National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL 339 (r 88/04) c

Canada

Neural Networks: Learning and Growth as different aspects of the same natural process; Neurons with variable firing strength and Formal Neurons

Constantinos Bassias

A Thesis

in

The Department

of

Physics

Presented in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy at

Concordia University

Montréal, Québec, Canada

August 1991

© Constantinos Bassias, 1991

National Library of Canada

Bibliothèque nationale du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-73680-1

Canada

ABSTRACT

Neural Networks: Learning and Growth as different aspects of the same natural process; Neurons with variable firing strength and Formal Neurons

Constantinos Bassias, Ph. D.
Concordia University, 1991

Two neural network models, namely the Hopfield model and a variant of it with low levels of activity, are studied employing neurons with variable firing strength. Appropriate storage prescriptions are found for their synaptic efficacies. Their storage capacity is discussed using a noise analysis and it is proven that the noise in the network becomes minimal when all neurons fire with the same strength.

A very broad and general learning process for any stable neural network with an energy function defined in the linear space of all polynomials of N variables is derived and serves as an appropriate foundation on which growth in feedback networks can be studied. The two processes, growth and learning, are viewed as different aspects of the same classification mechanism controlled by the magnitude of a single parameter λ. The existence of a "parent" system as the condition under which growth can take place without disturbing the stability of the neural network is established. Results from simulations show that the learning process enhances the performance of a network as associative memory, in agreement with previously reported calculations done for special cases of the process. When, in addition, growth is allowed to take place efficient networks with economy of space are developed able to perform tasks of memorization and pattern recognition. The scale and font invariant recognition of English letters is given as an example.

A response function for neurons that store all the information needed for their processing in addressable memory locations is proposed and the limitations of single-layered networks made up of those neurons are explored. Multi-layered networks are suggested as an alternative and a training program is developed for the implementation by them of any Boolean function. This program is applied in the computation of the parity function of 4 binary variables. The result is an extremely compact and memory efficient network which highlights the advantages of using such general purpose neurons and networks made up from them.

ACKNOWLEDGEMENTS

The author expresses his sincere gratitude to Professor B. Frank for his continued help and guidance throughout the entire course of the research presented in this thesis. He is also thankful to him for providing partial financial support from his NSERC grant and moral support when needed the most.

The teaching assistantship provided by the Department of Physics is also gratefully acknowledged.

TABLE OF CONTENTS

Chapter I

Introduction

This thesis consists of three distinct topics in neural network (NN) research and a brief description of each one of them is given in their corresponding chapters. The three topics are:

(i) *Derivation of two well-known models of neural networks as the minimum noise models for neurons with variable firing strength.* Binary neurons are used for many neural network models. Their behaviour consists in either firing at maximum strength or not firing at all. A model neural network is proposed wherein each neuron has its own firing strength. From this general model, models with homogeneous firing strengths can be derived as the ones that minimize the noise in the network.

(ii) *Learning and growth in neural networks with feedback.* Learning is the most significant stage in the design of a network. It represents basically the effect of the environment on the network. It allows it to respond differently to different stimuli. In this thesis, learning is seen as a gardening process: the gardening of an energy landscape. This point of view is not new but it is brought here to its full potential via a linearization of the energy function. On the other hand, growth in neural networks is a very recent subject. The position taken here is that growth occurs as a developmental process: the development of new synapses among neurons. For the first time a common mechanism is suggested for both learning and growth. These two different processes are characterized by different values of a common parameter, the learning strength.

(iii) *An algorithm for the computation of Boolean functions by a multi-layered network of a new type of neuron.* The formal neuron is a new type of neuron. It stores all the information it requires in addressable memory locations instead of storing it in the synaptic weights. Formal neurons with a specific response function are proposed in this thesis. Some of their properties are studied and an algorithm for the computation of Boolean functions by multi-layered networks of formal neurons is evaluated.

Information processing with neural networks is definitely a new subject for physicists but it is also a 40-year-old discipline. It is a field that now unites a very broad scientific community: mathematicians, physicists, parallel processing experts, optical technologists, neurophysiologists, experimental biologists and cognitive scientists. The fact, though, that from the point of view of physics the interest in neural nets is less than a decade old, suggests that a relatively long introduction to the subject in this thesis would be useful.

Chapter II serves this purpose. It contains the necessary background information and the principles that constitute a foundation for neural network studies. It also gives an account of the most recent discoveries in biology that serve as the body of experimental facts by which the proposed theories should be judged. And finally it discusses the scope and limit of applicability of methods from statistical physics to the study of neural nets.

The derivation of two NN models, namely the Hopfield model and a model with low levels of activity, from a more general model that employs neurons with variable firing strengths, is carried out in Chapter III. This is accomplished using a noise analysis argument for the total noise in the general model.

In Chapter IV the generalized learning method is developed. First a linearization of the energy function is carried out. Then a learning scheme is proposed that can be applied to any system with such an energy function. The learning process is finally studied on a specific model. A description is provided of the computer simulations used for the study of the model as well as an analysis of the results.

The learning process is carried further to encompass growth for a certain group of neural networks in Chapter V. This group contains the networks that are derivable from a larger "parent" network by switching off some of its interactions. An example of a growing network is given and the results of the computer simulations done on it are discussed.

Chapter VI deals with formal neurons. An introduction to them is carried out in the first part of that chapter. The equivalence of formal neuron networks to conventional neural networks is established, and a formal neuron with a particular response function is proposed. A training algorithm for multi-layered networks of those neurons is applied to the parity problem of 4 binary inputs and a discussion of its performance is provided. Finally, the conclusions are given in Chapter VII.

Chapter II

A general framework for neural network research

Information processing with neural networks (NN) is a relatively new discipline. The natural phenomenon that this discipline studies is the brain. The brain computes by absorbing experience; therefore, one can define information processing with neural networks as the discipline which studies the class of machines that compute by absorbing experience. Moreover these machines are cellular, are composed of smaller units, the neurons, and have a natural propensity for storing knowledge gained by experience.

The drive behind neural network research is the desire for a better understanding of the complex processes that take place in the central nervous systems of animals. The drive behind information processing with neural networks is the fascination one has for the speed with which the brain performs tasks of pattern recognition using components much slower than those found in computers. Recognition is one of several stages on a scale going from early to late processing in the brain. One starts with sensory processing, then a fuzzy stage of preprocessing (fuzzy because it is not clear when it starts and when it ends), recognition, and finally attention. It is safe to assume[1,2] that pattern recognition is the 'typical' cortex activity and it is done well. Here are some facts to prove it. (i) Presumably[3], in a typical pattern recognition computation, much smaller cell assemblies are involved than in attention states. (ii) Also time intervals involved in recognition may be between 20 ms and 200 ms, instead of 10s for the attention states[4]. (iii) The cortex performs its recognition of the vast number of patterns, in a typical human environment, using neurons that can fire at a maximum rate of 200 spikes per second and usually fire at a rate of 20 spikes per second[4] (in contrast to a 40MHz clock speed for modern personal computers).

There is a number of reasons for the recent interest in neural networks. The first reason is that information processing with neural networks is now believed to be computationally complete[5]. This means that given a computational task there is an appropriate neural network, and appropriate training, that will perform it. On the other hand neural nets promise rapid solutions for certain tasks which would require much longer time in conventional computers. This is enough reason for a financial justification of the development of neural computing methods. Another reason is that neural nets promise a functional use of knowledge gained from experience[5]. It is in speech, language and scene understanding that such ability is essential and it is here that the neural net can perform functions beyond the capability of conventional systems.

A neural network can be generally described in mathematical terms as a self-organising map[6]. What one means by this will be explained in the following paragraphs.

Let $\mathbf{M_i}$ be the space of inputs to the network (usually a probability space). Let $\hat{X}_i$ be an N_i-dimensional vector (or pattern) in this space, the input vector. Let $\mathbf{M_o}$ be the space of outputs and $\hat{X}_o$ an N_o-dimensional vector (or pattern) in this space, the output vector. Let $\mathcal{F}$: $\mathbf{M_i} \rightarrow \mathbf{M_o}$ be a function that maps $\hat{X}_i$ to $\hat{X}_o$. The map $\mathcal{F}$ is called self-organising if it is adapted during training, i.e. it is modified by experience. Training is a period when one presents an input vector $\hat{\xi}$ to the network, causing the map $\mathcal{F}$ to change in a specific way, according to a learning algorithm. The vector $\hat{\xi}$ is drawn from a subset of the input space called the training set.

The map $\mathcal{F}$ can be chosen as one representing as closely as possible the processes taking place in the brain, or it can represent a learning machine with abstract components having nothing to do with naturally occurring computational systems. As physicists we should try to discover the laws of neural computing and not try to explain the behaviour of a particular brain or a robot. Thus, our models should take into account the known facts, (as few as they may be), about the brain but not be limited by them. Our models should help to extract crucial parameters of neural computing from the inessential details, and stimulate new experimental activity in neurophysiology and cognitive science.

II.1 The neuron

The neuron is the structural unit from which neural nets are built. Thus one interprets the map $\mathcal{F}$ as representing a finite set of processing units. This means that the map $\mathcal{F}$ describes the processing done to the input $\hat{X}_i$ by a set of N neurons. It is useful to distinguish between three types of neurons: input, output, and hidden. Input neurons receive inputs from sources external to the system under study. The output neurons send signals out of the system. The hidden neurons are those whose only inputs and outputs are within the system. They are not "visible" to outside systems.

Schematically, a neuron, **i**, may be represented as in the following figure:

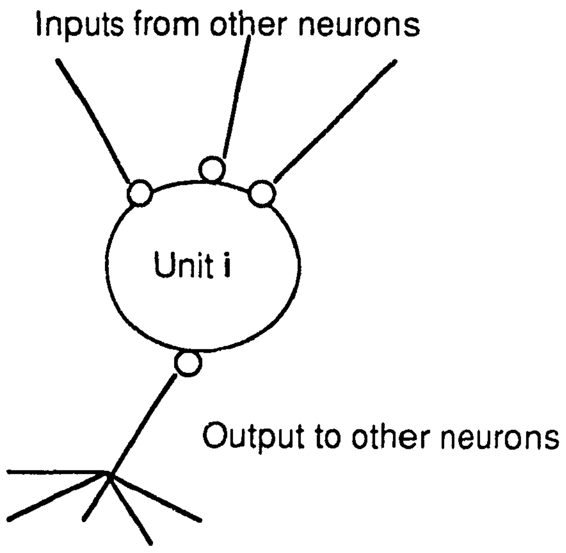


Figure 1. Schematic representation of a neuron.

The well known and used dendrite-soma-axon model of the neuron[7] is still valid with rare exceptions. A neuron receives its input from other neurons through a set of dendrites. These inputs are integrated in the soma of the neuron and the response is transmitted through the axon to other neurons. The junction between an axon of one cell and a dendrite of another is called a synapse.

Recent observations indicate that axons form synapses with the individual segments of other axons in certain cells of the cerebral cortex[4]. This fact justifies the use of multi-neuron interactions in Chapters IV and V. Also dendrites sometimes form synapses with other dendrites[8] (but not in the cerebral cortex). In general the known types of direct connections between neurons are presented in Figure 2 in the next page[9].

These direct connections, though, are not the only way of communication between neurons. Peptides are recently discovered 'special' neurotransmitters that appear to modulate synaptic functions rather than actually cause them[8]. They do so over relatively long periods of time (seconds or minutes as opposed to milliseconds). They act over larger areas by the process of diffusion, and provide an alternative slow, but broadcast-like means for neurons to communicate. A single neuron can produce several peptides. They might provide a mechanism for the realization of the generalized learning processes proposed in Chapters IV and V.

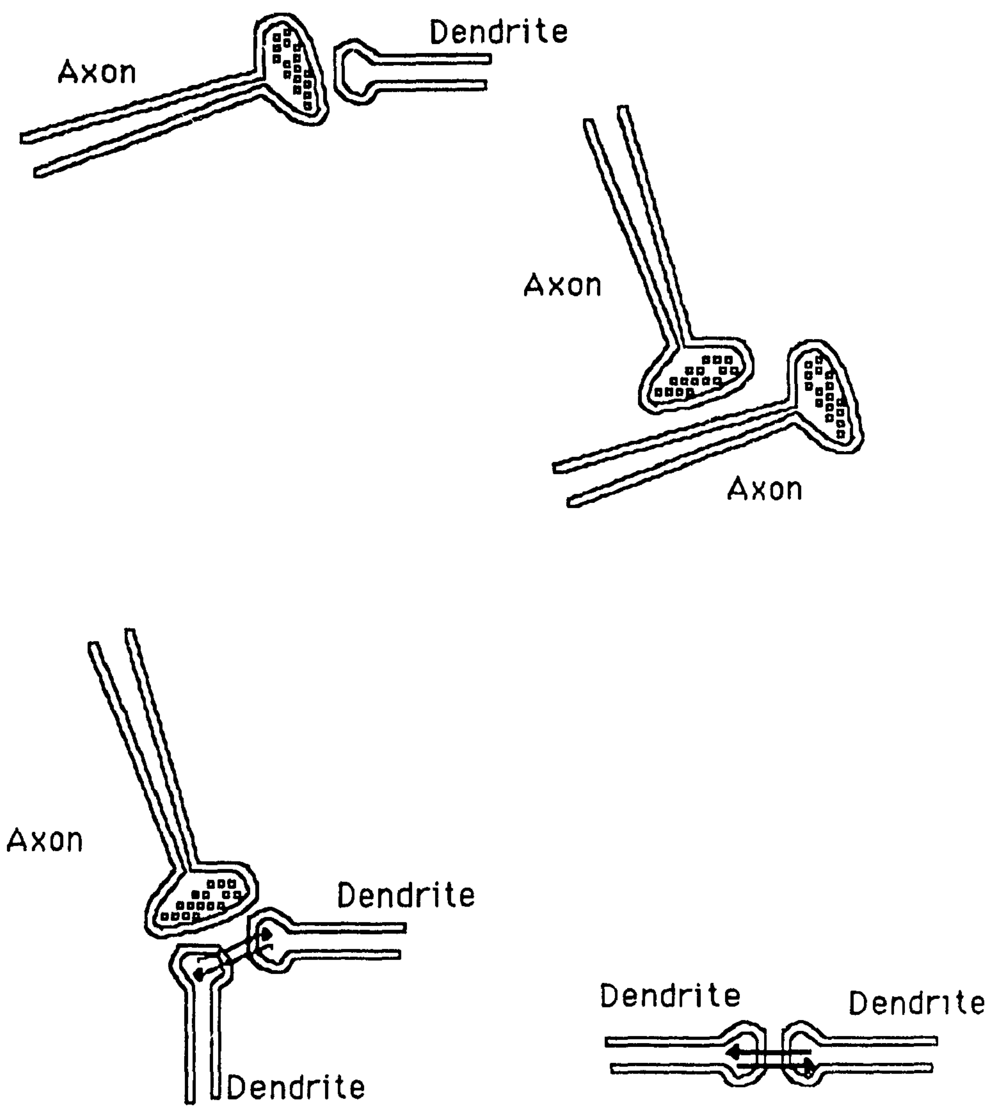


Figure 2. Diagrammatic representation of the four different types of direct connections between neurons.

How many types of neurons are there? At least two types of neural cells in the cortex are anatomically different. The pyramidal and the non-pyramidal stellate cells[10]. The former are excitatory (they make only excitatory synapses) the latter are inhibitory and appear to have only local connections and not to interconnect cortical areas. Another division appears to be into cells that are 'spiny' and those that are not[8]. The latter are characterized by receiving both sorts of inputs (excitatory and inhibitory) on both soma and dendrites, while the former receive non-excitatory somatic inputs.

The output of a neuron in most cases is a spike. The exceptions are neurons with short axons where the output consists of graded potentials instead of spikes[4]. This makes models sensible where a continuous map $\mathfrak{F}$ is used. The question of the relevance of the spikes, though, is an open one. Spikes may be totally irrelevant to the functional behaviour of the cortex[11]. The firing frequency of a neuron may be the relevant variable[12]. On the other hand the spike structure may be absolutely essential, due to synchronicity effects[13]. Some problems may be overcome by seeing the probability of firing (i.e. a stochastic variable) rather than the average firing rate as being the carrier of information[8].

Another characteristic of a neuron seems to be its threshold. This is the value the potential has to reach at the axon hillock (the part of the cell membrane near the axon), before the axon can fire. Finally one should note the fact that normally in the cortex a neuron is switching between two low-activity states[3] (say, one with 5 spikes per second and the other with 20).

Neurobiologists have a long list of parameters characterizing a single neuron and it is an ever-increasing one. The above-mentioned parameters are those considered essential in this thesis, the ones that will be used to further determine the nature and the structure of the map $\mathfrak{F}$.

II.2 The synapses

Small stellate neurons usually receive about 10^2 inputs[8], larger pyramidal cells 10^3 and the largest pyramidal cells have about 10^4. The axons of stellate cells make all synapses inhibitory on their contact sites. On the other hand the axons of pyramidal cells make all synapses excitatory on their contact sites. Modern discoveries have changed very little the Hebbian[14] idea that synapses are the main sites for modifications due to learning. The question of on which side (the presynaptic or the postsynaptic) the change occurs, remains unanswered. At issue is the existence of a Hebbian-like mechanism[14] for synaptic modification, which will be assumed in Chapter III, because the convergence of signals from the presynaptic and postsynaptic neurons will take place more naturally on the postsynaptic membrane[3] as shown in Figure 3.

If, however, a presynaptic mechanism takes place, then the information from the postsynaptic neuron must arrive on the presynaptic membrane via a retrograde signal through the

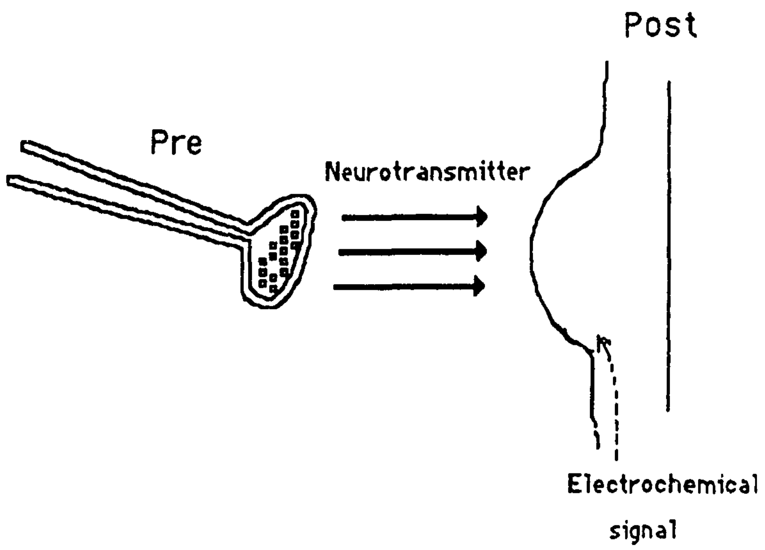


Figure 3. The Hebbian mechanism for synaptic modification. The response of the postsynaptic neuron and presynaptic signal determine the changes in the postsynaptic membrane.

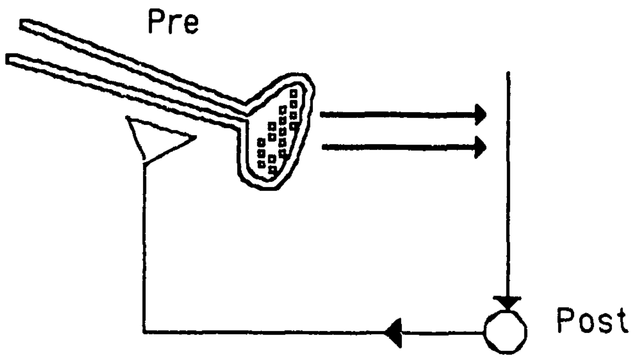


Figure 4. A possible circuit by which presynaptic modification can take place.

junction[3] (Figure 4) or via other mechanisms (maybe even ad hoc circuitry).

The mechanisms of synaptic modification are very important in neural network modeling. They provide insights in developing efficient learning algorithms. They affect the most significant period in the development of a neural net, namely training. They tell us how the map $\mathcal{F}$ should be changed in order to respond to a multitude of patterns in its environment.

II.3 The cerebral cortex

The neo-cortex part (where, crudely, most of the 'clever' processing must be done) is organized in major layers[8]: (a) a superficial layer which mainly receives axons from other layers; (b) an upper layer of small pyramidal neurons; (c) a middle layer that contains small stellate neurons; and (d) a deep layer where the larger neurons reside[4,7]. Axons generally connect towards the surface (vertically) across layers[7]. Horizontal structures appear to be much more local. The neo-cortex, besides being layered, has been divided into more than 100 distinct areas on each side of the human brain, each being both anatomically and functionally distinct[4,15].

The architecture of the cell assemblies in the cerebral cortex have given rise to an ensemble of neural network models[16,8], from which two simple extremes deserve to be distinguished: the feed-forward 'perceptron'[17] and the feedback 'ganglion'[18]. The 'perceptrons' are layered structures[8,19], where information flows from the first layer (input) to the last layer (output). They simulate the layered structure of the cortex and they will be the point of discussion in Chapter VI.

In such a case the map $\mathcal{F}$ can be considered as a q-stage mapping process:

$$\hat{X}_i \to \hat{X}_1 \to \ldots\ldots \to \hat{X}_{q-1} \to \hat{X}_o \qquad (1)$$

where $\hat{X}_1 \equiv \hat{X}_1(\hat{X}_i)$, with dimension N_1, is the first stage of the map. It maps the input vector $\hat{X}_i$ to the output $\hat{X}_1$ of the first processing layer. It represents the first layer of the network. Similarly the vectors $\hat{X}_k \equiv \hat{X}_k(\hat{X}_1, \hat{X}_m, \ldots)$ $(1<k, m<k, ..)$, with $\dim(\hat{X}_k)=N_k$, represent the hidden layers of the network. The vector $\hat{X}_k$ depends only on the previous stages of the map since that is required for the network to be a causal one.

On the other hand the 'ganglions' are defined here as richly interconnected structures[8,20] where the inputs govern the initial network state. The 'ganglions' simulate the local horizontal structures of the cortex and they are the major topic of investigation in this thesis.

