An Introduction to Artificial Neural Networks

Carsten Peterson[1)] *and Thorsteinn Rögnvaldsson*[2)]

Department of Theoretical Physics, University of Lund, Sweden

Abstract

A general introduction to artificial neural networks is given assuming no previous knowledge in the field. Properties of the multilayer perceptron, feature maps, the Hopfield model and the Boltzmann machine are discussed in some detail. Also novel methods of finding good solutions of difficult optimization problems with feed-back networks and so-called elastic nets are described. Throughout the lectures practical hints on how to use the algorithms are given. Potential hardware implementations, both VLSI and optical, are briefly mentioned.
The power of the artificial neural network approach is illustrated in three high energy physics applications — quark jet tagging, mass reconstruction and track finding.

1 Introduction

There has been an upsurge in interest in Artificial Neural Networks (ANN) over the last 5 years. There are several reasons for this. One is that theoretical obstacles within the so-called perceptron, which was popular in the late sixties, have been overcome. Also the recently available inexpensive access to powerful and inexpensive CPU has facilitated the development of the field both with respect to model development, and very importantly, to deal with real-world problems. Indeed, results from impressive "product" quality application work keeps appearing both from the commercial and academic sector.

These lectures are intended as an introduction to the basic concepts of ANN for students in high energy physics (HEP). However, since the physics applications are presented in a separate section, they can be read by a more general audience. We focus mostly on architectures and algorithms relevant for HEP and have for example left out those specifically designed to handle speech recognition problems etc.. We refer to ref. [1] for a more comprehensive treatment of ANN. The lectures only assume very basic knowledge of ordinary differential equations, linear algebra (vector spaces, matrices, etc.) and statistical physics.

These lectures are organized as follows. Section 2 deals with neural systems in general, biological neural networks and the abstraction to artificial neural networks. Sections 3 to 6 describe the main ANN models and algorithms; supervised feed-forward networks, self-organizing networks, feed-back networks and networks used for optimization problems. In section 7 a very brief discussion on hardware implementations can be found. Section 8 describes some applications of these ANN algorithms to high energy physics problems and section 9 contains a short summary.

The reader who is merely interested in getting started with some HEP application can omit the Hopfield model and the Boltzmann machine (essentially section 5) with no loss in content. Also the lectures can be read coherently when ignoring the sections on variations and practical hints etc. in case the reader is only interested in the basic concepts.

2 Neural Networks

Artificial neural networks is a computational paradigm that differs substantially from those based on the "standard" von Neumann architecture. ANN's generally learn

1) thepcap@seldc52 (bitnet) carsten@thep.lu.se (internet)
2) thepdr@seldc52 (bitnet) denni@thep.lu.se (internet)

from "experience", rather than being explicitly "programmed" with rules like in conventional artificial intelligence (AI).

2.1 Biological Neural Networks

ANN is inspired from the structure of biological neural networks and their way of encoding and solving problems. We will here briefly review the basic components and functionality of the vertebrate central nervous system (CNS), whose details were revealed around 1940 with the emergence of the electron microscope. For more extensive literature on this subject we refer the reader to refs. [2, 3].

The human brain contains approximately 10^{12} *neurons*. These can be of many different types, but most of them have the same general structure (see fig. 2.1). The *cell body* or *soma* receives electric input signals to the *dendrites* by means of ions. The interior of the cell body is negatively charged against a surrounding *extracellular fluid*. Signals arriving at the dendrites depolarize the resting potential, enabling Na^+ ions to enter the cell through the membrane, resulting in an electric discharge from the neuron — the neuron "fires". The accumulated effect of several simultaneous signals arriving at the dendrites is usually approximately linearly additive whereas the resulting output is a strongly non-linear all-or-none type process. The discharge propagates along the *axon* to a *synaptic junction*, where *neurotransmitters* travel across a *synaptic cleft* and reach the dendrites of the postsynaptic neuron. A synapse which repeatedly triggers the activation of a postsynaptic neuron will grow in strength; others will gradually weaken. This *plasticity*, which is known as the *Hebb rule*, plays a key part in *learning*.

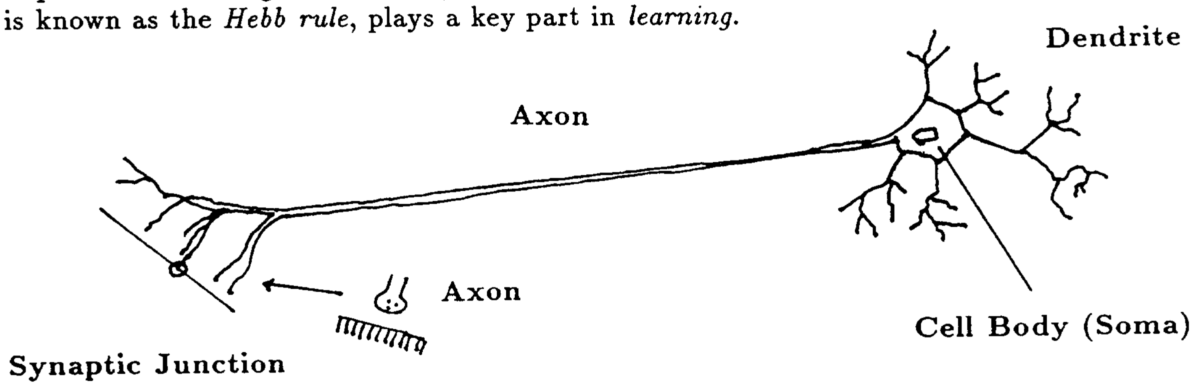


Figure 2.1: Schematic structure of a neuron

This general neuron structure is implemented in many different sizes and forms with different functionalities. Cell bodies have diameters in the range 5–80 μm and dendrite "trees" extend from 10 μm up to 2–3 mm. Axons can be up to 1 m in length, why sizes of entire neurons vary from 0,01 mm to 1 m. *Primary sensory neurons* connect muscles or receptors to neurons, *secondary sensory neurons* and *interneurons* connect neurons with neurons, while *motor neurons* connect neurons with muscle fibers.

The *connectivity* (number of neurons connected to a neuron) varies from a $\mathcal{O}(1)$ to $\mathcal{O}(10^5)$. For the *cerebral cortex* $\mathcal{O}(10^3)$ is an average. This corresponds to $\mathcal{O}(10^{15})$ synapses per brain. Synapses can be either *excitatory* or *inhibitory* of varying strength. In the simplified binary case of just two states per synapse the brain thus has $\mathcal{O}(2^{10^{15}} \approx 10^{10^{14}})$ possible configurations! The neural network is consequently in sharp contrast to a von Neumann computer both with respect to architecture and functionality (see table 2.1).

The von Neumann computer was originally developed for "heavy duty" numerical computing but has later also turned out to be profitable for data handling, word processing and suchlike. However, when it comes to matching the vertebrate brain in terms of performing "human" tasks it has very strong limitations. There are therefore strong reasons to design an architecture and algorithm that shows more resemblance with that of the vertebrate brain.

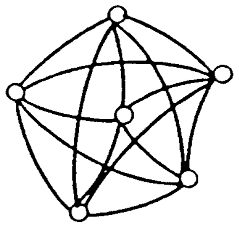
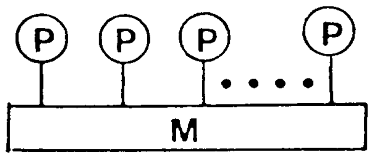
Vertebrate Brain (Parallel Distributed Proc.)	Conventional Computer (von Neumann machine)
	
Power dissipation: $\approx$ 100W	Power dissipation: $\approx 10^5$W
Good at: - recognition - adaption - optimization (rough)	Good at: - a+b, a $\cdot$ b,... - if (....) else - state space exploration
$\tau = \mathcal{O}(ms)$	$\tau = \mathcal{O}(ns)$
Parallel Robust Fault Tolerant	Serial/parallel Fragile

Table 2.1: Comparison of characteristics of neural networks and conventional computers.

2.2 Artificial Neural Networks

The philosophy of the ANN approach is to abstract some key ingredients from biology and out of those construct simple mathematical models that exhibit most of the above-mentioned appealing features. In physics one has good experience of model building out of major abstractions. For example, details of individual atoms in a solid can be lumped into effective "spin" degrees of freedom in such a way that good description of *collective* phenomena (phase transitions etc.) are obtained. It is exactly the collective behavior of the neurons that is interesting from the point of view of intelligent data processing.

2.2.1 Basics

The basic computational entities of an ANN are the neurons [4] v_i, which can take real values within the interval [-1,1] (or [0,1]). Sometimes the even simpler binary neuron is used, where $v_i = \{-1, 1\}$ (or $\{0, 1\}$). For binary neurons the notation s_i is normally used.

These are simplifications of the biological neurons described in the previous section. The most common form is the "thresholding" neuron (see fig. 2.2):

$$v_i = g(\sum_j \omega_{ij} v_j + \theta_i) \tag{2.1}$$

where v_j are all neurons that are feeding to neuron v_i, through weights (synapses) ω_{ij}. These weights can have both positive (excitatory) and negative (inhibitory) values. The θ_i term is a threshold, corresponding to the membrane potential in a biological neuron. The non-linear *transfer function* $g()$ is typically a sigmoid-shaped function like

$$v_i = g(a_i) = \tanh(a_i/T) \tag{2.2}$$

for [-1,1] neurons or

$$v_i = g(a_i) = \frac{1}{2}[1 + \tanh(a_i/T)] \tag{2.3}$$

for [0,1] neurons. The argument a_i is the linearly summed input signal to neuron v_i;

$$a_i = \sum_j \omega_{ij} v_j \tag{2.4}$$

and the "temperature" T sets the inverse gain of the function $g()$; a low temperature corresponds to a very steep sigmoid and a high temperature corresponds to an approximately constant $g()$ (see fig. 2.2). The limit ($T \longrightarrow 0$) corresponds to binary neurons s_i .

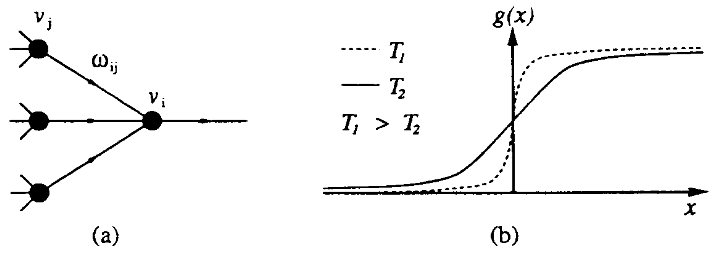


Figure 2.2: A thresholding neuron: (a) Neuron updating and (b) sigmoid response functions of eqs. (2,3) for different temperatures T.

The simple "thresholding neuron" mimics the main features of real biological neurons in terms of linear additivity for the inputs and strong non-linearity for the resulting output. If the integrated input signal is larger than a certain threshold θ_i the neuron will "fire".

There are two different kinds of architectures in neural network modeling; feed-forward (fig. 2.3a) and feed-back (fig. 2.3b). In feed-forward networks signals are processed from a set of input units in the bottom to output units in the top, layer by layer, using the local updating rule of eq. (2.1). In feed-back networks, on the other hand, the synapses are bidirectional. Activation continues until a fixed point has been reached, reminiscent of a statistical mechanics system. In our HEP applications feed-forward architectures will be used for pattern recognition, whereas track-finding will be based on feed-back architectures.

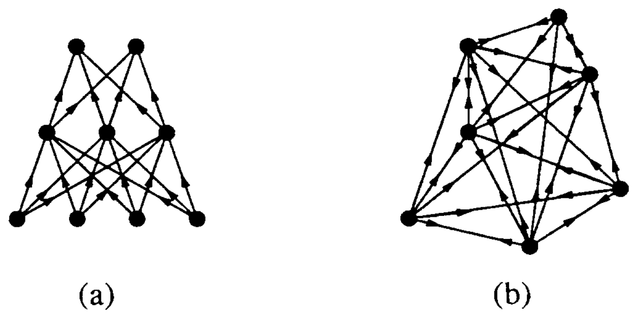


Figure 2.3: (a) Feed-forward and (b) feed-back architectures

2.2.2 Application Areas

How can the power of these networks be exploited? There are four major application areas with soft borderlines in between — feature recognition, function approximation, associative memories and optimization. Feature recognition is the most exploited application domain to date.

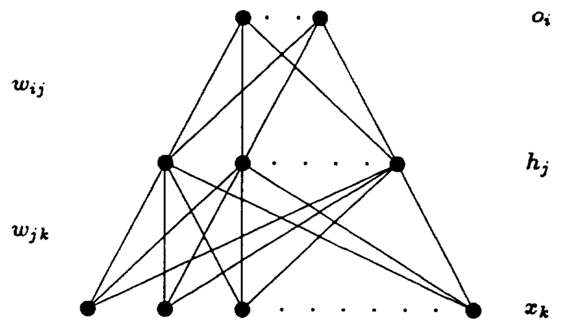


Figure 2.4: A one hidden layer feed-forward architecture.

Feature recognition

In feature recognition (or pattern classification) situations one wants to categorize a set of input patterns $\vec{x}^{(p)} = (x_1, x_2, ..., x_N)^{(p)}$ in terms of different features o_i. The input patterns are fed into an input layer (receptors) and the output nodes represent the features. For the architecture depicted in fig. 2.4, o_i depends on $\vec{x}^{(p)}$ through the functional form (cf. eq. (2.1))

$$o_i(\vec{x}^{(p)}) = g(\sum_{j=0} \omega_{ij} g(\sum_{k=0} \omega_{jk} x_k^{(p)})) \tag{2.5}$$

where ω_{ij} and ω_{jk} are the parameters. Eq. (2.5) can of course be generalized to any number of layers. The thresholds, or bias terms, θ_i appearing in eq. (2.1) have in eq. (2.5) been generalized to weights ω_{i0} by adding an extra "dummy" unit v_0 to each layer, which is constantly firing. Fitting ω_{ij} and ω_{jk} to a given data set (or *learning*) takes place with gradient descent on a suitable *error function*[3]. In this process the *training patterns* are presented over and over again with successive adjustments of the weights. Once this iterative learning has reached an acceptable level, in terms of a low error, the weights are frozen and the ANN is ready to be used on patterns it has never seen before. The capability of the network to correctly characterize these *test patterns* is called *generalization* performance. This procedure is equivalent to "normal" curve fitting where a smooth parameterization from a training set is used to interpolate between the data points (generalization).

Function approximation

Rather than having a "logical" output unit (o_i) with a threshold behavior described by eqs. (2.2,2.3), one could imagine having an output representing a real number, corresponding to replacing eqs. (2.2,2.3) with linear behaviors. In this case one adjusts the weight parameters to parameterize an unknown real-valued function. Such an approach can be useful in *time-series predictions*, where one aims at predicting future values of a series given previous values [5, 6, 7, 8], e.g.

$$x_t = \mathcal{F}(x_{t-1}, x_{t-2}, ...) \tag{2.6}$$

where x_t is the real-valued output node and $x_{t-1}, x_{t-2}, ...$ are the values of the series at previous times. Another application with linear output nodes is mass reconstruction in high energy physics [31], which is discussed in section 8.

Associative memory

In this kind of application there is no input/output distinction. Rather the network learns a set of identity mappings by fitting ω_{ij} to memory patterns or "words" $\vec{s} =$

3) The error function is sometimes referred to as *Lyapunov function* or *cost function*.

$(s_1, s_2, s_3, \ldots s_N)$. If an incomplete or partly erroneous memory is presented to the network, it completes the pattern. In a certain sense feature recognizers are special cases of associative memories — pattern completion.

Optimization

Feed-back networks have shown great promise in finding good solution problems to difficult combinatorial optimization problems. In this case again non-linear updating equations (eq. (2.1)) are allowed to settle. This corresponds to minimizing an energy function, which typically looks like

$$E = -\frac{1}{2}\sum_{ij}\omega_{ij}s_i s_j \tag{2.7}$$

The problem is mapped onto the form of eq. (2.7) by a clever choice of ω_{ij}. In this case ω_{ij} are fixed once and for all for each problem — they are not adaptive as in the other domains of application. The neurons s_i are then allowed to settle into a stable state, where the solution to the problem is given by the configuration $\vec{s} = (s_1, s_2, \ldots)$ with minimum energy.

3 Feed-forward Networks

In this section we discuss feed-forward networks with respect to what mappings they can perform for a given complexity of the architecture. Also, different error measures and the back-propagation learning algorithm with variants and refinements are explained.

3.1 The Simple Perceptron

The *simple perceptron* [9], which dates back to 1961, has one layer of input units and one layer of output (or feature) units (see fig. 3.1). Starting from randomly distributed ω_{ij}'s each pattern is processed through the network and the output nodes are updated according to eq. (2.1). For each training pattern $\vec{x}^{(p)}$ there is a known *target pattern* $\vec{t}^{(p)}$. The goal is to produce a true mapping $\vec{x}^{(p)} \mapsto \vec{t}^{(p)}$ for all p.

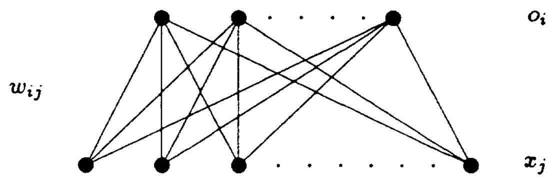


Figure 3.1: The simple perceptron.

3.1.1 A Perceptron Learning Algorithm

The training of the perceptron is performed by minimizing the *summed square* error function

$$E = \frac{1}{2}\sum_p\sum_i (o_i^{(p)} - t_i^{(p)})^2 \tag{3.1}$$

with respect to the weights. This is done with a gradient descent method where the weights are incrementally updated in proportion to $\partial E/\partial\omega_{ij}$. In practice this can be done either in *batch-mode* (or *off-line* mode) where $\partial E/\partial\omega_{ij}$ are computed for all patterns before updating the weights or with an *on-line* procedure, where $\partial E/\partial\omega_{ij}$ is computed pattern by pattern. In what follows we assume the latter alternative (in order to suppress the index p). The derivative of E with respect to ω_{ij} is given by

$$\frac{\partial E}{\partial\omega_{ij}} = \frac{\partial E}{\partial o_i}\frac{\partial o_i}{\partial\omega_{ij}} \tag{3.2}$$

where o_i is (cf. eq. 2.1)

$$o_i = g(\sum_{j=0} \omega_{ij} x_j) = g(\vec{\omega}_i \cdot \vec{x}) \tag{3.3}$$

Gradient descent (or forward Euler) updating then reads

$$\Delta\omega_{ij} = -\eta \frac{\partial E}{\partial \omega_{ij}} = -\eta \delta_i x_j \tag{3.4}$$

where η is the learning rate parameter ($\eta < 1$) and δ_i is given by

$$\delta_i = (o_i - t_i) g'(\vec{\omega}_i \cdot \vec{x}) \tag{3.5}$$

When no more changes (within some accuracy) occurs, i.e. $\Delta\omega_{ij} \approx 0$, the weights are frozen and the network is ready to use for data it has never "seen". The procedure is summarized in fig. 3.2.

Note that this algorithm is not the only training algorithm for perceptrons (for other alternatives see refs. [10, 11]).

```
1. Initialize $\omega_{ij}$ with $\pm$ random values.
2. Repeat until $\omega_{ij}(t+1) \approx \omega_{ij}(t)$:
   2.1 Pick pattern $p$ from training set.
   2.2 Feed input $\vec{x}^{(p)}$ to network and
       calculate the output $\vec{o}$ (eq. (3.3)).
   2.3 Update the weights according to
       $\omega_{ij}(t+1) = \omega_{ij}(t) - \eta\delta_i x_j$
       where $\delta_i$ is given by eq. (3.5).
```

Figure 3.2: A perceptron learning algorithm.

3.1.2 Examples: Boolean Functions

To demonstrate the computational capabilities of the simple perceptron, let us consider the case of boolean functions of two binary inputs. Two such functions, the AND and OR functions, are defined in table 3.1.

AND			OR		
x_1	x_2	t	x_1	x_2	t
1	1	1	1	1	1
1	0	0	1	0	1
0	1	0	0	1	1
0	0	0	0	0	0

Table 3.1: The AND and OR functions.

Both the AND and OR functions are easily learned by the simple perceptron. Using a network with two inputs and an output unit with a [0,1] transfer function $g()$ (eq. (2.3)) one might get the weight values $\vec{\omega} = (\omega_0, \omega_1, \omega_2) = (-1.5, 1.0, 1.0)$ for the AND problem and $\vec{\omega} = (-0.5, 1.0, 1.0)$ for the OR problem, where ω_0 corresponds to the threshold term θ in eq. (2.1). It is easily verified that these weight values really compute the logical functions in table 3.1, by computing the argument $\vec{\omega} \cdot \vec{x}$ and using the step-function limit of $g()$ ($T \to 0$) for the output[4].

4) The threshold "dummy" unit x_0 for the input layer is a constant = 1.

3.1.3 Limitations

What are the limitations of the simple perceptron? Let us examine how it deals with the boolean exclusive-or [XOR], or the two-bit parity problem, which is defined in table 3.2. We again use a network with two inputs and one output node. To simplify the discussion we use the step-function limit of $g()$. The functional dependence reads:

$$o = g(\vec{\omega} \cdot \vec{x}) = \begin{cases} 0 & \text{if } \omega_0 + \omega_1 x_1 + \omega_2 x_2 < 0 \\ 1 & \text{if } \omega_0 + \omega_1 x_1 + \omega_2 x_2 > 0 \end{cases} \tag{3.6}$$

x_1	x_2	t	$\vec{\omega} \cdot \vec{x}$
1	1	0	$\omega_0 + \omega_1 + \omega_2 < 0$
1	0	1	$\omega_0 + \omega_1 > 0$
0	1	1	$\omega_0 + \omega_2 > 0$
0	0	0	$\omega_0 < 0$

Table 3.2: The XOR problem and the conditions on weights for the simple perceptron.

It is clear that the inequalities in the right column of table 3.2 cannot be simultaneously satisfied. The three parameters are not sufficient to describe the problem. Another way to phrase this fact is that the XOR problem is not *linearly separable*, which is illustrated in fig. 3.3.

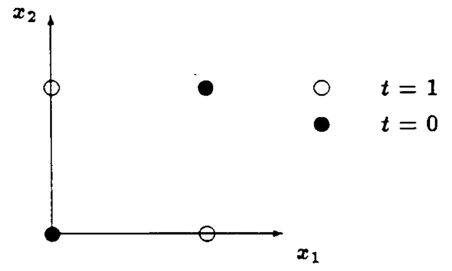


Figure 3.3: State space of the XOR problem.

