_j(t)\}) = \prod_{i=1}^{N} [1 + e^{-2\beta x_i(t) H_i}]^{-1}$$

where $P(\{x_i(t+1)\}|\{x_j(t)\})$ is the probability for neurons to have the values $x_i(t+1)$ at time $t+1$, given the network configuration $\{x_j(t)\}$ at time t, and

$$H_i = \frac{\sqrt{M}}{N} \sum_{j=1}^{N} w_{ij} x_j(t)$$

For symmetric coupling $w_{ij} = w_{ji}$ one can also show (Ref. 16) that Eq. (2.3) tends to a stationary Gibbs distribution:

$$P(\{x(t+1)\}) \propto e^{-\beta E(t)}$$

therefore the standard methods of equilibrium statistical mechanics can be successfully applied.

2.1.2. Associative memories in HEP

An interesting way to think of associative memory is to think of data in pixel format, like an image. An image consists of a set of on and off pixels. Suppose we connect all the on pixels together with reinforcing coefficients. If we turn off some of them, the other ones will turn them back on. Thus this image is "stored" in the pixel array. In this way multiple capacity and efficiency is actually a very active field of research.

There are two potential uses in HEP (Ref. 17). One is in interrogation of HEP data bases. The information in HEP events could be represented in pixel format. If we are looking for events with certain characteristics but don't care about what other characteristics they may have, we simply specify these characteristics to the network, and if an event fulfilling them is stored in the network, it displays the event. The other use is in storing patterns for patterns recognition problems, such as storing templates for a template matching track finding algorithm. One would present an observed hit pattern to the network to see if it corresponds to one of the stored templates.

2.2. Optimization Problems

The analogies between statistical mechanics and the Hopfield model result in an interesting application of this type of neural networks in optimization problems.

The energy function minimized by a neural network can be any function expressible in terms of the states of the neurons. For this reason, the Spin Glass model is also a good model for solving optimization problems, i.e., problems where one seeks simultaneous satisfaction of a maximum number of *constraints* among hypotheses with a minimum resulting *cost*. If every hypothesis is assigned to a neuron of a neural network, then a contradiction between two hypotheses can be expressed by a negative connection between the corresponding neurons (Ref. 18). There are several *"near-optimal"* solutions, each of them can be associated with a local minimum of the

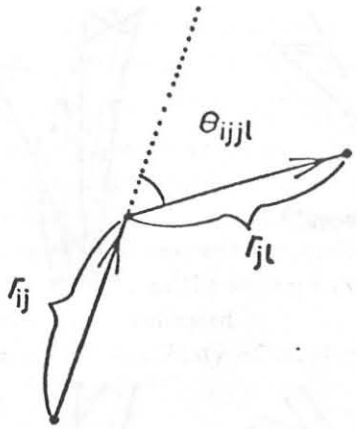

Fig. 8. Definition of segment lengths r_{ij} and angles Θ_{ijjl} between segments (from Ref. 20).

energy function. In such a case, the energy function can be interpreted as a *cost function* that must be minimized.

An example of optimization problem that can be treated with a neural network, is the *track finding* problem (Ref. 9). It will be described in the next section.

2.2.1. The track finding problem

A typical pattern recognition problem in experimental HEP is that of reconstructing tracks of charged particles in gaseous detectors (Ref. 9). The typical detector measures a set of N space points i along the track, which may be curved by a magnetic field. From each of these points a set of segments $i \rightarrow j$ will be defined subject to these conditions:

- A segment $i \rightarrow j$ connects two points i and j in the set, with $i \neq j$.
- The length r_{ij} of each segment is less than some a maximum value R_{cut}.

Thus, each point can be thought as interacting only with the points within a radius R_{cut}, to form possible track segments. A natural choice is to represent each possible segment with a neuron (Ref. 19,20).

On a track, no point should have more than one directed segment entering or leaving it, and no point should appear more than once. For this reason, inhibitive connections are imposed between neurons violating these constraints.

The reinforcing connections will be set up such that the joining together of short segments of similar direction will be favored, in order to ensure a smooth track. To define an appropriate cost function for this network, two measurements are needes: length of the segments and angles between adjacent segments, indicated in Fig. 8 (from Ref. 20).

In order to "code" this problem onto a neural network (Ref. 19,20), each segment $i \rightarrow j$ can be represented by a binary neuron ij, the state of which is $x_{ij} = 1$ if the segment $i \rightarrow j$ is part of the track, and $x_{ij} = 0$ if this is not the case. An energy function is to be minimized, that favours pairs of neurons with similar slopes, with paths as short as possible, and inhibits bifurcations of tracks.

Time evolution of the network in Ref. 20 is shown in Fig. 9.

We evaluated the performance of this approach to the track finding problem (Ref. 21), as recently proposed in the literature. The result was that this neural network approach produces solutions of good quality for modest size problems (number of tracks 5-7, number of points per track around 10). Typical results show "confusion" in regions where tracks were very close together, leading to incorrect or illegal solutions. In fact, the final solution often violates the constraint that there should be no bifurcation in the tracks. To remove these spurious solutions, a *greedy heuristic procedure* (Ref. 22) was needed, which examines all track points with more than one segment entering or leaving and removes those segments which correspond to a larger cost.

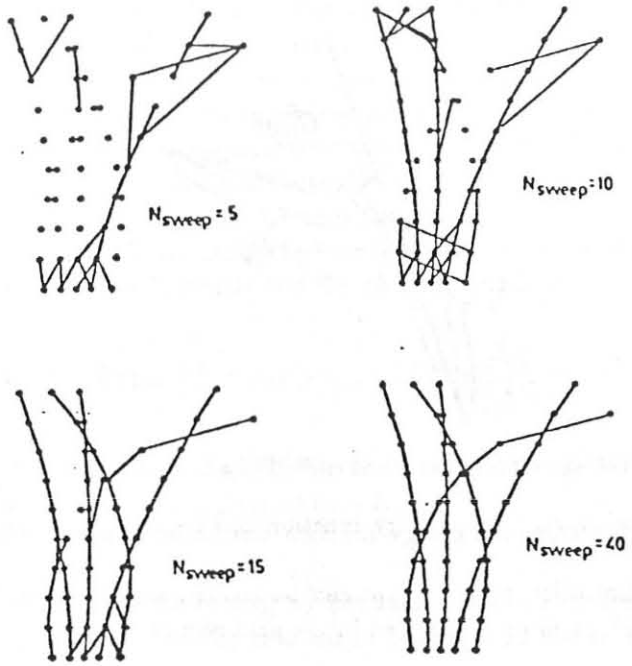

Fig. 9. Segments at different evolution stages (from Ref. 20).

An attempt to apply this method to real data from a the TPC of the DELPHI detector at LEP gave rather poor results: only a fraction of tracks were recognized, and at high costs (memory, CPU consumption).

The method is not competitive with other conventional methods for the track finding problem (Ref. 23). The real interest of this neural network approach lies in the future possibilities of implementing it in parallel hardware (Ref. 20), for fast on-line triggers.

Considerable effort was put into trying to optimize the performance by varying the form of the synaptic weights (Ref. 24). In particular, one will probably never obtain perfect performance without somehow incorporating curvature information. In this case, a possible choice for the synaptic weights is:

$$w_{ijkl} \propto f_{ijkl} \cos^n \alpha_{ijkl} \cos^n \beta_{ijkl}$$

where α_{ijkl} is the angle between the segments $i \rightarrow j$ and $k \rightarrow l$ placed tail to tail, β_{ijkl} is the angle between radial vectors to the midpoints of $i \rightarrow j$ and $k \rightarrow l$, $n = 100$, $f_{ijkl} = +1$ if $j = k$ (the two segments are tail to head), $f_{ijkl} = -1$ if $j = l$ or $i = k$ (the two segments are tail to tail or head to head) and $f_{ijkl} = 0$ otherwise. Also in this case the parallel implementation provides the best results (Ref. 24).

A further extension of the method presented is described in Ref. 25.

2.2.2. Secondary vertex trigger

The possibility to use a Hopfield neural network as a beauty trigger was investigated in the literature (Ref. 26,27), for the case study of the SVX vertex detector of the CDF experiment at FNAL.

One method on triggering on beauty are the detection of secondary vertices from the beauty decay. The ISAJET (Ref. 28) Monte Carlo was used to generate $b\bar{b}$ events and QCD background. The $b\bar{b}$ particles are allowed to decay into their secondary decay products.