In this case processing of the information occurs via the internal dynamics of the network. The dynamics are a product of the iteration of the map $\mathcal{F}$. It is well known[21] that such systems, in the case of a nonlinear $\mathcal{F}$, may behave periodically or may show chaotic behaviour thus being quite unpredictable. Let $\mathcal{F}: \mathbf{M_i} \to \mathbf{M_i}$ (in this case the input and the output spaces are identical). One is interested in the behaviour of points in $\mathbf{M_i}$ under iteration of $\mathfrak{F}$. Let $\mathfrak{F}^n$ denote the nth iterate of $\mathfrak{F}$. That is,

$$\mathfrak{F}^2 = \mathfrak{F} \circ \mathfrak{F},$$
$$\mathfrak{F}^3 = \mathfrak{F} \circ \mathfrak{F} \circ \mathfrak{F},$$

and so forth.

If $\hat{z} \in \mathbf{M_i}$, the sequence of points

$$\hat{z},\ \mathfrak{F}(\hat{z}),\ \mathfrak{F}^2(\hat{z}),\ldots.$$

is called the orbit of $\hat{z}$. The main question in dynamics is: can one predict the fate of all orbits of $\mathfrak{F}$? That is, what can be said about the behaviour of $\mathfrak{F}^n(\hat{z})$ as $n \to \infty$? There are three different patterns of behaviour[22] for the orbits of $\mathfrak{F}$:

(i) They eventually settle into an attracting fixed point (attractor) and this persistent state then contains the output of the computation.

(ii) They eventually enter an attractive periodic orbit or cycle with a certain period T. This cycle then is the output of the neural network.

(iii) The orbit behaves erratically, without ever converging to a periodic orbit.

So far the experimental evidence for persistent states comes either from very primitive circuits (such as invertebrate central pattern generators), or from event-related recordings (such as delayed template matching) in tasks involving short-term memory and attention[3].

Finally another important feature of neural information-processing in the cortex is that there is no time discretization. The neurons seem to provide continuously available output. That is, there does not seem to be an appreciable decision phase during which a unit provides no output. Rather it seems that the state of a unit reflects its current input[8].

II.4 Learning

The ability to learn from experience is the single most important characteristic of neural networks. The position taken here will be that the learning sites are both the neurons and the synapses. Assume a network of N neurons each of which is described by a set of parameters (for example threshold, type, etc.). This set of parameters constitutes the first learning site. The neurons are connected to one another via synaptic junctions. This pattern of connectivity constitutes the other part of the knowledge encoded into the network. It is convenient to represent the synaptic pattern of connectivity by a matrix **J**, which will be called the synaptic matrix. Changing the processing in a neural network involves modifying the synaptic matrix **J**. It constitutes the second learning site. Three types of changes are conceivable: development of new connections (Chapter V), loss of existing ones or modification of the strengths of connections that already exist (Chapters III and IV).

Two different learning schemes are to be found in nature and they have their counterparts in the design of learning algorithms for artificial neural networks. The two schemes are supervised learning[5,8,23] and unsupervised learning[5,8]. It is worth exploring the fundamental differences between them.

In supervised learning a human supervisor examines each training pattern (a vector $\hat{\xi}$ from the training set) before it is applied to the learning machine, and assigns a class label to that pattern. To be of any use, the class label must have some perceptual significance to the human supervisor. The following is an example of what we consider to be supervised human learning; one says to a child, "Look! There is a bird", and points to an eagle. On another occasion one points to a duck and says, "Look! There is a bird." The visual patterns presented by these two examples of class 'bird' are quite different in some ways. The rule of the supervised training has been to link these two examples into the single class 'bird'. In mathematical terms during supervised learning one changes the map in such a way that the network will produce a target output $\hat{X}_0 = tg(\hat{\xi})$ corresponding to a given training pattern $\hat{\xi}$.

However, it is hard to believe that the above-mentioned child, faced with an eagle and a duck, would not have recognized their similarity and associated them in its mind. In that case, all that the supervision did was to cause the word 'bird' to be associated with these two examples of what was already a naturally distinct class in the mind of the child. A class is naturally distinct if the description of its members in some feature space is much closer together than to any other image[24]. This suggests that any group of patterns that is worthy of a name in our perception will form a distinct cluster in a naturally occurring feature space in our brains. In mathematical terms if $tg(\hat{\xi}) = \hat{X}_o = \hat{\xi}$ then the training program is unsupervised and the output of some hidden neurons should be thought of as the output of the network[6]. This output can be visualized as a reduced or internal representation of $\hat{\xi}$ in some feature space. The clustering in the feature space is achieved by setting an overall goal or purpose for the network. An example[6] of such a goal is the minimization of the weighted squares of the differences between the input $\hat{\xi}$ and the output $\hat{X}_o$:

$$\int P_{\hat{\xi}}(\hat{\xi}) \left|\hat{X}_o - \hat{\xi}\right|^2 d\hat{\xi} \qquad (2)$$

where $P_{\hat{\xi}}(\hat{\xi})$ is a probability measure induced on the input space by the training set.

Let $S_i(t)$ be a dynamical variable describing the temporal evolution of neuron **i**. This variable may represent particular conceptual objects such as words, letters, features and concepts, or it may represent an abstract element (like a picture element [pixel]). Whatever it represents $S_i(t)$ alone is just a part of a pattern, the vector

$$\hat{Y} = (S_1(t), S_2(t), \ldots, S_i(t), \ldots, S_N(t)),$$

and it is the pattern as a whole that is the meaningful level of analysis. In general an equation can be constructed to describe the temporal evolution of the neural network

$$\hat{Y}(t+dt) = \mathbf{G}(\hat{Y}(t-\tau), \mathbf{J}), \qquad (3)$$

where τ is a delay interval. The state of the network at time t+dt is determined by its condition during the time interval between $t-\tau$ and t. In the whole of this thesis τ will be taken equal to zero.

One of the most common cases is when **J** represents connections of only one type, and therefore J_{im} represents the strength and sense of the connection from neuron **i** to neuron **m.** If one adds to that the constraint that each neuron simply sums the signals it receives from the other neurons without delay one gets the well-known[8], class of models

$$S_i(t+dt) = \mathbf{G}\left(\sum_m J_{im} S_m(t)\right). \qquad (4)$$

Virtually all learning rules for models of this type can be considered a variant of the Hebbian learning rule suggested by Hebb in his classic book Organization of Behavior[14]. Hebb's basic idea is this: If a neuron **i** receives an input from another neuron **m** , then if both are active the synaptic strength J_{im} should increase. This idea has been extended and modified so that it has taken a more general form[8]

$$\Delta J_{im} = q(S_i(t), t_i(t)) * r(S_m(t), J_{im}), \tag{5}$$

where $t_i(t)$ is a kind of teaching input to neuron **i**.

Eq. (5) states that the change in the synaptic strength J_{im} is given as a product of two functions q() and r(), where q() depends on the state of neuron **i** and a teaching input to it, and r() depends on the state of neuron **m** and the synaptic strength J_{im}. There are many versions of this rule. In Chapter III the simplest version of Hebbian learning will be employed. This version is an unsupervised learning process (there is no teaching input to neuron **i**) and the functions q() and r() are simply proportional to their first arguments. Thus one has

$$\Delta J_{im} = \lambda S_i(t) * S_m(t) \tag{6}$$

where λ is the constant of proportionality and represents the learning rate.

Another common variation of Eq. (5) is the delta rule. This rule (known also as the Widrow-Hoff[25,26] rule) is a supervised learning process and is mostly used as a learning mechanism for perceptrons. The change in J_{im} is proportional to the difference between the actual state of neuron **i** and a desired or target state, $t_i(t)$ as in Eq. (5), provided by the teacher. The function r() is again simply proportional to its first argument. In this case one has

$$\Delta J_{im} = \lambda(t_i(t)-S_i(t))*S_m(t) \qquad (7)$$

The perceptron learning rule for which the perceptron convergence theorem has been proved[17] is a special case of Eq. (7) for binary $S_i(t)$. As a final point one should note that recent challenges[27] to the Hebbian mechanism for learning based on coincidence of incoming signals from two presynaptic neurons, still obey Eq. (5).

<u>II.5 The connection between neural networks and Statistical Mechanics</u>

It is far from evident that the type of systems we have described should in any way resemble those encountered in physics. Surprisingly enough there is an argument[28] (Changeux[29] , Purves and Lichtman[30], Eigen[31] and Kauffman[31]) that supports the position that the above systems could be viewed as disordered systems. This despite the fact that these systems look like they are goal oriented, with subtle interrelationships and use experience to achieve efficiency. The argument states that while a part of a biological system may have been perfectly modelled to perform a function, another part, or even the same part but looked at on a different scale, can be analyzed assuming randomness. In the particular case of a mammalian brain one can distinguish different pieces whose interconnections seem like the result of a careful design. Looking inside each of these pieces one discovers a great amount of randomness. The argument also stresses the fact that it is impossible for every single detail in the mammalian brain to be optimized because the amount of information needed for this optimization surpasses by a large factor the information content of the DNA (compare 4^{10^9}, the information content of a DNA strand composed of 10^9 nucleotides of 4 different types, with the information contained in the way 10^{10} neurons are connected by 10^{14} synapses, which can be as large as $\left[\left(10^{10}\right)^{10000}\right]^{\left(10^{10}\right)}$ if all the synapses are assumed to be different). In other words the mammalian brain has many more variables than the ones that have been put under control by the genetic code.

The situation is therefore not very different from the one encountered in statistical mechanics with macroscopic variables under control and microscopic variables being random. The difficult task is to identify the relevant macroscopic variables that control the emergence of a typical behaviour as a collective manifestation of the large number of microscopic degrees of freedom.

An approach to information processing with neural networks closer to statistical mechanics was initiated by

Little[32,33] and later developed by many authors (Kohonen[24], Cooper et al.[34] , Hinton and Anderson[35]). All their models are high-feedback models, the map $\mathfrak{F}$ is iterated, and employ the idea that recognition is like falling into the basin of attraction of a fixed point or limit cycle. This is also the basic idea behind the models proposed here in Chapters III, IV and V. The existence of a fixed point is a major topic of the stability theory of all non-linear dynamic systems. Stability is of course a fundamental requirement of all neural networks if they are to serve any useful purpose. A great variety of techniques for the study and prediction of stability have been developed, but the most important approach to this problem remains that introduced by A. M. Lyapunov[36] in 1907. The basis of this methnd is the association of a Lyapunov function, H, to a dynamic system of the form:

$$\frac{d(\hat{z})}{dt} = F(\hat{z}) \tag{8}$$

where $\hat{z}$ is an N-dimensional vector. This is essentially Eq. (3). Then Lyapunov's theorem states that the system described by Eq. (8) is (asymptotically) stable if and only if there exists an H such that

$$\exists\, M: \qquad H(\hat{z}) \geq M \qquad \forall\, \hat{z}$$

and

$$\frac{d\left(H(\hat{z})\right)}{dt} \equiv (\nabla_{\hat{z}} H(\hat{z}))\frac{d\left(\hat{z}\right)}{dt} = (\nabla_{\hat{z}} H(\hat{z}))F(\hat{z}) \leq 0 \qquad (9)$$

The general problem of finding a Lyapunov function for any non-linear system is not solved yet and thus one relies on intuition and analogy in the search for such a function.

If, though, a Lyapunov function is found for a particular feedback network it can be associated with the Hamiltonian function of an equivalent physical system[37,38]. In such a case the evolution of the network is equivalent to a path in the phase space of the corresponding physical system. In the absence of noise the system will wander in the phase space till it reaches an energy minimum and will stay there. The configuration of the dynamical variables corresponding to that minimum is a fixed point of the network. The very important role of noise[1] can be played by a heat bath of temperature T with which the network is supposed to be in contact. Then a free energy function can be defined for noisy networks and their macroscopic behaviour can be studied using methods from statistical mechanics.

II.6 Discussion

The ideas and concepts discussed in this chapter form the backbone on which the theories and models presented in the following chapters will be based. As a concluding remark one should mention that neural network research at its present state mainly consists of a large collection of models from a variety of disciplines. An attempt was made in this chapter to present what could be thought of as a minimum common set of ideas and concepts necessary for the development of any neural network model.

Chapter III

Neurons with variable maximum firing rates

The study of neural networks within the framework of statistical mechanics that was sparked by Little[32] was carried out further, after the introduction by Hopfield[18], of a model that turned out to be exactly soluble[38,39] and had a strong affinity with statistical mechanics. This model is related to models used in the theoretical analysis of certain materials called spin glasses[40,41,42]. These are magnetic materials which have a random orientational ordering (glass) of magnetic moments (spins). The spin sites are randomly interconnected by positive and negative competing interactions. One can see from that the relation to neurons connected with excitatory and inhibitory synapses.

The Hopfield model consists of a fully connected set of neurons with feedback. Such a network is represented diagrammatically in Figure 5. Hopfield's introduction of an energy func ion that governs the dynamics of the collective activity of such a set of N neurons, has produced many interesting and diverse results[43,44,45].

In almost all the models derived from Hopfield's basic model, the maximum firing strength of the neurons is taken to be the same for all neurons, and to be unity (the exception being the threshold-linear neurons introduced by Treves[46]). We are interested in the question of diverse firing strengths

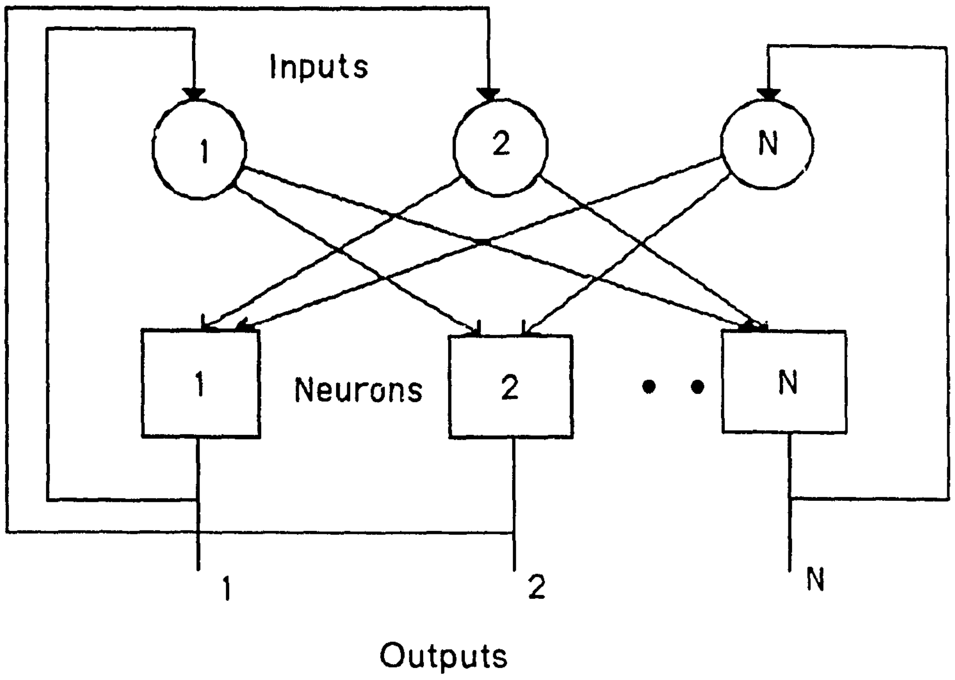


Figure 5. A schematic representation of a fully connected neural network with feedback. The inputs, represented by circles, do not do any processing.

and their consequences. Although the most impressive results in neural network research come from models where a Hamiltonian function cannot be defined[8,5], we here study a system for which such a function exists; such a system is more amenable to theoretical treatment.

The neurons will be modeled as bi-valued devices[47]. A set of positive quantities a_i are here introduced, one for each neuron, such that the **i**th neuron can be either in the firing state with firing strength $V_i = a_i > 0$, or in the quiescent state with $V_i = 0$. (In Hopfield's model a_i is taken to be 1 and is interpreted as the maximum firing rate of the neuron.) A network composed of N such neurons could be used to model the behaviour of *gestalts* in which specialization has driven the neurons into a diversified standard behaviour with different firing strengths. These different strengths can be taken to represent different maximum firing rates for the various neurons. (It is of importance to note here that this is not the absolute maximum firing rate ($\approx$250 Hz) that a neuron can in fact achieve (which latter is independent of the function which that neuron performs) but rather an experimental assessment of the behaviour of a neuron that does a particular job within the context of its gestalt. Such a neuron can be observed to fire with a specific maximum firing rate characteristic of its operation.) The neurons are interconnected through synaptic junctions of strength J_{ij}, and the input potential on each neuron is the sum of all post-synaptic potentials delivered to it.

Assuming, for the moment, $V_i(t)$ to be the dynamical variable that describes the temporal evolution of neuron **i**, then the dynamics of the model, in the absence of noise ($T=0$), is determined by the equation,

$$V_i(t+1) = a_i * \Theta(\sum_{j(\neq i)} J_{ij}V_j(t) - U_i) \tag{10}$$

where Θ is the Heaviside function

$$\Theta(x) = \begin{cases} 1 \text{ if } x > 0 \\ 0 \text{ if } x < 0 \end{cases} \tag{11}$$

U_i is the threshold for neuron **i**, time has been digitized and the neurons are updated asynchronously (Eq. (10) is applied to one neuron chosen randomly at each time step). Eq. (10) is just a particular realization of Eq. (3).

If $J_{ij}=J_{ji}$, one can define the Hamiltonian function

$$H = -1/2 \sum_{\substack{i,j \\ (i\neq j)}} J_{ij}V_iV_j + \sum_i V_iU_i \quad . \tag{12}$$

The change ΔH due to the change $\Delta V_i(t) = V_i(t)-V_i(t-1)$ (with $\Delta V_j(t) = 0$ for $j\neq i$) is

$$\Delta H = -(\sum_{j(\neq i)} J_{ij}V_j(t-1) - U_i)\Delta V_i(t) \tag{13}$$

and the dynamics lead to local minima of the Hamiltonian in Eq. (12), as can be seen from the fact (using Eq. (10)) that ΔH is either negative or zero, and thus H is a decreasing function of time. The other part of Lyapunov's theorem (Eq.

(9)) is also satisfied because the above Hamiltonian is bounded below by M, where

$$M = -1/2 \sum_{\substack{i,j \\ (i \neq j)}} |J_{ij}| a_i a_j - \sum_i a_i |U_i| \quad . \tag{14}$$

An equivalent model is obtained under the transformation,

$$S_i = 2V_i - a_i \tag{15}$$

where S_i takes the values $\pm a_i$, and replaces V_i as a dynamic variable. Note that S_i can be viewed as an Ising spin variable.

The dynamics of this system is described at T=0 by

$$S_i(t+1) = a_i \text{ sign } \Big(\sum_{j(\neq i)} J_{ij} S_j(t) \Big) \tag{16}$$

with Hamiltonian

$$H = -1/2 \sum_{\substack{i,j \\ (i \neq j)}} J_{ij} S_i S_j \quad . \tag{17}$$

The Hamiltonian in Eq. (17) is equivalent to the one in Eq. (12) to within a configuration-independent constant if after the transformation in Eq. (15) one sets

$$\sum_i J_{ij} a_i = 2U_j \quad . \tag{18}$$

In this chapter we will investigate the properties of the Hamiltonian in Eq. (17) with respect to deviations of the firing strength a_i from unity, the value that it takes in the Hopfield model.

III.1 Noise analysis and the storage capacity of the network with Hebbian learning: the Hopfield model

Assume that the network learns according to the Hebbian rule:

$$J_{ij} = (1/\sum_k a_k^2) \sum_{\mu=1}^{p} (2V_i^\mu - a_i)(2V_j^\mu - a_j) \tag{19}$$

or,

$$J_{ij} = (1/\sum_k a_k^2) \sum_{\mu=1}^{p} \xi_i^\mu \xi_j^\mu \tag{20}$$

after the transformation in Eq. (15) with ξ_i^μ in place of S_i. Here each ξ_i^μ is an independent, quenched, random variable, which takes the values $+a_i$ and $-a_i$ with equal probability; the set of ξ_j^ν with ν fixed will be called the pattern $\hat{\xi}^\nu$.

This is an unsupervised learning process. The network starts with initial synaptic strengths equal to 0 and proceeds to learn a set of p patterns by modifying its synapses as follows

$$J_{ij}^{new} = J_{ij}^{old} + (1/\sum_k a_k^2)\xi_i^\mu \xi_j^\mu \tag{21}$$

whenever a new pattern $\tilde{\xi}^\mu$ is presented to it.

The local field h_i acting on the site **i**, when the system is in the state $\hat{Y} = \tilde{\xi}^\nu$, is

$$h_i = \sum_{j(\neq i)} J_{ij}S_j = (1/\sum_k a_k^2) \sum_{j(\neq i)} \sum_{\mu=1}^{p} \xi_i^\mu \xi_j^\mu \xi_j^\nu =$$

$$= (1/\sum_k a_k^2) \left\{ \left[\sum_{j(\neq i)} a_j^2\right]\xi_i^\nu + \sum_{j(\neq i)} \sum_{\substack{\mu=1\\(\mu\neq\nu)}}^{p} \xi_i^\mu \xi_j^\mu \xi_j^\nu \right\} =$$

$$= \xi_i^\nu \left\{ (1/\sum_k a_k^2) \left[\sum_{j(\neq i)} a_j^2\right] + (1/a_i^2)(1/\sum_k a_k^2) \sum_{j(\neq i)} \sum_{\substack{\mu=1\\(\mu\neq\nu)}}^{p} \xi_i^\mu \xi_j^\mu \xi_j^\nu \xi_i^\nu \right\}. \tag{22}$$

The second term in braces is the noise n_i at the lattice site **i** coming from the other patterns; when it becomes sufficiently large it will destroy the stability of the stored pattern $\tilde{\xi}^\nu$. This second term is a random variable with zero mean, and with variance

$$\overline{((1/\sum_k a_k^2)\ (1/a_i^2) \sum_{j(\neq i)} \sum_{\substack{\mu=1\\(\mu\neq\nu)}}^{p} \xi_i^\mu \xi_j^\mu \xi_j^\nu \xi_i^\nu)^2} = (p-1) \sum_{j(\neq i)} a_j^4/(\sum_k a_k^2)^2 \tag{23}$$

The translationally invariant noise n_t is defined as the sum over the whole lattice of all the noises n_j

$$n_t = \sum_j n_j \quad , \qquad (24)$$

and is also a random variable with zero mean and with variance

$$\mathrm{Var}\,(n_t) = (N-2)\,(p-1)\sum_j a_j^4/(\sum_k a_k^2)^2 + p-1 \quad . \qquad (25)$$

To have maximum storage capacity one must minimize Eq. (25) with respect to the a_j. The minimum occurs when (see Appendix A)

$$a_j = a \text{ for every } j. \qquad (26)$$

Substituting $a_j = a \;\forall\; j$, into Eq. (22) one obtains the Hopfield model result

$$h_i = \xi_i^\nu[1-1/N + n_i], \qquad (27)$$

where the noise n_i has variance $(1-1/N)(p-1)/N$ (cf. result of ref 48, which is the same as the one above to $O(1/N)$).