The input space is two-dimensional and the possible inputs are corners of a square. The two "classes" $t = 1$ and $t = 0$ cannot be separated with a single line in the *input space*. For the linearly separable AND and OR functions, the separating line between the classes is orthogonal to the vectors $\vec{\omega} = (\omega_1, \omega_2)$ for the two functions, which is easily verified, and the threshold term ω_0 is a measure of the perpendicular distance to the origin. In the general case, the simple perceptron is capable of classifying all classes that are separable by a hyperplane ($(N-1)$-dimensional subspace) in the N-dimensional input space.

The failure of the simple perceptron is thus caused by its linearity[5)] and this criticism [12] against it was vigorously pursued in the late sixties, which effectively put ANN research on hold for some 20 years. One might argue that a XOR problem is too academical to serve as a testbed for an adaptive learning algorithm but many real-world pattern recognition problems contain similar *higher order constraints* which make them linearly inseparable. To circumvent this we have to provide the perceptron with non-linear qualities. One way to do this is of course to use a more complex transfer function $g()$, like a Gaussian. In that case the simple perceptron would be able to compute the XOR problem. Another way, which is more powerful and general, is to introduce extra *hidden layers* between the input units and the output units, which is usually referred to as the *multilayer perceptron* (MLP).

5) The non-linearity of the sigmoid function $g()$ is irrelevant for the functionality; it is only used to provide a feature representation.

3.2 The Multilayer Perceptron

To improve the simple perceptron we augment it with a layer of *hidden units* (see fig. 2.4). Since these have non-linear response functions (eq. (2.5)) the network should be able to perform non-linear mappings. The hidden nodes in fig. 2.4 build up an *internal representation* of the data. In fig 3.4 we show an explicit solution to the XOR problem with one layer of two hidden nodes.

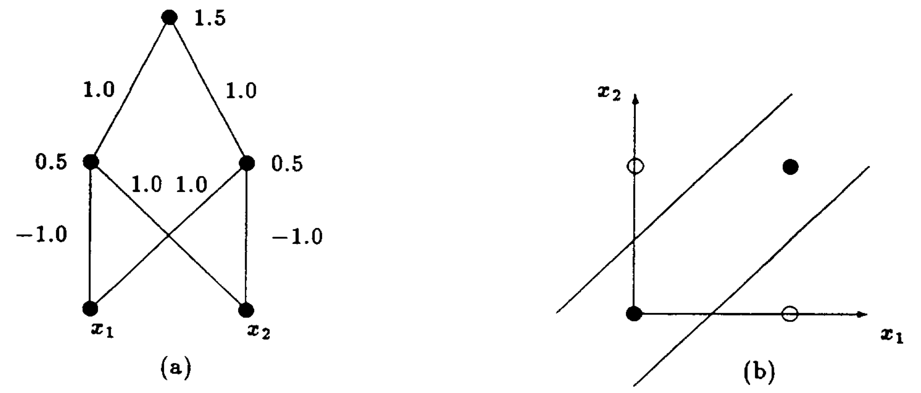


Figure 3.4: (a) An explicit solution to the XOR problem. (b) The representation (hyperplanes) of the hidden units; $\vec{\omega}_1 = (0.5, -1, 1)$ and $\vec{\omega}_2 = (0.5, 1, -1)$. The output unit performs a NAND function on h_1 and h_2.

3.2.1 The Back-propagation Learning Algorithm

Let us now generalize the gradient descent procedure of eq. (3.4) to the case with hidden layers [13]. For simplicity we perform the derivations for architectures with one hidden layer - the generalization to several hidden layers is straightforward. At each layer we introduce the notation a_i and a_j for the weighted input sums (the units h_0 and x_0 correspond to the threshold "dummy" units).

$$a_i = \sum_{j=0} \omega_{ij} h_j = \vec{\omega}_i \cdot \vec{h} \tag{3.7}$$

$$a_j = \sum_{k=0} \omega_{jk} x_k = \vec{\omega}_j \cdot \vec{x} \tag{3.8}$$

The hidden and output nodes are thresholded like

$$\begin{aligned} h_j &= g(a_j/T) \\ o_i &= g(a_i/T) \end{aligned} \tag{3.9}$$

In addition to $\partial E/\partial\omega_{ij}$ we now also need $\partial E/\partial\omega_{jk}$ in order to minimize E. With the "chain-rule" one gets

$$\frac{\partial E}{\partial \omega_{jk}} = \sum_i \frac{\partial E}{\partial o_i}\frac{\partial o_i}{\partial a_i}\frac{\partial a_i}{\partial h_j}\frac{\partial h_j}{\partial a_j}\frac{\partial a_j}{\partial x_k} = \sum_i \omega_{ij}\delta_i g'(a_j) x_k \tag{3.10}$$

where δ_i is defined in eq. (3.5).

With gradient descent, the ω_{ij}'s are again updated as in eq. (3.4). For ω_{jk} one gets

$$\Delta\omega_{jk} = -\eta \sum_i \omega_{ij}\delta_i g'(a_j) x_k \tag{3.11}$$

Here we see how the error δ_i is *back-propagated* down through the network.

```
1. Initialize $\omega_{ij}$ with $\pm$ random values.
2. Repeat until $\omega_{ij}$ and $\omega_{jk}$ have converged
   or the desired performance level is achieved:
    2.1 Pick pattern $p$ from training set.
    2.2 Present input $\vec{x}^{(p)}$ and calculate $\vec{h}$ and
        the output $\vec{o}$ according to eq. (3.9)
    2.3 Update the weights according to
        $\omega_{ij}(t+1) = \omega_{ij}(t) - \eta \delta_i h_j$
        $\omega_{jk}(t+1) = \omega_{jk}(t) - \eta \sum_i \omega_{ij} \delta_i g'(a_j) x_k$
        (...etc...for extra hidden layers).
```

Figure 3.5: The back-propagation learning algorithm.

Back-propagation is not limited to the simple case of layer to next-layer connections, it generalizes just as easily to network architectures with input to output connections besides the input to hidden and hidden to output.

The training of a multilayer perceptron using BP can be thought of as a walk in "weight space" along an energy surface, defined by eq. (3.1), trying to find the global minimum and avoiding local minima. There is, in contrast to the simple perceptron, no guaranty that the global minimum will be reached with this method. However, in most cases the energy landscape is smooth without too many local minima.

3.2.2 Refinements and Variations

One major criticism against this "vanilla" version of BP is that the convergence time grows very rapidly with the problem size. Several variations have been suggested to speed it up [14]. One straightforward improvement is to use "on-line" updating (see above) where the weights are updated for each n^{th} pattern presentation (where $n \sim \mathcal{O}(10)$) instead of for the whole training set [15] (for some problems [16], however, "off-line" updating has been reported to be quicker). In this section we describe some useful variations and extensions to the back-propagation algorithm in order to make it more powerful.

Alternative error measures

The *summed square* error in eq. (3.1) is not the only possible error measure. Any function that is differentiable with respect to o_i and has a minimum for $o_i = t_i$ can be used. Some commonly used alternative measures are:

– *Cross-entropy* error: [17]

$$E = -\sum_p \sum_i [t_i^{(p)} \log o_i + (1 - t_i^{(p)}) \log(1 - o_i)] \tag{3.12}$$

In this case the $g'()$ factor in the updating of ω_{ij} disappears, but the updating of the hidden-hidden and input-hidden connections are the same (eq. (3.10)).

– *Kullback measure:* [18]

$$E = \sum_p \sum_i t_i^{(p)} \log \frac{t_i^{(p)}}{o_i} \tag{3.13}$$

which is used in conjunction with K-valued sigmoids (see below). The back-propagation algorithm turns out to be the same as in the case of the entropy error above, again without a $g'()$ between top hidden and output layers.

Alternative output nodes

It is sometimes convenient to use other output nodes than the normal "thresholding" types in eqs. (2.2) and (2.3).

– *K-valued Potts neurons:* [52] If the target output vectors $\vec{t}^{(p)}$ equal the orthonormal basis vectors $(1, 0, 0, ...)$, $(0, 1, 0, ...)$, $(0, 0, 1, ...)$, etc., then a "winner-takes-all" interpretation of the output $\vec{o}^{(p)}$ is often beneficial. An efficient encoding of this is achieved by using K-valued neurons, analogous to *Potts* spin models in physics [19]. The sigmoid $g()$ for the output node is then replaced by

$$o_i = o_i(a_1, a_2, ..., a_n) = \frac{e^{a_i/T}}{\sum_l e^{a_l/T}} \tag{3.14}$$

satisfying

$$\sum_i^n o_i = 1 \tag{3.15}$$

Eq. (3.15) corresponds to a dimensional reduction of the solution space hypercube to a hyperplane (see fig. 3.6).

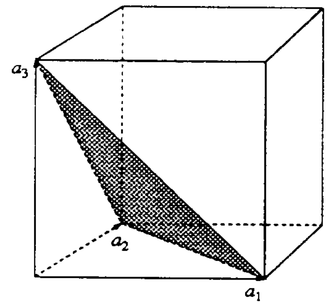


Figure 3.6: The volume of solutions corresponding to K-valued neurons for K=3. The shaded plane corresponds to eq. (3.15) for K=3.

– *Linear output nodes:* If one wants to use the network as a plain "fitting engine", like in the case of time-series prediction, the output nodes should encode any real number; i.e. be linear $o_i = a_i$. In this case $g'()$ also disappears from eqs. (3.2) but the lower layers are the same.

Adapting parameters

The back-propagation in its basic form has two parameters; the temperature (inverse gain) T and the learning rate η. During training, the network passes through different parts of the energy landscape, with possibly very different profiles. The learning parameters should therefore be allowed to change dynamically during training [1]. The ideal is to let each parameter change individually for each weight [20].

– *Annealing:* In the high-temperature limit ($T \to \infty$), the sigmoid $g(x)$ approaches a constant value. For large temperatures, the energy is thus approximately constant, independent of the weights. Slowly decreasing the temperature will increase the contours of the energy landscape, initially with small gradients, making it easier to find an appropriate learning rate. This is usually not necessary.

– *Momentum:* In situations, where the energy landscape is characterized by steep walls and "flat" valleys, it is beneficial to average over many updatings in order to quickly reach the bottom of the valley and increase speed along it. The most commonly used technique to do this is the addition of a *momentum term.* The idea is to provide each weight with some inertia in order to avoid oscillations. This is done by including an extra term in eqs. (3.4,3.11)

$$\Delta\omega(t+1) = -\eta\frac{\partial E}{\partial\omega} + \alpha\Delta\omega(t) \tag{3.16}$$

where $\Delta\omega(t)$ refers to the previous updating. The momentum parameter α can be varied during training.

– *Second order methods:* A more thorough approach to dynamically tune η with respect to the landscape is the *Newton rule*:

$$\Delta\omega_{ij} = -\eta H^{-1}\frac{\partial E}{\partial\omega_{ij}} \tag{3.17}$$

where the Hessian matrix **H** is given by

$$H = \frac{\partial^2 E}{\partial\omega_{ij}\omega_{i'j'}} \tag{3.18}$$

Needless to say this procedure is too CPU and memory consuming to be of practical use; for N weights it requires the storage and diagonalization of an N by N matrix. Also, for networks with many hidden layers, higher order methods sometimes perform worse than simple gradient descent [21].

Manhattan Learning

The time-dependence of the learning rate for the standard BP algorithm sometimes makes it difficult to reach optimum performance. In such cases *Manhattan updating* can be more efficient. Manhattan updating is bounded and uses only the direction of the gradient ∇E;

$$\Delta\omega_{ij} = -\eta\cdot\mathrm{sgn}[\frac{\partial E}{\partial\omega_{ij}}] \tag{3.19}$$

where η should be decreased with learning. In this way the same learning "step" is used for all weights.

Restricting the Number of Parameters

An important question is what ANN architecture (number of layers, hidden nodes and degree of connectivity) to use for a particular problem. Clearly one should use as few parameters (weights) as possible in order to have powerful generalization properties. In other words one wants to avoid overfitting. A straight-forward, but costly, way is of course trial-and-error. However, more algorithmic methods exist:

– *Weight decay:* The most simple such approach is weight decay, where weights which are rarely updated are allowed to decay according to

$$\Delta\omega_{ij} = -\eta\frac{\partial E}{\partial\omega_{ij}} - \varepsilon\omega_{ij} \tag{3.20}$$

where ε is the decay parameter, typically a very small number, $\mathcal{O}(10^{-4})$. This corresponds to adding an extra complexity term to the energy function

$$E \rightarrow E + \frac{\varepsilon}{2\eta}\sum_{ij}\omega_{ij}^2 \tag{3.21}$$

imposing a "cost" for large weights.

- *Pruning:* A more advanced complexity term [8] is

$$E \rightarrow E_0 + \lambda \sum_{ij} \frac{\omega_{ij}^2/\omega_0^2}{1+\omega_{ij}^2/\omega_0^2} \tag{3.22}$$

where the sum extends over all weights. For large $|\omega_{ij}|$, the cost is λ, whereas for small weights it is zero. The scale of the weights is set by ω_0. In this way the cost reflects the number of weights, instead of the size, hence the network gets pruned to only contain weights that are really needed to represent the problem.
- *Self-generating networks:* Instead of determining the optimal architecture for a network by starting with a large number of weights and then use the pruning procedure described above, one could allow the network to change its complete architecture during learning. Such self-generating networks [22, 23, 24, 25] aim at constructing the most economic architecture for a specific problem by some algorithmic method. These generative algorithms generally perform better than simple BP, both in sense of convergence time and generalization.

Stochastic Methods

There exists a number of methods with stochastic elements aimed at avoiding getting stuck in local minima in the energy landscape. These include randomly changing the order of pattern presentations, to avoid cyclic behavior.

- *Noisy input:* Adding noise to the inputs smears the inputs and improves generalization. Too much noise however deteriorates the network's performance.
- *Random changes of weights:* One can make stochastic steps increasing the energy by using e.g. the *Metropolis* [26] or *heat bath* [27] algorithms or the *Langevin* equations [28]. The latter case is natural since the gradient descent updating equations can be regarded as forward Euler approximations of the differential equations

$$\frac{d\omega_{ij}}{dt} = -\frac{\partial E}{\partial \omega_{ij}} \tag{3.23}$$

with $\Delta t = \eta = 1$. Augmented with a white appropriately normalized Gaussian random noise σ eq. (3.23) becomes the Langevin equation

$$\frac{d\omega_{ij}}{dt} = -\frac{\partial E}{\partial \omega_{ij}} + \sigma \tag{3.24}$$

3.2.3 Practical Issues and "Rules of Thumb"

Even though we have a powerful learning algorithm for the multilayer perceptron, there are still a number of questions that remain unanswered. How many hidden layers and units are needed? How should a network be trained to achieve optimal generalization performance? What is "optimal generalization performance"? Some of these questions have simple answers while others don't. We will in this section discuss these issues and give some rules of thumb for the questions with no firm answer.

Number of Layers

The "number of layers" issue of course depends on the specific task one wants the network to perform, but a general statement is that no more than two hidden layers are needed, even though very many units might be needed in these layers. The most common tasks for a BP feed-forward network are pattern classification (or feature recognition) and function approximation, which are discussed below.

- *Pattern classification:* Pattern classification means selecting regions of the input space and assigning them to classes. This is illustrated in fig. 3.7, where three classes occupy a two-dimensional input space. To separate these classes, the network has to make borders around them and "cut them out". To see how this can be done we make use of the vector representation of eqs. (3.7) and (3.9). In analogy with the simple perceptron the units $h_j^{(1)}$ in a first hidden layer are able to linearly separate the input space into two regions. The units $h_j^{(2)}$ in a second hidden layer, corresponding to the output layer of fig. 2.4, can perform AND or OR like functions on $h_j^{(1)}$ ($h_j^{(1)} + h_{j'}^{(1)}, h_j^{(1)} - h_{j'}^{(1)}, h_j^{(1)} \cap h_{j'}^{(1)}, h_j^{(1)} \cup h_{j'}^{(1)}$). The response region (in input space) of the units $h_j^{(2)}$ will thus be a convex subspace. Analogously the output units in such a four-layered MLP will respond to AND or OR like functions on convex subspaces. Such subspaces can be of any shape, even concave. Subsequently, any classification task can be performed using only two hidden layers. In the case depicted in fig. 3.7 one hidden layer is not enough to give a simple classification, class B needs two hidden layers to be optimally classified.

– *Function approximation:* [5] An arbitrary function $\mathcal{F}$ can be approximated by a linear combination of "splines" (a spline is a Gaussian-like bump). A spline-like response can be constructed by combining two layers, i.e. multiplying two sigmoids give a bump. This means that at most two layers are needed to approximate any function. Actually, it has been shown that any continuous function can be approximated with only one layer of sigmoidal units [29].

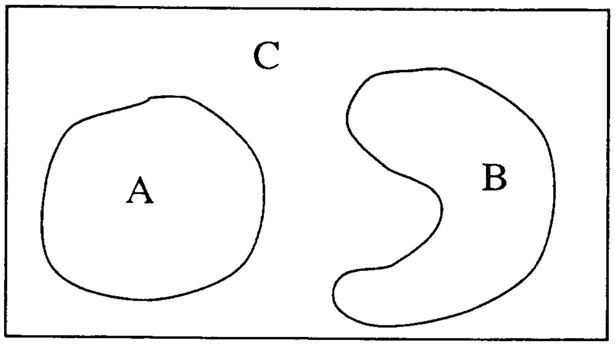


Figure 3.7: A pattern classification problem in two dimensions.

Number of Hidden Units

The approximate minimum number of hidden units to achieve a certain task can be determined from general arguments. It is however not wise to use the very minimum number of hidden units, since the number of paths to the global minima increases with the number of hidden units, decreasing the chance of getting stuck in a local minima. The addition of extra weights (= extra dimensions in weight space) adds the possibility of "going around the mountain instead of over it". But at the same time, more hidden units can give rise to lower generalization performance (see below). The ideal approach is to train with many hidden units and then decrease that number as the network is getting close to the global minima [25] (cf. the discussion on pruning above).

– *Pattern classification:* The view of pattern classification as a "cut out" procedure gives a clue to the minimum number of hidden units needed for a specific classification task. In the "hard" case where the first hidden layer has to define the border of a closed volume in the input-space, at least $(N + 1)$ hidden units are needed in the first layer for each class, usually more. This is because $(N + 1)$ is the minimum number of hyperplanes needed to cut out a closed N-dimensional volume. The more hyperplanes one uses, the smoother the border of the class. The second hidden layer is usually performing some kind of AND or OR function on the units of the first hidden layer, why a lower number of units is needed there (compare with the simple perceptron above).

- *Function approximation:* The number of hidden units needed to approximate a given function $\mathcal{F}$ is related to how many terms that are needed in an expansion of $\mathcal{F}$ in the function $g()$ [32].

Learning Parameters

The crucial parameter in BP is the learning rate η. The ideal η depends on two things; the scale of the activations and the "fan-in" of the network.

- *Activation scale:* If the average activation $\langle x_i \rangle$ of input node i is large, the optimal η for weights connecting to that unit will be small. The natural thing is to rescale the input data such that $\langle x_i \rangle \sim \mathcal{O}(1)$ for all i, in which case they will all have approximately the same optimal η (even the thresholds). This also simplifies for the case of the hidden units, where $\langle h_j \rangle \sim \mathcal{O}(1)$ for sigmoidal units, and the same learning rate can be used for all weight layers (if the "fan-in" is the same).
- *"Fan-in":* The "fan-in" to a unit is the number of units connecting to that unit, in the forward direction. For example, if a fully connected three-layered (one hidden layer) MLP has 20 input units, 10 hidden units and one output unit, then the "fan-in" to a unit in the hidden layer is 20 and to the output unit it is 10. The optimal learning rate for connections ω_{jk} scales like $\eta \propto 1/$("fan-in" to unit h_j) or stronger [30].

The momentum term α controls the "averaging" of the updatings and is closely connected to the learning rate. An increase in α means an increase of the "effective" learning rate. The optimum α depends on the updating procedure used. For the "off-line" method α is very useful and should be a number close to unity ($0.5 < \alpha < 1$). For the "on-line" updating, on the other hand, α is often (but not always) useless.

Initial Weight Values

The initial weights are important. From eqs. (3.4,3.11) we see that $\Delta\omega_{ij} \propto g'()$. The initial changes to the weights ω_{ij} should be large, i.e. $g'()$ should be large. This means that the initial summed inputs (eq. 3.7) should be small. As a "rule-of-thumb" the size of the initial weights should be $\omega \approx \delta/$(maximum "fan-in"), where $\delta \approx T/10$. The "temperature" T sets the inverse slope for $g()$ and the central part of $g'()$ increases with T. A very low T subsequently frustrates the weights. $T = 1$ is usually a good choice.

Input Representation

Input representation is crucial for achieving good performance [31]. Preprocessing of the data can be done in order to decrease the number of input units, thus decreasing the number of weights in the network. The representation of the data (using polar coordinates, etc.) influences the number of cuts needed and thus changes the number of hidden units. The network learns more easily, and less rescaling of parameters is needed, if all inputs are scaled to $\mathcal{O}(1)$. If one of the inputs is very much larger in magnitude than the others, the network will concentrate on that one and take a very long time to learn the others. This has to do with different "time constants" for the weights to reach their asymptotic values. Small inputs need a larger η than large inputs, since the updatings (eq. (3.10)) are proportional to the input values. If the network needs to compare a very large input with a small one, the inputs to the hidden units should be of the same order for the two. This means that the weight from the large input should be small and the input from the small input should be large. Unfortunately eq. (3.10) promotes the opposite.

Generalization

Generalization is the networks performance on a set of test patterns $\vec{x}^{(p)}$ it has never seen before. The network learns from a number of training examples, creating a parameterization of the actual "function". This parameterization is then used for the test patterns, much like interpolating in a lookup-table. The networks generalization performance is usually lower than its performance on the training set, although for very large data sets the performances can be approximately equal. Typical learning curves are depicted in fig. 3.8. The lower curve (test set) reaches a maximum and then falls off, while the upper curve (training set) asymptotically climbs upwards. This is due to *over learning*, i.e. the network is learning the individual training examples, and subsequently losing generalization ability.

- *Optimal generalization:* The upper limit of generalization performance for a classification problem is given by the *Bayes limit* [33], which is the minimum overlap of the multidimensional distributions. This is illustrated in fig. 3.9 for the case of two Gaussian-like distributions in one dimension. For academic problems with overlapping Gaussian distributions, where Bayes limit is easily calculated, BP networks generalize to values very close to this limit [34]. However, for "real" problems with many dimensions, Bayes limit is usually practically impossible to calculate due to the CPU consumption involved.
- *Achieving optimal generalization:* The generalization performance depends mainly on the ratio N_ω/N_p, where N_ω is the number of weights in the network and N_p is the number of training patterns. For a feedforward network with one hidden layer of "thresholding" neurons the generalization error ε is of order $\mathcal{O}(N_\omega/N_p)$ [35]. As a "rule of thumb", one should use at least 10 times more training patterns than the number of weights in the network.
- *Improving generalization:* Adding random noise to the input training patterns smears them and improves generalization. The improvement is however sensitive to the noise level. It is also possible to improve generalization performance in classification tasks after the initial training by using so-called *border patterns.* These are patterns in the training set whose outputs are close to the midpoint value of $g()$ and subsequently are close to the border between the classes. Additional training on such border patterns generally improves the classification performance.