The neural network approach used in this case is similar to that used for the track finding above

presented. Also in this case the neurons reinforce each other to the degree that the angles of their segments are similar, but the reinforcing connections are

$$w_{ijkl} \propto \exp(-A\theta_{ijkl}^2 - B\Delta\phi^2)$$

where A and B are positive constants, θ_{ijkl} is the absolute angle in $D-\phi$ space (D is the distance of closest approach to the origin and ϕ is the azimuthal angle of the track) between the segment $i \rightarrow j$ and the segment $k \rightarrow l$, and $\Delta\phi$ is the difference between the ϕ intercepts ($D = 0$) of the two neurons. A leakage term causes the neurons without reinforcement connections to decay to zero. Initially, all neurons are activated and as the network evolves only those segments which have neighbors of similar orientation remain activated.

Encouraging results are found on the possibility of tagging the presence of beauty via the detection of secondary vertices.

2.2.3. Other applications

A review of other applications of Hopfield-type neural networks is presented in Ref. 29.

2.3. Non-adaptive Networks

As we said in the introduction, NAN have the immense advantage of being easier to implement in hardware. This simplicity has, of course, to be payed for: non-adaptive systems are unable to tackle situations for which they were not designed for, and so they can only be used in problems where the input/output functions are completely defined. On the other hand, one trades dynamic complexity for architecture complexity, which is not always possible.

However, they can be quite useful for some specific class of problems like calorimeter cluster recognition and particle identification in Cherenkov detectors (Ref. 30). The basic idea behind both problems is the same: how to classify points which are on a given pattern. In the case of cluster recognition on reduces a spot to a single point which its center of gravity and contains all the information (total energy deposited on the spot and size of the spot). In the case of particle identification in Cherenkov detectors for every three points on a given output image one calculates the center of the circle that passes by these three points; one gets, thus, a set of spots (composed of possible centers for a circle created by a given particle) which can be reduced with the above technique.

3. Feed-forward Neural Networks

The problem in *feed-forward* (Ref. 31) neural networks is again to model some mapping between input pattern x and output pattern y. The architecture of this kind of networks is the one described in Section 1.2. for layered networks. There are $l = 1, \ldots, L$ layers with $i = 1, \ldots, N_l$ neurons per layer. Each neuron is connected to all neurons of the neighboring layers in a forward directed manner, so that the output of one neuron is connected to the inputs of neurons in the subsequent layer (feed-forward architecture). The output of each neuron is assumed to be a bounded semi-linear function of its inputs. That is, if $o_j^{(l-1)}$ denotes the output of the j-th neuron of layer $l-1$ and w_{ij} denotes the weight associated with the connection of the output of the j-th neuron to the input of the i-th of layer l, then the i-th neuron takes the value

$$o_i^{(l)} = \begin{cases} x_i, & \text{if } l = 1; \\ g\left(\sum_{j=1}^{N_l} w_{ij} o_j^{(l-1)} - \Theta_i\right) & \text{if } l = 2, \ldots, L. \end{cases} \tag{3.1}$$

where g is a bounded, differentiable and non decreasing activation function. In most cases the activation function is the sigmoid *logistic function*

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

Note that for this activation function, the output can not reach its extreme values of 1 or 0 without infinitely large weights. It is necessary to use a certain *tolerance* τ in reading out the network output in situations in which the desired outputs are binary. A typical way to implement this is to read out as 0 all output values between 0 and $0 + \tau$ and as 1 all output values between $1 - \tau$ and 1. This is equivalent to adding another threshold function to each output neuron as shown in Fig. 4. The weights w_{ij} have real values that can be positive (excitatory), negative (inhibitory) or zero (no connection). In this model, the outputs of neurons in different layers have different meanings. A set of "input neurons" are assembled in the first layer, and hence only in this layer there is the representation of the externally dictated patterns. "Hidden neurons" are assembled in one or more hidden layers. On these layers, the system is free to choose an internal representation of the input pattern. "Output neurons" constitute the last layer. This layer collects the representation of the output pattern associated to the input pattern received. The first and the last layer serve respectively as the input and output of the network. Input sets the first layer in an initial state. After its forward propagation through the network layers, the state of the last layer is read out as output. There is no meaning to stable state, however, every final state of the last layer is read out, whether it is near the right output pattern or not. This may pose the same difficulties as the spurious state of the Hopfield model. The feed-forward network described above is also called *multi-layer perceptron*. The perceptron has a long, interesting and instructive history, which is the topic of the next section.

3.1. Perceptron

The *perceptron*, proposed by Rosenblatt (Ref. 3,4) in the 1950s and 1960s, is a special case of feed-forward network with only one input layer, a single output neuron and no hidden layers. It is often called a one-layer network, referring to the single layer of weights connecting input to output. In this case, there is no internal representation of the input pattern, but only the coding provided by the external world is used. Conceptually, the perceptron may be described as a machine with an input channel for patterns, a NO/YES output indicator and a mechanism to indicate if the machine response is correct or not (see Fig. 10).

The input patterns belong to two possible classes, I_0 and I_1, and one would like the perceptron to respond NO (0) to all patterns in I_0 and YES (1) to all patterns in I_1. This can be viewed as a classification task: the perceptron classifies all possible input to either I_1 or *not*-I_1. Furthermore, the perceptron acquires this skill in a training stage, during which it adapts its parameter of decision if the given response is wrong. A first idea to implement such a machine is as a memory where the patterns are stored in two separate groups. This kind of implementation never makes a mistake on a previously seen pattern, but it is not able to take a decision on new patterns not presented during the training-memorization session. A better approach is to find the "features" that distinguish one class of patterns from the other and to use them in the classification task.

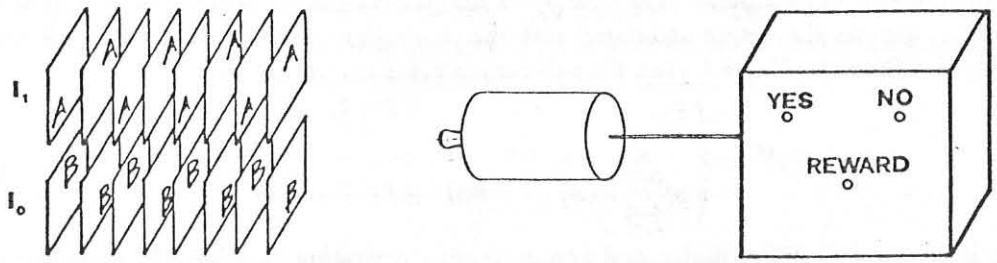

Fig. 10. A schematic representation of the perceptron architecture.

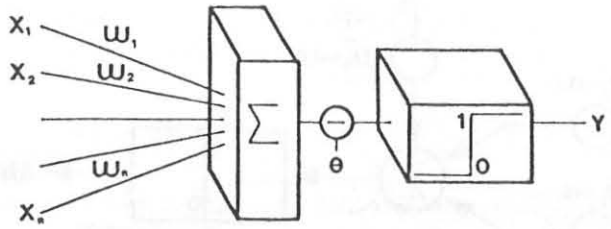

Fig. 11. Computations of the perceptron output.

In this case the machine is expected to perform well with inputs for which it was not specifically trained, i.e., to be able to *generalize*. Perceptron, as a pattern discriminating machine, lies between these two cases. It does not simply store the patterns, but it is also limited to what can be done with no internal representation of the features and linear threshold relations between these features. Therefore, as a compensation for this limitation, a simple learning procedure results from these restrictions. This is what will be shown in the following section.

3.1.1. Definition

Let $x_p = (x_{p1}, \ldots, x_{pN})$ be a binary array, i.e., $x_{pi} = 0, 1$ for $i = 1, \ldots, N$, represents the binary coding of a pattern p. This array is presented to the perceptron through the N units (neurons) that constitute its input layer, then it is forwarded into the single output unit that represents the NO/YES machine indicator. The output unit is a *linear threshold* element, which takes the value $y = 0, 1$ according to the rule

$$y = H(\sum_{i=1}^{N} w_i x_i - \Theta) \tag{3.3}$$

where Θ is the output threshold and w_i the "weight" of i-th input on the output. This computation is shown in Fig. 11.