III.2 A model with low levels of activity

To accommodate the fact that the neurons in the brain are most of their time quiet one might consider a modification of Hopfield's model that breaks the symmetry between the states +ai and -ai[49]. For instance, in such a case, every component ξ_i^μ can be chosen independently with probability $P(\xi_i^\mu)$,

$$P(\xi_i^\mu) = (1/2)(1+\alpha)\delta(\xi_i^\mu - a_i) + (1/2)(1-\alpha)\delta(\xi_i^\mu + a_i). \tag{28}$$

The average of each ξ_i^μ is αa_i and the stored patterns are necessarily correlated, though in a rather simple way. One has

$$\overline{\xi_i^\mu \xi_i^\nu} = (\delta^{\mu\nu} + \alpha^2(1-\delta^{\mu\nu}))\, a_i^2. \tag{29}$$

To consider the effects of neurons with maximum firing strength differing from unity on such a model, a modification of the learning rule is needed. The reason is simple. If one chooses Eq. (21) as a learning rule then the local field acting on the spin on site **i** has a contribution from the other patterns, which does not average to zero. In the state $S_i = \xi_i^\nu$,

$$h_i = \sum_{j(\neq i)} J_{ij} S_j = (1/\sum_k a_k^2) \sum_{j(\neq i)} \sum_{\mu=1}^{p} \xi_i^\mu \xi_j^\mu \xi_j^\nu \cong \xi_i^\nu (1+n_i) \tag{30}$$

and the noise

$$n_i = (1/a_i^2)(1/\sum_k a_k^2) \sum_{j(\neq i)} \sum_{\substack{\mu=1 \\ (\mu\neq\nu)}}^{p} \xi_i^\mu \xi_j^\mu \xi_j^\nu \xi_i^\nu \tag{31}$$

is a random variable with mean

$$\overline{n_i} = (p-1)\alpha^4 \frac{\sum_{j(\neq i)} a_j^2}{\sum_k a_k^2} \cong (p-1)\alpha^4 \tag{32}$$

On the other hand one might want to apply a rule proposed by Amit, Gutfreund and Sompolinsky[49]. In their model Hebb's rule was modified, and had taken the form

$$J_{ij}^{new} = J_{ij}^{old} + (1/N)\,(\xi_i^\mu - \alpha)\,(\xi_j^\mu - \alpha) \qquad (33)$$

which could be written for neurons with variable maximum firing strengths as

$$J_{ij}^{new} = J_{ij}^{old} + (1/\sum_k a_k^2)\,(\xi_i^\mu - \alpha)\,(\xi_j^\mu - \alpha) \quad . \qquad (34)$$

Eq. (34) leads to a new set of synaptic efficacies

$$J_{ij} = (1/\sum_k a_k^2) \sum_{\mu=1}^{p} (\xi_i^\mu - \alpha)\,(\xi_j^\mu - \alpha) \quad . \qquad (35)$$

With the above synaptic efficacies the local field at neuron **i** in pattern ν is

$$h_i = \sum_{j(\neq i)} J_{ij} S_j = (1/\sum_k a_k^2) \sum_{j(\neq i)} \sum_{\mu=1}^{p} (\xi_i^\mu - \alpha)\,(\xi_j^\mu - \alpha)\,\xi_j^\nu =$$

$$\xi_i^\nu \left[\left(\sum_{j(\neq i)} a_j^2 - (\xi_i^\nu \alpha / a_i^2) \sum_{j(\neq i)} a_j^2 - \alpha \sum_{j(\neq i)} \xi_j^\nu + (\alpha^2/a_i^2)\,\xi_i^\nu \sum_{j(\neq i)} \xi_j^\nu \right) / (\sum_k a_k^2) \right] +$$

$$+\xi_i^\nu \left[(1/(a_i^2 \sum_k a_k^2)) \sum_{j(\neq i)} \sum_{\mu(\neq \nu)}^{p} (\xi_i^\mu - \alpha)\,(\xi_j^\mu - \alpha)\,\xi_j^\nu \xi_i^\nu \right] =$$

$$=\xi_i^\nu\left(\frac{\sum\limits_{j(\neq i)} a_j^2-\alpha^2\sum\limits_{j(\neq i)} a_j}{\sum\limits_k a_k^2}\right)\left(1-\alpha\frac{\xi_i^\nu}{a_i^2}\right)+$$

$$+\xi_i^\nu\left[(1/(a_i^2\sum_k a_k^2))\sum_{j(\neq i)}\sum_{\mu(\neq\nu)}^{p}(\xi_i^\mu-\alpha)(\xi_j^\mu-\alpha)\xi_j^\nu\xi_i^\nu\right]$$

where we have used the fact that $\overline{\xi_j^\nu} = \alpha a_i$ and the approximation

$$\sum_{j(\neq i)}\xi_j^\nu = \alpha\sum_{j(\neq i)} a_j \quad . \tag{36}$$

The noise term in the expression for the local field is

$$n_i = (1/(a_i^2\sum_k a_k^2))\sum_{j(\neq i)}\sum_{\mu(\neq\nu)}^{p}(\xi_i^\mu-\alpha)(\xi_j^\mu-\alpha)\xi_j^\nu\xi_i^\nu$$

which again does not average to zero.

Thus one has to modify the above learning rule in such a way that the contribution from the other patterns to the local field on site **i** will have a zero mean. The following assignment for the synaptic efficacies will serve this purpose

$$J_{ij}= (1/\sum_k a_k^2)\sum_{\mu=1}^{p}(\xi_i^\mu-\alpha a_i)(\xi_j^\mu-\alpha a_j) \quad . \tag{37}$$

Using the synaptic strengths Eq. (37) and Eq. (36) one obtains for the local field h_i

$$h_i = \sum_{j(\neq i)} J_{ij}S_j = (1/\sum_k a_k^2) \sum_{j(\neq i)} \sum_{\mu=1}^{p} (\xi_i^\mu - \alpha a_i)(\xi_j^\mu - \alpha a_j)\xi_j^\nu =$$

$$= \xi_i^\nu \left(\frac{\sum_{j(\neq i)} a_j^2}{\sum_k a_k^2} \right) (1-\alpha^2)\left(1 - \alpha \frac{\xi_i^\nu}{a_i}\right) +$$

$$+ \xi_i^\nu \left[(1/(a_i^2 \sum_k a_k^2)) \sum_{j(\neq i)} \sum_{\mu(\neq \nu)}^{p} (\xi_i^\mu - \alpha a_i)(\xi_j^\mu - \alpha a_j)\xi_j^\nu \xi_i^\nu \right]$$

The first term in the above expression is the signal f_i on site **i** and the second term is the noise n_i.

The signal term has a minimum and

$$f_{i,\min} = \left(\frac{\sum_{j(\neq i)} a_j^2}{\sum_k a_k^2} \right) (1-\alpha^2)(1-|\alpha|) \qquad (38)$$

while the noise term is a random variable with zero mean and variance

$$\mathrm{Var}(n_i) = (p-1)(1-\alpha^2)^2 \left(\left(\sum_{j(\neq i)} a_j^4 \right) / (\sum_k a_k^2)^2 \right) \qquad (39)$$

The minimum of the noise to signal ratio $NS = \sqrt{\mathrm{var}(n_i)} \, / \, f_{i,\min}$ with respect to the set of site firing strengths a_i occurs again when all a_i's are equal as can be seen from the similarity of form between Eq. (23) and Eq. (39). Substituting $a_j = a \ \forall \ j$, into Eq. (38) and (39) one obtains for NS

$$NS = \frac{1}{(1-|\alpha|)}\sqrt{\frac{(N-1)\,(p-1)}{(N-1)^2}}$$

recovering the result of ref 49.

III.3 Conclusion

In this chapter it has been proven that the Hopfield model and one of its variations with low levels of activity, can be found from a more general model which employs neurons with variable maximum firing strength, as that particular realization which exhibits minimal total noise (or, equivalently, maximal storage capacity). This has been done using a noise analysis argument for the total noise n_t in the case of the Hopfield model and the same argument for the noise to signal ratio NS in the case of its variant.

Chapter IV

Generalized learning in Neural Networks with a parametric Lyapunov function

The discussion in the previous chapter involved high-feedback models for which a Lyapunov or energy function existed. The learning method employed was an unsupervised Hebbian type learning which gave the synaptic strengths as functions of the patterns one wished to store in the network. The existence (which had to be demonstrated) of the energy function ensured the stability of the networks.

In this chapter it is assumed that for a given dynamic system described by Eq. (8), a Lyapunov function exists, thus ensuring its stability. Moreover it is assumed that this Lyapunov function can be written in parametric form, with the parameters depending on the patterns one stores in the neural network. That means that this energy function characterizes a whole class of stable dynamical systems differing from each other only in the coefficients of their Hamiltonian functions. Under the previous assumptions falls a wide category of networks which includes the Hopfield network and all its proposed variations (e.g. ref 44 and 50), feedback networks with multi-neuron interactions[51], Potts-glass models[52] of neural nets, and networks in the presence of external fields[53].

For all these models a learning process is proposed which is superimposed on any unsupervised learning that initially defines the synaptic efficacies. It is a supervised learning process and it is a natural and rather simple local error correction process involving one or many patterns at a time. The basic idea behind this generalized learning is a natural modelling of the energy landscape. This idea is not new[54,55,56] but it finds here its most general and unrestricted application. Through this modelling the stored patterns are forced to become local minima (attractors) of the energy landscape, while other "unwanted" states of the system gradually gain energy thus moving away from the local minimum. The process results in a choosing, from the set of all stable systems, of the one which optimizes the given goal, namely to store and accurately retrieve a given set of patterns even in the presence of noise.

The question concerning the depth of the minima and the height of the barriers between them (the content-addressability) is also addressed and it is found that the basins of attraction of the different stored patterns can be systematically enlarged by the generalized learning process if as it has been argued also in ref 56 and 57 one teaches the system to recognize the original stored pattern even in the presence of noise. For example one presents to the system a noisy modification of a pattern (or equivalently a pattern at a temperature $T\neq0$) and through the generalized learning process teaches the system to recognize it.

The above mentioned learning process also offers an alternative to previously proposed mechanisms that switch the network from recognition to learning when the pattern presented is too distant from all the stored memories. One mechanism proposed by Parisi[57] to overcome the problem arising from the fact that Hopfield networks will always 'recognize' an input_ iterating down to the nearest attractor_, is the introduction of asymmetry into the synaptic couplings. In such a case, though, there is a risk of destabilizing the system.

IV.1 Linearization of the Hamiltonian function

Given a set of N neurons arranged in a general architecture with feedback, an example of which is shown in Fig.6, described by a state vector

$$\hat{Y} = (S_1(t), S_2(t), \dots, S_i(t), \dots, S_N(t)),$$

$\hat{Y} \in \mathbf{R}^N$, and evolving under Eq. (3) (or equivalently Eq. (8)), assume that there exists a Lyapunov function H guaranteeing the stability of the system. Moreover assume that the Lyapunov function has the form:

$$H = \sum_{i_1=0}^{m_1} \dots \sum_{i_N=0}^{m_N} J_{i_1 \dots i_N} S_1^{i_1} \dots S_N^{i_N} \tag{40}$$

If one defines the functions:

$$\theta_{i_1 \dots i_N} = S_1^{i_1} \dots S_N^{i_N} \tag{41}$$

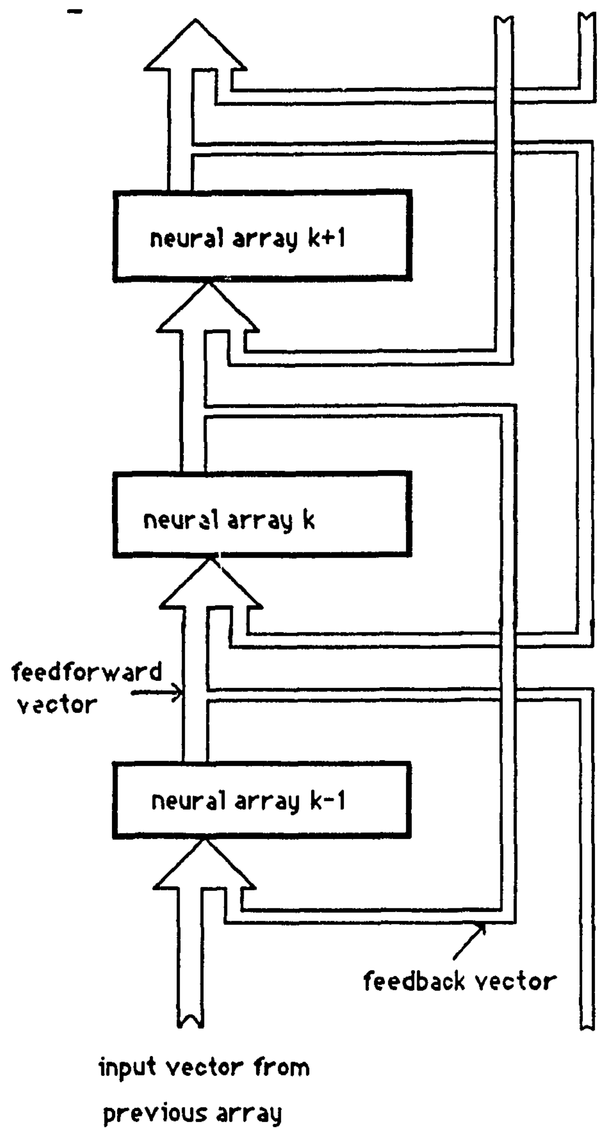


Figure 6. A general multi-layer neural network with feedback. The neural arrays in each layer are groups of interconnected neurons. Their output is communicated to the next layer through the feedforward connections and to the previous layer through the feedback connections.

then Eq. (40) can be written as

$$H = \sum_{i_1=0}^{m_1} \cdots \sum_{i_N=0}^{m_N} J_{i_1 \ldots i_N}\, \theta_{i_1 \ldots i_N} \quad . \tag{42}$$

The coefficients $J_{i_1 \ldots i_N}$ must be arbitrary and thus Eq. (42) describes a class of stable systems each system corresponding to a different choice of $J_{i_1 \ldots i_N}$'s.

The unsupervised training the system undergoes initially can be expressed as a relation between a given set of patterns and the coefficients of the Hamiltonian function. Let $S_\xi = \left(\hat{\xi}^1, \hat{\xi}^2, \ldots \hat{\xi}^p\right)$ be the set of p patterns one wishes to embed in the network. Then any general unsupervised training can be realized as:

$$J_{i_1 \ldots i_N} = \Phi(\hat{\xi}^1, \ldots, \hat{\xi}^p) \tag{43}$$

Φ being any mapping from the set of patterns to the space of the connection strengths. Usually (e.g. ref 49 and 52) the mapping Φ is just a polynomial and Eq. (43) becomes

$$J_{i_1 \ldots i_N} = \sum_{\mu=1}^{p} \sum_{j_1=0}^{k_1} \cdots \sum_{j_N=0}^{k_N} \beta^{i_1 \ldots i_N}_{j_1 \ldots j_N;\mu} \xi_1^{j_1;\mu} \cdots \xi_N^{j_N;\mu} \tag{44}$$

Both learning rules of the previous chapter are special cases of Eq. (44). For example Eq. (20) is recovered if one sets in Eq. (44) all $\beta^{i_1\ldots i_N}_{j_1\ldots j_N;\mu}$ equal to zero except the ones that have two superscripts equal to one. For those one must set

$$\beta^{i_1\ldots i_N}_{j_1\ldots j_N;\mu} = (1/\sum_k a_k^2) \quad \delta_{i_1\ldots i_N\,,\,j_1\ldots j_N}$$

where

$$\delta_{\alpha,\beta} = \begin{cases} 1 \text{ if } \alpha = \beta \\ 0 \text{ otherwise} \end{cases} \tag{45}$$

(the Kronecker δ) and $i_1\ldots i_N$, $j_1\ldots j_N$ have to be considered as N-bit binary words.

For a given input vector $\hat{\psi}^1 \in \mathbb{R}^N$ the system will evolve under Eq. (3) until eventually trapped in a local minimum of the Hamiltonian function. Let $\hat{\eta}^1$ be that minimum (and note that this is the limit point of an orbit starting with $\hat{\psi}^1$), and thus is the output vector of the network. The input vector $\hat{\psi}^1$ can be taken either as a pattern $\hat{\xi}^k$ or some noisy modification of it. Let $\hat{\xi}^k$ be the desired output of the network for input $\hat{\psi}^1$ as decided by the supervisor.

If $\hat{\eta}^1 \neq \hat{\xi}^k$ then there should be a modification of the synaptic strengths $J_{i_1\ldots i_N}$. Let

$$J^{new}_{i_1\ldots i_N} = J^{old}_{i_1\ldots i_N} + \lambda\Delta J_{i_1\ldots i_N} \tag{46}$$

if $J^{old}_{i_1 \ldots i_N}$ is different from zero. Otherwise (i.e., if $J^{old}_{i_1 \ldots i_N} = 0$) no change takes place in the synaptic efficacies. This means that no development of new connections is allowed in the network. One might want later on to remove this restriction and investigate the effects of growth in this class of models (this is done in the next chapter). In Eq. (46) λ is a positive parameter determining the learning strength.

This modification must result in a certain remodeling of the energy landscape. In the new energy landscape the desired output $\hat{\xi}^k$ should be energetically more favourable than the vector $\hat{\eta}^1$. The energy change introduced by the change in the $J_{i_1 \ldots i_N}$'s is

$$\Delta H(\hat{Y}) = \lambda \sum_{i_1=0}^{m_1} \ldots \sum_{i_N=0}^{m_N} \Delta J_{i_1 \ldots i_N} \, \theta_{i_1 \ldots i_N} \quad . \tag{47}$$

The proposed modeling of the energy landscape implies that for $\hat{Y} = \hat{\eta}^1$

$$\Delta H(\hat{\eta}^1) = \lambda \sum_{i_1=0}^{m_1} \ldots \sum_{i_N=0}^{m_N} \Delta J_{i_1 \ldots i_N} \, \eta_1^{i_1;1} \ldots \eta_N^{i_N;1} > 0 \tag{48}$$

while for $\hat{Y} = \hat{\xi}^k$

$$\Delta H(\hat{\xi}^k) = \lambda \sum_{i_1=0}^{m_1} \ldots \sum_{i_N=0}^{m_N} \Delta J_{i_1 \ldots i_N} \, \xi_1^{i_1;k} \ldots \xi_N^{i_N;k} < 0 \quad . \tag{49}$$

After completing the modification of the $J_{i_1 \ldots i_N}$'s one proceeds by supplying the network with another input vector $\hat{\psi}^2$ and obtaining another output $\hat{\eta}^2$ as a local minimum of the new Hamiltonian function. Assume that one wants to assign $\hat{\psi}^2$ to the pattern $\tilde{\xi}^m$. If $\tilde{\xi}^m \neq \hat{\eta}^2$ another modification of the $J_{i_1 \ldots i_N}$'s takes place. And so on. The process is repeated until all the inputs have been classified correctly.

Let S_η be the set of outputs of the network during the learning process

$$S_\eta = \left(\hat{\eta}^1, \hat{\eta}^2, \ldots\right). \tag{50}$$

Each pattern in $S_\xi \subset \mathbb{R}^N$ is represented by a point in the N-dimensional space $\mathbb{R}^N$. Each output in $S_\eta \subset \mathbb{R}^N$ is represented by a point in the same space. Eq. (48) and (49) are linear forms in the $(m_1+1)\times(m_2+1)\times\ldots\times(m_N+1)$- dimensional θ-space. This space is a subset of $\mathbb{R}^M$ where $M = (m_1+1)\times(m_2+1)\times\ldots\times(m_N+1)$. In order to simplify the notation one can introduce the following symbolism:

$$\begin{aligned} i &= (i_1, \ldots, i_N) \\ m &= (m_1, \ldots, m_N) \\ J_i &= J_{i_1 \ldots i_N} \\ \theta_i &= \theta_{i_1 \ldots i_N} \end{aligned}$$

Also

$$\sum_{i=0}^{m} = \sum_{i_1=0}^{m_1} \cdots \sum_{i_N=0}^{m_N}$$

and

$$\phi_i^k = \xi_1^{i_1;k} \ldots \xi_N^{i_N;k} \tag{51}$$

$$\omega_i^\ell = \eta_1^{i_1;\ell} \ldots \eta_N^{i_N;\ell} \tag{52}$$

In this notation Eq. (42), (46), (48) and (49) become:

$$H = \sum_{i=0}^{m} J_i \ \theta_i \tag{53}$$

$$J_i^{new} = J_i^{old} + \lambda \Delta J_i \tag{54}$$

$$\Delta H(\hat{\eta}^1) = \lambda \sum_{i=0}^{m} \Delta J_i \ \omega_i^1 > 0 \tag{55}$$

$$\Delta H(\hat{\xi}^k) = \lambda \sum_{i=0}^{m} \Delta J_i \ \phi_i^k < 0 \tag{56}$$

Let ρ be the number of zeros in the **J** matrix. Let Ω be the space spanned by the θ_i's which have nonzero coefficients in the expression for the Hamiltonian (Eq. (53). Then $\dim(\Omega) = M-\rho$ and $\Omega \subseteq \mathcal{R}^{M-\rho}$. Let $\mathcal{L} = M-\rho$, then each $\hat{\phi}^k$ is represented by a point in the $\mathcal{L}$-dimensional Ω space and each $\hat{\omega}^\ell$ is represented by another point in the same space. All $\hat{\omega}^\ell$ belong to one class; let it be named class **A**. All $\hat{\phi}^k$ belong to another class; let it be named class **B**. Now, it should be recalled that the coefficients ΔJ_i in Eq. (47) are unknown and developing a method to find them is the whole purpose of this discussion. The problem of finding them can be stated now though, in a different way. What we actually want to do is to separate classes **A** and **B** by a decision boundary in Ω such that $\Delta H(\hat{Y}) = 0$ on the boundary while $\Delta H(\hat{Y}) > 0$ on one side of the boundary and $\Delta H(\hat{Y}) < 0$ on the other side of the boundary. The fact that $\Delta H(\hat{Y})$ is a linear function imposes the condition that the decision boundary must be a hyperplane in Ω.