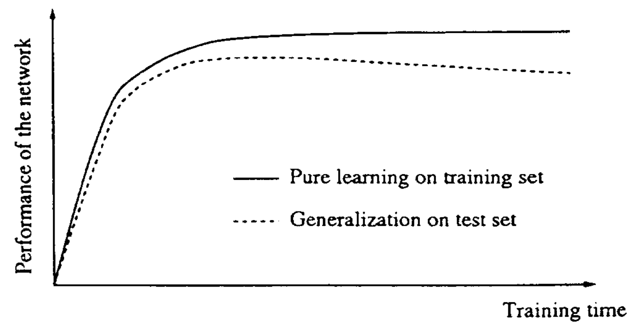


Figure 3.8: Typical behavior of learning curves.

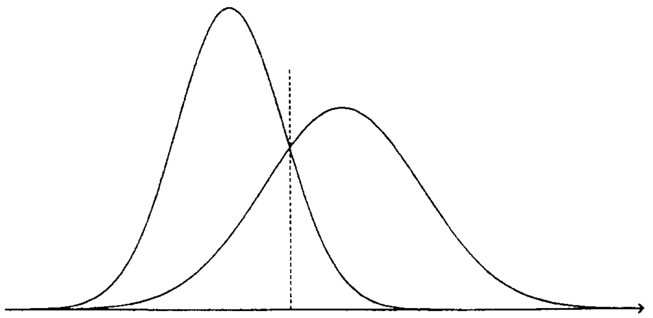


Figure 3.9: Two overlapping Gaussian distributions in one dimension. The optimal cut for the distributions is indicated.

Constraining the network

Restricting the number of weights in the network is important from many aspects. The generalization performance is better, without the need for a huge number of training patterns. The convergence time is shorter, since the number of weights to update is smaller. The latter is very important since one might need to try some different network architectures before the optimal performance is achieved. The best way to restrict the number of weights is to make use of knowledge about the task the network is supposed to perform.

- *Preprocessing:* Preprocessing the data can decrease the number of input units needed and simplify the problem such that fewer hidden units are needed.
- *Symmetries:* Consider the problem of discriminating between two patterns independent of translation and rotation. The possible patterns are represented in a $n \times m$ pixel matrix, with all possible rotations. These patterns can occur anywhere in a $N \times M$ input matrix ($N > n$, $M > m$), but the number of inputs needed to compute the task is only $n \cdot m$ (see fig. 3.10). In this case it is wise to use *receptive fields* [13, 36], i.e. overlapping fields with linked weights. The weights going from corresponding units in the fields to the hidden units are made equal, meaning that each updating to one of the weights is also made to the others. The effective number of weights is thus decreased from $N_\omega \sim \mathcal{O}(N \times M \times N_h)$ down to $N_\omega \sim \mathcal{O}(n \times m \times N_h')$, where N_h is the number of hidden units in the "big" network and N_h' is the number of hidden units that each receptive field connects to.
- *Limited input view:* Even though the problem does not have translational symmetry, it might be "local", meaning that a hidden unit does not need to see all the input units. The hidden units are then connected to overlapping *selective fields* in the input layer.
- *Modularizing the network:* In several cases the actual task can be divided into subtasks, with known target values. For example, a vision task can be divided into a "what" and a "where" subtask [37]. Each subtask might need a small network and be easy to learn. The resulting networks are then merged, i.e. their output is used as input to a new network which is trained to perform the full task. Such modular networks often outperform networks that are trained on the full task from the start [15].

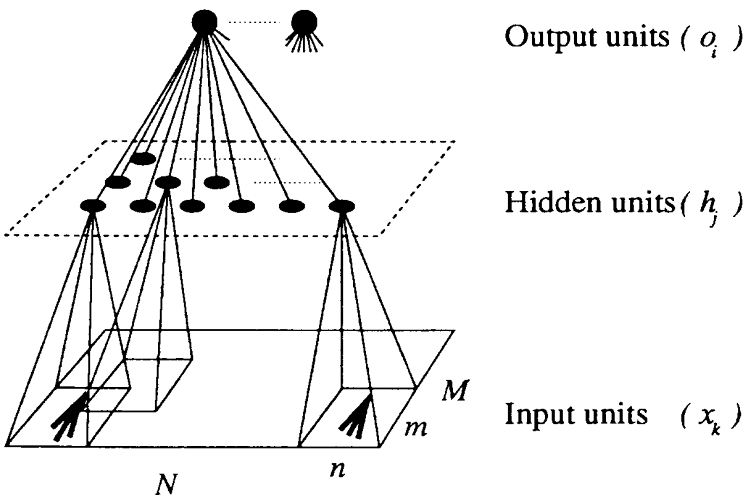


Figure 3.10: Receptive fields; weights connecting corresponding parts of the receptive fields to hidden units are linked together. The two arrow patterns should give similar responses in the output units.

3.2.4 Example: Mirror Symmetry

To illustrate the power of including hidden units we can study the one-dimensional *mirror symmetry* problem [13], which deals with the symmetry of binary patterns $\vec{x} = (x_1, x_2, ..., x_N)$, where $x_i \in \{0,1\}$. The task is to tell whether a specific pattern is symmetric with respect to reflection in its mid-point coordinate or not. For the case of 2 coordinates; the patterns (1,1), (0,0) are symmetric while (1,0) and (0,1) are not, which coincides with the XOR problem. This problem is not linearly separable [12] but can (in theory) be solved for any dimensionality of $\vec{x}$ using only two hidden units. This is a consequence of the fact that all symmetric patterns lie in the same $\lceil N/2 \rceil$-dimensional multiplane Π_S. To make a border around Π_S, the network only has to use two hyperplanes, i.e. two hidden units, that are parallel to Π_S and slightly displaced from it (cf. fig. 3.4b).

A MLP easily learns the one-dimensional mirror symmetry for moderate input dimensions. A possible solution for 4 inputs is shown in fig. 3.11. The weight vectors $\vec{\omega}_1$ = (-1,-2,-4,4,2) and $\vec{\omega}_2$ = (1,-2,-4,4,2) for the hidden units are linear combinations of $\vec{x}_1 = (1,1,-1,-1)$ and $\vec{x}_2 = (1,-1,1,-1)$, which span the subspace orthogonal to Π_S (the coordinate ω_{j0} is the threshold). The output unit is performing a simple OR type function on the two hidden units, it fires only when h_2 is active and h_1 is inactive.

If we train a MLP with 10 hidden units and apply the pruning procedure above (cf. eq. (3.22)), the net decreases the number of hidden units to 2 by cutting the weights to the other units, showing that only 2 hidden units are needed.

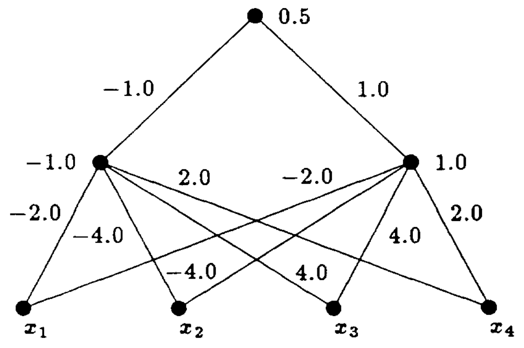


Figure 3.11: A three-layered MLP that solves the one-dimensional mirror symmetry problem with 4 input nodes.

4 Self-organization

The multilayer perceptron is very appealing with its simple structure and proven impressive track record. However, it suffers from some limitations:

- The learning has to be done in a supervised manner, the features o_i have to be known beforehand. This is the most severe limitation.
- It is somewhat cumbersome to analyze the representation that the hidden nodes have built up. For sigmoidal units, the decision regions are not strict polyhedral regions, but rather "polyhedrals" with smooth corners.

A *self-organizing* (SO) network operates in a different way which complements the multilayer perceptron. Self-organizing networks are networks that organize themselves according to the "natural structure" of the data without supervision. This "structure" can be clusters, principal components, prototypes, and/or other typical features. A SO network has usually only one layer of units besides the input units and we will term these units *feature units* from now on and denote them by h_j (see fig. 4.1). The resulting output from these feature units is a reduced representation of the data, reflecting the main features of the data — SO networks can thus be used as *feature extractors* on data.

Self-organizing networks exist on two levels of sophistication. The simple one is the *competitive* or *winner-takes-all*, where only one feature unit is allowed to be on at the time, whereas the more sophisticated *collective* version allows several units to react at the same time and utilizes this to extract relationships in the data. There exist a number of SO algorithms but we will only discuss a few of them here (see ref. [1] for a general review).

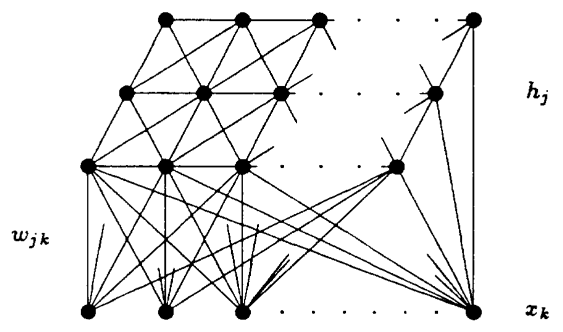


Figure 4.1: A self-organizing network. Note that possible interactions among the feature nodes are of feedback nature.

4.1 Competitive Self-organization

In a competitive network a pattern $\vec{x}^{(p)}$ is presented and the responses of the feature nodes h_j are computed,

$$h_j = G(\vec{x}^{(p)}) \tag{4.1}$$

where the *response function* $G()$ can be of different form, depending on how the input is coded. For each presented pattern, a *winner unit* h_m is picked out among the feature nodes.

$$h_m = \max_j(h_j) \tag{4.2}$$

The weights of this winner unit are then updated such that h_m becomes even more representative of the presented pattern $\vec{x}^{(p)}$. How this updating is done depends on how the "representativity" is measured (clusters, principal components, etc.).

4.1.1 Vector Quantization

Perhaps the most used form of competitive self-organization is *vector quantization* (VQ) [38], which uses an algorithm identical to the *adaptive k-means* clustering algorithm [39]. In vector quantization, each unit h_j is associated with a weight vector $\vec{\omega}_j = (\omega_{j1}, \omega_{j2}, ..., \omega_{jN})$ of the same dimensionality as the input patterns $\vec{x}^{(p)} = (x_1^{(p)}, x_2^{(p)}, ..., x_N^{(p)})$. The "representativity" is measured in (Euclidean) distance between $\vec{\omega}_j$ and $\vec{x}^{(p)}$. For each presented pattern $\vec{x}^{(p)}$, the distance between the weights vectors and the pattern is computed and the winner unit is the unit closest to the pattern.

$$h_m = \min_j(|\vec{\omega}_j - \vec{x}|) \tag{4.3}$$

This is analogous to using a response function

$$G(\vec{x}) = \exp(-|\vec{\omega}_j - \vec{x}|^2/T) \tag{4.4}$$

for the feature units, where the "temperature" T sets the width of the Gaussian. The weight vector $\vec{\omega}_m$ belonging to the winner unit is then moved closer to $\vec{x}^{(p)}$ according to

$$\Delta\vec{\omega}_m = \eta(\vec{x}^{(p)} - \vec{\omega}_m) \tag{4.5}$$

where η is the learning rate. This basic algorithm is summarized in fig. 4.3.

More insight into the VQ learning algorithm can be gained by studying a more general formulation of eq. (4.5)

$$\Delta\vec{\omega}_j = \eta\delta_{jm}(\vec{x}^{(p)} - \vec{\omega}_j) \tag{4.6}$$

where δ_{jm} is Kronecker's delta. This corresponds to gradient descent on an error function defined as

$$E^{(p)} = \frac{1}{2}\sum_j \delta_{jm}(\vec{x}^{(p)} - \vec{\omega}_j)^2 \tag{4.7}$$

Summing (4.7) over all patterns p we get

$$E = \sum_p E_j^{(p)} = \frac{1}{2}\sum_{p,j} \delta_{jm}(\vec{x}^{(p)} - \vec{\omega}_j)^2 \tag{4.8}$$

which is minimized when $\vec{\omega}_j = \langle\vec{x}^{(p)}\rangle$ for all p $\in \mathcal{J}$, where $\mathcal{J}$ is the set of patterns giving unit h_j as a winner. The weight vectors will thus converge towards "cluster centers" in the data distributions. The $\vec{\omega}_j$'s quantize (hence the name) the input space into polyhedral "compartments" centered around $\vec{\omega}_j$ (see fig. 4.2c). The final network will be extremely easy to analyze; one simply inspects the weight vectors $\vec{\omega}_j$ and they will mimic the specific features that the units are most sensitive to. This is the key virtue of this method.

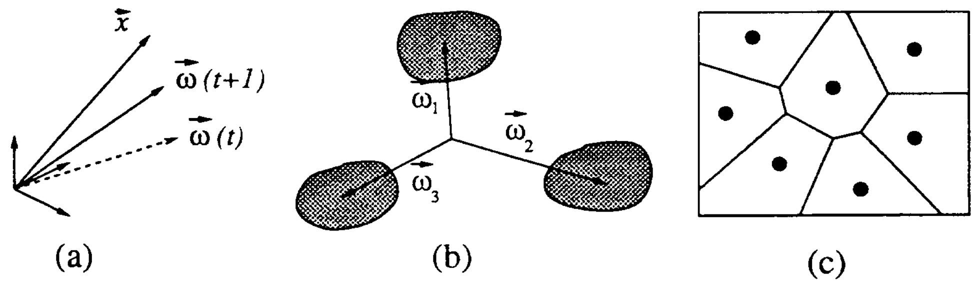


Figure 4.2: **(a)** A weight vector $\vec{\omega}_j$ lining up with an input vector $\vec{x}$. **(b)** Three feature vectors averaging different clusters of input categories. **(c)** The resulting quantization of the input space; each dot corresponds to one weight vector $\vec{\omega}_j$.

```
1. Initialize $\vec{\omega}_j$'s.
2. Repeat until $\vec{\omega}_j$'s have converged:
    2.1 Pick pattern $p$ from the data set.
    2.2 Present input $\vec{x}^{(p)}$ and calculate $h_j$
        according to eqs. (4.1,4.4).
    2.3 Pick out the winner unit $h_m$ (eq. (4.2)).
    2.4 Update the weight vector of the winner accord-
        ing to
        $\vec{\omega}_m(t+1) = \vec{\omega}_m(t) + \eta(\vec{x}^{(p)} - \vec{\omega}_m)$.
```

Figure 4.3: Vector Quantization.

4.1.2 Learning Vector Quantization

Although vector quantization is an unsupervised learning algorithm, for classification purpose one can augment it with supervised learning for fine tuning the units specific to certain features [38]. This is called *learning vector quantization* (LVQ) and amounts to learning correct answers and unlearning incorrect answers. The only change in the learning algorithm (fig. 4.3) is that the updating equation (4.5) is changed into

$$\Delta\vec{\omega}_m = \begin{cases} +\eta(\vec{x}^{(p)} - \vec{\omega}_m) & \text{if } \vec{x}^{(p)} \text{ is correctly classified} \\ -\eta(\vec{x}^{(p)} - \vec{\omega}_m) & \text{otherwise} \end{cases} \tag{4.9}$$

The relative performance of LVQ and BP on pattern classification problems strongly depends upon the dimensionality of the problem. LVQ is more efficient and faster for low dimensional problems whereas BP is more economical for high dimensional problems. This can be seen from the following argument. Consider the case of overlapping distributions in some N-dimensional space and let us assume that these are complex in the sense that the optimal "cut" between them is not one simple hyperplane. LVQ quantizes the input space by using "Gaussians" (cf. eq. (4.4)) and fig. 4.2c) and thus has to fill a volume in the input space to produce a good "cut". The BP feed-forward network on the other hand divides the input space using $(N-1)$-dimensional hyperplanes (see fig. 3.4). The number of units needed for LVQ to perform well on a specific task should therefore scale like a^N, while the number of units needed in the BP network should scale like $a^{(N-1)}$. From "benchmark" studies in refs. [34, 40] it is clear that LVQ has difficulties in reaching Bayes limit for synthetic problems consisting of overlapping high dimensional Gaussians distributions in contrast to BP and other supervised learning algorithms based on sigmoidal units like the Boltzmann machine [41] and Mean Field Theory learning [34, 42][6].

In high energy physics ANN applications where comparisons have been made between LVQ and BP [43, 44], LVQ has never been found to perform better than BP.

4.1.3 Practical Hints

Initial Weight Values

- *VQ:* The weights are initialized with random values, as in the BP case.
- *LVQ:* The simplest way to initialize the weights is to give them the values of some randomly picked data points.

Learning Parameters

The parameters are η and T (eq. (4.4)). The temperature T has only practical importance in computer simulations. It determines the "window" of each unit. It should be chosen $T \sim \mathcal{O}(\langle\vec{x}\rangle^2)$ such that the activations $G(\vec{x}) \sim \mathcal{O}(10^{-1})$.

- *VQ:* The learning rate η should be large in the beginning such that large changes $\Delta\vec{\omega}_j$ can be achieved. It is then lowered as the network converges in order for the network to settle into a stable state.
- *LVQ:* If the weights have been initialized as described above, η must be small all the time, since the initial weights are already quite close to the final solution. Also in this case η is allowed to decrease.

4.2 Collective Self-organization

The previous two sections dealt with competitive self-organizing, where only one unit was selected the winner. In the following section we discuss the case where several units are allowed to react and adapt to a given input $\vec{x}^{(p)}$. The general algorithm introduces a *neighborhood function* $\Lambda(h_j, j)$ defining the group of units that are allowed to be updated. The neighborhood function can depend both on the activation h_j and the index j for the

6) The conclusions with respect to performance of BP in [40] are in error since the architecture used for BP was inappropriate for the problem. We refer to ref. [34] for a proper discussion of this issue.

unit. In the competitive case the neighborhood function is simply the Kronecker delta δ_{jm} (cf. eq. (4.6)). This *collective* updating is then used to extract information on internal relations in the input data.

In section 6.2 (and 8.2.2) an alternative collective self-organizing algorithm is presented where the weights are updated in proportion to the relative strengths of the feature nodes h_j (cf. the Potts updating of eq. (3.14)).

4.2.1 Feature Maps

Feature mapping is a collective version of vector quantization that makes topologically correct maps of the input space onto a "feature map". For this, a geometrical topology has to be defined for the feature units h_j, usually a N-dimensional lattice (see fig.4.1 where $N = 2$). The error function (4.7) is modified into [45]

$$E^{(p)} = \frac{1}{2} \sum_j \Lambda_{jm} (\vec{x}^{(p)} - \vec{\omega}_j)^2 \tag{4.10}$$

where Λ_{jm} is a function of the (Euclidean) distance d_{jm} between units j and m in the lattice. The typical shape for Λ_{jm} is Gaussian

$$\Lambda_{jm} \propto e^{-d_{jm}^2/\lambda^2} \tag{4.11}$$

where λ is a parameter determining the width of the neighborhood.

When a pattern $\vec{x}^{(p)}$ is presented to the network, the winner unit h_m is selected and the weight vectors are updated according to

$$\Delta\vec{\omega}_j = \eta \Lambda_{jm} (\vec{x}^{(p)} - \vec{\omega}_j) \tag{4.12}$$

where the neighborhood width λ (eq. (4.11)) is decreased as training proceeds. This causes close neighbors to the winner unit to be updated in the same direction as the winner. In this way neighboring units will end up with similar weight vectors and the resulting network will be topologically correct (if possible), i.e. neighboring points in the input space activate neighboring units in the network.

```
1. Initialize the $\vec{\omega}_j$'s.
2. Repeat until $\vec{\omega}_j$'s have converged:
    2.1 Pick pattern $p$ from the data set.
    2.2 Present input $\vec{x}^{(p)}$ and calculate $h_j$
        according to eqs. (4.1,4.4).
    2.3 Pick out the winner unit $h_m$ (eq. (4.2)).
    2.4 Update the weights according to
        $\vec{\omega}_j(t+1) = \vec{\omega}_j(t) + \eta\Lambda_{jm}(\vec{x}^{(p)} - \vec{\omega}_j)$

        where $\Lambda_{jm}$ is given by eq. (4.11).
```

Figure 4.4: Feature mapping.

4.2.2 Practical Issues and "Rules of Thumb"

Even though the general mapping algorithm (fig. 4.4) is very simple, there exist a number of factors that are crucial for the final result. These include the learning parameters and the neighborhood function Λ_{jm}.

Convergence of the Map

An important theoretical question is if the algorithm converges to a stable state at all. There is no proof for the general N-dimensional map, but it can be shown for one- and two-dimensional maps that they will converge to an equilibrium state [45] if the learning rate goes to zero with time ($\eta(t) \overset{t\to\infty}{\longrightarrow} 0$).

Features of the Map

The quality of the resulting map is controlled by the topology of the map in relation to the topology of the problem (see examples below). The dimensionality and shape of the feature unit setup must resemble the dimensionality and shape of the problem to produce a high quality mapping.

- *Dimensionality of the map:* For the mapping to be useful, the feature map must be of lower dimensionality than the input space. The dimension of the problem is however not necessarily equal to the input dimension, the data points could be lying on a $(N-1)$ dimensional hypersurface and the problem dimensionality would thus be $(N-1)$. To map all the features of the problem, the feature map has to have the same dimensionality as the problem. Usually, one needs to map only the most important features of the data and a low-dimensional map is sufficient.
- *Shape of the map:* The shape of the map should be as close to the shape of the data distribution as possible (see fig. 4.5). A triangular data distribution maps optimally on a triangular feature map. Furthermore, for closed surfaces, the map must have periodical boundaries for a topologically correct mapping to be possible. In most cases, however, the topology of the data is unknown and the aim of the network is to extract this knowledge. In such cases one just has to make a "qualified" guess, which can be verified/falsified and followed by a new mapping.
- *The density of units:* The resulting density $P_\omega(\vec{x})$ of weight vectors will reflect the density $P_p(\vec{x})$ of data points (see fig. 4.6). The feature unit density P_ω is usually an increasing function of P_p, but the relationship is not linear. For a one-dimensional map the relationship can be shown to be $P_\omega \propto P_p^{2/3}$ [46].

The Neighborhood

The shape and width of Λ_{jm} is important for good performance. The shape influences the handling of "conflicts" and the width controls the *plasticity* of the map.