There is quite a lot that such a simple perceptron can do. For example, if $w_i = 1$ for $i = 1, 2$ and $\Theta = 1.5$, the output unit will have the value $y = 1$ only in response to an input in which both binary elements are equals to 1. In this case, the logic AND is computed:

$x_1 x_2$	$\sum_{i=1}^{2} w_i x_i - \Theta$	y
0 0	$0 - 1.5 < 0$	0
0 1	$1 - 1.5 < 0$	0
1 0	$1 - 1.5 < 0$	0
1 1	$2 - 1.5 > 0$	1

Neuronal implementation of AND and OR function are shown in Fig. 12.

But the most interesting thing is the adaptability of the weights w_i and threshold Θ. The perceptron modifies these decision parameters to conform better its correct response.

In the following, the perceptron output rule is assumed to be

$$y = H(\sum_{i=1}^{N} w_i x_i) \tag{3.4}$$

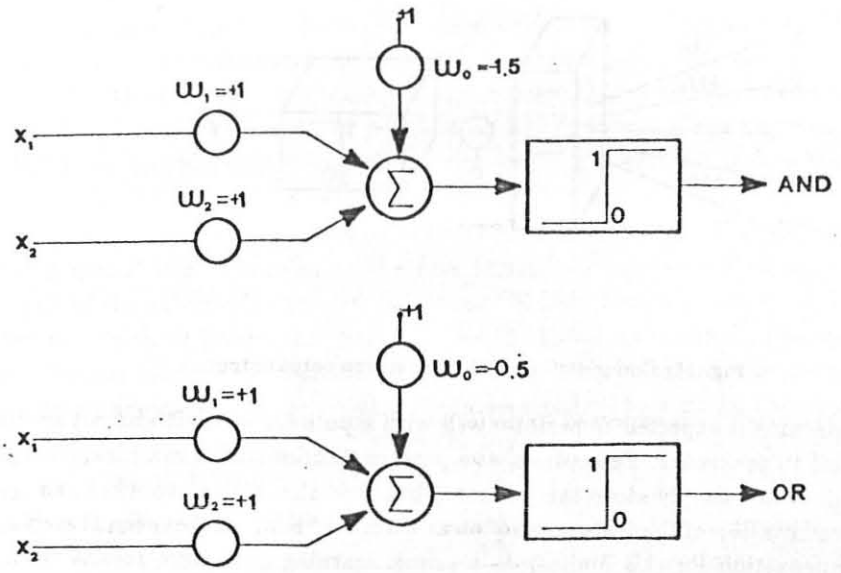

Fig. 12. Neuronal implementation of AND and OR functions.

instead of rule (3.3). This simplification does not affect the discussion, since Θ can be thought as the weight from an input unit that is always set to one. In this case, Θ can be learned just like any other weights. The perceptron ability to determine a set of weights w_i that will ensure solution of an assigned task, will be demonstrated in the next session.

3.1.2. Perceptron learning rule

To start the learning session, the perceptron is initialized: all parameters are assigned some random value. Now, two groups I_1 and I_0 of input patterns are prepared: I_1 is constituted by the patterns for which the perceptron response must be 1, and I_0 by the pattern that should elicit the opposite response. The set of these two groups of patterns is called the *training set.* Now, pick randomly a pattern from one of the two groups and present it to the perceptron. Eq. (3.4) determines the corresponding response of the output unit. Depending on this response, the weights are modified according to the *perceptron learning rule*, which can be stated as follows:

(i) Correct responses generate no change.

(ii) If the response is 0 while the right one is 1, then

$$\Delta w_i = \eta x_i$$

(iii) If the response is 1 while the right one is 0, then

$$\Delta w_i = -\eta x_i$$

where $0 < \eta < 1$ is the *learning coefficient.* This learning rule is easy to understand. If x belongs to I_1 (I_0) and the weighted sum $\sum w_i x_i$ is positive (negative), the perceptron response is 1 (0) and all is well. Case 2 occurs when the sum is too small and then it must be increased. The $x_i = 0$ does not affect the sum, then their weights can not be blamed for the bad response. Changing these weights might do harm in relation to the other patterns and does no good in relation to the current one. Thus, the weights w_i to increase are only those associated to the $x_i = 1$. The opposite is achieved in the case 3. The previous learning rule can be expressed in a single updating equation

$$\Delta w_i = \eta(t - y)x_i \tag{3.5}$$

where y is the output produced by the input x, while t is the desired one, or *target*. The most interesting aspect of this extremely simple rule is the existence of the associated *perceptron convergence theorem* (Ref. 5).

3.1.3. Perceptron convergence theorem

Theorem 1. If there exists a solution w^*, then the perceptron learning rule will converge to some solution w in a finite number of steps for any initial choice of the weights.

How does the perceptron solve a classification problems? Each input pattern is translated into a N-dimensional array, with binary components. Hence, the patterns belonging to I_1 and I_0, can be represented as points in a N-dimensional space. The perceptron tries to find a hyperplane in this space separating the two sets I_1 and I_0. In these terms, the theorem 1 can be reformulated as

Theorem 2. If the sets I_1 and I_0 are separable, then the perceptron learning rule will separate them.

Now, one can observe that the restriction on x that its components be either 0 or 1, is not really necessary. Let x be an array whose components may take any real, positive or negative value. Since the weights updating, given by learning rule (3.5), is proportional to x, it may be overwhelmed by x too large or stalled by x too small. To solve this problem, instead of using x itself, the unit-length array $\hat{x}$ can be used

$$\hat{x} = (\hat{x}_1, \ldots, \hat{x}_N) = (\frac{x_1}{|x|}, \ldots, \frac{x_N}{|x|}) \text{ so that } |\hat{x}| = 1$$

The problem of finding a separation between the two real sets I_1' and I_0' is not really different from the problem of separating the two binary sets I_1 and I_0. In this case also, there exists a convergence theorem

Theorem 3. If there exists a unit solution w^* and a number $\delta > 0$ such that $\sum w_i^* \hat{x}_i > \delta$ for all $\hat{x} \in I_1'$ and $\sum w_i^* \hat{x}_i < -\delta$ for all $\hat{x} \in I_0'$, then the perceptron learning rule will converge to some solution w in a finite number of steps for any initial choice of the weights.

A more substantial variation is obtained by allowing more than two sets of patterns. Let $I_1, I_2, \ldots, I_M$ be sets of patterns and suppose that there are vectors w_j^* and $\delta > 0$ such that

$$x \in I_j \text{ implies that } \sum_{i=1}^{N} w_{ji}^* x_i > \sum_{i=1}^{N} w_{ki}^* x_i + \delta \text{ for all } k \neq j$$

The perceptron convergence theorem generalized to this case assures that vectors w_j with the same property can be found by following the usual principle of updating

$$\text{If } x \in I_j \text{ and } \sum_{i=1}^{N} w_{ji}^* x_i < \sum_{i=1}^{N} w_{ki}^* x_i \text{ for some } k$$
$$w_j^* \text{ must be increased and } w_k^* \text{ must be decreased}$$

Note that this generalization involves more than one single output unit in the output layer. The output layer must collect M output units, one unit for each class.

Having a simple and transparent learning rule and an associated convergence theorem, the perceptron is quite impressive. However, Minsky and Paper (Ref. 5) demonstrated that there is a large class of tasks that perceptron is unable to perform. First of all, the convergence theorem starts with an *"if"*, and then when there is no solution, the learning algorithm will not converge. Not every sets of points are linearly separable, hence the class of problems that can not be solved by the perceptron. A classic example of insolvable problem is the *exclusive-or* (XOR) problem as shown in Fig. 13.

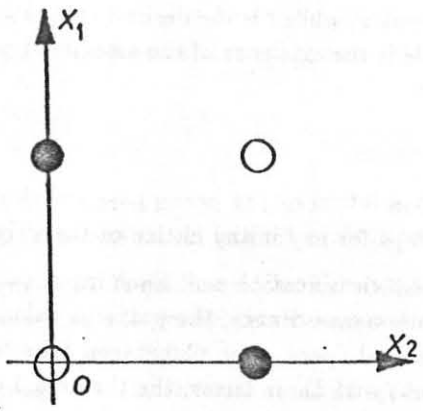

Fig. 13. No straight line can separate the solid points versus open points.

In what follows, a heuristic discussion on a well known statistical approach to classification based on Bayes' rule (Ref. 32), is proposed. The aim is to find points of contact with perceptron and then to understand what are the extensions that enable perceptron to solve problems like XOR problem.