Schematically:

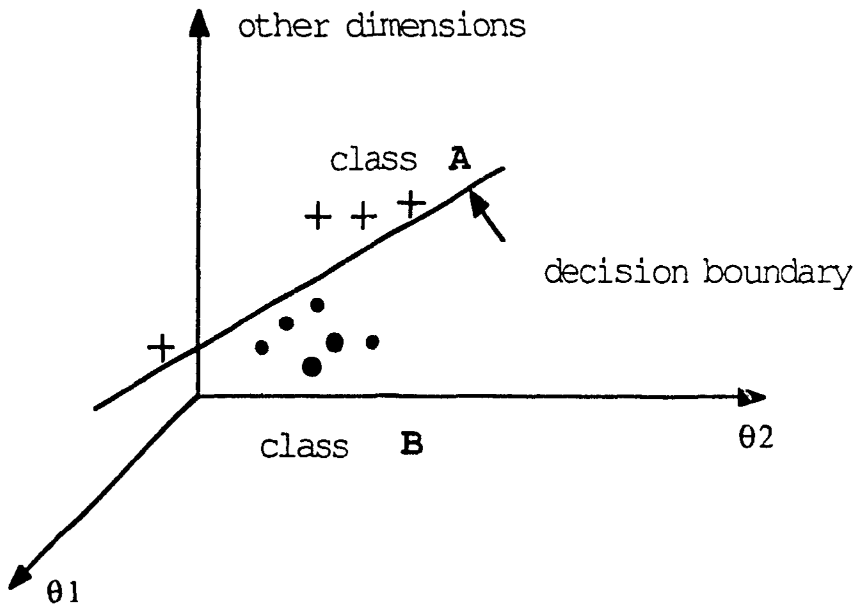


Figure 7. Diagram showing classes **A** and **B** in Ω and a hyperplane separating them. The third axis represents here all the L-2 remaining dimensions.

Then the problem of finding the coefficients ΔJ_i that satisfy Eq. (55) and (56) is simply that of finding the decision boundary in Figure 7, if such a boundary exists. But this is a very well known classification problem within the neural network framework. For example if the two classes **A** and **B** are linearly separable a simple perceptron algorithm[17] will suffice.

IV.2 The choice and implementation of the classification algorithm

Let HP be a hyperplane in Ω as shown in Figure 7,

$$HP = \left\{ \hat{x} \in \Omega : \sum_{i=0}^{m} \Delta J_i \; x_i = 0 \right\}. \tag{57}$$

This hyperplane divides Ω into two subspaces, $(+)$ and $(-)$, with

$$(+) = \left\{ \hat{x} \in \Omega : \sum_{i=0}^{m} \Delta J_i \; x_i > 0 \right\} \tag{58}$$

and

$$(-) = \left\{ \hat{x} \in \Omega : \sum_{i=0}^{m} \Delta J_i \; x_i < 0 \right\}. \tag{59}$$

The kth stage of the proposed algorithm consists of supplying the network with an input vector $\hat{\psi}^k$, in $\mathbf{R}^N$, which one desires to associate with the pattern $\hat{\xi}^k$. Let $\hat{\eta}^k$ in $\mathbf{R}^N$ be the output of the network at this stage. Then one forms the vectors $\hat{\phi}^k$ and $\hat{\omega}^k$, in Ω, according to Eq. (51) and (52) respectively. Given $\hat{\phi}^k$ and $\hat{\omega}^k$ one has the new sets S_ξ and S_η as well as the classes **B** and **A**. One then can give these two classes as input to an $\boldsymbol{L}$ -neuron single-layered perceptron. If $\hat{\omega}^m$ in S_η belongs to $\langle - \rangle$ and thus is misclassified, then the perceptron algorithm suggests that one should change the ΔJ_i's as follows:

$$(\Delta J_i)^{m+1} = (\Delta J_i)^m + \omega_i^m \;, \quad \forall\ i \;,\ 0 \leq i \leq \boldsymbol{I} \;. \qquad (60a)$$

By the same token if $\hat{\phi}^m$ in S_ξ belongs to $\langle + \rangle$ then it is also misclassified. The perceptron algorithm, for an input vector equal to $\hat{\phi}^m$, then gives for the change in the ΔJ_i's

$$(\Delta J_i)^{m+1} = (\Delta J_i)^m - \phi_i^m \;, \quad \forall\ i \;,\ 0 \leq i \leq \boldsymbol{I} \;. \qquad (60b)$$

If $\hat{\omega}^m$ or $\hat{\phi}^m$ are correctly classified then no change in the ΔJ_i's occurs. In the case that only two vectors are given to be classified at the kth stage of the algorithm and if in addition the vectors $\hat{\omega}^m$ and $\hat{\phi}^m$ are of the same magnitude (as is the case with binary neurons) then Eq. (60a) and (60b) can be combined into one and the total change in the ΔJ_i's, at the kth stage of the algorithm will be:

$$(\Delta J_i)^{k+1} = (\Delta J_i)^k + \omega_i^k - \phi_i^k \;, \quad \forall\ i \;,\ 0 \leq i \leq \boldsymbol{I} \;. \qquad (60c)$$

Eq. (60a) and (60b) or Eq. (60c) supplied with the initial condition $(\Delta J_i)^0 = 0 \ , \ \forall \ i$ is the essence of the learning scheme proposed in this chapter. This is a local learning rule in the sense that changes in a synapse are caused by the condition of the neurons at the endpoints of that synapse.

By the general perceptron convergence theorem[17] this algorithm will converge in finite time if the two classes of points in Ω, class **A** and class **B** are linearly separable. Otherwise the learning algorithm will cycle[58].

At this point one should make the additional assumption that every component $S_i(t)$ of the state vector $\hat{Y}$ is bounded. This is a very plausible assumption given the limitations of the actual biological networks. Then the Hamiltonian function itself is bounded and one can argue as follows :

If the Hamiltonian H is bounded, then as long as the classes **A** and **B** remain linearly separable, the network will eventually store the set S_ξ of p patterns one wishes to embed in it, and the learning process will stop. Let $\hat{\xi}^{\nu}$ be any pattern in S_ξ . Let $H(\hat{\xi}^{\nu})^{in}$ be its initial energy

$$H(\hat{\xi}^{\nu})^{in} = \sum_{i=0}^{m} J_i^{in} \, \sigma_i^{\nu} \qquad . \tag{61}$$

It is expected that, for a small enough learning strength λ, at each step of the generalized learning algorithm the energy landscape in the vicinity of the pattern $\hat{\xi}^{\nu}$ will be modified in such a way that the energy of $\hat{\xi}^{\nu}$ is systematically decreased. Because H is bounded it will eventually reach a minimum $H_{min}(\tilde{\xi})$. The above argument holds for every pattern in S_ξ; thus all the p patterns will become local minima of the Hamiltonian H and the learning algorithm will stop.

Of course as the number of patterns in S_ξ increases, the number of points in the classes **A** and **B** increases. The number of points in class **A** increases much faster than the number of points in **B** because class **A** contains all the outputs of the network for all the input vectors which are fed many times into the network while class **B** contains only p patterns. Thus eventually the two classes will not be linearly separable, and one expects a critical value α_c of the ratio p/N to exist above which no generalized learning is possible (for binary neurons and random patterns one can consult ref 45, 56, 59 and 60 on the value of α_c).

Another advantage of this method is that it can be applied to effective Hamiltonians used to describe synchronous dynamics, i.e., networks of neurons that are updated in parallel (Eq (3) is applied to all the neurons at each time step). An example of such a Hamiltonian is the one characterizing Little's[32] model. It has been shown[61] that as long as the connectivity matrix **J** is symmetric, Little's model leads to a stationary Gibbs distribution of states, $\exp(-\beta\bar{H})$, with the effective Hamiltonian

$$\bar{H} = -\frac{1}{\beta}\sum_i \ln\left[2\cosh\left[\beta\sum_j J_{ij}S_j\right]\right] \quad . \qquad (62)$$

One could use a Taylor expansion to linearize (62) and then apply the generalized learning algorithm to the resulting linear effective Hamiltonian.

IV.3 A particular example : Neurons with self-interactions

A network consisting of a fully connected set of neurons with feedback and in which the neurons are allowed to self-interact is studied under the framework of the generalized

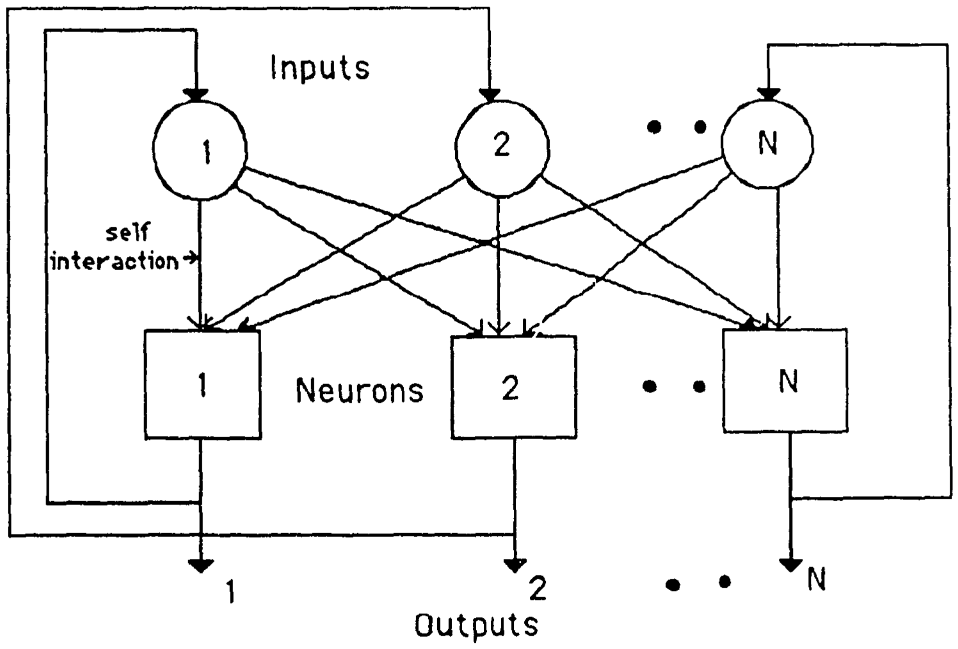


Figure 8. A fully connected network with feedback in which the neurons are allowed to self-interact.

learning algorithm presented above. Such a network is represented diagrammatically in the following figure.

Assuming, as before, $S_i(t)$ to be the dynamical variable that describes the temporal evolution of neuron **i** (it could be called a spin), then the dynamics of the network under discussion, in the absence of noise (T=0), is determined by the equation,

$$S_i(t+1) = \mathrm{sign}\left(\sum_j J_{ij} S_j(t)\right) \tag{63}$$

As is obvious from Eq. (63) the time evolution of the network is considered at discrete time intervals. We will consider the case where the neurons are updated asynchronously. This means that at time t one neuron is selected at random, its local field

$$h_i = \sum_j J_{ij} S_j(t) \tag{64}$$

is evaluated and the spin S_i is flipped if it was misaligned with its local field, and otherwise remains unchanged. All the other spins remain unchanged.

Such a dynamical system is definitely stable if $J_{ij}=J_{ji}$ and $J_{ii} > 0$. Then one can define the Hamiltonian function (see Appendix B)

$$H = -1/2 \sum_{\substack{i,j \\ (i \neq j)}} J_{ij} S_i S_j \quad . \tag{65}$$

(This system might be stable even if $J_{ii} < 0$ but governed by a different Hamiltonian.)

The Hamiltonian in Eq. (65) is identical to the Hopfield Hamiltonian. Thus one does expect the general thermodynamic properties of the two systems to be identical. The advantage of using a system with self interactions is, though, three-fold. First, it can be shown[62] that on a fully connected network a diagonal term of order 1 (for non-diagonal terms of order $1/\sqrt{N}$) influences the convergence properties favourably and one would like to evaluate the effects of non-zero self interacting terms in systems with finite N. Second, one can see and compare how the two systems (one with and the other without self-interactions) behave under the generalized learning algorithm. And third, one would like to remove the conditions of stability for both systems and thus to assess the robustness of the generalized learning.

The neurons here are binary units, $S_i = \pm 1$, and the space of inputs is restricted to $\{1,-1\}^N = \{1,-1\}\times\{1,-1\}\ldots\times\{1,-1\}$. In order to apply generalized learning to this Hamiltonian one has to transform Eq. (65) into the form of Eq. (40). Eq. (65) can be written as

$$H = -\sum_{j>i} J_{ij} S_i S_j = \sum_{i_1=0}^{1} \ldots \sum_{i_N=0}^{1} J_{i_1\ldots i_N} S_1^{i_1}\ldots S_N^{i_N} \tag{66}$$

with,

$$J_{0\ldots0} = 0 \tag{67}$$

$$J_{0\ldots i_k\ldots 0} = 0\ ,\ i_k=1,\quad \forall\ i_k\ ,\quad 1\leq k\leq N \tag{68}$$

$$J_{0\ldots i_k.0.i_m\ldots 0} = -J_{km}\ ,\ i_k = i_m = 1,\ \forall\ (k,m),\ 1\leq k,m\leq N \tag{69}$$

and all other $J_{i_1\ldots i_N}$'s with more than two indices equal to 1 vanishing.

One also selects a set S_ξ of binary patterns ($\xi_i^\mu = \pm 1$) to be stored in the network. For the initial J_{ij}'s one can choose a simple Hebbian assignment, thus selecting a particular network having

$$J_{ij} = (1/N) \sum_{\mu=1}^{p} \xi_i^\mu \xi_j^\mu \tag{70}$$

This is of the form of Eq. (44) with

$$\beta^{i_1\ldots i_N}_{j_1\ldots j_N;\mu} = (1/N)\ \delta_{i_1\ldots i_N\ ,\, j_1\ldots j_N}$$

whenever two and only two of the superscripts $i_1,\ldots,i_N$ equal to one. In all other cases

$$\beta^{i_1\ldots i_N}_{j_1\ldots j_N;\mu} = 0$$

Now to the model characterized by Eq. (66) to (70) one can apply the generalized learning process as described in the previous section.

One should mention here that in this particular case, Eq. (60c) which expresses the generalized learning algorithm for binary neurons and error correction involving only one pattern at a time, can be written in the simplified form

$$(\Delta J_{ij})^{k+1} = (\Delta J_{ij})^{k} + (\eta_i^k \eta_j^k - \xi_i^k \xi_j^k) \quad , \quad \forall \ (i,j) \quad , \ 0 \le i, j \le N \ . \qquad (71)$$

From this form one can see that the learning process of Krey and Pöppel[56] is just a special case of the generalized learning that is proposed here.

IV.4 Simulations and Results

The first part of the simulations was done in order to evaluate the effect of the inclusion of self-interactions in the performance of small networks armed with generalized learning. The computer program first (for a listing see Appendix C) generated an array of N neurons. Each neuron in this array was a structure that contained all the relevant information to the particular neuron. For the above example the relevant information was the value of the dynamical variable $S_i(t)$ that describes the neuron and the values of the synaptic strengths that connect it with other neurons. After that the program generated the set S_ξ of p patterns one wished to embed in the network. For this part of the simulation this set included p random uncorrelated binary patterns. Each component of each pattern was chosen independently according to:

$$P(\xi_i^\mu) = (1/2)\delta(\xi_i^\mu-1) + (1/2)\delta(\xi_i^\mu+1)$$

by the use of a random number generator. The function $\delta(x)$ is Dirac's delta function.

This set of patterns was then used for the computation of the initial synaptic strengths for each neuron according to Eq. (70) and then stored in memory. For the rest of the simulation this set constituted just the reference set. After that a set of input vectors S_ψ was created. This set contained the original patterns and several noisy modifications of them. The noisy modifications were obtained by randomly flipping a certain percentage of the neurons in the original pattern (this percentage was taken between 1% and 35%). An input vector $\hat{\psi}$ was then selected randomly from S_ψ and fed into the network. The processing by the network was done as follows. A neuron **i** was selected at random from the N neurons in the network. Its value then was determined by Eq. (63). Then another neuron was selected at random and so on until no neuron changed its value when Eq. (63) was applied to it.

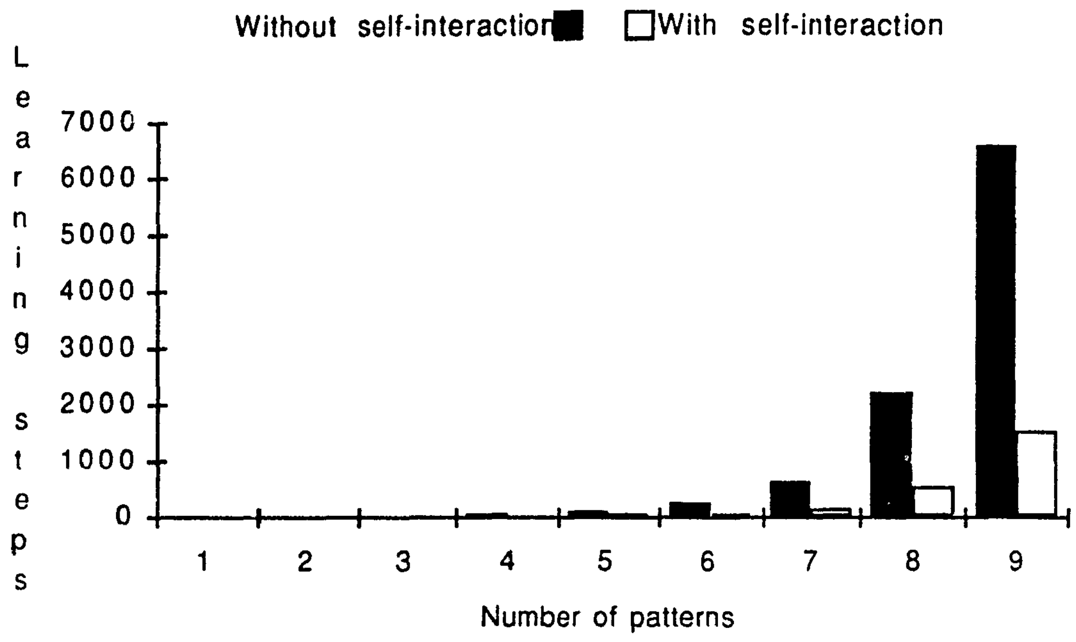
The values of the N neurons in the network at that point constituted the output of the network. That output was compared to the original pattern $\hat{\xi}$ in the reference set from which $\hat{\psi}$ was generated. If the output and the original pattern were identical no change occurred, otherwise the synaptic strengths of each neuron were modified as specified by the generalized learning process Eq. (60c). As it is easy to see for the particular model of binary neurons employed here there can be no change in the self-interaction strengths by the learning process.

After that another input vector was selected and fed into the network and the process was repeated until all the input vectors were properly memorized. The value of the learning strength was taken to be 0.0001 and the self-interaction strengths were taken to be positive and of order p/N.

In the first five of the following graphs the number of learning steps needed for the memorization of p patterns is plotted against p for networks of different sizes. For comparison, results without self-interaction terms are also given in the same graphs. The critical number of patterns for which generalized learning is necessary can be also found from these graphs. It is the value of p for which the learning steps become different from 1.

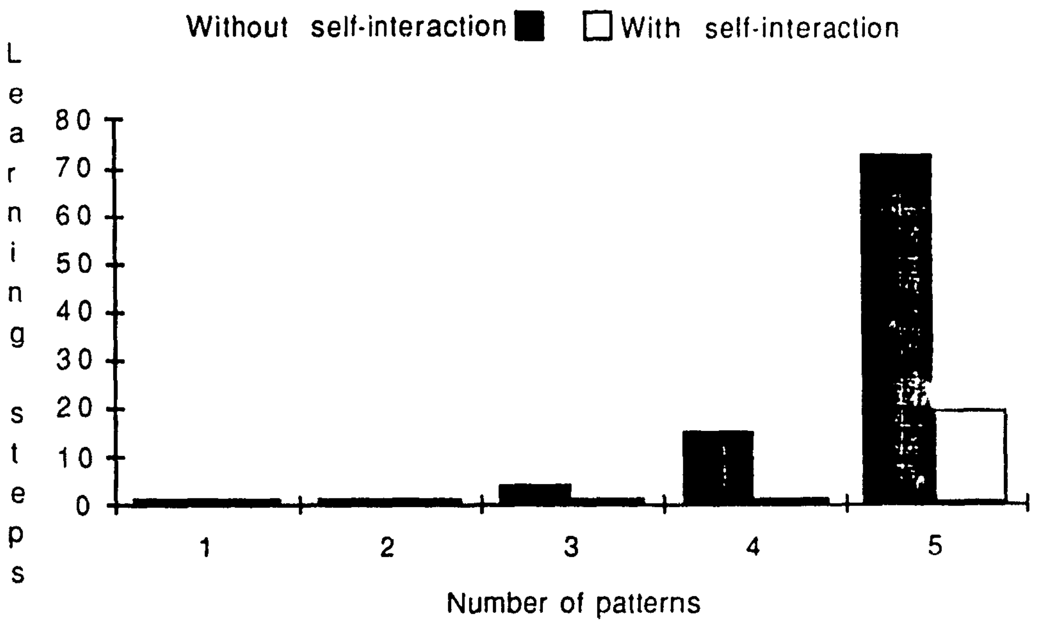
For the results of the simulations presented in Graph 1 a network with 16 neurons was used. Each bar on this graph represents an average over 100 networks. Each network starts with a different set of synaptic strengths corresponding to the different set of random patterns it stores. It is evident that the advantages of using self-interactions persist even when one applies the generalized learning algorithm. The networks that used self-interactions took one third of the time (measured in number of learning steps) taken by the networks without self-interactions to perform the same task.

The ratio of the time taken by networks with self-interactions to that taken by networks without self-interactions seems to be independent of the storage ratio p/N at least up to p = 0.6*N.