- *Shape:* Λ_{jm} is usually either a Gaussian (eq. (4.11)) or a simple square function

$$\Lambda(j,m) = \begin{cases} 1 & \text{if } d_{jm} \leq \lambda \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

  where λ is the width[7]. For the one-dimensional map Λ_{jm} should be a convex function with maximum for $j = m$ in order to speed up convergence. If Λ_{jm} is concave, a number of metastable states occur and convergence is very slow [47]. Also, if Λ_{jm} is asymmetric, "conflicts" are removed faster and convergence speeded up [48].
- *Width:* The width λ of the neighborhood determines the plasticity of the net. If λ is large, the feature units are strongly coupled together, the network is "stiff". If λ is zero, the feature units are completely free and are allowed to converge to their respective cluster centers, the net is "soft". In the beginning of learning, λ should be large ($\mathcal{O}$(*size of map*)) in order for the network to find the general location of the data. As learning proceeds, λ is decreased (linearly) such that the units can map more details of the data distribution. In cases where the data distributions have a larger dimensionality than the feature map, the final quality of the map can be controlled with λ. This is illustrated in figure 4.5 that shows a map where $\lambda \to 0$ as training proceeds, the feature units converge fully and the map shows much detail, but the topological order is not correct.

Initial Weight Values

The weight values should be initialized such that $\vec{\omega}_j \approx \langle \vec{x} \rangle$.

7) Using the square form is referred to as the "short cut" algorithm.

Learning Parameters

The principal parameters for the feature map are λ, which is discussed above, η and the "temperature" T (eq. (4.4)). The same comments on T as for VQ and LVQ apply also here. The initial η should be large ($0.5 < \eta < 1.0$) such that large changes can be made to the weights. It is then lowered during training such that a stable equilibrium is achieved (see above on convergence).

4.2.3 Examples: Shapes and Densities of Distributions

Triangular Distribution

The input to the network consists of points (x_1, x_2) that are evenly distributed over a triangular area (fig. 4.5a). The feature map is one-dimensional with 50 units. The weights are initialized with random values. The formation of the map is shown at five different stages. It is obvious that a one-dimensional network cannot capture the topology of a two-dimensional distribution perfectly, but the network tries to fill the triangle as well as possible. If the neighborhood size λ is allowed to shrink to zero (fig. 4.5f) the network units are allowed to converge fully, resulting in the loss of topological order. However, if λ is small but non-zero (fig. 4.5d), the main topology can be extracted, but with a loss of detail. Detail is thus achieved through loss of topological information and vice versa.

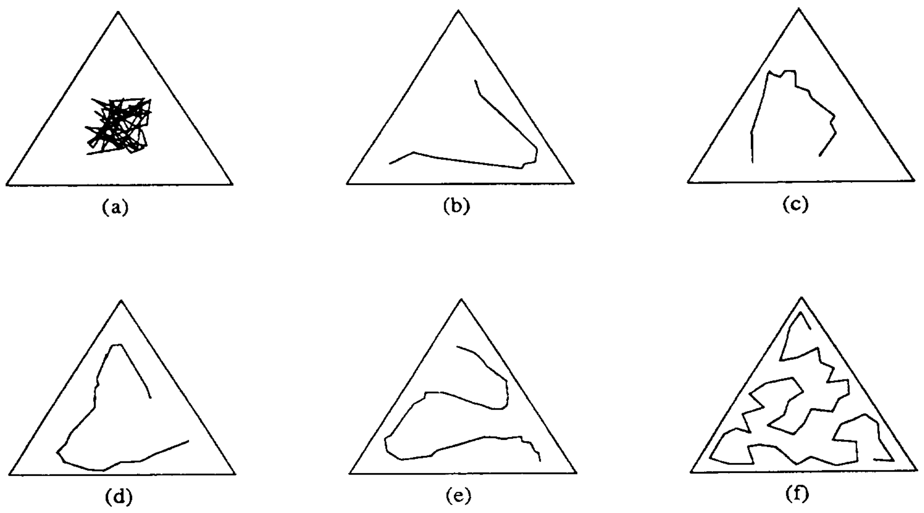


Figure 4.5: Mapping a two-dimensional triangular distribution on to a one-dimensional network with 50 units; from (a) random initial weights to (f) final state. The neighborhood size λ decreases linearly from 15 down to 0.

Gaussian Distribution

The input to the network are points (x_1, x_2) drawn from a two-dimensional Gaussian distribution (fig. 4.6a). The topology of the network is now a 10×10 lattice. The weights are initialized with random values around the center point of the distribution. As the map is being formed several "conflicts" can occur and have to be removed. It is often the removal of these conflicts that takes most time. Convergence can thus be considerably speeded up by smart choices (and adaption during learning) of parameters so that conflicts are more easily solved. The final density of feature units increases towards the center point — the average distance between units reflects the density of data points.

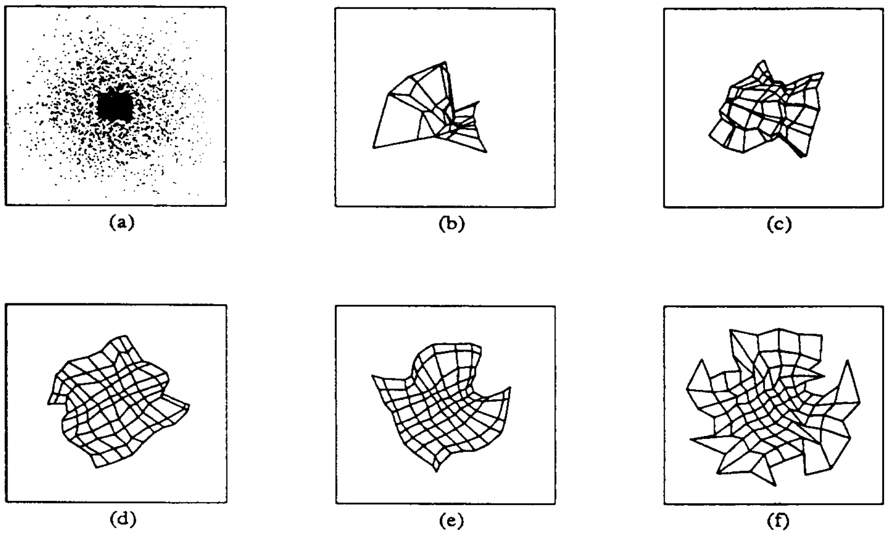


Figure 4.6: The formation of a map of a two-dimensional Gaussian distribution; **(a)** distribution and initial random map, **(f)** final map. The neighborhood size λ decreases linearly from 5 to 0.

5 Feed-back Networks

Feed-back networks appear in the context of associative memories (the Hopfield model [49, 50]) and difficult optimization problems [51, 52] but also in feature recognition applications (the Boltzmann machine [41] and its mean field approximation [34])8). Simple models for magnetic systems have a lot in common with feed-back networks and have hence been the source of much inspiration. We therefore start this section by familiarizing the reader with *Ising models* for magnetic systems.

5.1 Magnetic Systems

The Ising model describes a magnetic system in terms of binary spins $s_i \in \{-1, 1\}$ that are effective variables for the individual atoms. The two spin states at each site represent the possible magnetization directions. The Ising system is governed by the energy function

$$E = -\frac{J}{2} \sum_i s_i s_{i+1} \qquad (5.1)$$

where nearest neighbors interact pairwise with a constant attractive force of strength J. The lowest energy state is reached by iterative updating of

$$s_i = \mathrm{sgn}[J(s_{i-1} + s_{i+1})] \qquad (5.2)$$

which leads to a state where all spins point in one of the two possible directions (see fig. 5.1a). If the system is embedded in a temperature environment (the situation in fig. 5.1a assumes T=0) fluctuations will appear subject to the Boltzmann distribution

$$P(\vec{s}) \propto e^{-E(\vec{s})/T} \qquad (5.3)$$

where the dynamics of eq. (5.2) is replaced by some stochastic procedure leading to fluctuating configurations (see fig. 5.1b).

8) There is also a feed-back version of back-propagation: recurrent back-propagation [53].

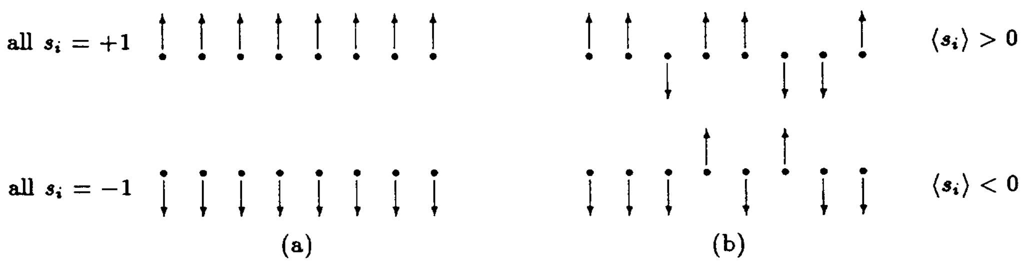


Figure 5.1: (a) The two possible E=0 states for the Ising model at temperature T=0. (b) Two $T \neq 0$ configurations.

Depending on T, the time-averaged values $\langle s_i \rangle$ can point in one direction or the other. At very high temperatures, above the *phase transition* point, there is no alignment at all. All the spins are completely random. A phase transition behavior from an *disordered* phase to an *ordered* one is depicted in fig. 5.2.

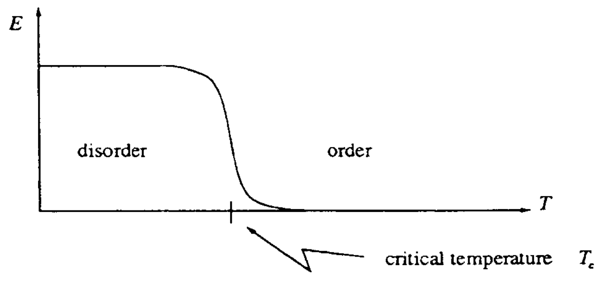


Figure 5.2: E as a function of T illustrating a phase transition.

Such transitional phenomena represent *global* properties of the system. Following the dynamics of individual spins does not tell us very much. The transition into the ordered phase will play an important role in feed-back network applications below — information is order.

Similar features hold for more realistic modeling of magnets in three dimensions . It is interesting and reassuring that spin models are effective theories; all atomic physics details are lumped into *effective* spin variables and the energy function E. This is sufficient for capturing the global properties, or collective properties, of the system.

Let us generalize the Ising model to a *spin glass* system[9] by:

- Allowing for non-local interactions $s_i s_{i+1} \to s_i s_j$ for all $j \neq i$.
- Allowing for different, but symmetric, bond strengths $J \to \omega_{ij} = \omega_{ji}$ between s_i and s_j.

Thus eq. (5.1) is replaced by

$$E = -\frac{1}{2} \sum_{i \neq j} \sum_{j} \omega_{ij} s_i s_j \tag{5.4}$$

The fact that the bonds can be of different signs has the effect that such systems contain conflicting interests — *frustration.* This is illustrated in fig. 5.3; all the constraints (spins connected with positive bonds pointing in the same direction) cannot be satisfied simultaneously. This leads to many "ground states" with almost the same energy in the ordered phase ($T < T_c$). There exist $\approx e^{0.2N}$ such "almost ground states" for a N spin

9) Spin-glasses model certain alloys like AuFe.

system . It is suggestive that such a system could be exploited in information technology — a feed-back network.

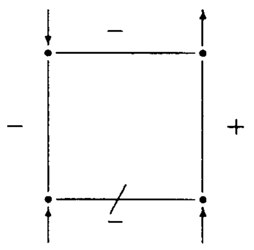


Figure 5.3: 4 spins connected with different sign bonds.

5.2 The Hopfield Model

The Hopfield model [49, 50] is based on the energy function of eq. (5.4) with binary neurons $s_i = \pm 1$. By appropriate choice of ω_{ij} the idea is to let the model function as an associative memory (cf. Section 1.2.2). The dynamics that locally minimizes eq. (5.4) is given by (cf. eq. (5.2)).

$$s_i = \text{sgn}[\sum_{j \neq i} \omega_{ij} s_j] \tag{5.5}$$

Given a set of N_p patterns $\vec{x}^{(p)}$ ($\vec{x}^{(p)} = (x_1, x_2, ..., x_N)^{(p)}$; $p = 1, ..., N_p$), the *Hebb rule* [10]

$$\omega_{ij} = \sum_{p=1}^{N_p} x_i^{(p)} x_j^{(p)} \tag{5.6}$$

is used for learning. For example with

$$\begin{aligned} \vec{x}^{(1)} &= (1, 1, -1, 1, -1) \\ \vec{x}^{(2)} &= (-1, 1, -1, -1, -1) \end{aligned}$$

one gets

$$\begin{aligned} \omega_{12} &= 1 \cdot 1 + (-1) \cdot 1 = 0 \\ \omega_{23} &= 1 \cdot (-1) + 1 \cdot (-1) = -2 \end{aligned}$$

etc.. Thus the Hebb rule increases weights (synapses) between neurons whose activities are correlated and decreases weights between anticorrelated neurons.

With $\{0, 1\}$-neurons, the Hebb rule takes the form

$$\omega_{ij} = \sum_{p=1}^{N_p} (x_i^{(p)} - 1)(x_j^{(p)} - 1) \tag{5.7}$$

The Hebb rule is very appealing since it is both *local* and *incremental.* Since ω_{ij} can take either sign it gives rise to an "interesting" energy landscape like in the spin glass case. To understand how the $\vec{x}^{(p)}$'s are related to the stable states of the energy of eq. (5.1), we compute the *local field* $a_i = \vec{\omega}_i \cdot \vec{s} = \sum_j \omega_{ij} s_j$ when $\vec{s} = \vec{x}^{(q)}$ (q = one of the stored states) using the Hebb rule for ω_{ij}

$$a_i = \sum_{j=1}^{N} \omega_{ij} s_j \quad = \quad \sum_{j=1}^{N} \sum_{p=1}^{N_p} x_i^{(p)} x_j^{(p)} x_j^{(q)}$$

$$\begin{aligned} &= [p=q] + [p \neq q] \\ &= x_i^{(q)} \sum_{j=1}^{N} x_j^{(q)} x_j^{(q)} + \sum_{p \neq q}^{N_p} x_i^{(p)} \sum_{j=1}^{N} x_j^{(p)} x_j^{(q)} \\ &= N x_i^{(q)} + \delta_i \end{aligned} \tag{5.8}$$

where we have used $|\vec{x}^{(q)}|^2 = N$. The two terms in eq. (5.8) represent signal and noise ("crosstalk") with respect to having $\vec{s}=\vec{x}^{(q)}$ as a stable state to eq. (5.5). For completely random patterns (x_i=rand{ -1,1}) one has $\langle\delta_i\rangle=0$ and $\langle\delta_i^2\rangle = N(N_p-1)$, which in the $N \rightarrow \infty$ limit yields a signal/noise ratio

$$R = \frac{\text{signal}}{\text{noise}} = \frac{N}{\sqrt{N(N_p-1)}} = \sqrt{\frac{N}{N_p}} \tag{5.9}$$

R sets the capacity limit for storing patterns in the Hopfield network. For random patterns the capacity is $N_p/N \approx 0.14$ [54], corresponding to $R = 2.7$. Attempting to store more patterns than $0.14N$ will cause "spurious states" (non-memory stable states) $\vec{\xi}$ to appear.

The key advantages with a "Hopfield memory" is that it is *associative* and *robust.* If a pattern $\vec{x}^{(p)}$ is presented to the network where some bits are distorted as compared to a stored pattern $\vec{x}^{(q)}$, the dynamics of eq. (5.2) will complete the pattern (or "correct" it). Also, if some of the weights ω_{ij} are lost, the network will still perform well, since the memory is distributed over all the weights. It is illuminating to think about this in terms of dynamics in the energy landscape[10]. In fig. 5.4 a schematic one-dimensional view of E as a function of different configurations $\vec{s}$ is shown. Stored patterns are represented by local minima in E. A distorted pattern (e.g. some of the bits off as compared to the memorized one) is a state located uphill from the pure pattern. The updating dynamics of eq. (5.5) makes the network slide downhill towards the correct pattern.

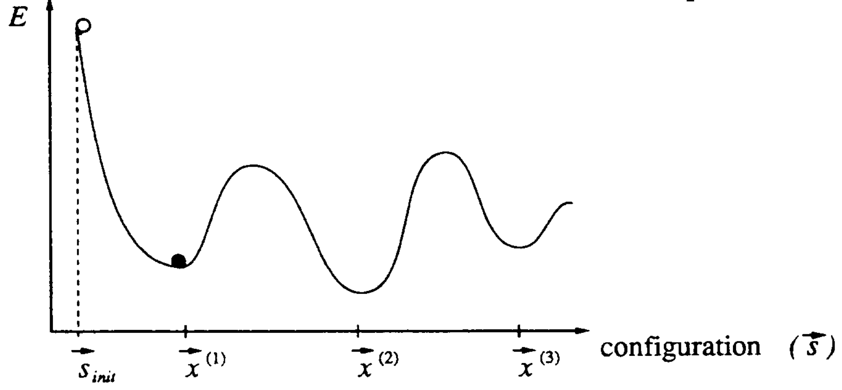


Figure 5.4: Schematic energy landscape with patterns stored with Hebb rule indicated. Open and closed circles denote starting and final states respectively.

The basic steps for the Hopfield model are in fig. 5.5.

Various modifications of eq. (5.6) have been suggested to improve the storage capacity. They fall into two classes: local [55, 56] and non-local [57]. In non-local approaches one computes a pattern correlation matrix C_{pq}

$$C_{pq} = \sum_i x_i^{(p)} x_i^{(q)} \tag{5.10}$$

10) Not the same energy landscape as for BP networks, where E as a function of the ω_{ij}'s is considered.

```
1. "Program" $\omega_{ij}$ using the Hebb rule (eq. (5.6)).
2. Choose a starting state $\vec{s} \approx \vec{x}^{(p)}$.
3. Repeat until $\Delta\vec{s} = 0$:
   3.1 Pick a neuron $s_i$ at random
   3.2 Update according to
         $s_i = \mathrm{sgn}[\sum_{j \neq i} \omega_{ij} s_j]$
```

Figure 5.5: A Hopfield model algorithm

which is then used in a modified Hebb rule

$$\omega_{ij} = \sum_{pq} C^{-1} x_i^{(p)} x_j^{(q)} \tag{5.11}$$

Clearly this procedure results in a more optimized energy landscape. However this is at the expense of not having a local learning rule, which is attractive both from the point of view of biological plausibility and feasibility of designing custom made VLSI hardware.

It is demonstrated in ref. [55] that the performance improves if the rule of eq. (5.6) is supplemented by "unlearning" of the spurious states $\vec{\xi}^{(p)}$. It is also possible to let the network run freely in order to detect the spurious states and remove them "by hand";

$$\omega_{ij} = -\varepsilon \sum_{p=1}^{N'_p} \xi_i^{(p)} \xi_j^{(p)} \tag{5.12}$$

where ε is a parameter and N'_p the number of spurious states. Subsequent work has verified the power of this method with storage capacities in the range $30 - 40\%$ as a result [58]. It has been suggested that during *REM sleep*, mammals let the brain run freely (dream) as a means for removing spurious unwanted memories [59]. This "unlearning" procedure is closely related to the Boltzmann machine learning.

5.3 The Boltzmann Machine

The Hopfield model is very appealing with its local learning rule. However, as stated above, it suffers from two shortcomings:

- Learning is not optimized — spurious states are not removed.
- It cannot be extended to difficult recognition tasks since Hebbian learning cannot handle hidden units — $h_j^{(p)}$ unknown in $\sum_p s_i h_j^{(p)}$.

The Boltzmann machine (BZ) [41] is a learning algorithm that handles these problem while still preserving local learning. It was originally intended for pattern recognition applications but can also be "extended" to the case of no hidden units in the context of content addressable memories [62]. BZ learning learning is unacceptably slow in serial simulations, but (as will be discussed later) its deterministic version, *mean field theory (MFT) learning*, is quite competitive. Furthermore, it reduces to the back-propagation algorithm in the limit of few outputs. The dynamics of BZ is based on the Hopfield energy function (eq.(5.4)). The model learns by making an internal representation of its environment. The learning procedure changes weights so as to minimize the distance between two probability distributions, as measured by the G-function or the so-called Kullback measure [18] (cf. eq. (3.13))

$$G = \sum_{\alpha} P_\alpha \log \frac{P_\alpha}{P'_\alpha} \tag{5.13}$$

where P_α is the probability that the visible units, input and output (see fig. 5.6) are collectively in state α when their states are determined by the environment. P_α represents the *desired* probabilities for these states. The corresponding probabilities when the network runs freely are denoted P'_α. G is zero if and only if the distributions are identical; otherwise it is positive. The word "free" either means that all visible units are free or that only the output units are free. In our presentation we will stick to the latter alternative, the formalism is basically the same.

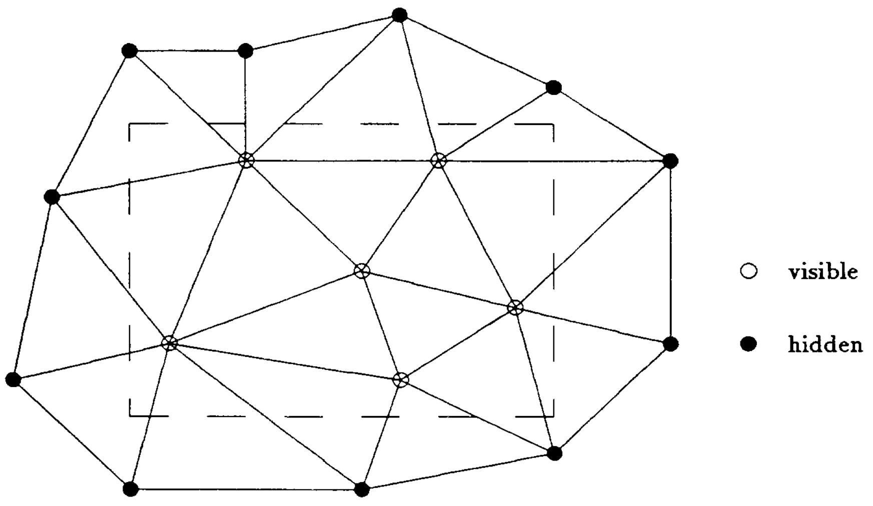


Figure 5.6: Visible and hidden (internal) units in a Boltzmann machine.

The probability P'_α is given in terms of a Boltzmann distribution of the energy $E_{\alpha\beta}$ as

$$P'_\alpha = \frac{1}{Z}\sum_\beta e^{-E_{\alpha\beta}/T} \tag{5.14}$$

where α denotes states of the visible units and β of hidden ones.