3.1.4. Bayes' decision rule

Suppose one picks a pattern from one of some possible classes C_j, with $j = 1, \ldots, M$. Let the selection be based on some methods like, for example, tossing a coin in the case of two classes. Each class is thus selected with a priori probability $P(C_j)$, where, of course

$$\sum_{j=1}^{M} P(C_j) = 1$$

The extracted pattern x is an array of random variables. Since each x_i has been chosen to try to separate the classes, then its distribution is presumably different from the patterns of one class than from the patterns of the other classes. Thus, for each x_i, M *conditional densities* $p(x_i|C_j)$ exist, and for each pattern x, M *joint conditional densities* $p(x|C_j)$. Now, it is possible to compute the *a posteriori densities* using Bayes' rule (Ref. 32)

$$p(C_j|x) = \frac{p(x|C_j)P(C_j)}{p(x)} \tag{3.6}$$

where $p(x) = \sum_{j=1}^{M} p(x|C_j)P(C_j)$ assures that $\sum_{j=1}^{M} p(C_j|x) = 1$. The Bayes' decision rule states

$$\text{Decide } C_j \text{ if } p(C_j|x) > p(C_k|x) \text{ for all } k \neq j \tag{3.7}$$

In other words, if a particular x has occurred, this procedure looks for which C_j is the most likely and then asserts that x belongs to that C_j. This rule minimizes the probability of error in such cases where all the a priori probabilities are known and there is no other information. There are serious practical obstacles in the use of Eq. (3.6). A solution (Ref. 5) can be obtained by making the critical assumption that the random variables x_i are statistically independent over each class. For this hypothesis

$$p(x|C_j) = \prod_{i=1}^{N} p(x_i|C_j) \text{ for } j = 1, \ldots, M$$

Define

$$P_j = P(C_j)$$
$$p_{ij} = p(x_i = 1|C_j)$$
$$q_{ij} = 1 - p_{ij} = p(x_i = 0|C_j)$$

Now, suppose a x has just been observed, the decision rule (3.7) will choose that C_j which maximizes

$$P_j \prod_{x_i=1} p_{ij} \prod_{x_i=0} q_{ij} = P_j \prod_{i=1}^{N} p_{ij}^{x_i} q_{ij}^{(1-x_i)} = P_j \prod_{i=1}^{N} (\frac{p_{ij}}{q_{ij}})^{x_i} \prod_{i=1}^{N} q_{ij}$$

Since log is an increasing function, the decision rule (3.7) will select the largest of

$$\sum_{i=1}^{N} x_i \log(\frac{p_{ij}}{q_{ij}}) + (\log P_j + \sum_{i=1}^{N} \log q_{ij})$$

Because the term enclosed in parenthesis is a constant that depends only upon the class C_j and not upon the observed x, it is possible to write decision (3.7) as

$$\text{Decide } C_j \text{ if } \sum_{i=1}^{N} w_{ij} x_i + \Theta_j > \sum_{i=1}^{N} w_{ik} x_i + \Theta_k \text{ for all } k \neq j \tag{3.8}$$

and then

$$\text{Decide } C_j \text{ if } \sum_{i=1}^{N} (w_{ij} - w_{ik}) x_i - (\Theta_k - \Theta_j) > 0 \text{ for all } k \neq j$$

which has the form of the familiar linear threshold function. This suggests to design a *layer-machine* like that of Fig. 14.

Each pattern to be classified is collected from the first layer, while each element $\sum_j$ of the second layer computes the "decision quantity" $\sum w_{ij} x_i + \Theta_j$. The single element of the last layer simply decides which of its inputs is the largest. If the a priori probabilities (weights w_{ij}) are not known, they can be "estimated" in a *training* phase, during which a sequence of n patterns is considered and the number of "favourable" patterns is counted for each class. All these analogies with perceptron theory lead to a question: what happens introducing other layers between the two input and output layers of perceptron?

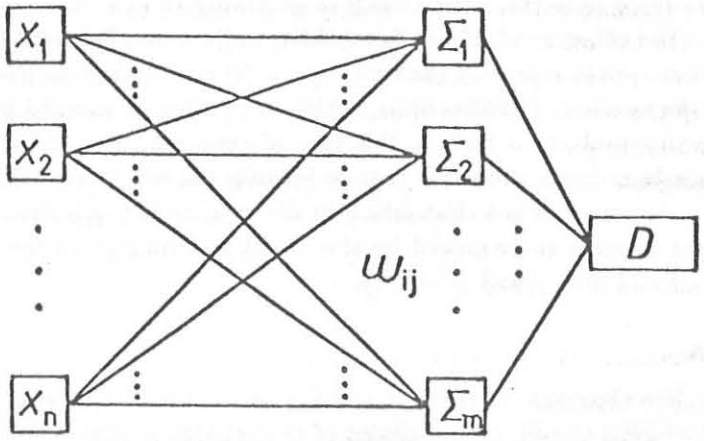

Fig. 14. Layered machine implementing Bayes' decision rule.

3.1.5. Multi-layer perceptron

Since input units and output units assume values in the same domain, one can think of connecting perceptrons together in such a way that the outputs of one are the inputs for the others. In

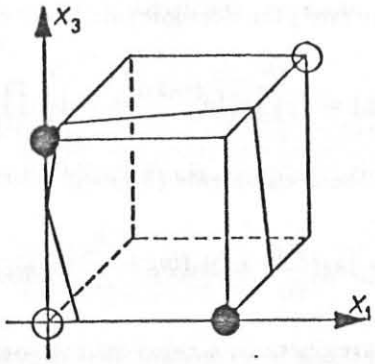

Fig. 15. A linear separation between solid and open circles is possible.

1960, 1961 Gamba (Ref. 33,34) described a type of perceptron in which each x_i is itself computed by the threshold function of Eq. (3.3)

$$y_k = H(\sum_{i=1}^{N} w_{ki} H(\sum_{j=1}^{N} w'_{ij} x'_j - \Theta'_i) - \Theta_k)$$

Gamba machines could be described as a two layers perceptron. Indeed, Fig. 15 demonstrated that inserting a hidden layer of three neurons between input layer and output layer, with the weights and thresholds indicated, a perceptron solution to the XOR problem is possible.

In this way, the four points of two-dimensional space are projected in four points in three-dimensional space. These four points are now easily separable by a plane in the two desired groups. This demonstrates that adding hidden layers increases the class of problems that are solvable by perceptron networks.

A first question one can ask is: how many hidden units would be optimal for a particular problem? If too many are allocated, it is not only wasteful but could also negatively affect the performance of the network. In fact, since too many hidden units imply too many free parameters to fit specifically the training patterns, their ability to generalize to new "test" patterns would be adversely affected. On the other hand, if too few hidden units were allocated, then the network would not have the power even to represent the training set. There exists no satisfactory theoretical basis for determining the number of hidden units, which must often be decided by trial and error.

Another more worrying problem is that, in this type of networks, the perceptron learning rule is not suitable. In single layer perceptron it is easy to identify the weight that is too strong or too weak. For multi-layer networks it is not clear which of the weights is responsible for mistakes and successes. This problem appears to be solved by the recent introduction of the *back-propagation* algorithm by Rumelhart and Maclelland (Ref. 31).

3.2. Back-Propagation

In the *back-propagation* algorithm (Ref. 31) a training pattern is first forward propagated from input to output, after which a certain measurement of the network output error is *backward* propagated from output to input. The general idea is to update the weights on the basis of two quantities: one depending on the input to that weight (like in the perceptron learning rule) and the other depending on the desired output (target) and the actual one. A natural choice for the second quantity is simply a difference between target and actual output, i.e., the output error. An updating rule having these characteristics is

$$\Delta_p w_{ij} = \eta x_{pj}(t_{pi} - o_{pi}) = \eta x_{pj} \delta_{pi} \qquad (3.9)$$

This is called Widrow-Hoff or δ-rule (Ref. 35). It can only be used in networks without hidden units.For each pattern p, the input x_p produces the output o_p, which is compared with the desired one, the target t_p. If there is no difference, no learning takes place, otherwise, the weights are updated to reduce the difference $(t_{pi} - o_{pi})$.