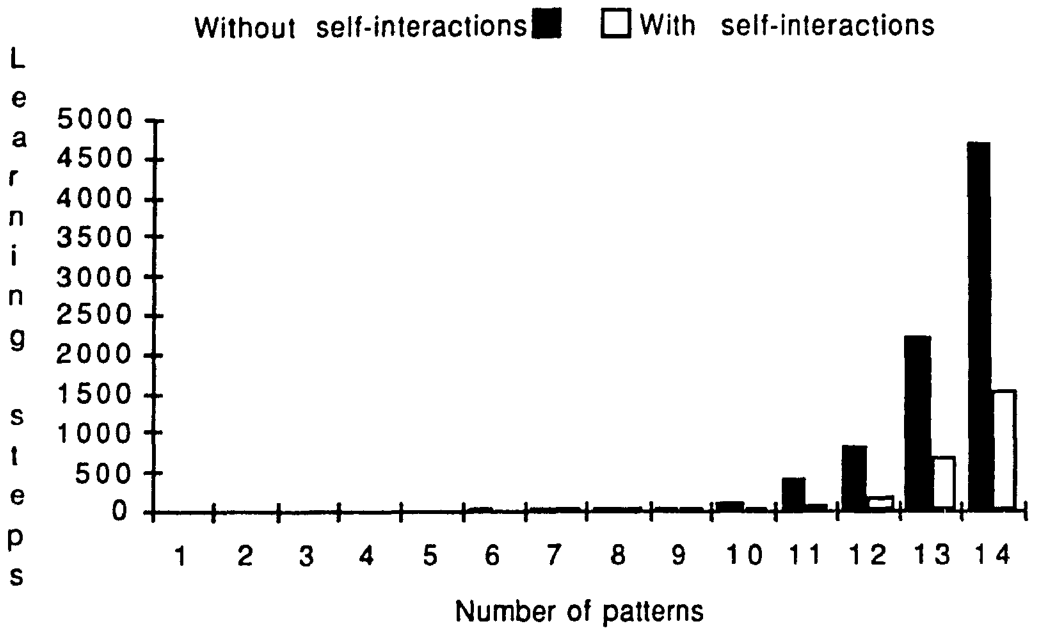


Graph 1. Learning steps versus number of memorized uncorrelated random patterns for a network with 16 neurons.

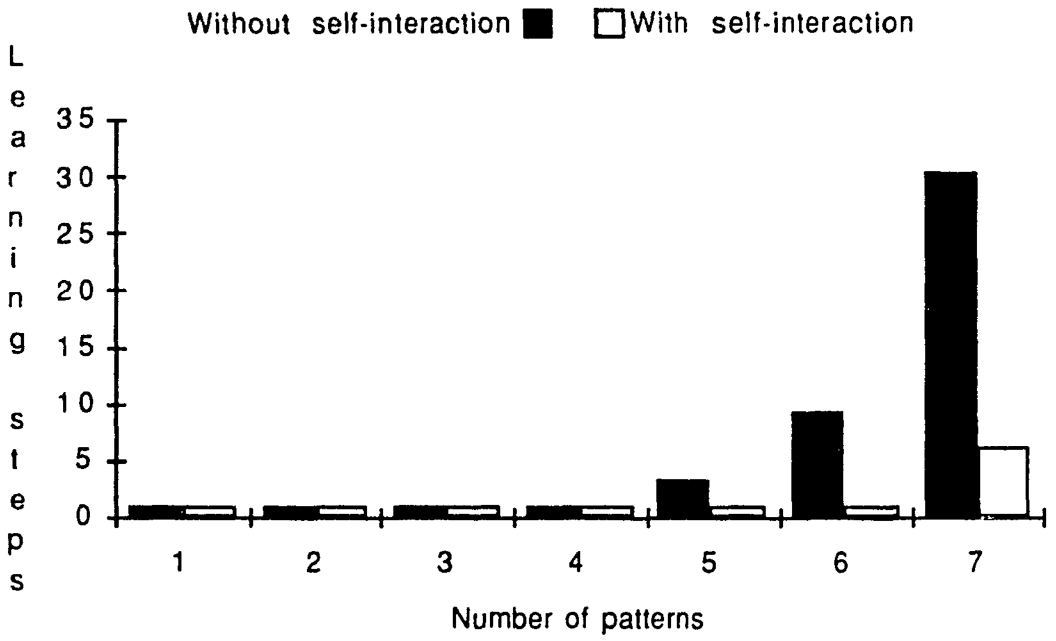
Graph 2 shows the critical number of patterns for which the generalized learning becomes necessary. The network used had 16 neurons and again each bar on the graph represents an average over 100 networks. One can see that the critical number of patterns is twice as much for the network with self-interactions as for the one without them. The critical number of patterns is actually the storage capacity of the network without generalized learning. Thus one can conclude that using self-interactions increases the storage capacity of the network for small N.



Graph 2. Generalized learning becomes necessary at p=3 for a network without self-interacting terms and at p=5 for a network with self-interactions. Both results are given for a network of 16 neurons.



Graph 3. Learning steps versus patterns for a network with 32 neurons.



Graph 4. The critical number of patterns for N=32.

For Graphs 3 and 4 the networks used had 32 neurons. While the ratio of learning steps remained 1/3 for networks with and without self-interactions, the ratio of the storage capacity of the corresponding networks without generalized learning increased to 1/1.5.

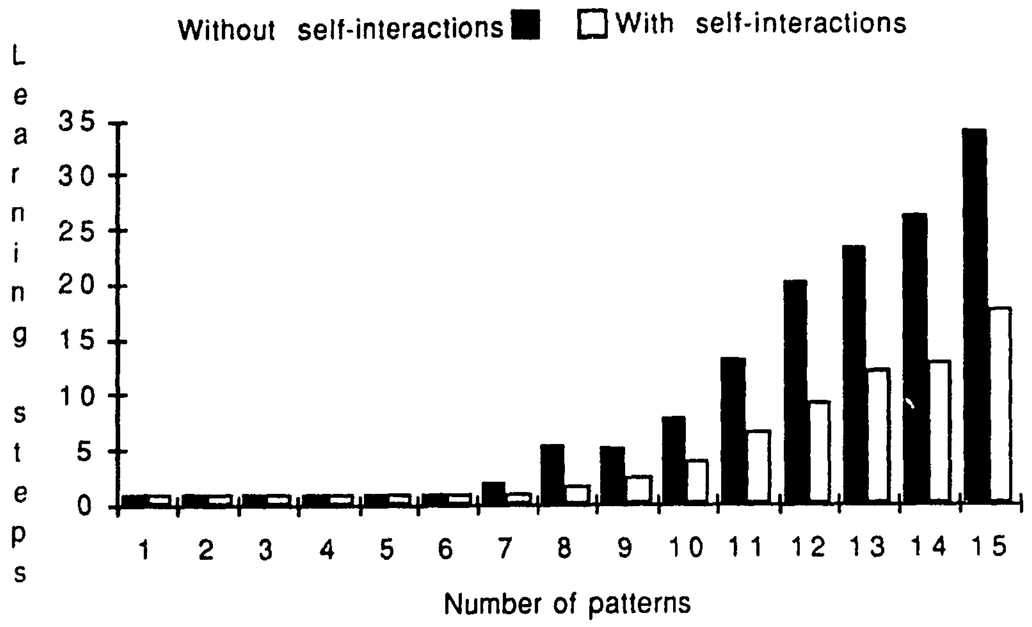
The critical number of patterns is almost equal for the two models when one uses a network with 64 neurons, as can be seen from Graph 5. The ratio of the learning times between the network with self-interactions and the one without remained almost constant between 1/2.5 and 1/2.7.

The above results are in accordance with ref 62. When one uses self-interactions the convergence properties are improved. Moreover this improvement persists during the application of the generalized learning algorithm. For small N even the storage capacity of the network without generalized learning was increased. In all the cases of course, generalized learning allowed the storage of more patterns in the network than would have been possible without it.

The average retrieval quality

$$\bar{m} \equiv (1/p) \sum_{\mu=1}^{p} m(\hat{\xi}^{\mu}, \hat{\eta}^{\mu}) \equiv (1/Np) \sum_{\mu=1}^{p} \sum_{i=1}^{N} \xi_i^{\mu} \eta_i^{\mu} \qquad (72)$$

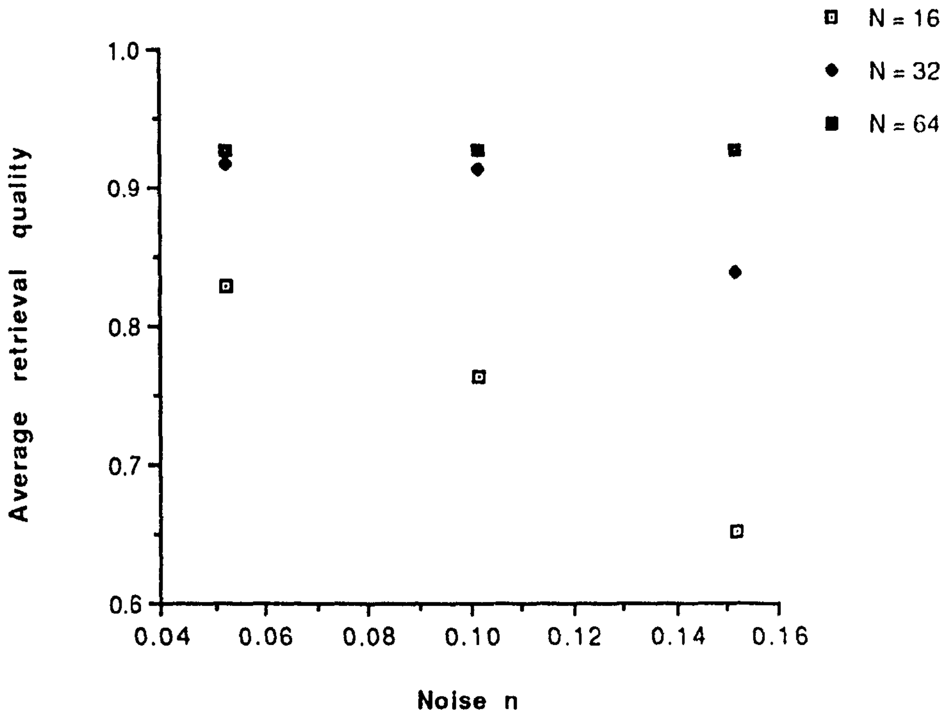
for networks of different sizes and for various noise levels was calculated and is plotted in Graph 6. The networks [illegible]



Graph 5. Time in learning steps versus the number of stored patterns in a network of 64 neurons. The critical number of patterns is p=7 for a network without self-interacting terms and p=8 for a network with self-interactions.

included self-interactions and the number of patterns stored in each network was p=9 (in all cases the use of generalized learning was necessary in order to store those patterns). A noisy modification of a pattern is presented to a trained network (a network that has learned the p patterns). The output of the network is used to evaluate the average retrieval quality via Eq. (72). Each point in the graph represents an average of 20 different noisy modifications of the same set of patterns at constant noise level. The noise n is defined as the percentage of randomly flipped spins in the original pattern.

As one can see, the average retrieval quality drops in general with the noise level _ one does not expect the network to recognize a highly distorted version of a pattern.



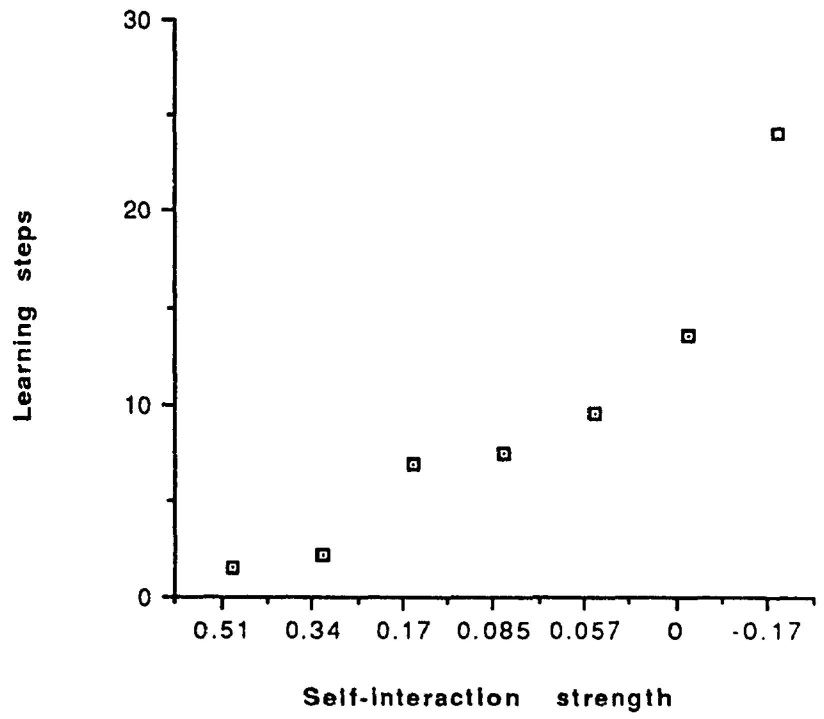
Graph 6. Average retrieval quality of 9 patterns stored in networks of various sizes versus noise level.

The drop, though, is much more prominent for high values of the storage ratio p/N. This reflects the well known fact[45] that as the storage ratio increases the basins of attraction of the stored patterns become smaller and smaller and eventually vanish for a critical storage ratio α_c.

The second part of the simulations had the purpose of assessing the effect of different self-interaction strengths on the generalized learning algorithm. The same computer program was used as in the first part. The value of the learning strength was taken again to be 0.0001 but this time the strength of the self-interaction was varied. The results of the simulations are presented in Graph 7.

A network with 64 neurons was used for these simulations and 11 patterns were embedded in it. Each point in the graph represents the average over 100 networks. As in the case where no generalized learning is present[62] increasing the self-interaction strength improves the convergence properties of the algorithm. A large negative value, though, of the self-interaction acts in reverse. This can be explained by realizing that the self-interaction at least for the first N time steps acts as a local magnetic field antiparallel to the input and as such it highlights the negative of the input pattern (the input pattern with all its spins reversed). This has as a result the delay of the convergence of Eq. (63) to its fixed point for every neuron.

Finally the question of robustness was addressed. For two networks, one made of 16 neurons and the other made of 32



Graph 7. Time in learning steps versus self-interaction strength for p=11 and N=64.

neurons, the symmetry condition of the synaptic matrix **J** was gradually relaxed. The same program was used as in the previous simulations but now the calculation of the initial synaptic strengths was not done according to Eq. (70). An asymmetry was introduced. This was done by using Eq. (70) to compute the initial value of J_{ij}, with i greater than j, and then assign a fraction of it as the value for J_{ji}. That fraction was then gradually diminished till it became zero and the robustness of the generalized learning was studied as a function of the increasing asymmetry in the synaptic strengths. The algorithm was proven robust even with extremely asymmetric connections, i.e., when $J_{ij} \neq 0$ and $J_{ji} = 0$. In such a case the network does not converge to a fixed point to begin with but it is eventually trapped at a metastable point. This point is regarded as the initial output of the network. After the application of the generalized learning algorithm though, the synaptic matrix becomes increasingly more symmetric and more fixed points of the network dynamics start to appear.

IV.5 Concluding Remarks

In this Chapter a powerful and general method was proposed for the training of stable feedback networks. Its application is not restricted to binary inputs which is predominantly the case when one deals with neural networks from the point of view of statistical mechanics. It can be applied to graded or even continuous inputs which is the usual case in engineering applications (signal and image processing, robotics, etc.).

It is a natural process because it involves only local adjustments of the synaptic connections and thus does not require a mechanism by which a single neuron will be informed about the activity of all the other neurons in the network.

(Such a mechanism is not known to exist in the central nervous systems of animals.)

The method was tested through simulation on a particular feedback network and was found to improve its storage capacity as well as its retrieval properties. Training of the network in the presence of noise, $(T \neq 0)$ was though the crucial factor in enlarging the basins of attraction of the stored patterns (memories), thus improving the retrieval quality of the network and its function as associative memory.

The method assumes that it trains a stable network but if as in the case of the previous network one were to remove the conditions of stability the generalized learning method would be found to improve the stability of the network. The generalized learning enforced a set of patterns to become attractors of the dynamics of the network when only chaotic orbits were present.

Chapter V

Growth in stable feedback systems derived from a generalized learning algorithm

Changing the knowledge structure in a neural network involves modifying the pattern of interconnections, i.e., the matrix **J**. In general there are three kinds of possible modifications: (i) the development of new connections; (ii) the loss of existing connections; and (iii) the modification of the strengths of connections that already exist. Very little work has been done on the first two types of modifications. The interest in growth algorithms is presently concentrated in feed-forward networks like multi-layer perceptrons[63,64] and neural network decision trees[65]. The type of growth in those models is not merely a growth of new synapses but also a growth in size: new neurons are employed as they are needed and their synaptic connections are computed. The growth of new synapses can to a first order approximation, be considered as a special case of the modification of existing synapses. For example whenever one changes the strength of a connection between two neurons away from zero to some positive or negative value, it has the same effect as growing a new connection.

Within this approximation, a modification of the generalized learning process that was proposed in the previous chapter can be used to explore growth processes in feedback networks described by a parametric Lyapunov function. Thus here learning and growth are viewed as different aspects of the same mechanism. In certain cases if the architecture and the dynamics permit one can move smoothly from one dynamical system with a certain connectivity matrix **J** (given in the parametric form of Eq. (43) or (44)) to another dynamical system with connectivity matrix **J'**.

This can happen if there is a "parent" stable system from which all the other systems can be derived as special cases by reducing certain elements of its connectivity matrix to zero. If such a "parent" system exists then one can say that this system forms a framework in which growth can take place in the spirit mentioned in the above paragraph, without disturbing the stability of the system.

V.1 The "parent" system

Consider the system of N binary neurons in which the dynamical behaviour of neuron i_1 is governed by

$$S_{i_1}(t+1)=\mathrm{sign}\Big(h_{i_1}+\sum_{i_2} J_{i_1 i_2}S_{i_2}(t)+\sum_{i_2<i_3} J_{i_1 i_2 i_3}S_{i_2}(t)S_{i_3}(t)+\dots$$

$$+\sum_{i_2<i_3<\dots<i_N} J_{i_1\dots i_N}S_{i_2}(t)S_{i_3}(t)\dots S_{i_N}\Big) \qquad (73)$$

and the index i_1 is different from all other indices.

The Hamiltonian for this system, within a configuration-independent constant, is

$$H = \sum_{m_1=0}^{1} \cdots \sum_{m_N=0}^{1} J_{m_1 \ldots m_N} S_1^{m_1} \cdots S_N^{m_N} \qquad , \qquad (74)$$

where $J_{m_1 \ldots m_N}$ is invariant under permutations of the indices $m_1, \ldots, m_N$.

The systems discussed in Chapter III as well as the example of Chapter IV are special cases of the above system with certain elements of its synaptic matrix **J** turned off. The fact that such a system exists allows one to take a subsystem of Eq. (74) and let it grow as the needs of the task at hand demand. Thus one does not waste memory space and communication time by choosing a very big totally interconnected network with multi-spin interactions to perform a given task that might require a much simpler network with a smaller number of synapses and probably only pair-interactions.

For example, one can start with a partially connected Hopfield-type network (Eq. (17)) . A partially connected network proposed by Canning and Gardner[66] employs connection strengths defined by

$$J_{ij} = (1/N) \sum_{\mu=1}^{p} D_{ij} \xi_i^\mu \xi_j^\mu \qquad i \neq j \quad J_{ii} = 0$$

(75)

where **D** is a symmetric matrix which defines the connection architecture. It has a 0 or 1 at position i,j depending on whether site **i** is connected to site **j** or not. Such a system has an increased storage capacity per connection but requires many more neurons to have any significant storage capacity over a fully connected system. Assume now that someone wants to store p patterns in a network. Then if a partially connected network is not sufficient to store those patterns, one must increase the storage capacity by switching on the magnetic fields h_i or by switching on more pair connections or even by switching on multi-neuron interactions.

V.2 A growth algorithm for feedback systems

One starts with a certain subsystem $\mathbf{H_S}(\mathbf{J_S})$ of a "parent" system $\mathbf{H}(\mathbf{J})$, where $\mathbf{J_S}$ and $\mathbf{J}$ are the connectivity matrices of the subsystem and the "parent" system respectively. One embeds the patterns to be stored in the subsystem $\mathbf{H_S}(\mathbf{J_S})$ according to Eq. (43). Then one proceeds with the generalized learning process described in the previous chapter. If the classes **A** and **B** generated by the process are not linearly separable, the process will cycle, meaning that the subsystem $\mathbf{H_S}(\mathbf{J_S})$ is not capable of storing the desired patterns under Eq. (43) supplemented with the use of generalized learning. Then one relaxes the restriction that if $J^{old}_{i_1 \ldots i_k} = 0$ no change takes place in the synaptic efficacies and allows the system to grow. This means that we decrease the number of zeros ρ in the $\mathbf{J_S}$ matrix thus increasing dim(Ω). The new subsystem

$\mathbf{H_{S'}}(\mathbf{J_{S'}})$ is now defined in a higher-dimensional space Ω'. The final values of $\mathbf{J_S}$ may or may not be of any use to the new subsystem. One must remember that the classes **A** and **B** generated by $\mathbf{H_S}(\mathbf{J_S})$ were not linearly separable in Ω and thus will not be linearly separable in Ω'. The new subsystem must generate its own class **A'** (while **B**, the class containing the patterns remains the same) in Ω'. $\mathbf{H_{S'}}(\mathbf{J_{S'}})$ is actually a new Hamiltonian system with its own energy landscape. The advantage is that in the higher-dimensional space Ω' one expects that the set of points generated will be more sparsely distributed; thus in a sense one can store more patterns before reaching the point where **A'** and **B** are not linearly separable.

Starting anew with subsystem $\mathbf{H_{S'}}(\mathbf{J_{S'}})$ but retaining any initial assignments of the synaptic strengths due to an unsupervised learning, one has

$$\mathbf{H_{S'}}(\mathbf{J_{S'}}) = \mathbf{H_S}(\mathbf{J_S}) + \Delta\mathbf{H} \tag{76}$$

and $\Delta\mathbf{H} = 0$ to begin with. One must allow $\Delta\mathbf{H}$ to grow fast till it reaches at least the order of magnitude of $\mathbf{H_S}(\mathbf{J_S})$ evaluated at any of its minima if one really wants to have a new Hamiltonian system with its own class **A'**. If $\Delta\mathbf{H}$ grows slowly it will be for the greatest part of the generalized learning a small correction to $\mathbf{H_S}(\mathbf{J_S})$; thus the set of points in class **A** is more likely to be generated again. (Simulations done with different growth rates for $\Delta\mathbf{H}$ are in complete agreement with the above statement.)