Z is the partition function over all possible states of the network

$$Z = \sum_{\alpha\beta} e^{-E_{\alpha\beta}/T} \tag{5.15}$$

G can be rewritten as

$$G = \sum_\alpha P_\alpha \log\frac{P_\alpha}{P'_\alpha} = \sum_\alpha P_\alpha(\log P_\alpha + \log Z - \log Z_\alpha) \tag{5.16}$$

where Z_α is the partition function summed over all possible states whe n the visible units are clamped in state α;

$$Z_\alpha = \sum_\beta e^{-E\alpha\beta/T} \tag{5.17}$$

In order to minimize G we need $\partial G/\partial\omega_{ij}$,

$$\frac{\partial G}{\partial\omega_{ij}} = \sum_\alpha P_\alpha(\frac{\partial\log Z}{\partial\omega_{ij}} - \frac{\partial\log Z_\alpha}{\partial\omega_{ij}}) \tag{5.18}$$

where P_α is constant. For the two terms one gets

$$\frac{\partial\log Z}{\partial\omega_{ij}} = \frac{1}{Z}\sum_{\alpha\beta} s_i s_j e^{E_{\alpha\beta}/T}\cdot(-\frac{1}{T}) = -\frac{\langle s_i s_j\rangle}{T} \tag{5.19}$$

and

$$\frac{\partial \log Z_\alpha}{\partial \omega_{ij}} = \frac{1}{Z}\sum_\beta s_i s_j e^{E_{\alpha\beta}/T} \cdot \left(-\frac{1}{T}\right) = -\frac{\langle s_i s_j\rangle_\alpha}{T} \tag{5.20}$$

and thus for $\partial G/\partial \omega_{ij}$ one has

$$\partial G/\partial \omega_{ij} = -\frac{1}{T}\sum_\alpha P_\alpha[\langle s_i s_j\rangle - \langle s_i s_j\rangle_\alpha] \tag{5.21}$$

Usually one wants all the M stored patterns to have the same probability $P_\alpha = 1/M$, leading to

$$\partial G/\partial \omega_{ij} \propto -(\langle s_i s_j\rangle^c - \langle s_i s_j\rangle) \tag{5.22}$$

where we have introduced the notation $\langle s_i s_j\rangle^c = \langle s_i s_j\rangle_\alpha$ (c = "clamped"). Gradient descent on G gives the updating rule

$$\Delta\omega_{ij} = \eta(\langle s_i s_j\rangle^c - \langle s_i s_j\rangle) \tag{5.23}$$

where η is the learning rate. At some low temperature T_o, $\langle s_i s_j\rangle^c$ is computed when both input and output nodes are clamped to an input pattern and $\langle s_i s_j\rangle$ is computed when only the input nodes are clamped. The first term in eq. (5.23) corresponds to Hebbian learning since the visible units are clamped to the patterns to be learned. The second term is closely related to the REM sleep "unlearning" in eq. (5.12) — spurious states are removed. In this way the Boltzmann machine optimizes learning at the same time as it handles hidden units. The calculation of $\langle s_i s_j\rangle^c$ and $\langle s_i s_j\rangle$ should take place at an approximate minimum of E (eq. (5.4)). In order to ensure this one needs to anneal the system by thermalizing configurations with some stochastic method like the *heat bath* algorithm [27] for a sequence of decreasing temperatures

$$P(s_i \rightarrow 1) = [1 + \exp(\sum_j \omega_{ij} s_j/T)]^{-1} \tag{5.24}$$

1. Initialize ω_{ij} with $\pm$ random values.
2. Repeat until ω_{ij}'s have converged:
   - 2.1 **Clamped phase.** The values of the input and output units of the network are clamped to a training pattern, and for a sequence of decreasing temperatures $T_n > T_{n-1} > \ldots > T_o$, the network of eq.(3.1) is allowed to relax. At $T = T_o$ statistics are collected for the correlations $\langle s_i s_j\rangle^c$.
   - 2.2 **Free phase.**The same procedure as above, but this time the network runs freely or with only the input units clamped. Correlations $\langle s_i s_j\rangle$ are measured at $T = T_o$.
   - 2.3 Update ω_{ij} according to $\omega_{ij}(t+1) = \omega_{ij}(t) + \eta(\langle s_i s_j\rangle^c - \langle s_i s_j\rangle)$

Figure 5.7: A Boltzmann machine algorithm.

The Boltzmann machine algorithm is summarized in fig. 5.7. Needless to say this is a very CPU time consuming process due to the annealing and thermalization. In the next section we will discuss the *mean field approximation*, which replaces the stochastic updating of eq. (5.24) with a set of deterministic equations.

5.4 The Mean Field Approximation

Consider the Hopfield energy function (eq. (5.4)) and the corresponding updating equation (eq. (5.5)), which in terms of the *local field* u_i[11] (cf. eq. (3.7))

$$u_i = -\frac{\partial E}{\partial s_i} = \sum_{j \neq i} \omega_{ij} s_j \tag{5.25}$$

reads

$$s_i = \mathrm{sgn}[u_i] \tag{5.26}$$

This updating equation is based on gradient descent and is hence suitable for associative memory applications; given an initial configuration it takes us to the closest local minimum. Other applications like feature recognition in BZ or optimization problems (see next section) require that an approximate global minimum of eq. (5.4) is reached. In BZ we use the very time consuming *simulated annealing* stochastic procedure. The key idea in the MFT approach is to approximate u_i by its thermal average

$$u_i \approx \langle u_i \rangle_T = \sum_{j \neq i} \omega_{ij} \langle s_j \rangle_T = \sum_{j \neq i} \omega_{ij} v_j \tag{5.27}$$

where $v_j = \langle s_j \rangle_T$. We will not derive the MFT approximation here in detail but refer the reader to refs. [42, 52, 60]. To obtain this approximation one starts by rewriting the partition function sum Z (eq. (5.15)) as

$$Z = const \cdot \int (\prod_i^N dv_i) e^{-\frac{1}{T}\sum_{ij} \omega_{ij} v_i v_j + \sum_i \log\cosh(\sum_j \omega_{ij} v_j / T)} \tag{5.28}$$

One then assumes (*saddle point* approximation) that the contribution to Z is dominated by the minima of the exponent (E') of eq. (5.28). Taking $\partial E'/\partial v_i = 0$ yields the MFT equations

$$v_i = \tanh(\sum_j \omega_{ij} v_j / T) \tag{5.29}$$

In other words; the stochastic updating process is being emulated by a set of deterministic equations with sigmoid gain functions (cf. eq. (2.2,2.3)). At T=0 the Hopfield step-function updating rule is recovered. In general, this approximation is supposed to hold for large N. In ANN applications the approximation is extremely reliable (see e.g. refs. [42, 52]). For [0,1] neurons the MFT equations read

$$v_i = \frac{1}{2}[1 + \tanh(\sum_j \omega_{ij} v_j / T)] \tag{5.30}$$

The key advantage of the MFT approach is speed - stochastic updating is replaced by a set of deterministic equations. Another virtue is that one gets a transparent probabilistic interpretation of neuronic outputs since $v_i = \langle s_i \rangle_T$. Also these equations are isomorfic to RC-equations and hence naturally map onto custom made hardware (see Section 7).

5.5 Mean Field Learning

Armed with the MFT approximation we now revisit the Boltzmann machine, where the origin of the time consumption is the stochastic computation of $\langle s_i s_j \rangle_T$ etc. stochastically. From the definition of Z (eq. (5.15)) one has

$$\langle s_i s_j \rangle_T = -T \frac{\partial \log Z}{\partial \omega_{ij}} \tag{5.31}$$

11) In feed-back networks it is customary to denote the local field by u_i rather than a_i, which is the feed-forward (fan-in) counterpart.

Using eq. (5.28) one can then evaluate $\langle s_i s_j \rangle_T$ directly in terms of the MFT variables v_i

$$\begin{aligned} \langle s_i s_j \rangle_T &= -v_i v_j + 2v_j \tanh(\sum_k \omega_{ik} v_k / T) \\ &= v_i v_j \end{aligned} \qquad (5.32)$$

This means that the BZ updating equation takes the simple form

$$\Delta\omega_{ij} = \eta(v_i^c v_j^c - v_i v_j) \qquad (5.33)$$

In summary this MFT algorithm follows the steps of fig. 5.8.

1. Initialize ω_{ij} with $\pm$ random values.
2. Repeat until ω_{ij} has converged:
   - 2.1 **Clamped phase.** The values of the input and output units of the network are clamped to a training pattern, and for a sequence of decreasing temperatures $T_n > T_{n-1} > \ldots > T_o$, eqs. (5.29) are solved iteratively. At $T = T_o$ $v_i v_j$ is computed.
   - 2.2 **Free phase.**The same procedure as in step 3, but this time the network runs freely or with only the input units clamped. Again $v_i v_j$ is computed at $T = T_o$.
   - 2.3 Update ω_{ij} according to $\omega_{ij}(t+1) = \omega_{ij}(t) + \eta(v_i^c v_j^c - v_i v_j)$.

Figure 5.8: A mean field learning algorithm.

The performance of this MFT learning algorithm is comparable with BZ and BP [42]. The expected speedup as compared to BZ has been verified [42]. In the limit of few output units it reduces to BP if the cross-entropy error (eq. (3.12)) is used for the latter [61, 87]. In associative memory applications it gives rise to capacities $N_p \sim \mathcal{O}(10-20)N_H$, where N_H is the number of hidden units [62]. This number should be compared with $N_p \approx 0.14 N_H$ for the Hopfield model.

6 Optimization Problems

Neural networks can also be used to find good approximate solutions to difficult combinatorial optimization problems [51, 52, 64, 65], which are NP-complete[12]. Exact solutions to these problems require state space exploration leading to $n!$ or a^n computations for a system with n degrees of freedom. Different kind of heuristic methods are therefore often used to find reasonably good solutions. The ANN approach, which is based on feedback networks, falls within this category. It has advantages in terms of solution quality and a "fuzzy" interpretation of the answers through the MFT variables. Furthermore it is inherently parallel, facilitating implementations on concurrent processors and with MFT equations custom made hardware is straight-forward to design.

There are two families of ANN algorithms for optimization problems:

– *"Pure" neural approach* based on either binary [51] or multi-state [52] neurons with MFT equations for the dynamics.

12) A problem is NP if there is no known algorithm that solves it in polynomial time. *NP-complete* is a class of NP problems such that if one finds a polynomial solution to one of the members, all the others are solved as well.

- *Deformable templates* [66], where the neuronic degrees of freedom have been integrated out and one is left with coordinates for possible solutions.

The latter pathway is a winner for low-dimensional geometrical problems like the traveling salesman problem (TSP), whereas the former is the only possibility for high-dimensional problems like scheduling. The following sections describe both these approaches illustrated with suitable problems.

6.1 The Neural Approach

6.1.1 The Graph Bisection Problem

The "pure" neural approach is well illustrated with the *graph bisection* problem. This problem is defined as follows (see fig. 6.1a): Partition a set of N nodes with given connectivity into two halves such that the connectivity (cutsize) between the two halves is minimized.

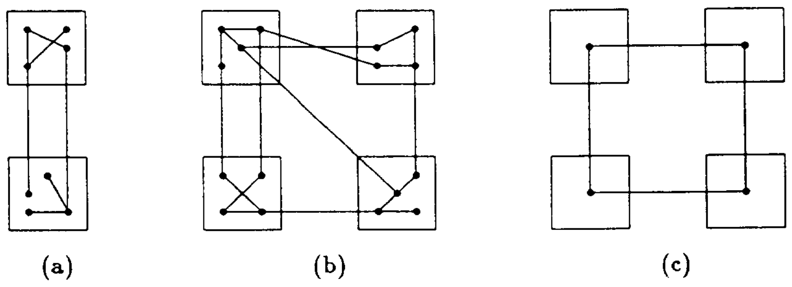


Figure 6.1: (a) A graph bisection problem. (b) A $K = 4$ graph partition problem. (c) A N=4 TSP problem

The problem is mapped onto the Hopfield energy function (cf. eq. (5.4)) by the following representation: For each node, assign a binary neuron s_i and for each pair of vertices $s_i s_j$ $(i \neq j)$ we assign a value $\omega_{ij} = 1$ if they are connected, and $\omega_{ij} = 0$ if they are not. In terms of fig. 6.1a, we let $s_i = \pm 1$ represent whether node i is in the left or in the right position. With this notation,

$$\omega_{ij} s_i s_j \Rightarrow \begin{cases} > 0 & \text{whenever } i \text{ and } j \text{ are in the same partition} \\ = 0 & \text{if } i \text{ and } j \text{ are not connected at all} \\ < 0 & \text{whenever } i \text{ and } j \text{ are in different partitions} \end{cases} \tag{6.1}$$

Minimization of the energy function (5.4) will maximize the connections within a partition while minimizing the connections between partitions, with the result that all nodes are forced into one partition. Hence we must add a "constraint term" to the right hand side of eq. (5.4) that penalizes situations where the nodes are not equally partitioned. We note that $\sum s_i = 0$ when the partitions are balanced and a term proportional to $(\sum s_i)^2$ will subsequently increase the energy whenever the partition is unbalanced. Our energy function for graph bisection thus takes the form:

$$E = -\frac{1}{2} \sum_{ij} \omega_{ij} s_i s_j + \frac{\alpha}{2} (\sum_i s_i)^2 \tag{6.2}$$

where the *Lagrange multiplier* (imbalance parameter) α sets the relative strength between the cutsize and the balancing term. This balancing term represents a *global constraint.* The generic form of eq. (6.2) is

$$E = \text{“}cost\text{”} + \text{“}global\ constraint\text{”} \tag{6.3}$$

which is typical when casting "difficult" optimization problems onto neural networks. The origin of the difficulty is very transparent here; the problem is frustrated in the sense that the two constraints ("cost" and "global constraint") are competing with each other with the appearance of many local minima. (cf. the discussion of spin-glasses above). These local minima can to a large extent be avoided by applying the MFT technique to eq. (6.2) yielding

$$v_i = \tanh(\sum_j(\omega_{ij} - \alpha)v_j/T) \tag{6.4}$$

The generic form of the energy function in eq. (6.2) is very different from a more standard heuristic treatment of the optimization problem. For example in the case of graph bisection one typically starts in a configuration where the nodes are equally partitioned and then proceeds by swapping pairs subject to some acceptance criteria. The constraint of equal partition is respected throughout the updating process. This is in sharp contrast to neural network techniques (eq. (6.2)), where the constraints are implemented in a "soft" manner by a Lagrange multiplier. The final MFT solutions are therefore sometimes plagued with a minor imbalance, which is easily remedied by applying a *greedy heuristic* to the solutions [87].

Very good numerical results were obtained for the graph bisection problem in ref. [67] for problem sizes ranging from 20 to 2000. The quality of the solutions were comparable with those of the CPU demanding simulated annealing method. The time consumption is lower than any other known method. This holds even though the possibility of parallel execution has not been used. In fig. 6.2 we show typical evolutions of systems as functions of number of iterations, it illustrates well the "fuzziness" of the system; some v_i-values never converge to 1 or -1.

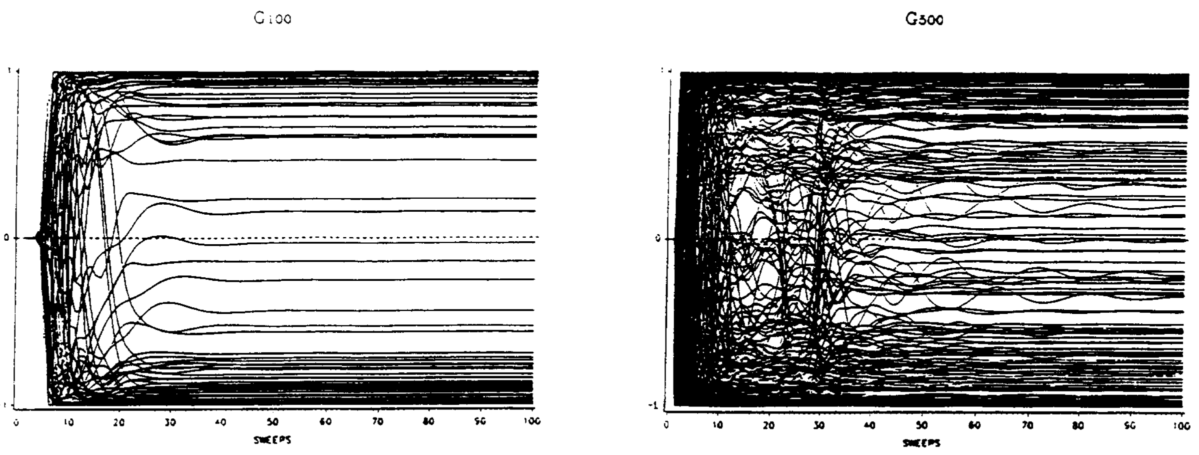


Figure 6.2: Evolution of v_i for a N=100 and N=500 graph bisection problem.

The MFT equations (eq. (5.29)) can for these applications either be solved by iterations at a fixed low temperature T or by annealing in T. Very good numerical results are obtained with this method [67]. The temperature is a free parameter of the system (in addition to α). The system has two phases (cf. fig. 5.2); at large enough temperatures ($T \to \infty$) the system relaxes into the trivial fixed point $v_i^{(0)} = 0$. As the temperature is lowered a phase transition is passed at $T = T_c$ and as $T \to 0$ fixed points $v_i^{(*)}$ emerge representing a specific decision made as to the solution to the optimization problems in question.

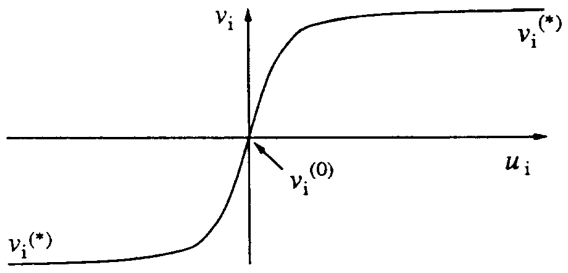


Figure 6.3: Fixed points in $\tanh(u_i)$.

The position of T_c, which depends on ω_{ij} and α, can be estimated by expanding the sigmoid function (tanh) in a power series around $v_i^{(0)} = 0$ (see fig. 6.3). The fluctuations around $v_i^{(0)}$

$$v_i = v_i^{(0)} + \varepsilon_i \tag{6.5}$$

satisfy

$$\varepsilon_i = \frac{1}{T}\sum_j m_{ij}\varepsilon_j \tag{6.6}$$

where $m_{ij} = \omega_{ij} - \alpha$. For synchronous updating it is clear that if one of the eigenvalues to $\mathbf{m}/T$ in eq. (6.6) is > 1 in absolute value the solutions will wander away into the nonlinear regions. In the case of serial updating the philosophy is the same but the analysis slightly more complicated. We refer the reader to ref. [52] for a more detailed discussion. Finding the largest eigenvalue to $\mathbf{m}$ could be computationally explosive by itself. Is there an approximate way of doing it? Yes, there is. Providing $\langle\omega_{ij}\rangle$ and the corresponding standard deviation σ turns out to be sufficient for obtaining estimates within 10% of T_c. Also, this analysis is important for avoiding oscillatory behavior [68], which appears for eigenvalues < -1.

With this method of estimating T_c in advance we thus have a "black box" reliable, parallelizable powerful algorithm for solving problems of this kind.

6.1.2 The Traveling Salesman Problem

When generalizing the graph bisection problem to *graph partitioning* (GP) the N nodes are to be partitioned into K sets, each with N/K nodes, again with minimal cutsize (see fig. 6.1b). This requires introducing a second index for the neurons (*neuron multiplexing*)

$$s_{ia} = 0, 1 \tag{6.7}$$

where the index i denotes the node ($i = 1, ..., N$) and a the set ($a = 1, ..., K$). The neuron s_{ia} takes the value 1 or 0 depending on whether node i belongs to set a or not[13]. The same structure is present in TSP (see fig. 6.1c), where N cities are to visited exactly once each with a minimal total tour length. Let s_{ia} be 1 if city i has the a:th tour number and let d_{ij} be the distance between city i and j. With neuron multiplexing the energy can be written as [51]

$$E = \sum_{ij} d_{ij} \sum_a s_{ia}s_{j(a+1)} + \frac{\beta}{2}\sum_i\sum_{a\neq b} s_{ia}s_{ib} + \frac{\alpha}{2}\sum_a(\sum_i s_{ia} - N)^2 \tag{6.8}$$

where the first term measures the total distance of the tour, the second term penalizes situations where a city appears more than once in the tour and the third term ensures

13) We here use $\{0, 1\}$ notation in order to get a more convenient form of the energy function.

that all cities are visited at least once. Again mean field variables v_{ia} can be defined and the corresponding MFT equations solved. It turns out that the results are extremely parameter sensitive [51, 52].

In ref. [52] an alternative encoding method, *multi-state* or *K-valued* neurons, was proposed. A Potts condition (see section 3.2.2) was imposed to account for

$$\sum_a s_{ia} = 1 \tag{6.9}$$

Thus for every i, s_{ia} is one for only one value of a, and zero for the remaining values of a. So, the allowed values of the vector $\vec{s_i} = (s_{i1}, s_{i2}, ...s_{iK})$ are the principal unit vectors $\vec{e}_1, \vec{e}_2, ..., \vec{e}_K$ in an obvious vector notation (cf. sect. 3.2.2). The number of states available at every node is thereby reduced from 2^K to K, and technically we have a K-state Potts model at our hands. In fig. 3.6 we show the space of states at one node for the case K=3. The energy function of eq. (6.8) can now be rewritten, using the constraint of eq. (6.9), as[14].

$$E = \sum_{ij=1}^{N} d_{ij} \sum_{a=1}^{K} s_{ia} s_{j(a+1)} - \frac{\beta}{2} \sum_{i=1}^{N} \sum_{a=1}^{K} s_{ia}^2 + \frac{\alpha}{2} \sum_a (\sum_i s_{ia})^2 \tag{6.10}$$

We next need MFT equations for Potts neurons (eq. (3.14)). These can be derived in a manner similar to the binary neuron case [52]. They read (cf. eq. (3.14))

$$v_{ia} = \frac{e^{u_{ia}}}{\sum_b e^{u_{ib}}} \tag{6.11}$$

with

$$u_{ia} = \frac{\partial E}{\partial v_{ia}} \frac{1}{T} \tag{6.12}$$

which for K=2 gives reduces to a tanh-function as expected. This expression automatically satisfies the constraint $\sum_a v_{ia} = 1$. Thus, when iterating eq. (6.11), the mean field variables v_{ia} will be forced to live in this (K-1)-dimensional subspace of the original K-dimensional unit hypercube, as shown in fig. 3.6 for the case K=3.