Given a training set of patterns pairs (x_p, t_p), the delta-rule attempts to adjust the weights so as to minimize the squares of the differences between the actual output o_p and the target t_p, summed over the output units and all pairs of input/target patterns. Let

$$E = \sum_p E_p = \sum_p \frac{1}{2} \sum_{i=1}^{N} (t_{pi} - o_{pi})^2 \tag{3.10}$$

be the measure of the error on all input/output patterns p. The delta-rule implements a *gradient descent* in E when the outputs are linear, that is to say

$$o_{pi} = \sum_{j=1}^{N} w_{ij} x_{pj} \tag{3.11}$$

The derivative of E_p with respect to the weight w_{ij} can be computed, using the chain rule, as the product of two quantities: the derivative of the error with respect to the output multiplied by the derivative of the output with respect to the weight

$$\frac{\partial E_p}{\partial w_{ij}} = \frac{\partial E_p}{\partial o_{pi}} \frac{\partial o_{pi}}{\partial w_{ij}}$$

The first quantity tells how much the output of the i-th unit affects the error and the second quantity tells how much this output is changed by changing w_{ij}. From Eq. (3.10)

$$\frac{\partial E_p}{\partial o_{pi}} = -(t_{pi} - o_{pi}) = -\delta_{pi} \tag{3.12}$$

Moreover, from Eq. (3.11)

$$\frac{\partial o_{pi}}{\partial w_{ij}} = x_{pj}$$

then, one can write

$$-\frac{\partial E_p}{\partial w_{ij}} = \delta_{pi} x_{pj} \tag{3.13}$$

Since

$$-\frac{\partial E}{\partial w_{ij}} = -\sum_p \frac{\partial E_p}{\partial w_{ij}} = \sum_p \delta_{pi} x_{pj}$$

it is possible to conclude that the delta-rule implements an approximation to gradient descent in E, since the weights are changed after each pattern p is presented. Nevertheless, if the learning coefficient η is sufficiently small, the delta-rule implements a very close approximation to gradient descent in E and then it finds a set of weights minimizing E. A choice of η too large manifests itself by oscillations around the minimum, leading to a non-convergence of the minimization process.

The important contribution due to Rumelhart, is how to implement Eq. (3.9) in hidden units, for which there are no target values directly available, and for not linear outputs. Rumelhart proposed a generalized version of Eq. (3.9), called *generalized delta-rule*. Define the *net total output*

$$net_{pi} = \sum_{j=1}^{N} w_{ij} o_{pj} \tag{3.14}$$

where $o_{pj} = x_{pj}$ if j is an input unit. Thus, a semi-linear unit is one in which

$$o_{pi} = g(net_{pi}) \tag{3.15}$$

and g is differentiable and not decreasing. To get the correct generalization of delta-rule

$$\Delta_p w_{ij} \propto -\frac{\partial E_p}{\partial w_{ij}}$$

Using the chain rule

$$\frac{\partial E_p}{\partial w_{ij}} = \frac{\partial E_p}{\partial net_{pi}} \frac{\partial net_{pi}}{\partial w_{ij}} \tag{3.16}$$

From Eq. (3.14)

$$\frac{\partial net_{pi}}{\partial w_{ij}} = o_{pj}$$

If δ_{pi} is defined as[a]

$$\delta_{pi} = -\frac{\partial E_p}{\partial net_{pi}} \tag{3.17}$$

Eq. (3.16) has the same form of Eq. (3.13). Moreover

$$\delta_{pi} = -\frac{\partial E_p}{\partial net_{pi}} = -\frac{\partial E_p}{\partial o_{pi}} \frac{\partial o_{pi}}{\partial net_{pi}}$$

By Eq. (3.15), the second factor of the previous derivative is

$$\frac{\partial o_{pi}}{\partial net_{pi}} = g'(net_{pi})$$

To compute the first factor, one must distinguish between the two cases that i is an output unit or not. In the first case

$$-\frac{\partial E_p}{\partial o_{pi}} = (t_{pi} - o_{pi})$$

and from the definition (3.17) of δ_{pi}, then

$$\delta_{pi} = (t_{pi} - o_{pi}) g'(net_{pi}) \tag{3.18}$$

from any output unit i. If i is not an output unit

$$-\frac{\partial E_p}{\partial o_{pi}} = -\sum_k \frac{\partial E_p}{\partial net_{pk}} \frac{\partial net_{pk}}{\partial o_{pi}} = -\sum_k \frac{\partial E_p}{\partial net_{pk}} w_{ki} = \sum_k \delta_{pk} w_{ki}$$

then

$$\delta_{pi} = (\sum_k \delta_{pk} w_{ki}) g'(net_{pi}) \tag{3.19}$$

Eq. (3.18) and (3.19) give a recursive procedure to compute δ_{pi}, by a *back-propagation* of the error signals through the network. These results can be summarized in three equations. First, the weights updating has the same form of delta-rule in Eq. (3.9)

$$\Delta_p w_{ij} = \eta \delta_{pi} o_{pj}$$

[a] Note that, since $o_{pi} = net_{pi}$ when the output of unit i is linear, this definition of δ_{pi} is consistent with that of Eq. (3.12).

The other two equations specify the error signal, determined recursively starting from the output unit

$$\delta_{pi} = \begin{cases} (t_{pi} - o_{pi}) g'(net_{pi}) & \text{if } i \text{ output unit;} \\ (\sum_k \delta_{pk} w_{ki}) g'(net_{pi}) & \text{otherwise.} \end{cases}$$

In the next section, a discussion on practical application of this generalized delta-rule is presented.

The implementation of the generalized delta-rule results in two phases, denoted by forward and backward propagation (Ref. 31). First, the weights are initialized with small random values. During the forward propagation, the input is propagated forward through the network to produce the output, that will be compared with the target to compute the error signal. A backward propagation follows, during which the error signal for a hidden unit (for which there is no target directly available) is computed recursively in terms of the error signal of the units directly connected and the weights of these connections. In this way, the larger the difference between output and target and the larger the error signals. The updating of the weights is then made with respect of these error signals.

Since the derivative of the activation function is used in the error signal computation, the discontinuous threshold function on which the perceptron is based, is not good. Moreover, a linear activation function is not sufficient either, because in this case only linear separation is achieved. A nonlinear, continuous activation function is then necessary, like, for example, the logistic function

$$g(net_{pi}) = \frac{1}{1 + e^{-net_{pi}}}$$

It is easy to demonstrate that the derivative or this function with respect to its total input net_{pi}, is given by

$$g'(net_{pi}) = g(net_{pi})(1 - g(net_{pi})) = o_{pi}(1 - o_{pi})$$

This derivative reaches its maximum for $o_{pi} = 0.5$ and, since $0 < o_{pi} < 1$, approaches its minimum as o_{pi} approaches zero or one. Since the weights updating is proportional to this derivative, weights will be changed more for those units whose output is near 0.5, i.e., not yet on or off.

The learning coefficient η must be sufficiently small to better approximate the gradient descent procedure and to avoid oscillation. Nevertheless, a greater η implies a more rapid learning. One way to increase η avoiding oscillation danger, consists of including a *momentum* term in generalized delta-rule updating

$$\Delta w_{ij} = \eta(\delta_i o_j) + \alpha \Delta^{old} w_{ij}$$

where $0 < \alpha < 1$ is the so-called momentum coefficient, which determines the effect of past weights updating on the current ones.

In case of linear output and no hidden layers, the error surface is concave with only one minimum, so the gradient descent procedure implemented by delta-rule, is guaranteed to find it. This is not the case with the generalized delta-rule, which suffers from the problems of hill-climbing procedures, i.e., the danger of getting stuck in some local minimum. In fact, no convergence theorem of those mentioned before regarding the single-layer perceptron, exists for back-propagation. However, in many problems the generalized delta-rule has demonstrated good performances in finding a solution. One of these problems is the *parity problem*. In this problem, one wants a classification scheme that differentiates input with an even number of 1's from those with an odd number. This problem is a fairly difficult one, since changing any single input unit throws the output from one class to the other. The XOR problem is a parity problem of size two. Rumelhart studied this problem for different numbers of input units (Ref. 31). A solution found by the back-propagation procedure is shown in Fig. 16.

A feed-forward network solution requires at least N hidden units for input patterns of size N. In Fig. 16, the unbroken lines indicate weights $w_{ij} = 1$, while the broken ones indicate $w_{ij} = -1$.