The above discussion implies that Eq. (46) should be initially applied with much greater learning strength λ (of order unity) for the new connections than for the old ones. We will denote this value of λ during growth as λ_g . Eq. (54) should read during growth

$$J_i^{new} = J_i^{old} + \lambda(\Delta J_i)^S + \lambda_g(\Delta J_i)^{S'} \qquad (77)$$

where the superscript S refers to the connection matrix $\mathbf{J_S}$ while the superscript S' to the matrix $\mathbf{J_S} - \mathbf{J_{S'}}$. The matrices $\mathbf{J_S}$ and $\mathbf{J_{S'}}$ belong to the same linear space, the linear space to which the connectivity matrix of the "parent system" belongs. The fast-growth period should stop when

$$\Delta\mathbf{H}(\hat{\xi}^{\nu}) = O(\mathbf{H_S}(\hat{\xi}^{\nu})) \qquad \nu=1,\ldots,p \qquad (78)$$

where O() denotes order of magnitude. After that the learning strength should be homogenized to a small value for fine tuning of the energy landscape. Eq. (77) is replaced by Eq. (54) and the generalized learning continues as in Chapter IV.

Here one should mention that the question of optimal learning rate is task dependent and one cannot infer much from specific situations. On the other hand the learning strength λ is kept small because one does not want the generalized learning process to affect drastically the basins of attraction of the original energy landscape imposed on the system by the unsupervised learning. The reverse reasoning dominates when one allows the system to grow. One wants large scale changes in the energy landscape thus $\Delta\mathbf{H}$ should be on equal footing with $\mathbf{H_s}$ and λ_g should be of order unity.

The way the new connections grow is always governed by Eq. (60a) and (60b) or Eq. (60c), the fundamental equations of our learning scheme, thus ensuring that the desired patterns are always lowering their energy while the unwanted ones increase theirs. The method described here ensures that the new Hamiltonian subsystem $\mathbf{H_{s'}}(\mathbf{J_{s'}})$ will create its own class of points $\mathbf{A'}$. Whether or not $\mathbf{A'}$ and $\mathbf{B}$ will be linearly separable depends on the set S_ξ of the p patterns. If it is not, though, one can proceed by allowing the network to grow again till one finds a network that will store and effectively retrieve the p patterns.

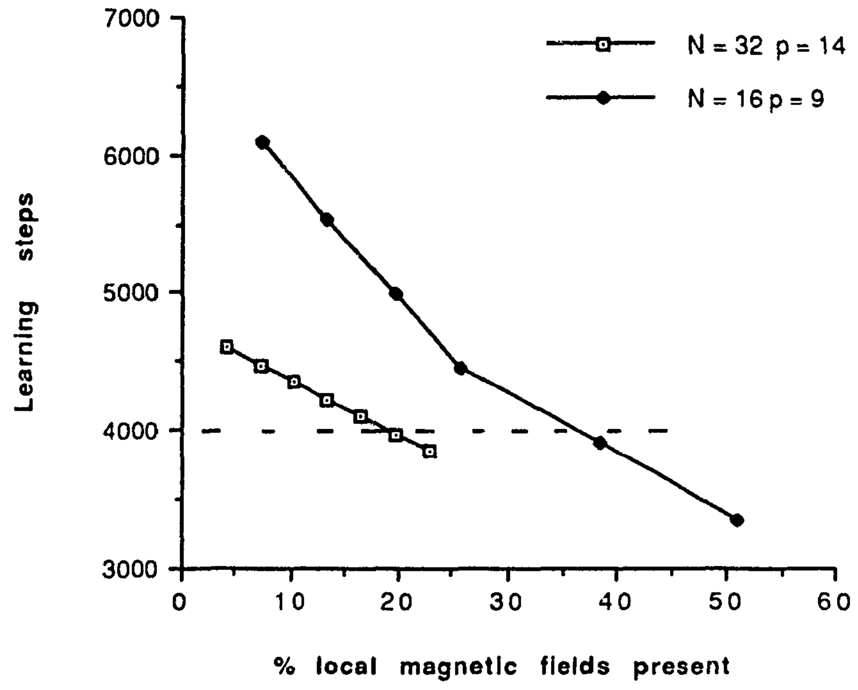
The problem of finding the minimal growth, starting from a given subsystem $\mathbf{H_s}(\mathbf{J_s})$, that will store a given set of patterns is a very difficult one. Thus we here suggest a *heuristic* approach for building a network. In the synaptic matrix $\mathbf{J} = [J_{i_1 \ldots i_N}]$ of the "parent" system there are N groups of elements with one index nonzero and all the other indices equal to zero, N^2 groups with two indices nonzero and N^N groups with all indices nonzero. Given an initial subsystem $\mathbf{H_s}(\mathbf{J_s})$ we assume that $\mathbf{J_s}$ has some elements with one index nonzero, some elements with two indices nonzero and so forth. One lets the network grow by first switching on the rest of the elements in $\boldsymbol{J}$ with only one index nonzero, then the rest of the elements in $\boldsymbol{J}$ with two indices nonzero, and so on.

In other words, one first allows more interactions with a local magnetic field[53], then allows more pair interactions before switching on the three-neuron interactions[67]. This is a crude way of growth that does not take into account the structure of S_ξ. For example with a few three-neuron interactions one might succeed in storing the p patterns while with a lot of interactions with local magnetic fields might not. On the other hand there are only N local fields but N^3 three-neuron synapses so with the above *heuristic* approach one hopes to achieve better space economy and maybe reach a network that is close to the minimal network that will store the p patterns.

V.3 Simulations and Results

The first problem one faces following the above algorithm is to decide whether or not a set of patterns can be stored by the generalized learning by a network $\mathbf{H}_{\mathbf{S}'}(\mathbf{J}_{\mathbf{S}'})$. One actually has to wait *ad infinitum* to find if the perceptron algorithm (Eq. (60a) and (60b)) will cycle or terminate (although one could use Linear Programming to detect non linear separability in polynomial time). This of course is not practical so one defines an upper bound for the number of learning steps and any task that requires more steps than this upper bound is considered unattainable by the given system. This upper bound was taken here to be 4000 learning steps. Many tasks of the simulations in the previous chapter required more than 4000 learning steps but those tasks were inherently difficult for the computers used for the simulations (for example it took approximately 9h of computing time to teach a network of 16 neurons to recognize 9 uncorrelated patterns on a Macintosh SE). But one can use the growth algorithm not only for achieving an otherwise impossible task but also to simplify a difficult (time-consuming) task. If a given task is too difficult for a given network, one can let this network grow in the expectation that the task will be easier for the network with a higher-dimensional Ω.

The purpose of the first part of the simulations in this chapter was to assess how the growth process transforms a problem unsolvable by a system $\mathbf{H_S}(\mathbf{J_S})$ to a problem solvable by a higher-dimensional system (i.e. a system with a higher-dimensional Ω). Two difficult problems were selected from the previous simulations. The problem of storing 9 uncorrelated patterns in a 16-neuron Hopfield-network and the problem of storing 14 uncorrelated patterns in a 32-neuron Hopfield-network (no self-interactions present). A program similar to the one used in Chapter IV was used reflecting the fact that the two processes, learning and growth, are not considered fundamentally different in the present work. An array of N neurons, a set S_ξ of p patterns and a set of input vectors S_ψ was again generated by the program in the same way as in the previous chapter. The processing by the network was done as before. The value of the learning strength was taken to be 0.0001 and the self-interaction strengths were switched off. At this point a counter of the learning steps was added. If the number of learning steps exceeded 4000 then a neuron was selected at random and its interaction with a local magnetic field h_i was allowed. The value of the magnetic field h_i started from 0 and grew according to Eq. (60c) and (77) with a growth strength $\lambda_g = 0.1$ while all the pair interaction strengths were modified with a learning strength of magnitude 0.0001. As one can see the growth strength is 1000 times greater than the learning strength. The order of magnitude of the newly introduced interaction was evaluated at the end of each learning step. When it became comparable to the magnitude of the pair-interaction Hamiltonian evaluated at the desired output of the network and for every such output, growth stopped and learning was resumed by setting $\lambda_g = \lambda$. The learning steps were again counted and if they exceeded 4000 another neuron was selected at random and its interaction with a local magnetic field h_i was allowed. And so on. The process of growth stopped when enough interactions with the local magnetic fields were introduced to make the task at hand feasible (less then 4000 learning steps). Graph 8 shows the results of this simulation.

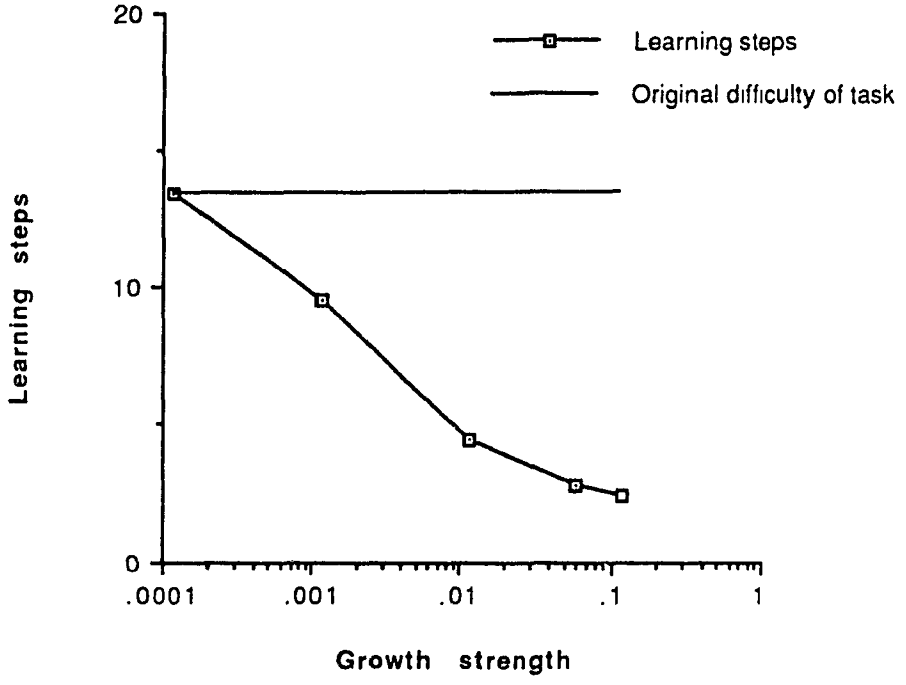


Graph 8. The percentage of interactions with local magnetic fields grown in order to make two different tasks feasible; one storing 9 patterns in a 16-neuron network, and the other storing 14 patterns in a 32-neuron network. For a task to be feasible the number of learning steps had to be less than 4000 (below the dotted line).

Each point in this graph represents an average of 50 networks, each with a different set of initial synaptic strengths corresponding to the different set of random patterns it stores. As one can see, for the more difficult task, i.e., storing 9 uncorrelated patterns in a Hopfield net with 16 neurons, approximately 37% of the neurons had to grow interactions with their local magnetic fields before the task became feasible. For the easier task, i.e., storing 14 uncorrelated patterns in a 32-neuron net approximately 20% of the neurons were interacting with their local magnetic fields when the task became feasible. Thus one concludes that the growth process selects with space economy a network that performs the given task.

The next part of the simulations was done in order to assess the effect of varying growth strengths in the performance of the growth algorithm. A network of 64 neurons was employed and the task was to embed 11 uncorrelated patterns in it. The original difficulty of this task, in other words, the number of learning steps it took the generalized learning algorithm to perform it, was 13. The network grew in one shot, meaning that all the neurons were suddenly allowed to interact with their local magnetic fields and the growth strength was a varying parameter. The results of this simulation are presented in Graph 9.

From this graph it can be seen that a growth strength of equal magnitude to the learning strength does not affect the difficulty of the task. In other words, if a task is



Graph 9. Difficulty of task in learning steps versus growth strength λ_g for storing 11 patterns in a 64-neuron network. The original difficulty is also shown as a straight line.

impossible, small growth strength will not make it possible. On the other hand a growth strength 100 times greater than the learning strength reduces the difficulty of the task to 1/3 of its original value, while for λ_g between 100λ and 1000λ, the difficulty of the task decreases slowly with increasing λ_g. The minimum number of learning steps for this simulation is 2, one step for the network to realize that the given task cannot be performed by the original subsystem with pair-interactions only, and another step to switch on all the magnetic fields and give them appropriate values so that the new network memorizes the given patterns. As in the previous simulation the points in Graph 9 represent averages over 50 different networks.

The last part of the simulations done for this chapter was goal oriented. The question was how the generalized learning and the growth algorithms will perform in an actual pattern recognition situation. The situation was set to be the recognition of characters from the English alphabet.

The human mind creates a concept for each letter of the alphabet which is scale and font invariant. The letters k, k, k, k, **k**, k, k, are all recognized as the same letter by a human. The situation is different for a computer. All these letters are different bitmaps in the computer memory. The concept of a letter has to be artificially created for the computer. Part of the program created for this simulation had to do that (for a listing refer to Appendix C).

First one selects a base font and size for the letters. The Geneva font of size 14 in a Macintosh SE computer was selected. This consists of the following set of letters.

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, g, k, l, m, o, p, q, r, s, t, u, v, w, x, y, z.

One chooses a portion of this set, let's say the set

S = {A, p, l, M, z, x, a, c}

and presents it to the network through a retina. A retina is

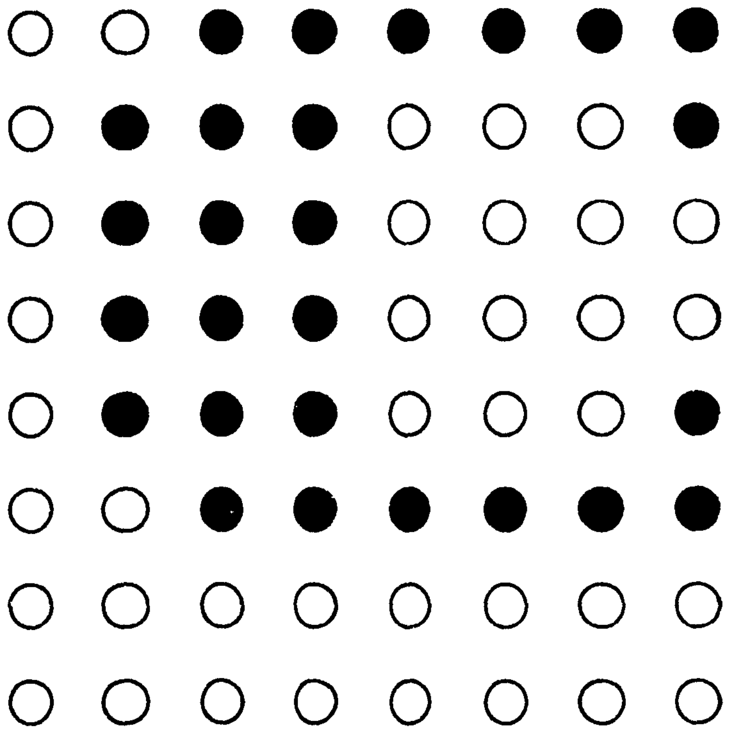
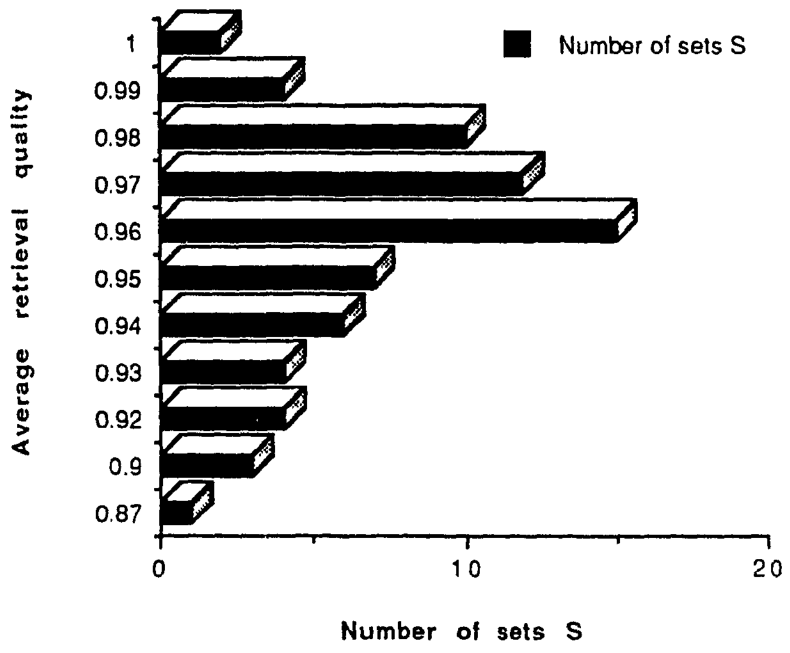


Figure 9. Representation of the letter c suitable for processing by a 64-neuron binary network.

a special part of the computer memory (part of the screen buffer) in which the letters are scanned one by one and then



Graph 10. Average retrieval quality versus the number of sets S of letters from the English alphabet.

are preprocessed. The use of binary neurons imposes on the retina the interpretation as a set of black and white pixels. The preprocessing of the letters consisted of a size renormalization down or up to the size of the network used, in this particular case a network with 64 neurons.

As an example, the form of the letter **c** is shown when it is presented to the network, in Figure 9. We defined this preprocessed 8x8 array as the concept of the letter **c**, for the computer. This array is then transformed into a 64x1 vector by putting each row of the array beside the previous one. This constitutes an input vector to the 64-neuron network. This is done for all the letters in the set S and the resulting set is the set S_ξ of the patterns to be memorized.

With this S_ξ one calculates the initial synaptic strengths. This was done for a Hopfield net Eq. (65) with Hebbian learning Eq. (70) without self-interactions. After that the set S_ψ was prepared by choosing the same letters as in S but of different fonts and sizes and present them to the network through the retina. For example one might choose for the set S_ψ the letters

A, p, l, M, z, x, a, c, *A, p, l, M, z, x, a, c,* A, p, l, M, z, x, a, c.

(in the actual simulations the set S_ψ contained approximately 100 variations of each letter in S). All these letters can be considered as noisy modifications of the letters in S. These letters were fed into the network and then the generalized learning algorithm and the growth algorithm governed the processing done on them. All this constituted the training phase of the network. After the network was trained all the letters in S were presented to it in a font and size chosen randomly_ not necessarily in a font and size used in the training set_ and the average retrieval quality for that set of letters was computed using Eq. (72).

Many different sets were used in the simulations. They varied in size from sets containing a few letters to sets containing the whole alphabet (larger networks were used to process larger sets of letters). They also varied in content: some sets had only small letters as members, some contained only capital letters and some consisted of mixtures of capital and small letters indiscriminately.

In order to assess the ability of the network to recognize letters that look similar, sets with different average correlation

$$\bar{Q} = \frac{1}{p(p-1)} \sum_{\substack{\mu,\nu=1 \\ (\mu>\nu)}}^{p} Q_{\mu\nu} = \frac{1}{Np(p-1)} \sum_{\substack{\mu,\nu=1 \\ (\mu>\nu)}}^{p} \sum_{i=1}^{N} \xi_i^\mu \xi_i^\nu$$

were employed. A plot of the average retrieval quality against the number of sets exhibiting that retrieval quality, for a total of 68 different sets of letters is given in Graph 10. Most of the sets (72%) had over 0.95 average retrieval quality, a very encouraging result for font and scale invariant character recognition.

V.4 Discussion

In this chapter growth is introduced as the development of new synapses between neurons. It is closely linked to learning in the sense that both processes (growth and learning) are governed by the same equations. On the other hand, growth becomes necessary when learning fails to make a network capable of performing a given task or if a given task is extremely difficult for the network. It is possible that the mechanisms described here are similar to the ones existing in the brain. When a task becomes impossible for the brain to perform (for example, injury to the motor cortex creates a loss of control of part of the body), new synapses are developed to overcome the problem and render the task possible once again.

Using the growth mechanisms one can construct networks that are suitable for performing particular tasks in pattern recognition. These networks are more efficient than their fixed architecture counterparts, as results from tests on recognition of the letters of the English alphabet prove.

Chapter VI

Formal neurons and logic nodes

Most of the approaches to neural networks try to be close to what is known of real neurons. Neural network research though is not limited to just mimicking what nature has already done (it might even be impossible). As mentioned in Chapter II the list of different types of neurons and the list of parameters characterizing each of them is an ever increasing one. Extending this experimental trend arbitrarily to its limit one might attempt to consider neurons of the foremost generality. Such a neuron will be a device that performs a logic function. How it performs it becomes unimportant. One does not care either about the details of the connections (e.g. their strength or their symmetry) or about the details of the neural function (e.g. sum-of-weighted-inputs).

A neuron like that may be represented diagrammatically as in Figure 10. There are n+1 inputs to the neuron namely $\mathbf{I}_0, \mathbf{I}_1, \ldots, \mathbf{I}_n$ and one output $\mathbf{O}$. What logic function the neuron performs is decided by a single number R. The meaning of R will be explained in the following paragraphs.

Assume that each input takes values from $\mathbf{Z}_k$ (the set of integers $\leq k$). Then the output can be any function in $\mathbf{Z}_k$. In other words

$$\mathbf{O} = f\ (\mathbf{I}_0, \ldots, \mathbf{I}_n)\ . \qquad (79)$$

There are $k^{\left(k^{n+1}\right)}$ such functions.

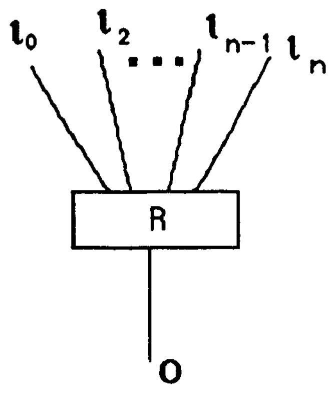


Figure 10. Typical representation of a formal neuron with n inputs and one output characterized by a single number R.

If one defines

$$v = \sum_{j=0}^{n} \mathbf{l}_j k^{n-j} \tag{80}$$

then v belongs to $\mathbf{Z}_{k^{n+1}}$. Essentially v is just the number $\mathbf{l}_0\mathbf{l}_1 \ldots \mathbf{l}_n$ expressed in the k basis. Now take any number R, $0 \leq R < k^{\left(k^{n+1}\right)}$. There is a one to one correspondence between the numbers R and the functions f in Eq. (79). One can actually rewrite Eq. (79) in terms of R

$$\mathbf{O} = \left[\frac{R}{k^v}\right] \bmod (k) \tag{81}$$

where [] denotes the integer part and mod() is the modulo function. Each different R in Eq. (81) will give a different function f in Eq. (79). A neuron of the form presented in Figure 10 and supplied with an R as in Eq. (81) will be called a formal neuron. A similar neuron represented though by a canonical logic expression has been introduced by I.Aleksander[5,68] and it is named a logic node.