As in the graph bisection case above, approximate values for T_c can be found by linearizing eq. (6.11). This algorithm for TSP, which is of "black-box" nature, is summarized in fig. 6.4 (for graph partition problems the algorithmic steps are identical).

```
1. Choose problem - $d_{ij}$
2. Find the approximate phase transition temperature by linearizing eq. (6.11).
   (for details see ref. [52])
3. Initialize the neurons $v_{ia}$ with $\pm$ random values.
4. Anneal until $\Sigma \equiv \frac{1}{N} \sum_{ia} v_{ia}$ is equal to 0.9:
   4.1 $T_n = 0.9 \times T_{n-1}$.
   4.2 At each $T_n$ update $v_{ia}$;
       $v_{ia} = \frac{e^{u_{ia}}}{\sum_b e^{u_{ib}}}$.
5. After $\Sigma = 0.9$ is reached perform a greedy heuristics to account for possible imbalances or rule violations.
```

Figure 6.4: A Potts neurons algorithm for TSP.

14) This is of course not the only way of coding the problem. One could e.g. interchange the roles of the labels i and a.

This TSP algorithm performs very well in comparison with others — *all* solutions are within $\sim$ 5% from the simulated annealing average [52, 71]. These results are for random distribution of cities. For realistic more structured problems results are even better. The disadvantage of this ANN approach is that N^2 degrees of freedom are needed for N cities. We will next discuss an ANN related approach that avoids this problem of large number of degrees of freedom.

6.2 Deformable Templates

Let us denote the cities in the TSP by $\vec{x}_i$ (see fig. 6.5). We are going to match these cities with template coordinates $\vec{y}_a$ such that $\sum_a |\vec{y}_a - \vec{y}_{a+1}|$ is minimized and that each $\vec{x}_i$ is matched by at least one $\vec{y}_a$. Define s_{ia} to be 1 if a is matched to i and 0 otherwise. The following energy expression then minimizes a valid tour

$$E(s_{ia}, \vec{y}_a) = \sum_{ia} s_{ia} |\vec{x}_i - \vec{y}_a|^2 + \gamma \sum_a |\vec{y}_a - \vec{y}_{a+1}|^2 \tag{6.13}$$

The Lagrange multiplier γ governs the relative strength between matching and tour length (cf. α and β in the neuronic descriptions above). Note that with eq. (6.13) the problem is parameterized in two ways — with the "neurons" s_{ia} and with the template coordinates $\vec{y}_a$. The Boltzmann distribution for the energy in eq. (6.13) reads

$$P(s_{ia}, \vec{y}_a, T) = \frac{e^{-E(s_{ia}, \vec{y}_a)}/T}{Z} \tag{6.14}$$

with the partition function Z given by

$$Z = \sum_{s_{ia}} \sum_{\vec{y}_a} e^{-E(s_{ia}, \vec{y}_a)}/T \tag{6.15}$$

We can now define so-called *marginal distributions* by either integrating out the neuronic or the template coordinate degrees of freedom. If we choose the latter alternative we end up with a pure neuronic description of the problem similar to the one in the previous section. We now choose the former alternative. Performing the sum over s_{ia} and using some "tricks" (see e.g. refs [69, 70]) one obtains the marginal distribution Z_M as

$$Z_M = \sum_{\vec{y}_a} \prod_i \left(\sum_a e^{-|\vec{x}_i - \vec{y}_a|^2/T}\right) e^{\gamma \sum_a |\vec{y}_a - \vec{y}_{a+1}|^2/T} \tag{6.16}$$

Only those configurations where s_{ia} is 1 for only one a for each i have been summed over. We rewrite eq. (6.16) as

$$Z_M = \sum_{\vec{y}_a} e^{-E_{eff}(\vec{y}_a)} \tag{6.17}$$

where the *effective energy* E_{eff} is given by

$$E_{eff}(\vec{y}_a) = -T \sum_i \log\left(\sum_a e^{-|\vec{x}_i - \vec{y}_a|^2/T}\right) + \gamma \sum_a |\vec{y}_a - \vec{y}_{a+1}|^2/T \tag{6.18}$$

Next we minimize E_{eff} with respect to $\vec{y}_a$ using gradient descent

$$\Delta \vec{y}_a = \eta [2 \sum_{ia} \tilde{v}_{ia} (\vec{x}_i - \vec{y}_a) + \gamma (\vec{y}_{a+1} - 2\vec{y}_a + \vec{y}_{a-1})] \tag{6.19}$$

where the Potts factor (cf. eq. (6.11)) $\tilde{v}_{ia}$ is given by

$$\tilde{v}_{ia} = \frac{e^{-|\vec{x}_i - \vec{y}_a|^2/T}}{\sum_b e^{-|\vec{x}_i - \vec{y}_b|^2/T}} \tag{6.20}$$

Thus in contrast to the Potts description in the previous section, the logical units here are not dynamical degrees of freedom, the analog variables $\vec{y}_a$ play that role instead.

How does this algorithm work? At high temperatures T the template cities are located on a closed string with an origin in the center of mass of the cities (see fig. 6.5). As T is lowered the first term in eq. (6.18) becomes more important and matching takes place.

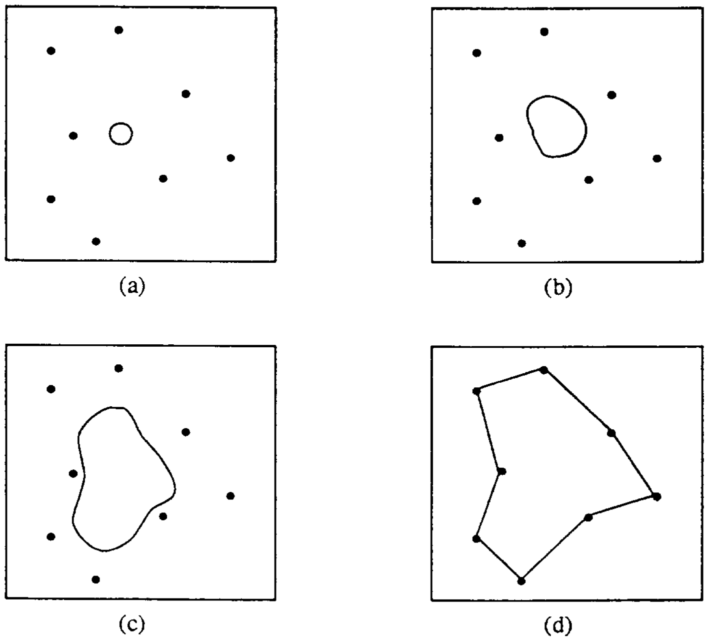


Figure 6.5: Development of the template "snake" from high to low temperatures.

The first term in eq. (6.18) is a sum of Gaussians around the templates with width T. At high T these Gaussians each attract all cities. As T decreases they focus on a smaller set of cities and finally each $\vec{y}_a$ tries to match one $\vec{x}_i$.

This algorithm, which is summarized in fig. 6.6, produces very high quality solutions [71]. And, very importantly, an N-city problem only requires $2N$ degrees of freedom. For problems embedded in low dimensional spaces like TSP the template method is always to prefer over the pure neuronic (Potts) one. However, for high dimensional problems like scheduling [64, 65] one must use the Potts neural network.

The structure of the first term in eq. (6.19) is similar to the one for collective updating of self-organizing networks (eq. (4.12)). The templates $\vec{y}_a$ play the role of Gaussian centers that are updated such that each of them will, at low temperatures (narrow peaks), match exactly one city. At very high temperatures they are attracted to (contain) all cities. The main difference between this approach and the self-organizing networks is the neighborhood function, which for this case is $\Lambda = \tilde{v}_{ia}$, i.e. each "weight" (template) is updated to some degree through the Potts updating equation (eq. (6.20)). At very low temperatures $\tilde{v}_{ia}$ approaches $\delta(\vec{x}_i - \vec{y}_a)$ and the competitive winner-takes-all updating is retrieved.

```
1. Choose problem - $\vec{x}_i$'s.
2. Choose number of templates $\vec{y}_a$, a=1,...,$M$; $M > N$.
3. Compute the center of gravity of the cities $\vec{x}_i$ and displace it slightly with a random seed.
   Place templates $\vec{y}_a$ with equal spacing on a circle around this center.
4. For a sequence of temperatures $T_n$ do:
   4.1 $T_n = 0.9 \times T_{n-1}$.
   4.2 At each $T_n$ update templates $\vec{y}_a$;
       $\vec{y}_a(T_n) = \vec{y}_a(T_{n-1}) + \Delta\vec{y}_a$
       where $\Delta\vec{y}_a$ is given by eq. (6.19).
```

Figure 6.6: An elastic net algorithm for the TSP.

7 Hardware

As we have demonstrated, the ANN approach is very powerful with respect to performance and generality. It is also very appealing from the point of view of concurrency — it easily lends itself to parallel execution. Most applications to date have been of off-line nature and mainly consist in simulating ANN on serial computers, without utilizing the inherent parallelism in the equations. Exploiting this parallelism can proceed in three different ways:

1. Using a general purpose multi-processor system like a CRAY, connection machine, hypercube, transputer system etc..
2. Realizing the ANN in special purpose VLSI electronic hardware, digital or analog.
3. Realizing the ANN with a non-linear optical system.

ANN's are characterized by very elementary processing units and any special purpose hardware should capitalize as much as possible on this fact. Another major property of an ANN is the large degree of connectivity, which poses a real challenge for VLSI implementations. In optical computing, on the other hand, large connectivity is natural. Optical computing has however not yet matured into massive product lines (but is rapidly gaining momentum).

This section is far from complete and exhaustive. It is merely intended as bringing up a few pointers. We refer the interested reader to the cited work.

7.1 VLSI Implementations

VLSI implementations follow two different approaches; digital and analog. Artificial neurons are analog by nature so the latter approach is natural. Digital implementations are expected to give slower chips than analog ones, but substantial efforts have gone into the development of digital transistor implementations of ANN (for a review see e.g. [72]). In what follows we will stick to analog implementations. An example of an analog electronic neuron is shown in fig. 7.1a. It is particularly interesting to look at feed-back analog systems since they are so closely related to the MFT equations (eq. (5.29,5.30)), which are heavily used in optimization applications and in the MFT version of the Boltzmann machine.

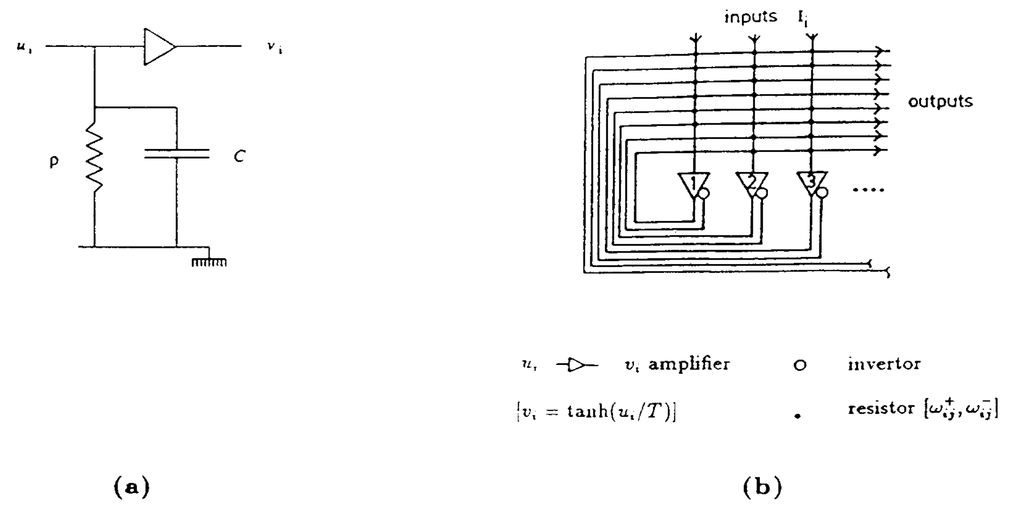


Figure 7.1: (a) An analog electronic neuron. (b) A VLSI implementation of a fully connected network

The MFT equations have a very important property from the VLSI implementation point of view. They are identical to the static limit of the resistance-capacitance (RC) charging equations of a circuit of analog amplifiers [50, 51]. Consider a circuit of nonlinear amplifiers that convert an input voltage u_i to an output voltage $v_i = \tanh(u_i/T)$ and are connected with resistors ω_{ij}^{-1} between amplifier i and j (see fig. 7.1b). Positive and negative[15)] ω_{ij}-elements are implemented by having a pair of connections between the amplifiers; $\omega_{ij}^{(+)}$ and $\omega_{ij}^{(-)}$. Each amplifier is given two outputs, a normal (> 0) and an inverted one (< 0). The weights $\omega_{ij}^{(+)}$ and $\omega_{ij}^{(-)}$ are both positive but interpreted with different signs depending on the sign of the outgoing voltage from the amplifier. The steady-state circuit equations are then given by

$$\frac{u_i}{R_i} = \sum_j \omega_{ij}(v_j - u_i) = 0 \tag{7.1}$$

where R_i are input resistors for the amplifiers leading to reference ground. With the replacement

$$\omega_{ij} \rightarrow (\frac{1}{R_i} - \sum_k \omega_{ik})^{-1}\omega_{ij} \tag{7.2}$$

and substituting $T\tanh^{-1}(v_i)$ for u_i, eq. (7.1) is identical to eq. (5.30). This close mapping between feed-back neural models and VLSI circuitry is indeed very appealing.

Feed-back networks along these lines have been developed. In ref. [74] N=256 fully connected neurons were implemented with e-beam lithography (fixed resistors). In refs. [75, 76] transistors were used for the ω_{ij} to make them modifiable. Ref. [75] also has on-chip learning using the local learning rule of the Boltzmann machine.

An analog feed-forward implementation with off-chip learning (software simulation) is commercially available (ETANN) [77].

7.2 Optical Implementations

Since light beams can cross each other, optical computing is well suited for parallel processing with high connectivity. A key element in ANN processing are matrix multiplications ($u_i = \sum_j \omega_{ij} v_j$), that are very natural to implement optically. To see this, consider

15) Resistors are of course always positive.

a set of light emitting diodes (LED) and a set of photodetectors (PD) with a spatial light modulator in between (see fig. 7.2). With appropriate lenses these devices can perform a matrix multiplication when the SLM is used to store the matrix elements ω_{ij}. As in the VLSI case the connection strengths need to be multiplexed ($\omega_{ij}^{(+)}$ and $\omega_{ij}^{(-)}$) in order to account for negative weights. In fig. 7.2 is indicated how to solve the MFT-equations in a feed-back network by closing a circuit with an amplifier (tanh). Instead of a SLM, which is typically an electronically loaded liquid crystal, one can use a photorefractive crystal to represent $\omega_{ij}^{(+)}$ [73, 78, 79]. In this case the mask is prepared with a volume hologram using coherent light. With this technology one can store approximately 10^9 weights per cm^2, which is smaller than the storage capacity in VLSI but without scaling limitations in the connectivity. The system is also slower than the VLSI alternative ($\mathcal{O}$(ms)) but the very large scale interconnection capacity is hard to beat!

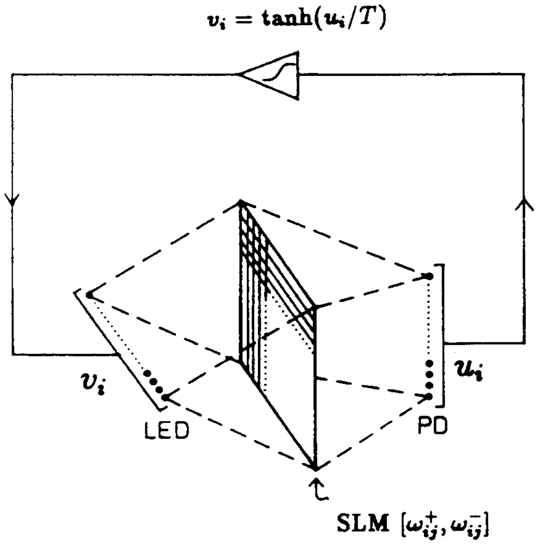


Figure 7.2: A generic optical neural network.

8 High Energy Physics Applications

High energy physics contains many challenging feature recognition problems ranging from off-line data analysis to low-level experimental triggers. Flavor tagging and Higgs detection are obvious examples. With powerful algorithms at our hands, signal-to-background ratios can be substantially reduced. In particular for the next generation of accelerators (LHS, SSC) the availability of algorithms that can be executed in real-time will be crucial. The event rate at these machines is of the order of one event per 10–100 ns. Another class of feature recognition problems is track finding. Again, with the high luminosity expected at LHS and SSC real-time track reconstruction would be advantageous, if not crucial.

It turns out that proof-of-concept for using artificial neural networks (ANN) in high energy physics can be established already in off-line analysis of data from existing accelerators. In the domain of pattern recognition; quark-gluon separation, heavy quark tagging and Higgs search, results have been achieved with feed-forward ANN that are superior to conventional approaches [43, 44, 81, 82, 83, 84, 85]. For function approximation tasks; like reconstructing the mass of a decayed particle from calorimeter information, feed-forward networks have outperformed standard methods [31]. Also promising approaches to use feed-back ANN for solving the optimization problem of track finding have been suggested [86, 87] and have recently been successfully confronted with realistic data [88]. Alternative ANN inspired algorithms based on deformable templates have also been successfully explored for tracking [70, 89].

8.1 Jet Identification

We here describe off-line applications; using ANN for discriminating quarks from gluons and heavy quark tagging.

8.1.1 Quark-gluon Separation

Predicting whether a jet of hadrons originates from a quark or a gluon is important from many perspectives. It can shed light on the hadronization mechanism. For example experimental studies of the so-called string effect needs identification of the gluon jet. Also a fairly precise identification of the gluon jet is required for establishing the existence of the 3-gluon coupling in e^+e^- annihilation reactions. In refs. [81, 82, 83] the back-propagation (BP) algorithm for MLP networks was used to do the separation both in e^+e^- reactions and in hadron-induced large $p_\perp$ processes.

Electron-positron reactions [81, 82]

In order not to reveal too much about the MC model dependent low momentum part of the jet, four-momenta $(\vec{p}_k, E_k)$ of the four leading particles in the jet were used as inputs to the network. The network was a three-layered MLP with 16 inputs, 10 hidden units and one output representing the quark/gluon option (quark $\rightarrow$ 0 and gluon $\rightarrow$ 1). Training and test sets were generated at 2 different energies (92 GeV and 29 GeV) with 3 different MC generators; JETSET [90] , ARIADNE [91] and HERWIG [92]. The inputs were either single jets defined by the LUCLUS clustering algorithm in JETSET [90] or entire 3-jet events (in which case the network was enlarged). After training was completed the network was tested with a middle point success criteria where > 0.5 for the output node was interpreted as a gluon jet and < 0.5 as a quark jet. For the 3-jet events, winner-takes-all criteria for three output units was used to determine the gluon jet.

On the average the network was able to correctly classify 85% of the test set jets. The MC model independence of the results were demonstrated by training on MC data generated by one model and tested on MC data from another. Almost no deterioration in performance was observed. Runs where detector acceptance effects were included showed only $\mathcal{O}(2\%)$ degradation.

The QCD matrix element suppresses gluon jet production as $1/E_{gluon}$. A fair part of the ANN discrimination originates from this property. In order to factor out this matrix element dependence from the intrinsic differences between quark and gluon jets one should train different networks with quark and gluon jets in different energy intervals and combine the answers with the appropriate matrix element predictions. In ref. [84] such a procedure was pursued using more kinematical information than in refs. [81, 82]. The classification power then increases to 92%[16].

Large $p_\perp$ processes [43]

In this case the network gets no lead from QCD matrix element information, since the kinematics of the incoming quarks is unknown. Hence we expect a lower classification performance. Another difference is that in hadron-hadron collisions the momenta and energies of the produced hadrons are available in a "raw" form in terms of towers in a calorimeter representing the transverse energies $E_\perp$. In ref. [83] a set of $p\bar{p}$ events at 630 GeV were generated with the PYTHIA MC [93]. The transverse energy of the fragmentation products was mapped onto a calorimeter with a granularity of $\Delta\eta = 0.20$ in pseudorapidity ($\approx$ longitudinal velocity) and $\Delta\phi = 0.26$ in azimuth angle. The setup corresponded to the UA2 calorimeter at CERN and had a complete coverage in ϕ and extended up to $|\eta| \leq 2$. The size of a jet was defined as a 7×7 matrix in the calorimeter, approximately corresponding to a cone of radius 0.8 in (η, ϕ)-space. The calorimeter information was presented to the network as follows: Take the $E_\perp$ of the leading cell in the 7×7 matrix

16) Including more kinematic information than the four leading hadrons might destroy the MC model-independence of the approach in refs. [81, 82].

and assign it to the first node x_1. Assign the η and ϕ coordinates relative to the center of the jet to x_2 and x_3 respectively. Then take the second leading cell and assign its $E_\perp, \eta$ and ϕ to x_4, x_5 and x_6 and so on for the largest 15 cells. This corresponds to 45 input nodes. The reason for choosing this representation of the input data rather than the 7×7 cells directly is that in this way invariances of the data is more efficiently incorporated. A three-layered MLP was used with 45 inputs, 10 hidden and one output unit. After training, the network correctly classifies 70 – 72% of the jets using the 0.5-criteria (midpoint) as above. However, a simple midpoint performance is not so interesting. Instead one should vary the *cut* of the output node and choose a value corresponding to an optimal efficiency and signal-to-background ratio. In fig. 8.1 the increase in signal-to-background ratio as a function of the signal efficiency is shown. In ref. [83] a similar encoding was successfully used for separating jets stemming from the intermediate vector boson W from those originating from QCD collision processes. Such a network is able to reduce the QCD background to the process $W/Z^0 \rightarrow jets$ by factors 20–30.

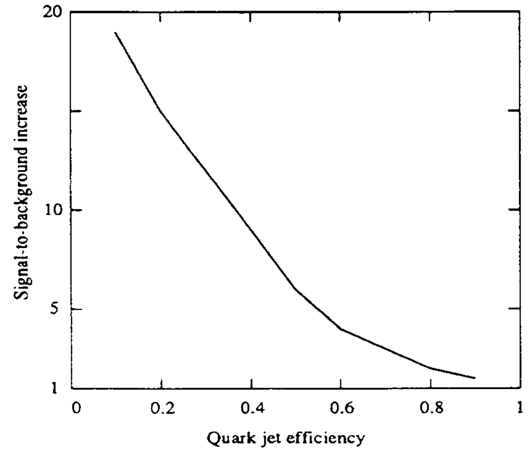


Figure 8.1: The signal-to-background increase versus the efficiency for a neural network quark trigger.