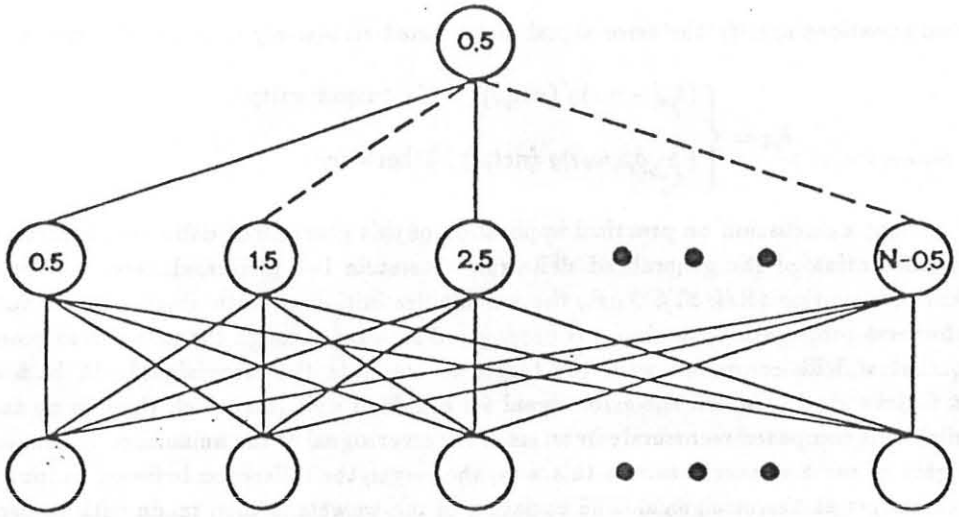

Fig. 16. Solution to the parity problem found by the back-propagation learning.

The number in the circles represent the threshold of the unit. The hidden units arrange themselves to count the number of 1 in the input. If there are m 1 in the input pattern, then the first m hidden units are on while the subsequent are off. The hidden units are connected to the output unit with alternately $+1$ and -1 weights, so that the weighted sum of hidden units outputs is 1 if there are an even number of 1's in the input pattern, otherwise is 0.

A certain number of theoretical analyses have been performed to determine the capabilities of classifiers formed from multi-layer networks. Similar constructive proofs, developed independently (Ref. 36,37,38), demonstrated that two hidden layers are sufficient to form arbitrary decision regions using multi-layer networks with step function. A careful mathematical proof (Ref. 39) demonstrated that using sigmoidal function in multi-layer networks with only one hidden layer, arbitrary decision regions can be approximated. This proof, however, is not constructive and does not indicate how many units are required in the hidden layer.

3.3. Optimizations of Back-Propagation

Although back-propagation with the generalized delta-rule has been very successful in a number of applications, it has a number of drawbacks. First of all the algorithm is not guaranteed to find the global minimum of the energy function. Therefore the network may get stuck in a local minimum. Next to this there exists no method of finding the optimal values for the network parameters. The number of hidden layers and hidden nodes, the temperature, the learning rate and the momentum term have to be determined by trial and error, thus leading often to non optimal values and slow convergence of the network. Finally back-propagation shows bad scaling behaviour. This means that when we increase the number of nodes, the computing time needed for training a network grows explosively, making the method useless for practical applications.

To overcome these problems a number of refinements to back-propagation have been proposed. We discuss some of these suggestions that leave the concept of local computation in the network intact.

3.3.1. Adaptive Back-Propagation

In this section we discuss some strategies to speed up back-propagation (BP) by finding the optimal learning rate for every weight individually. The heuristics for this approach are given by

Jacobs (Ref. 40).

(i) Every weight should have its own learning rate η. The optimal learning rate for a certain weight might be bad for another one.

(ii) The learning rate should be allowed to vary in time. As training goes on the error surface changes and so will the optimal learning rate.

(iii) When the change of a weight has the same sign for consecutive steps, the learning rate for that weight can be increased.

(iv) When the change of a weight changes sign, the learning rate for that weight should be decreased.

Three different algorithms based on these heuristics have been developed:
Self-Adapting back-propagation (SAB) (Ref. 41), SuperSAB (Ref. 42) and the delta-bar-delta rule (Ref. 40). We give the recipe for one of them, namely SuperSAB and present the benchmark results for the three algorithms.

In SuperSAB each weight w_{ij} has its own learning rate η_{ij}. They are all initialized to a certain value η_{start}. We define the increase factor η_+ and the decrease factor η_- for the learning rate. From here we take the following steps:

(i) Set all η_{ij} to η_{start}.

(ii) Do normal BP step n with a momentum term.

(iii) For every w_{ij}, as long as the weight change keeps the same sign, set
$\eta_{ij}^{n+1} = \eta_+ * \eta_{ij}^n$

(iv) If the weight change has a different sign:
   a) Undo the previous weight update
   b) Set $\eta_{ij}^{n+1} = \eta_- * \eta_{ij}^n$
   c) Set $\Delta w_{ij}^{n+1} = 0$.

A comparison of SuperSAB with SAB and standard BP has been made by Tollenaere (Ref. 42). The three algorithms were tested on a auto-association problem on a 10-10-10 network and on a set of random patterns on a 10-5-2 network. On these examples SuperSAB was mostly significantly faster than standard BP with optimal η and α. The speed of standard BP is also very sensitive to the choice of these parameters, where SuperSAB allows a wide range.

SuperSAB is slightly more instable. In general normal SAB is performing worse than SuperSAB and is behaving in a less stable way. SuperSAB shows a much better scaling behaviour than standard BP. The values user for η_- and η_+ used in the test were 2.0 and 1.05.

Tests done with Jacobs delta-bar-delta rule show similar results. Standard BP with and without momentum term has been compared to delta-bar-delta with and without momentum term on three different problems. One is the well known XOR task, one is a 6-6-6 network with should perform a multiplexer task and the third one is a 3-1-8-8 binary-to-local decoder (a three bit binary value should be converted to one bit set in a word of 8 bits). For all algorithms and problems the network parameters were tuned to get a good performance. The delta-bar-delta method converged twice as fast as BP on the first problem, slightly faster on the second one and 9 times as fast on the third one. Also the delta-bar-delta method proved to be less sensitive to the choice of the network parameters.

3.3.2. Other methods

In addition to adaptive BP some other methods have been tried to improve the network convergence. One of the simplest is derived from the simulated annealing method in Boltzmann networks. The temperature and the learning rate of the network is not kept constant during the training phase, but are slowly decreased. This allows faster learning in the beginning and avoids oscillations at the end of the training phase. An example of such a method can be found in Ref. 51.

Another method by Samad (Ref. 43) involves modifying the learning rule. In the delta-rule the weights are updated using the error on the destination node i and the actual value of the source node j ($\Delta_p w_{ij} = \eta \delta_{pi} o_{pj}$). In the hidden layer we can use the expected value of the source node

instead of the actual one since we can compute it with $\delta_{pi} = (\sum_k \delta_{pk} w_{ki}) o_{pk}$. The alternative weight update rule becomes then: $\Delta_p w_{ij} = \eta \delta_{pi} (o_{pj} + \delta_{pj})$.

A comparison between the standard and alternative learning rule has been made for the XOR task with a different number of layers (Ref. 43). In the test the alternative method converged twice as fast on a 2-10-1 network. This number is increasing dramatically with the number of hidden layers. The disadvantage of the alternative rule is that the errors δ_{pi} need to be computed twice. Once to correct the source node value and once as the error on the destination node of a weight.

3.4. Applications of Feed-Forward Neural Networks to Classification Problems

The feed-forward neural network model is well suited for pattern classification tasks and thus an interesting tool for the classification of, e.g., events, in High Energy Physics experiments. Pattern classification (Ref. 9) in general can be defined as follows:

Given an object o, one wants to associate this specific object with one of several classes $C_1, \ldots, C_M$. A class is formed by all those objects fulfilling certain criteria which define the class itself. First of all, the object must be described through a finite number n of quantities selected to be useful for distinguishing between classes. These quantities are the result of measurements with a *transducer* T, and computation based on this result. The n quantities form a pattern x, which can be seen as a point in an n-dimensional space: the *pattern space* P. In general, classes may share patterns, but the main interest is in disjoint classes. In such a term, the task of pattern classification consists of partitioning the pattern space P into disjoint regions, one region for each class. In many cases, the classification is not really done in P, but in another more convenient space F, called the *features space*. Each pattern x is transformed in an m-dimensional features vector z, where the transformation can be linear or non-linear. The major purposes of this transformation are

- To reduce the dimension of the vector/space to be studied ($m < n$), without losing significant information. In fact, frequently some components of x can be correlated in single data, since the objects will only have a rather small number of significant features.
- To obtain vector components which are better suited for pattern classification than the original ones.