As an example assume 4 binary inputs to the formal neuron and assign it the task to fire if the number of inputs that are firing are more than two (firing=1, not-firing=0). A formal neuron with R=59,520 will perform that task. In this case k=2 and

$$O = \left[\frac{59,520}{2^v}\right] \mathrm{mod}(2) \quad .$$

The outputs of the above neuron for all the possible inputs are shown in the Table 1.

All is fine as long as one is restricted to a small number of inputs. But R increases as $k^{\left(k^{n+1}\right)}$ with the number of inputs n and any significant number of inputs will make it totally unrealistic. The first thing one might want to try is to clamp R at a definite size. Assume that R is allowed to have only m digits in the k basis. Then Eq. (81) should be rewritten as

$$O = \left[\frac{R}{k^{v\mathrm{mod}(m)}}\right] \mathrm{mod}(k) \quad . \qquad (82)$$

v = 0	O = 0	f(0,0,0,0)=0
v = 1	O = 0	f(1,0,0,0)=0
v = 2	O = 0	f(0,1,0,0)=0
v = 3	O = 0	f(1,1,0,0)=0
v = 4	O = 0	f(0,0,1,0)=0
v = 5	O = 0	f(1,0,1,0)=0
v = 6	O = 0	f(0,1,1,0)=0
v = 7	O = 1	f(1,1,1,0)=1
v = 8	O = 0	f(0,0,0,1)=0
v = 9	O = 0	f(1,0,0,1)=0
v = 10	O = 0	f(0,1,0,1)=0
v = 11	O = 1	f(1,1,0,1)=1
v = 12	O = 0	f(0,0,1,1)=0
v = 13	O = 1	f(1,0,1,1)=1
v = 14	O = 1	f(0,1,1,1)=1
v = 15	O = 1	f(1,1,1,1)=1

Table 1. The values of a 4-input binary function that becomes one if more than two of its inputs are one.

This will be the response function associated with a formal neuron from now on. It might be considered as the definition of the response function of a formal neuron. One is interested in the type of functions Eq. (82) computes. In order to find an answer to that assume the natural ordering in the set $\mathbf{Z}_k{}^{n+1}$, the set to which v belongs. Moreover assume that the inputs v are arranged in increasing order, the function f is applied to them in that order and the outputs $\mathbf{O}$ are given in that order. Then it is easy to see that Eq. (82) computes functions f for which the outputs $\mathbf{O}$ are repeating themselves with period m (one just has to realize that when expressed in the k basis, the v^{th} digit of R gives the output $\mathbf{O}$ of the formal neuron). As the well-known perceptron computes linearly separable functions, the formal neuron of Eq. (82) computes some kind of periodic functions. The number of such periodic functions is k^m, a very small portion of the total number of functions. As in the case of perceptrons the solution is to introduce hidden units. An example of such a network with hidden units is given in Figure 11.

VI.1 Training a network of formal neurons with hidden units

First of all assume that one allows R to have a maximum number of digits m_{max}, in the k basis. Then suppose that a certain function f of n inputs has to be implemented. One orders the inputs v of the function in increasing order. If the outputs of f form a periodic sequence with period $m \leq m_{max}$ then one formal neuron suffices.

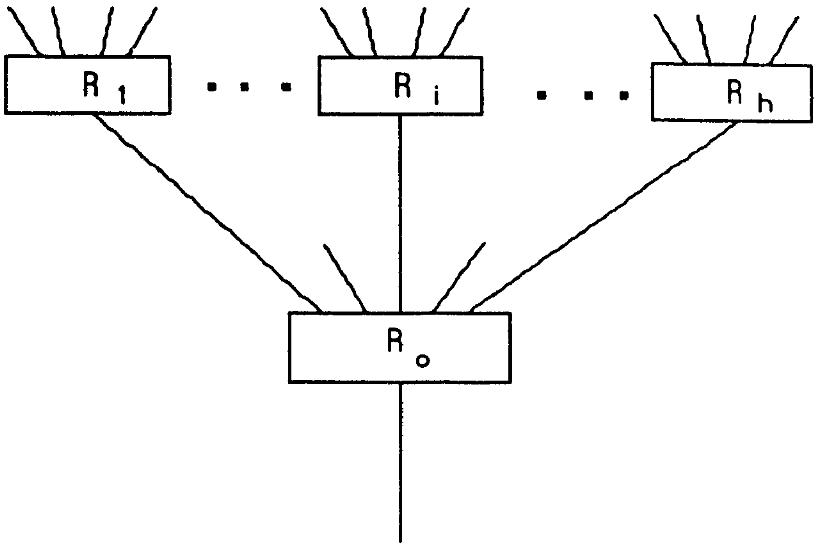


Figure 11. A network of formal neurons with hidden units.

Otherwise one divides the inputs into two equal groups, I_1 and I_2 (if this is not possible one group will contain one input more than the other). For simplicity and without loss of generality assume that the two groups are equal. Then one assigns a formal neuron R_1, to I_1 and a formal neuron R_2 to I_2. Assume O_1 to be the output of R_1 and O_2 to be the output of R_2. One assigns a formal neuron R_3 of two inputs to the outputs O_1 and O_2.

The formal neurons R_1, R_2 and R_3 are unknown and one would like to find the network with the smallest R's that will implement the function f. So one starts with m=2 for all R's and chooses random assignments for R_1, R_2 and R_3. Then the first input v is presented at the inputs. Let v_1 be the part of v that is presented to R_1, and v_2 the part of v that is presented to R_2. If the output O_3 of R_3 is the correct output for the function f then one locks the v_1^{th}mod(m) digit of R_1, the v_2^{th}mod(m) digit of R_2 and the resulting digit of R_3 into their present values and the second input is fed into the network (one has to keep in mind that all the R's must be expressed in the k basis for this computation).

If the output O_3 of R_3 is incorrect then one examines the v_1^{th}mod(m) digit of R_1, the v_2^{th}mod(m) digit of R_2 and the resulting digit of R_3, the ones that are locked retain their values while the others change in all possible combinations trying to satisfy the function f. If a satisfactory solution is found, one locks the values of the v_1^{th}mod(m) digit of R_1, the v_2^{th}mod(m) digit of R_2 and the resulting digit of R_3, for which the solution is found. If no solution is found one resets the v_1^{th}mod(m) digit of R_1, the v_2^{th}mod(m) digit of R_2 and all the digits of R_3 visited during the previous search to random numbers. Then one feeds another input to the network. When all the inputs are exhausted in this way one starts from the beginning and presents the first input to the network again.

If a set of R_1, R_2 and R_3 that will compute f is not found in this search one concludes that the given task is impossible for the present value of m. Then one increases m and repeats the above process with the condition

$$m \leq \min(m_{max}, k^{n_i}) \qquad (83)$$

where n_i is the number of inputs to the formal neuron **i**. If a solution is not found again one concludes that the given task is impossible for the given architecture and m_{max} . Then the number of hidden units is increased and the process starts from the beginning by dividing the number of inputs into three equal groups and so on.

Simulations have shown that the above training will eventually find a solution to the given task but there is no formal proof for the convergence of this algorithm. The solutions found are usually very memory-efficient. For example for the implementation of the parity function of 4 binary inputs with m_{max} set equal to 4 the solution found was $R_1 = R_2 = R_3 = 6$. There was no solution with m=2; the above solution has m=3. Computer times were also short. This is due to the fact that the computation in Eq. (82) is a fast one. It requires the calculation of vmod(m) and the isolation of the v^{th}mod(m) digit of R.

One drawback of this algorithm is that it produces networks that do not generalize. A possible solution to this problem is the use of larger R's than are actually necessary.

VI.2 Concluding Remarks

Formal neurons provide an alternative for memory efficient and faster neural networks. They can be implemented with existing technology because their information content can be stored in addressable memory locations (RAM) very common in conventional computers. The modification of their information content is also easy because it involves only a change in the value of a memory location. This solves the problem of plasticity occurring in conventional neural networks (their synaptic strengths have to be hardwired) and makes possible the construction of general purpose neural nets.

Chapter VII

Conclusion

The main points of the theories and simulations presented in this thesis can be summarized as follows:

(i) The Hopfield model and its variant with low levels of activity are the minimum noise models derived from more general models employing neurons with variable firing strengths.

(ii) The generalized learning algorithm Eq. (60) is a method for enhancing the performance of any feedback neural network with a Hamiltonian in the space of all polynomials of N variables, these variables being discrete or continuous. The enhancement is both in storage capacity and retrieval quality. In addition, application of the generalized learning to non-Hamiltonian systems improves their convergence properties in certain cases.

(iii) A common mechanism is possible for the two distinct processes of learning and growth. The two processes are characterized by different values of a single parameter λ, a small value characterizes the learning process and a large value the growing process. Growth will not disturb the stability of feedback networks if a "parent" network exists in which they can grow. They provide a means for construction of fast and memory-efficient associative memories and pattern recognizers.

(iv) Networks of formal neurons provide an alternative to conventional ones. The use of such neurons allows the implementation of general purpose NN with existing technology. The proposed response function for formal neurons and the training algorithm for multi-layered networks made up of them result in the construction of computationally fast and extremely compact networks for the implementation of integer functions.

References

[1]Amit, D. J. (1989) *Modeling Brain Function* (Cambridge:Cambridge University Press)

[2]Peretto, P. (1989) *The Modeling of Neural Networks* (Les Ulis: Editions de Physique)

[3]Toulouse, G. (1989) *J. Phys. A: Math. Gen.* **22**, 1959-1968

[4]Carlson, N. (1981) *Physiology of Behavior* (Allyn and Bacon, Inc.)

[5]Aleksander, I. (1989) *Neural Computing Architectures* (The MIT Press, Cambridge, Massachusetts)

[6]Luttrell, S. P. (1989) *First IEE International Conference on Artificial Neural Networks* (Great Britain)

[7]Readings from Scientific American (1971) *Perception: Mechanisms and Models* (W. H. Freeman ar. . Company)

[8]Rumelhart, D. E. & McClelland, J. L. (eds.) (1986) *Parallel Distributed Processing,* Vols. 1 and 2 (Cambridge Mass: MIT press)

[9]Freeman, W. J. (1991) *Scientific American Feb 1991*

[10]Gottlieb, D. (1988) *Scientific American Feb 1988*

[11]Haken, H. (1987) *Computational Systems, Natural and Artificial* (Springer, Berlin)

[12]Hopfield, J. J. (1984) *Proc. Natl. Acad. Sci.* **81,** 3088-3092

[13]Cotterill, R. M. J. (1986) *Physica Scripta.* **T13,** 161-168

[14]Hebb, D. O. (1949) *The organization of behavior.* (New York: Wiley)

[15]Brodmann, K. (1909) *Principle of Comparative Localization in the Cerebral Cortex Presented on the basis of Cytoarchitecture* (Leipzig: Barth)

[16]Wasserman, P. D. (1989) *Neural Computing: Theory and Practice.* (New York. Van Nostrand Reinhold)

[17]Rosenblatt, F. (1962) *Principles of neurodynamics* (New York: Spartan Books)

[18]Hopfield, J. J. (1982) *Proc. Natl. Acad. Sci.* **79**, 2554-2558

[19]Denker, J., Schwartz, D., Wittner, B., Solla, S., Hopfield, J., Howard, R., Jackel, L., (1987) *Complex Systems* **1**, 877-922

[20]Grossberg, S. (1987) *The Adaptive Brain,* Vols. 1 and 2. (Amsterdam: North-Holland)

[21]Devaney, R. L. (1989) *Proceeding of Symposia in Applied Mathematics* **39**

[22]Devaney, R. L. (1985) *An Introduction to chaotic dynamical systems,* (Menlo Park: addison-Wesley)

[23]Hansel, D. & Sompolinsky, H. (1990) *Europhys. Lett.* **11**, (7) 687-692

[24]Kohonen, T. (1984) *Self Organisation and Associative Memory* (Heidelberg: Springer Verlag)

[25]Widrow, B. & Hoff, M. E. (1960) *Adaptive switching circuits. IRE WESCON Convention Record,* part 4, pp. 96-104. New York: Institute of Radio Engineers

[26]Widrow, B. (1963) *A statistical theory of Adaptation. Adaptive control systems.* (New York: Pergamon Press)

[27]Alkon, D. L. & Rasmussen, H. (1988) A Spatial Temporal Model of Cell Activation in *Science,* **239**, No.4843, 998-1005

[28]Mézard, M., Parisi, G. and Virasoro, M. A. (1987) *Spin Glass Theory and Beyond.* (World Scientific)

[29]Changeux, J. P. (1985) *Neuronal Man* (New York: Pantheon)

[30]Purves, D. & Lichtman, J. W. (1985) *Principles of Neural Development* (Sunderland Mass.: Sinauer Associates Inc.)

[31]Bienenstock, E., Fogelman, F. & Weisbuch, G. (Eds). (1985) *Disordered Systems and Biological Organization* (Berlin: Springer)

[32]Little, W. A. (1974) *Math. Biosci.* **19**, 101-120

[33]Little, W. A. & Shaw, G. L. (1978) *Math. Biosci.* **39**, 281-289

[34]Cooper, L. N., Liberman, F. & Oja, E. (1979) *Biol. Cybern.* **33**, 9-28

[35]Hinton, G. E. & Anderson, J. A. (Eds). (1981) *Parallel models of associative memory* (Hillsdale, NJ : Erlbaum)

[36]Lyapunov, A. M. (1907) *Le problème général de la stabilite du mouvement* Ann. Fac. Sci. Toulouse **9**, 203-474

[37]Amit, D. J., Gutfreund, H., & Sompolinsky, H. (1985) *Phys. Rev. A* **32**, 1007-1018

[38]Amit, D. J., Gutfreund, H., & Sompolinsky, H. (1985) *Phys. Rev. Lett.* **55**, 1530-1533

[39]Bruce, A. D., Gardner, E. J. & Wallace, D. J. (1987) *J. Phys. A: Math. Gen.* **20**, 2909-2934

[40]Edwards, S. F. & Anderson, P. W. (1975) *J. Phys. F: Metal Phys.* **5**, 965-974

[41]Sherrington, D. & Kirkpatrick, S. (1978) *Phys. Rev. B* **17**, 4384-4403

[42]Parisi, G. (1980) *J. Phys. A: Math. Gen.* **13**, 1101-1112

[43]van Hemmen, J. L. & Zagrebnov, V. A. (1987) *J. Phys. A: Math. Gen.* **20**, 3989-3999

[44]Kanter, I. & Sompolinsky, H. (1987) *Phys. Rev. A* **35**, 380-392

[45]Gardner, E. & Derrida, B. (1988) *J. Phys. A: Math. Gen.* **21**, 271-284

[46]Treves, A. (1990) *Phys. Rev. A.*

[47]McCulloch, W. S. & Pitts, W. (1943) *Bulletin of Math. Biophys.* **5**, 115-133

[48]Amit, D. J., Gutfreund, H. & Sompolinsky, H. (1987) *Ann. Phys.* **173**, 30-67

[49]Amit, D. J., Gutfreund, H. & Sompolinsky, H. (1987) *Phys. Rev. A* **35**, 2293-2303

[50]van Hemmen, J. L., Keller, G. & Kühn, R. (1988) *Europhys. Lett.* **5**, 663-668

[51]Gardner, E. (1987) *J. Phys. A: Math. Gen.* **20**, 3453-3464

[52]Kanter, I. (1988) *Phys. Rev. A* **37**, 2739-2742

[53]Engel, A., Englisch, H. & Schütte, A. *Europhys. Lett.* **8**, 393-397

[54]Bruce, A. D., Canning, A., Forrest, B., Gardner, E., & Wallace, D. J. (1986) *Proc. Conf. on Neural Networks for Computing, Snowbird* (New York: AIP)

[55]Gardner, E., Stroud, N. & Wallace, D. J. (1987)

[56]Pöppel, G. & Krey, U. (1987) *Europhys. Lett.* **4**, 979-985

[57]Parisi, G. (1986) *J. Phys. A: Math. Gen.* L675-L680

[58]Minsky, M. & Papert, S. (1969) *Perceptrons: an Introduction to Computational Geometry* (Cambridge Mass: MIT Press)

[59]Forrest, B. M. (1988) *J. Phys. A: Math. Gen.* **21**, 245-255

[60]Gardner, E. (1989) *J. Phys. A: Math. Gen* **22**, 1969-1974

[61]Peretto, P. (1984) *Biol. Cybern.* **50**, 51-59

[62]Mézard, M., Nadal, J. P. & Krauth, W. (1988)

[63]Mézard, M. & Nadal, J. P. (1989) *J. Phys. A* **22**, 2191-2203

[64]Marchand, M., Golea, M. & Ruján, P. (1990) *Europhys. Lett.* **11**, 487-492

[65]Golea, M. & Marchand, M. (1990) *Europhys. Lett.* **12**, 205-210

[66]Canning, A. & Gardner, E. (1988) *J. Phys. A: Math. Gen.* **21**, 3275-3284

[67]Personnaz, L., Guyon, I. & Dreyfus, G. (1987) *Europhys. Lett.* **4**, 863-867

[68]Eckmiller, R. & Malsburg, C. (Eds). (1989) *Neural Computers* (Heidelberg: Spinger-Verlag)

APPENDIX A

The minimum of Var(n_t)

The first derivative of Var(n_t) with respect to a_ℓ is given by

$$\frac{\partial \,\mathrm{Var}\,(n_t)}{\partial \,a_\ell} = (N-2)\,(p-1)\,\frac{4a_\ell \left[a_\ell^2 (\sum_k a_k^2) \quad -\sum_j a_j^4\right]}{(\sum_k a_k^2)^3} \quad . \qquad \text{(A1)}$$

An extremum occurs when

$$\frac{\partial \,\mathrm{Var}\,(n_t)}{\partial \,a_\ell} =0 \quad , \qquad \ell = 1,\ldots,N$$

and from Eq. (A1) one gets a system of equations with solution

$$a_\ell^2 (\sum_k a_k^2) = \sum_j a_j^4 \Rightarrow a_1=a_2=\ldots=a_N=a$$

In order to prove that this extremum is a minimum one has to examine the Hessian matrix of the second derivatives of Var(n_t). One has

$$\frac{\partial^2 \,\mathrm{Var}\,(n_t)}{\partial \,a_\ell \,\partial \,a_m} = (N-2)\,(p-1)\,\frac{4\delta_{m\ell} \left[3a_\ell^2 (\sum_k a_k^2)^2 \quad -\left(\sum_j a_j^4\right)(\sum_k a_k^2) \quad \right]}{(\sum_k a_k^2)^4} +$$

$$+ (N-2)(p-1)\frac{8a_\ell a_m\left[3\left(\sum_j a_j^4\right)-2a_m^2(\sum_k a_k^2) \quad -2a_\ell^2(\sum_k a_k^2)\right]}{(\sum_k a_k^2)^4}$$

The diagonal elements of the above matrix at the extremum point $a_1=a_2=\ldots=a_N=a$, are equal to

$$\frac{8(N-1)(N-2)(p-1)}{N^3a^2}$$

while the non-diagonal elements at the same point are equal to

$$\frac{8(N-2)(p-1)}{N^3a^2} \quad .$$

As $N\to\infty$ and if $p=cN$ where c is a positive constant independent of N the off-diagonal elements of the Hessian matrix vanish while the diagonal elements become equal to $\frac{8c}{a^2} > 0$.

Thus the Hessian matrix is positive definite and the extremum point $a_1=a_2=\ldots=a_N=a$, is a minimum.

APPENDIX B

The Hamitonian function for the Neural Network using neurons with self-interactions

Take as a starting point the relation

$$-\mathbf{x}\,\mathrm{sign}(\mathbf{x}) \leq 0,$$

which implies that

$$-\sum_j J_{ij}S_j(t)\ \mathrm{sign}\left(\sum_j J_{ij}S_j(t)\right) \leq 0$$

and is equivalent to

$$-\ 2S_i(t+1)\sum_j J_{ij}S_j(t) \leq 0 \quad .$$

But $S_i(t+1)-S_i(t)$ can be either 0 or $2S_i(t+1)$ thus

$$-\left[\ S_i(t+1)-S_i(t)\right]\sum_j J_{ij}S_j(t) \leq 0$$

from which one gets using the fact that only the spin **i** might change at t+1

$$-\left[S_i(t+1)\sum_{j(\neq i)} J_{ij}S_j(t+1)\ -S_i(t)\sum_{j(\neq i)} J_{ij}S_j(t)\ \right] -$$

$$-J_{ii}(S_i(t+1)S_i(t)-1) \leq 0 \quad . \qquad \text{(B1)}$$

Eq. (B1) can be summed over all spins **i** if the synaptic matrix **J** is symmetric, leading to

$$- \sum_{\substack{i,j \\ (i\neq j)}} J_{ij}S_i(t+1)S_j(t+1) + \sum_{\substack{i,j \\ (i\neq j)}} J_{ij}S_i(t)S_j(t) -$$

$$- \sum_{i} J_{ii}(S_i(t+1)S_i(t)-1) \leq 0 \quad . \qquad \text{(B2)}$$

The energy for each pair-interaction in Eq. (B2) has been counted twice. Thus, if $J_{ii} > 0$ for every i the function

$$H = -(1/2) \sum_{\substack{i,j \\ (i\neq j)}} J_{ij}S_iS_j$$

can serve as a Lyapunov function for the network of neurons with self-interactions.

APPENDIX C

Listings of the computer programs

The computer programs used in this thesis are written in C, Pascal and FORTRAN programming languages. The C programs are written using the Think C compiler for the Macintosh SE computer while the FORTRAN and Pascal programs are written using the C.D.C. compilers for a Cyber 835.