8.1.2 Heavy Quark Jet Tagging

So-called *heavy* quarks (c and b) are produced at high energies. In contrast to u- and d-quarks they are unstable and decay weakly, emitting leptons. The conventional way of identifying heavy quarks in e^+e^- reactions is either through leptonic tagging or secondary vertex with efficiency/purity levels of approximately (5%/95%) and (25%/80%) respectively. We trained a network to identify b-quarks [82], entirely based upon hadronic information. As input variables was used the total jet energy and momentum along with the energy and direction for the 6 leading particles in the jet. Again a three-layered MLP was used, with 20 input units, 10 hidden and a single output neuron ($b \rightarrow 1$ and non-$b \rightarrow 0$). As in the large $p_\perp$ application above the option of varying the cut on the output node can be used to select the desired efficiency vs. purity performance. The network is then able to produce efficiency/purity numbers comparable with what is expected from vertex detectors. This remarkable result implies that a general purpose hadronic detector could be very efficient also for heavy flavor tagging. Even better efficiency/purity ratios were obtained in subsequent work [94] by preprocessing the kinematic variables in terms of different shape variables. In fig. 8.2 the distribution of events for the output node is shown from [94]. It is clear from this figure how choosing the output threshold governs the efficiency/purity ratio.

It is illuminating to study heavy quark tagging in a self-organizing feature map. In ref. [43] such a network consisting of a plane of 7×7 feature nodes was used to disentangle b-, c- and light (uds) quarks. The input layer has 12 nodes, corresponding to p_z and $p_\perp$

for the 6 leading particles in the jet. The resulting distributions over the feature nodes are shown in fig. 8.3. In fig. 8.4a the mapping is shown in terms of dominating quarks. Not surprisingly the two extremes in terms of quark masses occupy two distinct areas separated by the c-quarks. It is very interesting to inspect the corresponding weight vectors $\vec{\omega}_j$. The nodes in the lower right corner of fig. 8.4a have $\vec{\omega}_j$'s with an even distribution of momenta (see fig. 8.4b), which is exactly what one expects for b-quarks since they decay more isotropically.

The neighborhood size λ was only allowed to decrease down to 1 for this map, keeping some "stiffness" in the network. The resulting map (fig. 8.4b) is thus an approximate plane in the dominant (the largest variances) directions of the inputs. In this case the plane is oriented along the p_z for the first and second particles in the jet.

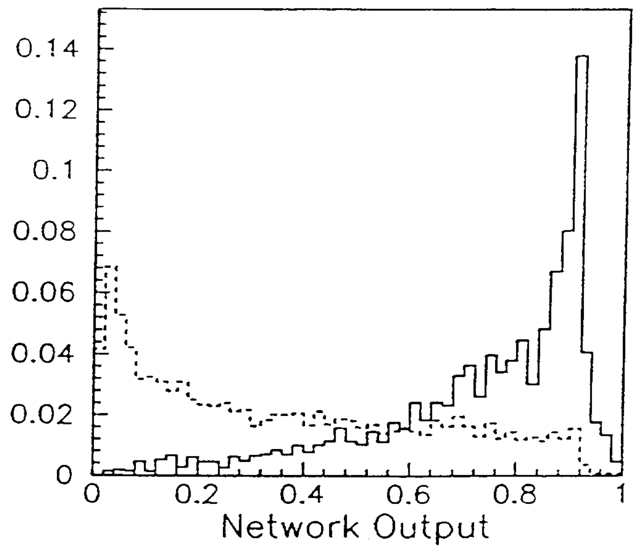


Figure 8.2: Output distributions for b (full line) and non-b (dashed line) hadronic jets (from ref. [84]).

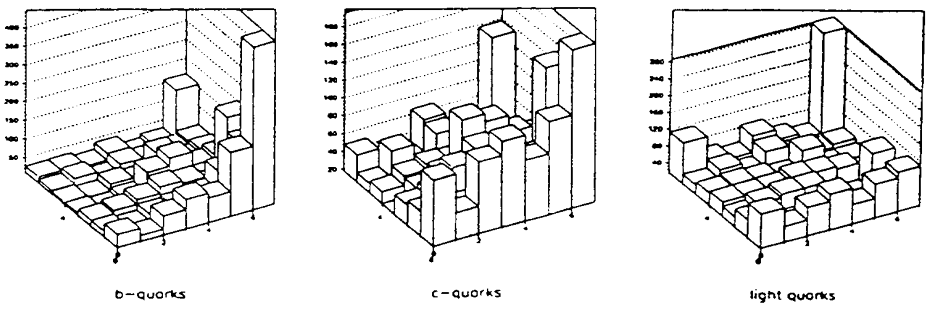


Figure 8.3: Distribution of b-quarks (a), c-quarks (b) and uds-quarks (c) over the self-organized 7×7 feature nodes.

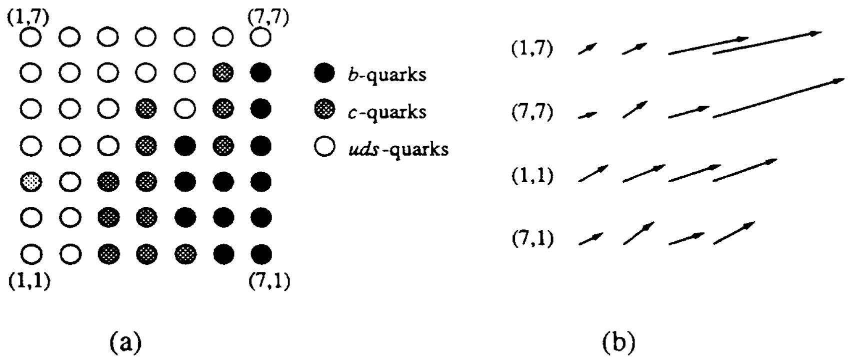


Figure 8.4: **(a)** The resulting map for *uds*- *c*- and *b*-quarks. The shading indicates the dominant flavor for the units. The units are numbered as in fig. 8.3. **(b)** The weight vectors, corresponding to $\vec{p}$ for the 4 leading hadrons, for the corner units; (7,1) is *b*-sensitive and (1,7) is *uds*-sensitive. The $p_\perp$ component has been multiplied by a factor of 5 relative to the p_z component.

8.2 Mass Reconstruction

When hunting for new particles or resonances one often encounters the problem of computing invariant masses of expected decay products. For example, in the case of the intermediate vector boson W (produced in $\bar{p}p$ collisions) in its hadronic decay channel, $W \to q\bar{q} \to$ hadrons, M_W is reconstructed relativistically from the momenta and assumed masses of the produced hadrons. The problem here is that the q and $\bar{q}$ jets are not the only hadrons in the collision - there are also remnants from projectile hadrons. An additional complication is that the q and/or $\bar{q}$ jets can give rise to additional jets through bremsstrahlung. Identifying the appropriate jets is thus crucial for a good reconstruction of the mass. The "standard" procedure [80] for doing this is by sweeping through the calorimeter with a "window" of a certain (ϕ, η) size. The two windows with largest total E_T are selected as containing the two jets and the hadrons in these jets are used to compute M_W.

The neural network approach to this problem is as follows [31]: This is an example of an function approximation task and a linear rather than non-linear output node is used for the answer (M_W). As inputs the 30 largest towers from the calorimeter are used (cf. quark/gluon separation above) together with the total $E_\perp$. A network with two hidden layers with 36 and 15 hidden units respectively is trained with the BP algorithm. As training set MC generated data with a flat distribution of M_W in the range [50,150] GeV is used. When tested on "real" data in terms of MC generated realistic W-masses the ANN approach produces a distribution which is more narrow and symmetric than the one using conventional methods (see fig. 8.5). The main reason why the ANN method does better than the conventional method is that it captures the gluon bremsstrahlung tails well.

8.3 Track Finding

The track finding problem is to construct smooth curves through a set of signal points subject to application specific requirements. Typical applications are determination of moving target trajectories and tracks in high energy physics experiments. Common for these applications is the existence of a set of N signal points. The task is to connect the N signal points into continuous smooth tracks.

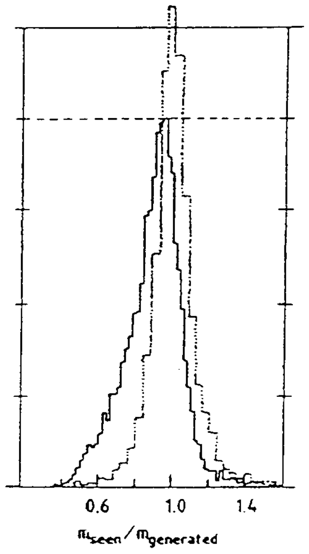


Figure 8.5: The reconstructed mass (M_W) divided by the true mass (M_W^0) using the ANN method (dotted line) and the conventional "window" method (full line).

8.3.1 The Neural Approach

In refs. [34, 86] binary neurons s_{ij} were introduced to represent whether the decision is to connect points i with j or not. Consider an energy function of the form

$$E = E^{(cost)} + E^{(constr)} \tag{8.1}$$

where the "cost" part $E^{(cost)}$ is chosen such that short adjacent segments with small relative angles are favored. In ref. [87] the following choice was made

$$E^{(cost)} = -\frac{1}{2}\sum_{ijkl}\delta_{jk}\frac{\cos^m\theta_{ijl}}{r_{ij}+r_{jl}}s_{ij}s_{kl} \tag{8.2}$$

where m is an odd integer and the line segments r_{ij} and angles θ_{ijl} are defined in fig. 8.6.

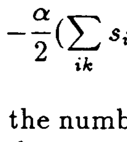


Figure 8.6: Definition of segment lengths r_{ij} and angles θ_{ijl} between segments

The "constraint" term in eq. (8.1) has two parts $E_1^{(constr)}$ and $E_2^{(constr)}$, where $E_1^{(constr)}$ takes care of the requirement that there should be no bifurcating tracks

$$E_1^{(constr)} = -\frac{\alpha}{2}(\sum_{ik} s_{ij}s_{kj} + \sum_{jl} s_{ij}s_{il}) \tag{8.3}$$

The other part, $E_2^{(constr)}$, ensures that the number of neurons that are "on" roughly equals the number of signals N. It takes the form

$$E_2^{(constr)} = \frac{\beta}{2}(\sum_{ij} s_{ij} - N)^2 \tag{8.4}$$

In eqs. (8.3,8.4) α and β are Lagrange multipliers.

The MFT equations

$$v_{ij} = \frac{1}{2}[1 + \tanh(-\frac{\partial E}{\partial v_{ij}}\frac{1}{T})] \tag{8.5}$$

are employed to minimize E. In refs. [86, 87] these equations produced good solutions to moderately sized problems with appropriate choice of parameters (α, β and T). In figs. 8.7 and 8.8 we show a typical evolution of the solutions and the energy behavior with respect to the number of iterations. There are in principle N^3 operations to be carried out at each iteration. However, due to the local nature of most track finding problems, this number can be substantially reduced; neurons need only to be defined within an interaction radius R_c. With on the average m "active" partners within R_c one needs only $\mathcal{O}(Nm^2)$ computations.

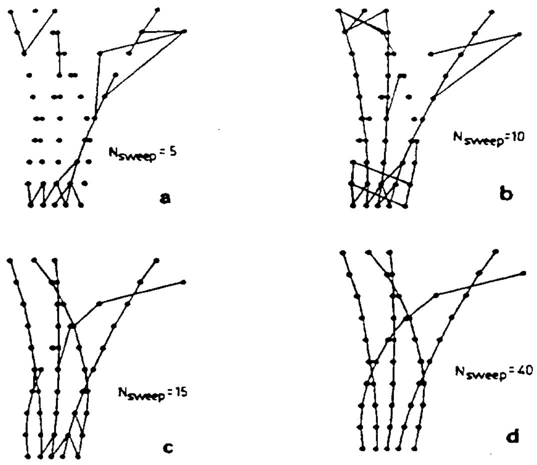


Figure 8.7: Segments with $v_{ij} > 0.1$ at different evolution stages of the MFT equations.

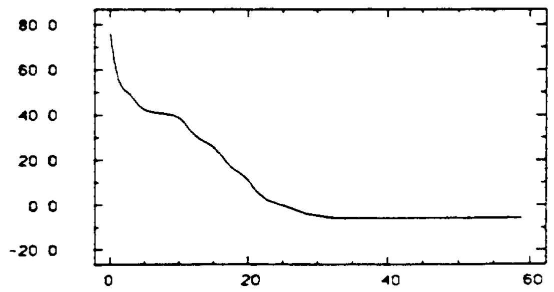


Figure 8.8: Energy as a function of the number of updatings for the problem in fig. 8.7.

In ref. [88] the number of active neurons was further reduced under realistic circumstances using data from the cylindrical ALEPH TPC detector at CERN. Only neurons connecting points satisfying the following cuts were included:

- $\Delta\phi < 0.15$ rad
- $\Delta\theta < 0.10$ rad
- difference in pad-row[17] < 4

17) In a cylindrical detector concentrical *pad-rows* around the origin detect the signals.

Also only those neurons corresponding to segments pointing inwards towards the origin exist. In fig. 8.9a segments corresponding to a $Z^0 \rightarrow$ *hadrons* event is shown prior to convergence. In fig. 8.9b the surviving $s_{ij}=1$ neurons are shown for the same event. In general the quality of the solutions are very good. The efficiency of the ANN algorithm is around 99% which is comparable with the conventional tracking method used in ALEPH (99.7%) [88]. In ref. [88] the authors also perform simulation studies with track densities corresponding to LHC and SSC detectors (~200 tracks/event). Fig. 8.10 shows the time consumption of the ANN algorithm and the conventional method [88].

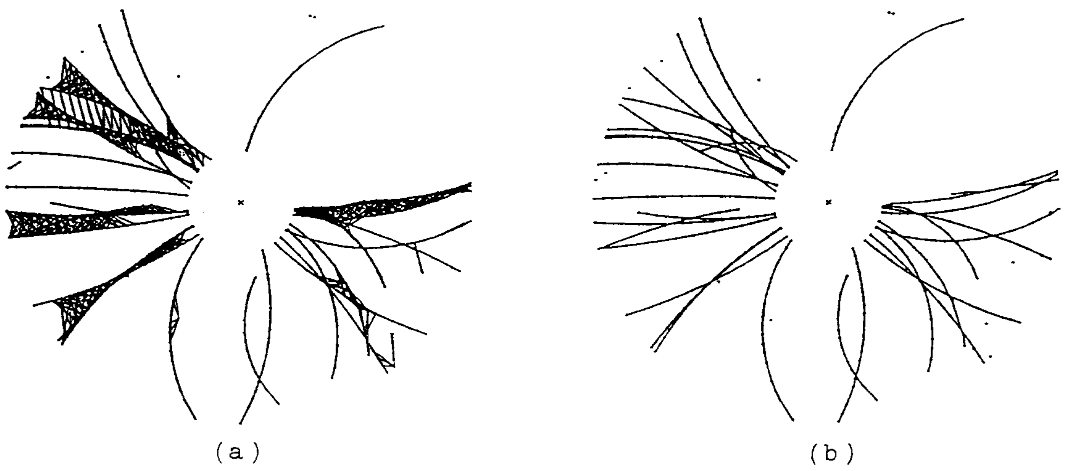


Figure 8.9: **(a)** Active neuron segments for a real $Z^0 \rightarrow$ *hadrons* event (x-y projection) from ref. [89]. **(b)** Final surviving $s_{ij}=1$ neurons for the same event.

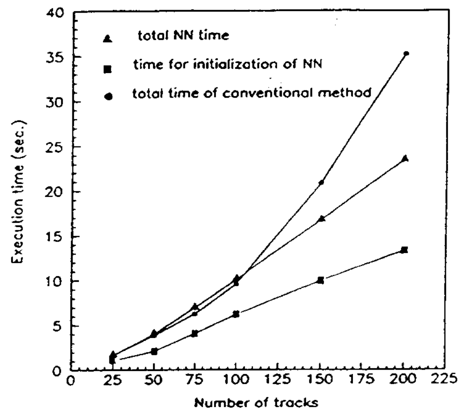


Figure 8.10: Execution time as a function of the number of tracks (from ref. [89]).

Two things emerge from fig. 8.10: One is that the ANN method scales better with the number of tracks than the conventional method. The other is that approximately 60% of the ANN time consumption is due to initialization of the network; computing weights from the observed coordinates.

A few variations of the neuronic approach are possible:

- If one knew the number of tracks in advance from e.g. histogramming (see next section) one could have chosen a different neural encoding where a neuron s_{ia} is "on" if point i is assigned to track a.
- One of the soft constraints in eq. (8.3) could have been realized in a "hard" way with Potts encoding, in which case one has $\sum_j s_{ij} = 1$.

Both of these items will be involved in development of the deformable templates method in the next section.

8.3.2 The Deformable Templates Approach

Even though the ANN approach above seems to work very well it may not be the optimal way to proceed for the high energy physics track finding problem:

- It only solves the combinatorial optimization part of the problem; assigns signals to tracks. In reality one also needs to know the momenta corresponding the tracks. In the neural approach one then has to augment the algorithm with some fitting procedure. It would be nice to have an algorithm that does both things simultaneously.
- The neural approach is presumably more powerful than what is needed for this problem. The parametric form of the tracks are known in advance – helices – but the network is given no prior information about this. The fact that the problem is fairly easy to solve for the ANN algorithm is reflected in a very smooth phase transition (see fig. 8.8). However, for other applications with no prior knowledge of the parametric form of the tracks, the very versatile ANN approach is the way to go.
- The number of degrees of freedom needed to solve the N signal problem is large even with the connectivity restrictions imposed in refs. [87, 88]. For a problem with N signals and M tracks one should only need N$\times$M degrees of freedom.
- As demonstrated in ref. [89] the neural approach is somewhat sensitive to noise. Again with prior knowledge of the parametric form one should be more robust with respect to noise.

All of these issues can be accounted for in the deformable templates (elastic nets) approach [70] discussed above for the TSP. A similar approach was independently pursued in ref. [89]. The strategy is to match the observed events to simple parameterized models where the form of the models contains *a priori* knowledge about the possible tracks — circles passing through the origin (the collision point). In addition, this formulation allows for some data points, hopefully those corresponding to sensor noise, to be unmatched. The mechanism involved is closely related to *redescending M-estimators* used in Robust Statistics [95].

Let us for simplicity limit the discussion to two dimensions, where θ_a and κ_a are the emission angle and the curvature for the a^{th} track. A *Hough transform* [96] is used to give initial conditions for the templates and to specify the number of templates required. Hough transforms are essentially variants of "histogramming" or "binning" techniques which have previously been applied to particle tracking.

Assume that we are armed with an initial set of M deformable templates with coordinates (θ_a, κ_a) from the Hough transform. A fitness measure is defined between the measurement points and the deformable templates as (cf. eq. (6.13))

$$E(s_{ia}, \theta_a, \kappa_a) = \sum_{i,a} s_{ia} M(\theta_a, \kappa_a, \vec{x}_i) + \lambda \sum_i (\sum_a s_{ia} - 1)^2, \tag{8.6}$$

where:

- $\vec{x}_i$ labels the positions of the measurement points ($i = 1, ..., N$).
- θ_a and κ_a are the angle of orientation at the origin and the curvature of the circles with $a = 1, ..., M$ indexing the circles.
- $M(\theta_a, \kappa_a, \vec{x}_i)$, which we abbreviate to M_{ia}, is a measure of distance between the i^{th} point and the a^{th} circle.
- s_{ia} is a binary decision unit (neuron) such that $s_{ia} = 1$ if the a^{th} circle goes through the i^{th} point and zero otherwise.

We want to minimize $E(s_{ia}, \theta_a, \kappa_a)$ with respect to $(s_{ia}, \theta_a, \kappa_a)$, subject to the global constraint that each point is either matched to a unique circle or not matched. More precisely; given i there exist a unique a such that $s_{ia} = 1$. The second term in eq. (8.6) imposes a penalty λ if a specific point is unmatched to any circle.

The Boltzmann distribution for $E(s_{ia}, \theta_a, \kappa_a)$ in eq. (8.6) reads

$$P(s_{ia}, \theta_a, \kappa_a) = \frac{e^{-E(s_{ia},\theta_a,\kappa_a)/T}}{Z} \tag{8.7}$$

As in the TSP case marginal distributions Z_M are obtained by integrating out the neuronic degrees (s_{ia}) of freedom. One gets

$$Z_M = e^{-E_{eff}(\theta_a,\kappa_a)/T} \tag{8.8}$$

where the effective energy E_{eff} is given by

$$E_{eff}(\theta_a, \kappa_a) = -T \sum_i \log(e^{-\lambda/T} + \sum_a e^{-M_{ia}/T}) \tag{8.9}$$

Gradient descent on eq. (8.9) gives

$$\frac{d\theta_a}{dt} = -\frac{\partial E_{eff}}{\partial \theta_a} = -\sum_i \frac{e^{-M_{ia}/T}}{e^{-\lambda/T} + \sum_b e^{-M_{ib}/T}} \frac{\partial M_{ia}}{\partial \theta_a} \tag{8.10}$$

$$\frac{d\kappa_a}{dt} = -\frac{\partial E_{eff}}{\partial \kappa_a} = -\sum_i \frac{e^{-M_{ia}/T}}{e^{-\lambda/T} + \sum_b e^{-M_{ib}/T}} \frac{\partial M_{ia}}{\partial \kappa_a} \tag{8.11}$$

These equations again have the interesting structure that decision elements are mixed with parameter fitting parts.

In ref. [70] it is shown how the Hough transform is a $T \to 0$ and $\lambda \to 0$ limit of eq. (8.6). In ref. [70] it is discussed how this approach is related to that of ref. [89], where also Hough transforms are used to specify the number of tracks and initial parameters values. The template "arms" corresponding to the different tracks are fitted to the data in a serial manner one by one; one elastic net is used per track and an explicit repulsive force is introduced between the tracks to ensure this. The approach here is somewhat different. First of all the elastic net consists of many "arms" finding the solution simultaneously. Repulsive forces between the tracks is implicitly present through the winner-takes-all mechanism of eqs. (8.10,8.11).

The approach of ref. [70] has been explored with simulated DELPHI TPC events with encouraging results (see fig. 8.11) and this algorithm is presently being investigated in depth for a variety of applications [97].

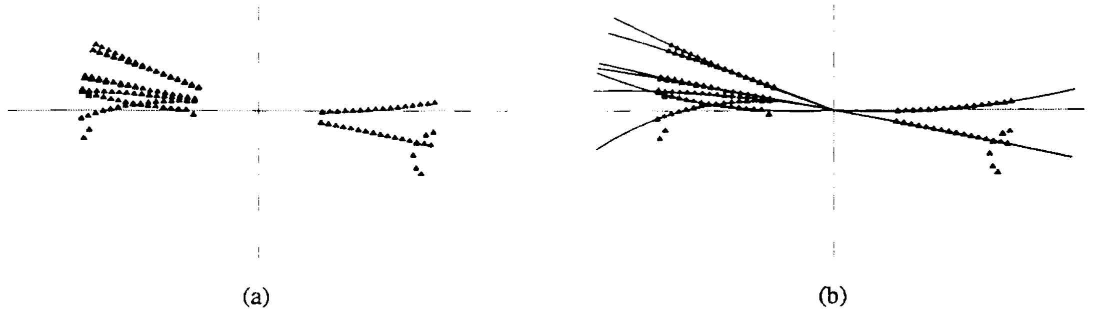


Figure 8.11: (a) Simulated data from DELPHI TPC. (b) Result from hybrid Hough/deformable template system.