This transformation is performed by a *features extractor* E, whose general aim is to reduce the complexity of pattern classification. Finally, a *classifier* C splits F into disjoint regions that indicate the classes to which the patterns belongs. The entire process can be mathematically described by a set of mappings. Each object is associated to a point o in object space O. The transducer represents each o mapping it into a pattern x in pattern space P. A features extractor transforms each x into a point z in features space F. Finally the classifier maps each z into a class designator d in decision space D:

$$\begin{array}{ccccccccc} O: & \xrightarrow{T} & P: & \xrightarrow{E} & F: & \xrightarrow{C} & D \\ o & & x & & z & & d \end{array}$$

This division of the problem into representation, features extraction and classification is arbitrary and the entire process can be viewed as a single mapping from object space to decision space.

About representation there is little to say, since this topic is extremely problem dependent. Generally the purpose of this pre-processing is to perform a first reduction from a mass of raw data to just those informations that are thought to be useful for distinguishing between classes. Unfortunately, automatic procedures which use the a priori knowledge about the specific problem are not always available. Normalization of input data and suppression of detail which may obscure the classification, are performed to reduce noise. Finally, the processed data are formatted to a form suitable to subsequent analysis.

Instead, there are general methods of approach to features extraction and classification. The features should be invariant or at least insensitive to irrelevant variation, such as limited amounts of translation, rotation, scale change, etc. while emphasizing difference that are important for distinguishing between pattern of different types. Assume that a sufficiently large amount of information x_i has somehow been obtained and assembled in pattern x. Taken together, these quantities are supposed to contain the information needed for classification, but some of them can be unpractical to use or less important than others. Feature selection methods seek a small number of x_i by obtaining a subset from the original one, by discarding irrelevant information while keeping the important features. Dimensionality reduction methods obtain a smaller number of x_i by forming, usually linear, combinations of the original ones.

Once a set of features has been selected, the only remaining problem is to design the classifier. The optimum classification is one for which all the patterns are associated to the proper class designator. Unfortunately, this is only possible in extremely simple situations. Besides that, a classifier that performs well on a set of patterns is not ensured to perform so well on a new set. This suggests that the classification problem has an important statistical component (Ref. 32) and that perhaps one should look for a classification procedure that minimizes the probability of error. In such terms, the pattern classification becomes a problem in statistical decision theory. The conventional Bayes' classifier (Ref. 32) characterizes classes by their probability density functions on the input features and uses Bayes' decision theory to decide to which class the input belongs. To implement Bayes' classifier the a priori probabilities and conditional densities must be known and in most pattern classification situations this is not the case. Usually, however, sample patterns from each class are available and the necessaries probabilities can be estimated from the samples.

In general, samples of training pattern can be used to design the features extractor and the classifier. After the training phase, new test patterns are used to evaluate the efficiency of classification. It is important to note that test data should never be used during training phase, since this produce an overly optimistic estimate of the real error rate. Test data must be independent data that are only used to asses the generalization, defined as the error rate on patterns never seen before. The more complete is the training set and better results are achieved. Rather than focusing on amount of training samples, it is better to concentrate to the quality and representativeness of them. A good training set should contain routine, unusual and boundary-condition cases. Gathering the best possible training data improves training and ensures the best possible result from the process.

Feed-forward layered networks and their learning procedure are well suited for pattern classification (Ref. 31). The internal representation of input pattern into the hidden layers can be seen as a sort of features extraction. In this case, the features are the result of weighted sums and linear or non-linear threshold functions. The weights, which can give some indication on the importance of an input data for classification, are learned during a training phase. The learning is performed with supervision, since each training pattern is associated to a label specifying the correct class (target). Features themselves are subject to subsequent elaboration and forward propagation through the layers until a convenient features representation is reached. In the last layer the classification is then performed, using simple threshold functions.

In the event classification problem, one tries to find an efficient mapping between some observed kinematical variables describing multiparticle production and well separable features. This map is learned using back-propagation on a set of training samples. After training, the network generalization is tested on an independent set. Both sets are generated with a Monte Carlo program. The procedure is then tested on different Monte Carlo models, to check its model independence (Ref. 44).

Public-domain software implementing the tuning of a feed-forward net is nowadays widely available (Ref. 45). Correspondingly, feed-forward nets have been applied to a large number of classification problems, in a "standard" way. We give below a summary of such applications. This

cannot be complete due to the growing interest on the subject.

- Quark-gluon Jet separation in e^+e^- collisions
  - (Ref. 44,45,46) *Three-layers Neural Network. No detector effects kept into account. Results on quark/gluon jet separation and on $b\bar{b}$ - other quarks separation are stable with respect to the model used for training and testing the network (ARIADNE 3.1 (Ref. 47), HERWIG 3.4 (Ref. 48) and JETSET 7.2 (Ref. 49)). Center of mass energies of 29 and 92 GeV.*
  - (Ref. 50) *Three-layers neural network. Tested on simulated data for the DELPHI detector at LEP.*
- b-quark tagging in e^+e^- collisions near the Z mass
  - (Ref. 44) *Three-layers Neural Network. No detector effects.*
  - (Ref. 51) *Three-layers network. Simulation of an average LEP detector, as in Ref. 52.*
  - (Ref. 53) *Three-layers network. Tested on simulated data for the ALEPH detector.*
- General flavour classification in e^+e^- collisions near the Z mass
  - (Ref. 54) *Separation into 4 classes : $u\bar{u}$ and $d\bar{d}$ (unresolved), $s\bar{s}$, $c\bar{c}$, $b\bar{b}$. Four binary 3 layer networks. Measurement of the branching fractions on DELPHI 1990 data.*
  - (Ref. 55) *Separation into 3 classes : $u\bar{u}$, $d\bar{d}$ and $s\bar{s}$ (unresolved), $c\bar{c}$, $b\bar{b}$. Four layer network, with 3 output nodes. Measurement of the branching fractions on ALEPH 1990 data.*
- W/Z classification in $p\bar{p}$ interactions
  - (Ref. 56) *Separation of W and Z decays from QCD background using simulated data in the case study of the UA2 detector at the $Sp\bar{p}S$. 3 layers neural network.*
- *b* jets identification in $p\bar{p}$ interactions
  - (Ref. 26) *4 layers neural network. Simulated data for the case study of the CDF detector at FNAL.*
- Electron identification in an electromagnetic calorimeter
  - (Ref. 57) *Segmented calorimeter with 5 longitudinal samplings. 3 layers feed-forward network with one input layer.*
- Particle identification in a Ring Imaging Cherenkov
  - (Ref. 58) *3 layers neural network. Used in simulated data from the DELPHI RICH.*

Some pattern recognition problems can be reconduced to pattern classification, by discretization of the output space. In such a way, feed-forward neural nets can be used to solve an analog problem after analog-to-digital conversion. This has the disadvantage of increasing the size of the output layer, with possible convergence problems. Some applications of such a technique are summarized below.

- Pattern recognition in a straw chamber
  - (Ref. 59) *3 layer neural network with 14 output nodes, representing the angles of a track. Simulated data without noise.*
- Vertex finding in a drift chamber
  - (Ref. 59,60) *3 layer neural network with 20 output nodes, representing the projections on a coordinate axis. Simulated data, plus real data from a chamber used in E-735 at FNAL.*

4. Conclusions

The study of the operation of brain has lead to artificial Neural Networks that, although using techniques far from the initial model of the study, can approximate arbitrarily complex functions.

Neural Networks can implement massively parallel and highly interconnected algorithms. Their architecture promises to allow a significant increase of speed in data processing and the possibility

to store memory in the architecture itself.

In this article, we have introduced the basics of Neural Networks, presented some simple examples of applications (centered on the most interesting feature for HEP: the possibility of learning by examples), and finally we have shortly illustrated some of the current case studies.

The use of Neural Networks in HEP has been shown to be especially fruitful when dealing with classification and optimization problems. The first case will imply in the next years the implementation of Neural Networks on dedicated hardware, today at the level of prototypes, for the use in fast on-line triggers. The application to optimization problems will probably result in a new generation of off-line algorithms.