Most results reported in this section [81, 82, 83, 43] were obtained using the F77 subroutine package JETNET [99].

Ongoing projects [98] include reproducing the theoretical "QCD calorimeter", given the experimentally measured hadronic and electromagnetic showers, i.e. finding the inverse to the QCD cascading, fragmentation and detector simulation processes. Another challenge [98] is to improve the signal/noise ratio for separating $H^0 \rightarrow W^+W^-$ at LHC/SSC energies where one of the W bosons decays into hadronic jets from a background consisting of $W + jets$, $t\bar{t} \rightarrow W + jets$ and $b\bar{b} \rightarrow l + jets$.

9 Summary

9.1 The Neural Network Approach — What is New?

The question that immediately arises is to what extent the ANN approaches really represent something new or whether they are just conventional methods in new "clothes". We discuss this issue separately for pattern recognition and optimization problem applications.

Feature Recognition

The most commonly used "conventional" method in this application field is *discriminant* analysis, where a discriminant function $\mathcal{F}(\vec{x}^{(p)})$ is formed such that

$$\mathcal{F}(\vec{x}^{(p)}) = \begin{cases} > 0 & \text{if } \vec{x}^{(p)} \in \Omega_1 \\ < 0 & \text{if } \vec{x}^{(p)} \in \Omega_2 \end{cases}$$

for the case of two classes Ω_1 and Ω_2. A summed squared error (cf. eq. (3.1)) is minimized with respect to the parameters occurring in $\mathcal{F}$. Typically a linear function for $\mathcal{F}$ is assumed. The supervised MLP, being able to approximate *any* continuous function for $\mathcal{F}$, subsequently represents a generalization of this discriminant procedure. The MLP is a "black-box" method that also has the advantage of being straightforward to realize in special purpose hardware.

We mentioned in section 4 that competitive the self-organizing vector quantization is identical to the adaptive *k-means-clustering* method. The novel part here is the collective updating variants; topological ordering that makes visualization easy by dimensional reduction onto a space with defined local topology, and soft Potts-type updating, which facilitates to disentangle difficult regions.

Optimization Problems

Optimization is the least explored ANN application area to date, and it is also the one where the gap to conventional methods is largest. By mapping these problems onto feed-back networks one is able to utilize the powerful MFT approximation; which is very efficient for avoiding local minima and leads to a set of deterministic updating equations (in contrast to stochastic methods). Furthermore, the MFT equations are isomorfic to VLSI RC-equations, making the hardware implementation straightforward. The use of self-organizing (or templates) approaches to combinatorial optimization problems is also new.

9.2 Other Algorithms and Applications

These lectures have not been complete with respect to covering all application areas and algorithms. We have confined ourselves to the ones displaying the basic concepts well. For example processing of time-sequenced data (e.g. speech) was not covered, neither was the recurrent back-propagation algorithm [53], which is intimately related to the MFT learning procedure. Also various hybrid schemes exist, among which one should mention the radial basis function approach in ref. [6]. There a set of self-organizing feature nodes (with different widths) are first used to cluster the data, followed by an addition of output nodes and BP learning. We refer to e.g. ref. [1] for treatment of the issues omitted here.

9.3 Finite Bit Precision Requirements

Ultimately, these ANN applications should be implemented in VLSI hardware, in which case one important question arises: To what extent are the solutions sensitive to the bit precision required for amplifiers and weights? This of course requires that both learning and testing is performed at the same level of precision. The option of cutting the precision exists in JETNET [99], and for quark triggering applications, it seems that 2 or 3 bits (except for the sign bit) in both amplifiers and weights are enough to retain good performance [98].

9.4 Outlook for High Energy Physics Applications

We have here demonstrated the power of the neural network paradigm for high energy physics applications. Needless to say the ANN approach looks very promising. The technology is certainly here to stay.

The neural network method is very efficient for extracting features in hadronic data. World record performance can be obtained for quark/gluon separation. With respect to heavy quark tagging the results are in parity with those expected from a vertex detector. A similar network is also able to reduce the QCD background to $W/Z^0 \rightarrow jets$ by a factor 20–30.

The neural network approach is in general very noise- and damage-resistant. Hence it is suitable for high energy physics experiments where various parts of a detector may malfunction. Also, with its inherent concurrency and simple structure, fast execution custom made hardware could be an asset for real-time triggering. Such systems are already underway [100].

In addition to feature recognition ANN will very likely also play an important role in track finding since feed-back networks easily lend themselves to real-time hardware implementations.

Acknowledgments:

We would like to thank L. Gislén, A. Nilsson and M. Ohlsson for valuable comments on the manuscript. These set of lectures benefitted from the nice atmosphere provided by the CERN School of computing organization and its very vital student body. We also thank DEC Sweden for providing the school with ample and up-to-date computer equipment for the hands-on exercises. In particular we thank Jan Andersson for installing the system.

REFERENCES

[1] J. Hertz, A. Krogh and R. G. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley, Redwood City, Ca. (1991).

[2] E.R Kandel and J.H. Schwartz, *Principles of Neural Science*, 3rd ed., Elsevier Publ. :New York (1991).

[3] Scientific American 3, **241**, Sept. 1979.

[4] W. S. McCullough and W. H. Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity", *Bull. math. Biophys.* **5**, 115 (1943).

[5] A. Lapedes and R. Farber, "Nonlinear Signal Processing using Neural Networks: Prediction and System Modeling", Los Alamos Report LA-UR 87-2662 (1987).

[6] J. Moody and C. Darken, "Learning with Localized Receptive Fields", in D. Touretzky, G. Hinton and T.S. Sejnowski (eds.), *Proceedings of the 1988 Connectionists Models Summer School, Carnegie Mellon University*, San Mateo, CA: Morgan Kaufmann (1988);
J. Moody and C. Darken, "Fast Learning in Networks of Locally Tuned Processing Units", *Neural Computation*, **1**, 281 (1989).

[7] R. D. Jones, Y. C. Lee, C. W. Barnes, G. W. Flake, K. Lee, P. S. Lewis and S. Qian, "Function Approximation and Time Series Prediction with Neural Networks", Los Alamos Report LA-UR 90-21 (1990).

[8] A. Weigend, B.A. Huberman and D. Rumelhart, "Predicting the Future: A Connectionist Approach", *International Journal of Neural systems* **3**, 193 (1990).

[9] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Spartan Books, Washington D.C. (1961).

[10] D. Hebb, *Organization of Behaviour*, Wiley, N.Y. (1949).

[11] H. D. Block, "The Perceptron: A Model for Brain Functioning", *Reviews of Modern Physics* **34**, No. 1, 123 (1962).

[12] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, MIT Press (1969 and 1988).

[13] D. E. Rumelhart and J. L. McClelland (eds.) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition (Vol. 1)*, MIT Press (1986).

[14] S. E. Fahlman, "Fast-Learning Variations on Back-Propagation: An Empirical Study", in D. Touretzky, G. Hinton and T. Sejnowski (eds.) *Proc. of the 1988 Connectionist Summer School*, Morgan Kaufmann (1989).

[15] F. Fogelman Soulie, "Neural Network Architectures and Algorithms: A Perspective", in T. Kohonen, K. Mäkisara, O. Simula and J. Kangas (eds.) *Artificial Neural Networks: Proc. of ICANN 91*, Elsevier North-Holland (1991).

[16] T. P. Vogl, J. K. Mangis, A. K. Rigler, W. T. Zink and D. L. Alkon, "Accelerating the Convergence of the Back-Propagation Method", *Biological Cybernetics* **59**, 257 (1988).

[17] E. B. Baum and F. Wilczek, "Supervised Learning of Probability Distributions by Neural Networks" in D. Z. Andersson (Ed.) *Neural Information Processing Systems*, American Institute of Physics, N.Y. (1988).

[18] S. Kullback, *Information Theory and Statistics*, Wiley, N.Y. (1959).

[19] F.Y. Wu, "The Potts Model", *Review of Modern Physics*, **54**, 235 (1983).

[20] R. A. Jacobs, "Increased Rates of Convergence Through Learning Rate Adaption", *Neural Networks* **1**, 295 (1988).

[21] S. Saarinen, R. Bramley and G. Cybenko. "The Numerical Solution of Neural Network Training Problems", Univ. of Illinois Report CSRD 1089 (1991).

[22] M. Mézard and J.-P. Nadal "Learning in Feedforward Layered Networks: The Tiling Algorithm", *Journal of Physics*, **A22**, 2191 (1989).

[23] T. Ash, "Dynamic Node Creation in Backpropagation Networks", Univ. of California, San Diego, Report ICS 8901 (1989).

[24] S. E. Fahlman and C. Lebiere, "The Cascade-Correlation Learning Architecture", Carnegie Mellon Univ. Report CMU-CS 90-100 (1990).

[25] Y. Hirose, K. Yamashita and S. Hijiya, "Back-Propagation Algorithm Which Varies the Number of Hidden Units", *Neural Networks* **4**, 61 (1991).

[26] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth and A.H. Teller, "Equation of State Calculation for Fast Computing Machines", *Journal of Chemical Physics* **21** 1087 (1953).

[27] C.-P. Yang, *Proceedings of Symposia in Applied Mathematics*, Vol. XV (Amer. Math. Soc., Providence, R.I.), 351 (1963).

[28] See e.g. P.T. Soong, *Random Differential Equations in Science and Engineering.* Academic Press: New York (1973).

[29] G. Cybenko, "Approximation by Superpositions of a Sigmoidal Function", *Mathematics of Control, Signals and Systems* **2**, 303 (1989);
K. Hornik, M. Stinchcombe and H. White, "Multilayer Feedforward Networks Are Universal Approximators", *Neural Networks* **2**, 359 (1989);
K. Hornik, M. Stinchcombe and H. White, "Universal Approximation of an Unknown Mapping and its Derivatives using Multilayer Feedforward Networks", *Neural Networks* **3**, 551 (1990).

[30] G. Tesauro and B. Janssens, "Scaling Relationships in Back-propagation Learning", *Complex Systems* **2**, 39 (1988).

[31] L. Lönnblad, C. Peterson and T. Rögnvaldsson, "Mass Reconstruction with a Neural Network", Lund Preprint *LU TP 91-25* (submitted to *Physics Letters* **B**) (1991).

[32] G. Cybenko, "Continuous Valued Neural Networks with Two Hidden Layers are Sufficient", Univ. of Illinois CSRD Report 935 (1988).

[33] R. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, Wiley: New York (1973).

[34] C. Peterson and E. Hartman, "Explorations of the Mean Field Theory Learning Algorithm", *Neural Networks* **2** 475 (1989) .

[35] E. Baum and D. Haussler, "What Size Net Gives Valid Generalization?", *Neural Computation* **1**, n:o 1, 151-160 (1989).

[36] Y. Le Cun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard and L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition", *Neural Computation* **1**, 541(1989).

[37] R. A. Jacobs, M. I. Jordan and A. G. Barto, "Task Decomposition through Competition in a Modular Connectionist Architecture: The What and Where Vision Tasks", Univ. of Massachusetts Technical Report COINS 90-27 (1990).

[38] T. Kohonen, "Self Organized Formation of Topologically Correct Feature Maps", *Biological Cybernetics* **43**, 59 (1982);
T. Kohonen, *Self-organization and Associative Memory*, 3^{rd} ed., Springer-Verlag, Berlin, Heidelberg (1990).

[39] J. MacQueen. "Some Methods for Classification and Analysis of Multivariate Observations", in L. M. LeCam and J. Neyman (eds.), *Proc. 5th Berkeley Symposium on Math. Stat. and Prob.*, U. California Press, Berkeley (1967).

[40] T. Kohonen, G. Barna and R. Chrisley, "Statistical Pattern Recognition: Benchmarking Studies", *Proceedings of the IEEE Second International Conference on Neural Networks*, San Diego, California (1988).

[41] D.H. Ackley, G.E. Hinton and T.J. Sejnowski, "A Learning Algorithm for Boltzmann Machines", *Cognitive Science* **9**, 147 (1985).

[42] C. Peterson and J.R. Anderson, "A Mean Field Theory Learning Algorithm for Neural Networks", *Complex Systems* **1**, 995 (1987).

[43] L. Lönnblad, C. Peterson, H. Pi and T. Rögnvaldsson, "Self-organizing Networks for Extracting Jet Features", Lund Preprint LU TP 91-4 (to appear in *Computer Physics Communications*) (1991).

[44] J. Proriol, J. Jousset, C. Guicheney, A. Falvard, P. Henrard, D. Pallin, P. Perret and B. Brandl, "Tagging B Quark Events in ALEPH with Neural Networks", (To appear in *Proceedings on Neural Networks; from Biology to High Energy Physics*, Elba), (1991).

[45] H. Ritter and K. Schulten, "Convergence Properties of Kohonen's Topology Conserving Maps: Fluctuations, Stability, and Dimension Selection", *Biological Cybernetics* **60**, 59 (1988).

[46] H. Ritter and K. Schulten, "On the Stationary State of Kohonen's Self-Organizing Sensory Mapping", *Biological Cybernetics* **54**, 99 (1986).

[47] E. Erwin, K. Obermayer and K. Schulten, "Convergence Properties of Self-Organizing Maps", in T. Kohonen, K. Mäkisara, O. Simula and J. Kangas *Artificial Neural Networks: Proc. of ICANN 91*, Elsevier North-Holland (1991).

[48] T. Geszti, *Physical Models of Neural Networks*, World Scientific, Singapore (1990).

[49] J.J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities", *Proceedings of the National Academy of Science, USA*, **79** 2554 (1982).

[50] J.J. Hopfield, "Neurons with Graded Response have Collective Computational Properties like those of Two-State Neurons", *Proceedings of the National Academy of Science, USA*, **81** 3088(1984).

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

[52] C. Peterson and B. Söderberg, "A New Method for Mapping Optimization Problems onto Neural Networks", *International Journal of Neural Systems* **1**, 3 (1989).

[53] F.J. Pineda, "Generalization of Backpropagation to Recurrent Neural Networks", *Physical Review Letters* **59**, 2229 (1987).

[54] D.J. Amit, H. Gutfreund and H. Sompolinsky, "Storing Infinite Numbers of Patterns in a Spin-glass Model of Neural Networks", *Physical Review Letters* **55**, 1530 (1985).

[55] J.J. Hopfield, D.I. Feinstein and R.G. Palmer, "Unlearning has Stabilizing Effects in Collective Memories", *Nature* **304**, 158 (1983).

[56] D.J. Wallace, "Memory and Learning in a class of Neural Network Models", *Lattice Gauge Theory - A Challenge to Large Scale Computing*, B. Bunk and K.H. Mutter (eds.), NY, Plenum (1986).

[57] I. Kanter and H. Sompolinsky, "Associative Recall of Memories without Errors", *Physical Review* **A35**, 380 (1987).

[58] D. Kleinfield and D.B. Pendergraft, "Unlearning the Storage Capacity of Content Addressable Memories", *Biophysical Journal* **51**, 47 (1987).

[59] F. Crick and G. Mitchison, "The Function of Dream Sleep", *Nature* **304**, 111 (1983).

[60] R.J. Glauber, "Time-Dependent Statistics of the Ising Model", *Journal of Mathematical Physics* **4:4** 294 (1963).

[61] J.J. Hopfield, "Learning Algorithms and Probability Distributions in Feed-forward and Feed-back Networks", *Proceedings of the National Academy of Science, USA*, **84** 8429 (1987).

[62] E. Hartman, "A High Storage Capacity Neural Network Content-addressable Memory", MCC Technical Report MCC-ACT-NN-173-90 (to appear in *Network: Computation in Neural Systems*).

[63] C. Peterson, "Mean Field Theory Neural Networks for Feature Recognition, Content Addressable Memory and Optimization", *Connection Science* **3**, 3 (1991).

[64] L. Gislén, B. Söderberg and C. Peterson, "Teachers and Classes with Neural Networks", *International Journal of Neural Systems* **1**, 167 (1989).

[65] L. Gislén, B. Söderberg and C. Peterson, "Scheduling High Schools with Neural Networks", Lund Preprint LU TP 91-9 (submitted to *Neural Computation*) (1991).

[66] R. Durbin and D. Willshaw, "An Analog Approach to the Traveling Salesman Problem using an Elastic Net Method". *Nature* **326**, 689 (1987).

[67] C. Peterson and J.R. Anderson, "Neural Networks and NP-complete Optimization Problems; A Performance Study on the Graph Bisection Problem", *Complex Systems* **2**, 59 (1988).

[68] M.Y. Choi and B.A. Huberman, "Digital Dynamics and the Simulation of Magnetic Systems", *Physical Review* **B 28**, 2547 (1983).

[69] A.L. Yuille, "Generalized Deformable Models, Statistical Physics, and Matching Problems", *Neural Computation.* **2**, 1 (1990).

[70] A. Yuille, K. Honda and C. Peterson, "Particle Tracking by Deformable Templates", *Proceedings of 1991 IEEE INNS International Joint Conference on Neural Networks*, Vol. 1, pp 7-12, Seattle, WA (July 1991).

[71] C. Peterson, "Parallel Distributed Approaches to Combinatorial Optimization Problems - Benchmark Studies on TSP", *Neural Computation* **2**, 261 (1990).

[72] J.B. Burr, "Digital Neural Network Implementations", Stanford Electrical Engineering Department Technical report (1990).

[73] M.S. Cohen, "Design of a New Medium for Volume Holographic Information Processing", *Applied Optics* **25**, 2228 (1986).

[74] H.P. Graf, L.D. Jackel, R.E. Howard, J.S. Denker, W. Hubbard, D.M . Tennant and D. Schwartz, "VLSI Implementation of a Neural Network Memory with Several Hundreds of Neurons". In *Neural Networks for Computing* (Snowbird 1986), ed. J.S. Denker, 182. New York: American Institute of Physics (1986).

[75] J. Alspector, R.B. Allen and S. Satyanarayana, "Stochastic Learning Networks and their Electronic Implementations". In *Neural Information Processing Systems* (Denver 1987), ed. D.Z. Anderson, 9. New York: American Institute of Physics (1987).

[76] C. Mead, *Analog VLSI and Neural Systems*, Reading: Addison Wesley (1989).

[77] M. Holler, S. Tam, H.Castro and R. Beson, "An Electrically Trainable Artificial Neural Network (ETANN) with 10240 "Floating Gate" Synapses". In *IJCNN International Joint Conference on Neural Networks*, II:191-196 (1989).

[78] D. Psaltis, K. Wagner and D. Brady, "Multilayer Optical Learning Networks", *Applied Optics* **26**, 5061 (1987).

[79] C. Peterson, S. Redfield, J.D. Keeler and E. Hartman, "An Optoelectronic Architecture for Multilayer Learning in a Single Photorefractive Crystal", *Neural Computation* **2**, 25 (1990).

[80] J. Alitti et al., "Measurements of the Transverse Momentum Distributions of W and Z bosons at the CERN $\bar{p}p$ Collider", *Zeitschrift für Physik* **C 47**, 523 (1990).

[81] L. Lönnblad, C. Peterson and T. Rögnvaldsson, "Finding Gluon Jets with a Neural trigger", *Physical Review Letters* **65**, 1321 (1990).

[82] L. Lönnblad, C. Peterson and T. Rögnvaldsson, "Using Neural Networks to Identify Jets", *Nuclear Physics* **B 349**, 675 (1991).

[83] P. Bhat, L. Lönnblad, K. Meier and K. Sugano, "Using Neural Networks to Identify Jets in Hadron-Hadron Collisions", Lund Preprint LU TP 90-13 (to appear in *Proc. of the 1990 DPF Summer Study on High Energy Physics Research Directions for the Decade*, Colorado), (1990).

[84] I. Scabai, F. Czakó and Z. Fodor, "Quark and Gluon Jet Separation using Neural Networks", ITP Budapest Report 477 (1990).

[85] A. Petrolini, "Use of Neural Network Classifiers in Higgs Search", DELPHI note, July 1 (1991)

[86] B. Denby, "Neural Networks and Cellular Automata in Experimenta l HighEnergy Physics", *Computer Physics Communications* **49**, 429 (1988) .

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

[88] G. Stimpfl-Abele and L. Garrido, "Fast Track Finding with Neural Nets", UAB-LFAE 90-06 (submitted to *Computer Physics Communications*) (1990).

[89] M. Gyulassy and H. Harlander, "Elastic Tracking and Neural Network Algorithms for Complex Pattern Recognition", *Computer Physics Communications* **66**, 31 (1991).

[90] T. Sjöstrand, JETSET 7.2 program and manual. See B. Bambah et al., "QCD Generators for LEP", CERN-TH.5466/89 (1989).

[91] L. Lönnblad, "ARIADNE-3, A Monte Carlo for QCD Cascades in the Color Dipole Formulation", Lund preprint LU TP 89-10 (1989) .

[92] G. Marchesini and B. R. Webber, "Monte Carlo Simulation of General Hard Processes with Coherent QCD Radiation", *Nucl. Phys.* **B310**, 461 (1988);
I. G. Knowles, "Spin Correlations in Parton-parton Scattering", *Nucl. Phys.* **B310**, 571 (1988).

[93] H-U. Bengtsson, T. Sjöstrand, *Computer Physics Communications* **46**, 43 (1987).

[94] L. Bellantoni, J. Conway, J. Jacobsen, Y.B. Pan and S.L. Wu, "Using Neural networks with Jet Shapes to Identify b Jets in e^+e^- Interactions", CERN-PPE/91-90 (submitted to *Nuclear Instruments and Methods* **A**) (1990).

[95] P.J. Huber, *Robust Statistics*, John Wiley and Sons, New York, (1981).

[96] R.O. Duda and P.E. Hart, *Pattern Classification and Scene Analysis*, John Wiley and Sons, New York, (1973).

[97] M. Ohlsson, C. Peterson and A. Yuille, in preparation.

[98] L. Lönnblad, C. Peterson and T. Rögnvaldsson, work in progress.

[99] L. Lönnblad, C. Peterson and T. Rögnvaldsson, "Pattern Recognition in High Energy Physics with Artificial Neural Networks - JETNET 2.0", Lund Preprint LU TP 91-18 (submitted to *Computer Physics Communications*) (1991)
[Program and manual available via BITNET request].

[100] B. Denby et. al., "Neural Networks for Triggering", FERMILAB-CONF-90/20 (1990).