References

1. M.L. Minsky, "Computation: Finite and Infinite Machines", Prentice-Hall, series in Automatic Computation, 1969.
2. W. McCulloch and W. Pitts, *Bull. Math. Biophysics* **5** (1943) 115.
3. F.Rosemblatt, "Two Theorems of Statistical Separability in the Perceptron", *Proc. of a Symposium of the Mechanization of Thought Processes* (London 1959), Her Majesty's Stationary Office.
4. F.Rosemblatt, "Principles of Neurodynamics", Spartan Books, New York 1962.
5. M.Minsky and S.Papert, "Perceptrons", MIT Press 1969.
6. J.J.Hopfield, *Proc. Nat. Acad. Sci. USA* **79** (1982) 2554.
7. K.Binder, "Fundamental Problems in Statistical Mechanics", E.G.D.Cohen, Amsterdam 1980.
8. D.J.Amit, H.Gutfreund and H.Spolinsky, *Phys. Rev.* **A 32** (1985) 1007.
9. R.K.Bock, H.Grote, D.Notz and M.Regler, "Data Analysis Techniques for High-Energy Physics Experiments", (Cambridge Universitity Press, 1990).
10. Y.Kamp and M.Hasler, "Réseaux de neurones récursifs pour mémories associatives", (Presses Polytechniques et Universitaires Romandes, Lausanne 1990).
11. D.O.Hebb, "The Organization of Behavior: a Neurophysiological Theory", (John Wiley & Sons, New York 1957).
12. D.Chowdhury, " Spin Glasses and Other Frustrated Systems", (World Scientific Publications, Singapore 1986).
13. S.S.Venkatesh and D.Psalties, "Information Storage and Retrieval in Two Associative Nets", (California Institute of Technology, 1985);
    D.J.Amit, H.Gutfreund and H.Spolinsky, *Phys. Rev. Lett.* **55** (1985) 1530.
14. R.J.Glauber, *J. Math. Phys.* **4** (1963) 294.
15. W.A.Little, *Math. Biosci.* **19** (1974) 101.
16. P.Peretto, *Biol. Cybern.* **50** (1984) 51.
17. B.Denby, "Neural Network and Cellular Automata Algorithms", Preprint FSU-SCRI-88-141 (1988).
18. J.J.Hopfield and D.W.Tank, *Biol. Cybernetics* **52** (1985) 141.
    J.J.Hopfield and D.W.Tank, *Science* **233** (1986) 625.
19. B.Denby, *Comp. Phys. Comm.* **49** (1988) 429.
20. C.Peterson, *Nucl. Instr. and Meth.* **A279** (1989) 537.
21. C.Bortolotto, thesis, Udine 1991.
22. C.Peterson and J.R.Anderson, *Complex Systems* **2** (1988) 59.
23. H.Grote, *Rep. Prog. Phys.* **50** (1987) 473.
24. B.Denby and S.L. Linn, "Status of HEP Neural NET Research in the U.S.A.", FERMILAB Conf-90/21, presented at the 1989 Conference on Computing in High Energy Physics, Oxford, England.
25. M.Gyulassy and M. Harlander, "Elastic Tracking and Neural Network Algorithms for Complex Pattern Recognition", Lawrence Berkeley Laboratory Preprint LBL-29654, 1991.
26. B.Denby *et al.*, "Neural Network for Triggering", FERMILAB Conf-90/20, presented at the 1989 Nuclear Science Symposium, San Francisco.
27. B.Denby, F.Bedeschi, "Investigation of a Beauty Trigger for the SVX", CDF/DOC CDF/PUBLIC/1146 (1990).
28. F.Paige and S.D. Protopopescu, ISAJET Monte Carlo, BNL 38034 (1986), Brookhaven National Laboratory.
29. B.Humpert, "On the Use of Neural Networks in High Energy Physics Experiments", ISU-CS/118 (1989).
30. T. Altherr and J. Seixas in *"New Computing Techniques in Physics Research"*, eds. CNRS (1990); T. Altherr and J. Seixas, CERN preprint CERN-TH-6133/91, July 1991.

31. D.E.Rumelhart, G.E.Hinton and R.J.Williams, "Learning Internal Representation by Error Propagation", Parallel Distributed Processing vol. 1, (MIT Press, Cambridge 1986).
32. R.O.Duda and P.E.Hart, " Pattern Classification and Scene Analysis" (John Wiley & Sons 1973).
33. G.Palmieri and R.Sanna, *Methods* **12** (1960).
34. A.Gamba, L.Gamberini, G.Palmieri and R.Sanna, *Nuovo Cimento Suppl.* no. 2 **20** (1961) 221.
35. B.Widrow and M.E.Hoff, 1960 IRE WESCON Conv. Record Part. 4 (1960) 96.
36. R.P.Lippmann, IEEE ASSP Mag. 4 (1987) 4.
37. S.J.Hanson and D.J.Burr, "Knowledge Representation in Connectionist Networks", Tech. Rep., Bell Comunication Research (1987).
38. I.D.Longstaff and J.F.Cross, "A Pattern Recognition Approach to Understanding the Multi-layer Perceptron", Memo 3, 936, Royal Signals and Radar Establishment (1986).
39. G.Cybenko, "Approximation by Superimpositions of a Sigmoidal Function", Mathematics of Control, Signals ans Systems 2 (4) (1989).
40. R.A.Jacobs, "Increased Rates of Convergence Through Learning Rate Adaption", *Neural Networks* **1** (1988) 295.
41. M.R.Devos and G.A.Orban, "Self Adaptive Backpropagation", Proceedings NeuroNimes (1988).
42. T.Tollenaere, *Neural Networks* **3** (1990) 561.
43. T.Samad, "Refining and redefining the back-propagation learning rule for connectionist networks", *Proc. IEEE Systems Man and Cybernetics Annual Conference* (1987).
44. L.Lönnblad, C.Peterson and T.Rögnvaldsson, *Nucl. Phys.* **B349** (1991) 675.
45. L.Lönnblad, C.Peterson and T.Rögnvaldsson, Phys. Rev. Lett. **65** (1990) 1321.
46. C.Peterson, "Neural Networks and High Energy Physics", Lund Preprint LU TP 90-3 (1990).
47. L.Lönnblad, "ARIADNE-3, a Monte Carlo for QCD Cascades in the Colour Dipole Formulation", Lund Preprint LU-TP 89-10
48. G.Marchesini and B.R.Webber, Nucl. Phys. **B310** (1988) 461.
49. T. Sjöstrand, *Comp. Phys. Comm.* **27** (1982) 243, *ibid.* **28** (1983) 229;
T. Sjöstrand and M. Bengtsson, *Comp. Phys. Comm.* **43** (1987) 367.
50. T.Åkesson and O.Barring, "Jet Classification with a Neural Network", DELPHI 90-59 PHYS 78 (1990).
51. C. Bortolotto, A. De Angelis and L. Lanceri, *Nucl. Instr. and Meth.* **A306** (1991) 459.
52. *Proceedings of the ECFA workshop on LEP200*, A. Böhm and W. Hoogland editors, CERN 87-08, June 1987.
53. L. Bellantoni *et al.*, *Nucl. Instr. and Meth.* **A310** (1991) 618.
54. C. Bortolotto *et al.*, INFN/AE 91-12 (September 1991), to be published in the proceedings of the Workshop on Neural Networks, Isola d'Elba 1991.
55. J. Proriol, presentation at the Workshop on Neural Networks, Isola d'Elba 1991;
P. Henrard, presentation at the 4^{th} International Symposium on Heavy Flavour Physics, Orsay 1991.
56. P.Bhat *et al.*, *"Using Neural Networks to Identify Jets in Hadron-Hadron Collision"*, DESY 90-144, LU TP 90-13 (1990).
57. D.Cutts *et al.*, "The Use of Neural Network in the D0 Data Acquisition System", presented at the conference Real Time 1989, Williamsburg.
58. N.DeGroot, presentation at the Workshop on Neural Networks, Isola d'Elba 1991.
59. B.Denby, E.Lessner and C.S Lindsey, "Tests of Track Segment and Vertex Finding with Neural Networks", FERMILAB-Conf-90/68, presented at the 1990 Conference on Computing in High Energy Physics, Santa Fe, New Mexico.
60. C.S Lindsey and B.Denby, "Primary Vertex Finding in Proton-Antiproton Events with a Neural Network Simulation", FERMILAB-Pub-90/192, Submitted to Nucl. Instr. and Meth. A.