The C language programs:

```
/*This is the header part of the program. It contains definitions of
constants and structures*/

#define MAXNEURONS 256
#define MAXBUFFER  15
#define MAXTIME    10
#define MAXLAYERS  5
#define MAXMEMORY  666000
#define NIL        0L

typedef int *net;

struct neuron{
      char type;
      char updating;
      char recurence;
      char feedback;
      int timepar[MAXTIME];
      int thresh;
      int magfield;
      int *responsefunct;
      int *values;
      int *instar[MAXLAYERS];
     };
```

```
/*  The declarations in the Network.c source file. */


#include "Network.h"

extern WindowPtr InpWindow,NetWindow,OutWindow,AnalWindow;
extern int      neurons[MAXLAYERS],bufinp,fon,siz;
extern int      hneurons[MAXLAYERS],vneurons[MAXLAYERS];
extern Rect      drInpRect,drNetRect,drOutRect,drAnalRect;
extern int      buffer[MAXNEURONS][MAXBUFFER];
extern int      store[MAXNEURONS][MAXBUFFER];


struct neuron mynet[MAXNEURONS];

net        thenet;


Rect r= {5,5,50,50};


char cc1[MAXBUFFER];
int coun;



/*  The main program. */

main()
{
   halfinit();
   init();
   initbuffer();
   InitAdress();
   for(;;)
     {events();
      InitCursor();}
}


/*  Initialization of the Macintosh Toolbox. */

init()
{
FlushEvents(everyEvent,0);
InitCursor();
InitFonts();
TEInit();
MaxApplZone();
}
```

```
/*  Buffer initialization. */

initbuffer()
{
register int i,j;
coun=1;
for (i=0;i<MAXNEURONS;++i)
 for(j=0;j<MAXBUFFER;++j)
   buffer[i][j]=-1;
}

/*   Initialization of the Macintosh screen. */

halfinit()
{
InitGraf(&thePort);
InitWindows();
InitMenus();
SetUpFiles();
ClearMenuBar();
fill_stmenus();
SetUpInpWindow();
SetPort(InpWindow);
InitDialogs(NIL);
}

/* Memory initialization. */

InitAdress()
{
long int i,j,c=MAXNEURONS*MAXNEURONS*MAXLAYERS;
MemTest(CompactMem(MAXMEMORY));
thenet= (net) NewPtr(MAXMEMORY);
for (i=0;i<MAXNEURONS;++i)
 {for (j=0;j<MAXLAYERS;++j)
 mynet[i].instar[j]=thenet+MAXNEURONS*(MAXLAYERS*i+MAXNEURONS*j);
 mynet[i].values=thenet+c+MAXTIME*i;}
}


/*  Memory test. */

MemTest(y)
Size y;
{
long int i;
if(y<MAXMEMORY)
  { MoveTo(6,52);
  TextFont(applFont);
  TextFace(bold);
  TextSize(12);
  DrawString(" NOT ENOUGH MEMORY     ");
  for(i=0;i<200000;++i)
      ;
  finish();}
}
```

/*   The event traping subroutine. */

```
events()
{
int ok;
EventRecord event;
HiliteMenu(0);
SystemTask();
ok=GetNextEvent (everyEvent,&event);
if (ok)
  switch (event.what)
  {case mouseDown:
       mouse_click(&event);
       break;
   case mouseUp:
   case keyDown:
   case keyUp:
   case autoKey:
       if (event.modifiers & cmdKey)
           do_menu(MenuKey( (char) (event.message &charCodeMask)));
       else
         {SetPort(InpWindow);
          ShowWindow(InpWindow);
          EraseRect(&r);
          do_input( (char) (event.message &charCodeMask),fon,siz);
          cc1[coun-1]=(char) (event.message &charCodeMask);
          coun = (coun-bufinp) ? coun+1 : 1 ;}
       break;
   case activateEvt:
       break;
  }
}
```

/*   This routine handles the events that are initiated by a mouse device. */

```
mouse_click(event)
EventRecord *event;
```

```
{
 WindowPtr mouse_window;
 int place=FindWindow(event->where,&mouse_window);
 switch(place)
   { case inSysWindow:
         SystemClick(event,mouse_window);
        break;
    case inMenuBar:
         do_menu (MenuSelect(event->where));
        break;
    case inContent:
         if (mouse_window == InpWindow)
          {if (mouse_window !=FrontWindow())
           SelectWindow(InpWindow);
           else
            InvalRect(&InpWindow->portRect);
          }
          if (mouse_window == NetWindow)
          {if (mouse_window !=FrontWindow())
           SelectWindow(NetWindow);
          else
            InvalRect(&NetWindow->portRect);
          }
          if (mouse_window == OutWindow)
          {if (mouse_window !=FrontWindow())
           SelectWindow(OutWindow);
          else
            InvalRect(&OutWindow->portRect);
          }
          if (mouse_window == AnalWindow)
          {if (mouse_window !=FrontWindow())
           SelectWindow(AnalWindow);
          else
            InvalRect(&AnalWindow->portRect);
          }
        break;
    case inDrag:
          if (mouse_window ==InpWindow)
           DragWindow(InpWindow, event->where, &drInpRect);
          if (mouse_window == NetWindow)
           DragWindow(NetWindow,event->where, &drNetRect);
          if (mouse_window == OutWindow)
           DragWindow(OutWindow,event->where, &drOutRect);
          if (mouse_window == AnalWindow)
           DragWindow(AnalWindow,event->where, &drAnalRect);
        break;
    case inGoAway:
if (mouse_window == InpWindow && TrackGoAway(InpWindow, event->where))
           HideWindow(InpWindow);
if(mouse_window ==NetWindow && TrackGoAway(NetWindow, event->where))
        HideWindow(NetWindow);
if(mouse_window ==OutWindow && TrackGoAway(OutWindow, event->where))
        HideWindow(OutWindow);
if(mouse_window==AnalWindow&&TrackGoAway(AnalWindow,event->where))
        HideWindow(AnalWindow);
      break;
   }
 }
```

/* The unsupervised assignment for the synaptic strengths. */

```
hebbJs()
{
register int s,k,j,i,tf;
tf=9000/neurons[0];
for (i=0;i<neurons[0];++i)
 {for (j=i+1;j<neurons[0];++j)
  {s=0;
   for (k=0;k<bufinp;++k)
     s+=store[i][k]*store[j][k];
   *(mynet[i].instar[0]+j)=tf*s;
   *(mynet[j].instar[0]+i)=tf*s;}
  *(mynet[i].instar[0]+i)=0;}
}
```

/* The representation of the network on the screen. */

```
ShowNeurons()
{
Rect rr;
register int i,j,vdim,hdim,rc;
vdim=115/(vneurons[0]-1)+1;
hdim=120/hneurons[0]+1;
  for(i=0;i<hneurons[0];++i)
   {rc=i*vneurons[0];
    for(j=0;j<vneurons[0];++j)
     { rr.top=10+vdim*j;
       rr.left=35+hdim*i;
       rr.bottom=10+vdim*(j+1)-3;
       rr.right=35+hdim*(i+1)-3;
       EraseOval(&rr);
       FrameOval(&rr);
       if(*(mynet[rc+j].values)==1)
         PaintOval(&rr);
    }
   }
 }
```

```
/*  The assignment of a pattern to the network. */

assign(c1)
register int c1;
{
register int i;
for (i=0;i<neurons[0];++i)
 *(mynet[i].values)=buffer[i][c1];
}


/*  The relaxation process. */

process(c2)
int c2;
{
msequencial();
mranchoice();
msequencial();
return(compare(c2));
}

/*   Sequential test of the neurons. */

sequencial()
{
register int s,i,j;
for (i=0;i<neurons[0];++i)
 {s=0;
   for (j=0;j<neurons[0];++j)
       {s+=*(mynet[i].instar[0]+j)*(*(mynet[j].values));}
    *(mynet[i].values) = (s>0) ? 1 : -1;}
}

/*  Random test of the neurons. */

ranchoice()
{
register int s,i,j,k;
for (i=0;i<neurons[0];++i)
{k=ran(neurons[0]);
 s=0;
  for (j=0;j<neurons[0];++j)
    {s+=*(mynet[k].instar[0]+j)*(*(mynet[j].values));}
  *(mynet[k].values) = (s>0) ? 1 : -1;}
}
```

/* Comparison of the stationary output of the network with the desired output of the network during learning */

```
compare(c2)
register int c2;
{
register int s,i;
s=0;
for (i=0;i<neurons[0];++i)
  s+=store[i][c2]-*(mynet[i].values);
if(s==0)
  return(1);
else
  {mdynamic(c2);
   return(0);}
 }
```

/* The generalized learning. */

```
dynamic(c2)
register int c2;
{
register int l=1,i,j;
for (i=0;i<neurons[0];++i)
 {for (j=0;j<neurons[0];++j)
*(mynet[i].instar[0]+j)-=l*(*(mynet[i].values)*(*(mynet[j].values))-
                                                     -store[i][c2]*store[j][c2]);
  *(mynet[i].instar[0]+i)=0;}
 }
```

/* Presentation of the results of the learning process on the screen.*/

```
ShowLearn(x)
int x;
{
int i;
Str255 s;
SetPort(OutWindow);
ShowWindow(OutWindow);
SelectWindow(OutWindow);
MoveTo(6,12);
TextFont(1);
TextFace(bold);
TextSize(12);
DrawString("  LEARNED LETTERS    ");
TextFace(italic);
for(i=0;i<bufinp;++i)
  {MoveTo(5+9*i,30);
  DrawChar(cc1[i]);}
 MoveTo(2,33);
 LineTo(160,33);
 MoveTo(6,47);
 NumToString(x,&s);
 DrawString(s);
 }
```

/* The testing of the network*/

```
neuprocess(c2)
int c2;
{
msequencial();
mranchoice();
msequencial();
return(compare1(c2));
}
```

/* Comparison of the stationary output of the network with the desired output of the network during testing */

```
compare1(c2)
register int c2;
{
register int s,i;
s=0;
for (i=0;i<neurons[0];++i)
  if (store[i][c2]-*(mynet[i].values))
    ++s;
if(s<5)
  return(1);
else
  return(0);
 }
```

/* Presentation of the results of the testing phase on the screen.*/

```
ShowOutput(x)
int x;
{
SetPort(OutWindow);
ShowWindow(OutWindow);
SelectWindow(OutWindow);
MoveTo(6,12);
TextFont(1);
TextFace(bold);
TextSize(12);
DrawString("     OUTPUT      ");
TextFace(italic);
if ( x==1)
  { MoveTo(5,30);
    DrawChar(cc1[0]);}
else
  {MoveTo(2,33);
   DrawString("   NOT RECOGNIZED      ");}
}
```

/* Smooth exiting from the program.*/

```
finish()
{
ExitToShell();
 }
```

/* Different cursor for waiting. */

```
please_wait()
{
CursHandle hCurs;
Cursor waitCursor;
hCurs=GetCursor(watchCursor);
waitCursor=**hCurs;
SetCursor(&waitCursor);
}
```

/* The following three routines control the growth of unary interactions for a growing network. */

```
msequencial()
{
register int s,i,j;
for (i=0;i<neurons[0];++i)
 {s=0;
   for (j=0;j<i;++j)
      {s+=*(mynet[i].instar[0]+j)*(*(mynet[j].values));}
    for (j=i+1;j<neurons[0];++j)
      {s+=*(mynet[i].instar[0]+j)*(*(mynet[j].values));}
    *(mynet[i].values) = ((s+*(mynet[i].instar[0]+i))>0) ? 1 : -1;}
}


mranchoice()
{
register int s,i,j,k;
for (i=0;i<neurons[0];++i)
{k=ran(neurons[0]);
 s=0;
  for (j=0;j<i;++j)
   {s+=*(mynet[k].instar[0]+j)*(*(mynet[j].values));}
  for (j=i+1;j<neurons[0];++j)
   {s+=*(mynet[k].instar[0]+j)*(*(mynet[j].values));}
 *(mynet[k].values) = ((s+*(mynet[i].instar[0]+i))>0) ? 1 : -1;}
}
```

```
mdynamic(c2)
register int c2;
{
register int l=1,i,j;
count6++;
for (i=0;i<neurons[0];++i)
  {for (j=0;j<neurons[0];++j)
    *(mynet[i].instar[0]+j)-=l*(*(mynet[i].values)*(*(mynet[j].values))-
                                                   -store[i][c2]*store[j][c2]);
  }
for (i=0;i<7;++i)
      {if (count6 < 4)
  *(mynet[i].instar[0]+i)-=1000*(*mynet[i].values-store[i][c2]);
   else
  *(mynet[i].instar[0]+i)-=l*(*mynet[i].values-store[i][c2]);}
 }
```

```
/*  The declarations in the Window.c source file. */

#include "Network.h"

WindowPtr InpWindow,OutWindow,NetWindow,AnalWindow;
Rect     drNetRect,drInpRect,drOutRect,drAnalRect;
Rect   winNetBounds = {40,40,215,250};
Rect   winInpBounds = { 40,300,200,450};
Rect   winOutBounds = {230,300,330,450};
Rect   winAnalBounds= {240,40,330,250};

/* This routine defines the Network Window. */

SetUpNetWindow()
{
 drNetRect=screenBits.bounds;
NetWindow=NewWindow(NIL,&winNetBounds,"\pNETWORK",true,noGrowDoc
                                        Proc,-1L,true,0);
 }

/* This routine defines the Output Window. */

SetUpOutWindow()
{
 drOutRect=screenBits.bounds;
OutWindow=NewWindow(NIL,&winOutBounds,"\pOUTPUT",true,noGrowDocPr
                                        oc,-1L,true,0);
 }
 /* This routine defines the Input Window. */

SetUpInpWindow()
{
 drInpRect=screenBits.bounds;
InpWindow=NewWindow(NIL,&winInpBounds,"\pINPUT",true,noGrowDocProc,
-                                       1L,true,0);
 }
/* This routine defines the Analysis Window. */


SetUpAnalWindow()
{
drAnalRect=screenBits.bounds;
AnalWindow=NewWindow(NIL,&winAnalBounds,"\pANALYSIS",true,noGrowD
                                        ocProc,-1L,true,0);
}


/*  The declarations in the Print.c source file. */



#include <PrintMgr.h>
#include "Network.h"

#define topMargin 20
#define leftMargin 20
#define bottomMargin 20
#define tabChar  ((char)'\t')


static      THPrint     hPrint = NIL;
static      int         tabWidth=7;

extern WindowPtr InpWindow;
```

```
char c='a';

/* Checking the printer. */

CheckPrintHandle()
{
        if (hPrint==NIL)
                PrintDefault(hPrint = (TPrint **) NewHandle( sizeof( TPrint )));
}

/* Setting up the page. */

DoPageSetUp()
{
        PrOpen();
        CheckPrintHandle();
        if (PrStlDialog(hPrint)) ;
        PrClose();
}

/* Drawing Text on the page. */

MyDrawText()
{
Point pt;

if ( (c !=tabChar))
  DrawText(&c, 0,1 );
else {
          GetPen(&pt);
          Move((tabWidth-(pt.h-leftMargin)%tabWidth), 0);
         }
}

/* Printing the document. */

PrDoc(hPrint)
THPrint     hPrint;
{
int                 font=1;
int                 size=12;

        Rect                printRect;
        int                 lineBase;
        int                 lineHeight;
        FontInfo            info;
        TPPrPort            printPort;
        printPort = PrOpenDoc( hPrint, 0L, 0L );
        SetPort(printPort);
        TextFont(font);
        TextSize(size);
        printRect = (**hPrint).prInfo.rPage;
        GetFontInfo( &info );
        lineHeight = info.leading+info.ascent+info.descent;
                PrOpenPage( printPort, 0L );
                MoveTo( printRect.left+leftMargin,
                        (lineBase = printRect.top+lineHeight) );
                        while (c != (char)'\r') ;
            MyDrawText();
                        MoveTo( printRect.left+leftMargin, (lineBase += lineHeight));
                PrClosePage( printPort );
        PrCloseDoc( printPort );
}

/* Printing the text in the document. */

PrintText()
{
        TPPrPort    printPort;
        GrafPtr             savePort;
        TPrStatus   prStatus;
        int                 copies;

    PrOpen();
        CheckPrintHandle();
        SetCursor( &arrow );
                        GetPort(&savePort);
                        PrDoc ( hPrint);
                        PrPicFile( hPrint, 0L, 0L, 0L, &prStatus );
                SetPort(savePort);
        PrClose();
}


/*  The declarations in the Menu.c source file. */


#include "Network.h"


extern WindowPtr InpWindow,NetWindow,OutWindow,AnalWindow;
extern int      buffer[MAXNEURONS][MAXBUFFER];

enum { APPLE_ID =1,
      START_ID,
      LAYER_ID,
          TIME_ID,
          STATES_ID,
          CONTINUE_ID,
          NEURON_ID,
          GEOMETRY1_ID,
          GEOMETRY2_ID,
          RECURENCE_ID,
          EXIT_ID,
          FILE_ID,
          INPUT_ID,
          PREPROCESS_ID,
          LEARNIN_ID,
          LEARNOUT_ID,
          NETWORK_ID,
          FONT_ID,
          SIZE_ID,
          DRAW_ID,
          NEURPROC_ID
         };

enum {DESIGN_NET =1,
     TRAIN_NET ,
     TEST_NET,
     QUIT1=5 };

enum {SAVE_SPECS=1,
     GOTOSTART=3,
     QUIT};

enum {LOAD_SPECS=1,
     LOAD_SYNAP,
     SAVE_SYNAP,
     PRINT_SYNAP,
     PRINT_ANALYSIS};


enum {RANDOM=1,
     DESIGN};


enum {RENORM=1,
     NATURAL};

enum {HEBB=1,
     SPINGLASS,
     SYNCHRONUS};

enum {ERROR=1,
     DYNAMIC};
```

```
enum {PROCES_SPECS=1,
    START=3,
    STOP,
    SHOW};

enum {SYSTEMF=0,
   APPLF,
   NEWYORKF,
   GENEVA,
   MONACO};

enum {MORE=1};

int      fon=3,siz=22,neurons[MAXLAYERS]={81,0,0,0,0},bufinp=15;
int      hneurons[MAXLAYERS]={9,0,0,0,0},flag1=1,state=2,
                   flag2[MAXLAYERS]={0,0,0,0,0};
int      vneurons[MAXLAYERS]={9,0,0,0,0},flag0=-1,layers=1,ntime=1;
int      store[MAXNEURONS][MAXBUFFER];

MenuHandle
applemenu,startmenu,neumenu,layermenu,timemenu,geo1menu,exitmenu;
MenuHandle filemenu,inputmenu,preprocessmenu,learninmenu,learnoutmenu,
MenuHandle
networkmenu,statesmenu,fontmenu,sizemenu,drawmenu,neurprocmenu;
MenuHandle continuemenu,recurmenu,geo2menu;
```

/* The start-up menu bar. */

```
fill_stmenus()
 {
 defineMenus();
 InsertMenu(applemenu,0);
 InsertMenu(startmenu,0);
 DrawMenuBar();
 AddResMenu(applemenu, 'DRVR');
AppendMenu  (startmenu,"\pDesign_NET/D;Train_NET/L;Test_NET/T;(-
;Quit/Q");
 }
```

/* The start-up menus. */

```
defineMenus()
{
applemenu=NewMenu(APPLE_ID,"\p\024");
startmenu=NewMenu(START_ID,"\pSTART");
layermenu=NewMenu(LAYER_ID,"\pLAYERS");
timemenu=NewMenu(TIME_ID,"\pTIME");
```

```
statesmenu=NewMenu(STATES_ID,"\pSTATES");
continuemenu=NewMenu(CONTINUE_ID,"\pCONTINUE");
neumenu=NewMenu(NEURON_ID,"\pNEURONS");
geo1menu=NewMenu(GEOMETRY1_ID,"\pINGEOMETRY");
geo2menu=NewMenu(GEOMETRY2_ID,"\pINTERGEOMETRY");
recurmenu=NewMenu(RECURENCE_ID,"\pRECURENCE-FB");
exitmenu=NewMenu(EXIT_ID,"\pEXIT");
filemenu=NewMenu(FILE_ID,"\pFile");
inputmenu=NewMenu(INPUT_ID,"\pInput");
learninmenu=NewMenu(LEARNIN_ID,"\pInLearn");
learnoutmenu=NewMenu(LEARNOUT_ID,"\pOutLearn");
networkmenu=NewMenu(NETWORK_ID,"\pNetwork");
neurprocmenu=NewMenu(NEURPROC_ID,"\pProcess");
}


int count1;

/*  The design menu bar. */

fill_desmenus()
{
DisposeMenu(startmenu);
ClearMenuBar();
InsertMenu(applemenu,0);
InsertMenu(layermenu,0);
InsertMenu(timemenu,0);
InsertMenu(statesmenu,0);
InsertMenu(geo2menu,0);
InsertMenu(continuemenu,0);
count1=0;
DrawMenuBar();
AppendMenu(layermenu,"\p1;2;3;4;5;(-;Other");
AppendMenu(timemenu,"\p1;2;3;4;5;10");
AppendMenu(statesmenu,"\pS=1:2;S=1;S=3:2;(-;Potts2;Potts3;Potts4;(-
                                    ;Continuous");
AppendMenu(geo2menu,"\pMFT;DILUTE");
AppendMenu(continuemenu,"\pMORE");
}

/* The change of menu bar. */

cont_menus()
{
DisposeMenu(layermenu);
DisposeMenu(timemenu);
DisposeMenu(statesmenu);
DisposeMenu(geo2menu);
DisposeMenu(continuemenu);
```

```
ClearMenuBar();
InsertMenu(applemenu,0);
InsertMenu(neumenu,0);
InsertMenu(geo1menu,0);
InsertMenu(recurmenu,0);
InsertMenu(exitmenu,0);
DrawMenuBar();
AppendMenu(neumenu,"\p16;32;64;128;256");
AppendMenu(geo1menu,"\pMFT;Dilute;N-Neighbours");
AppendMenu(recurmenu,"\pRecurrent;Non-Recurrent;(-;1;2;3;4;5");
AppendMenu(exitmenu,"\pSaveSpecs/S;(-;GoToStart/G;Quit/Q");
}
```

/* The training menu bar . */

```
fill_trainmenus()
{
DisposeMenu(startmenu);
DisposeMenu(neumenu);
DisposeMenu(geo1menu);
DisposeMenu(recurmenu);
ClearMenuBar();
InsertMenu(applemenu,0);
InsertMenu(filemenu,0);
InsertMenu(inputmenu,0);
InsertMenu(learninmenu,0);
InsertMenu(learnoutmenu,0);
InsertMenu(networkmenu,0);
InsertMenu(exitmenu,0);
DrawMenuBar();
AppendMenu(filemenu,"\pLoad Specs;Load SynMat;Save SynMat;Print
SynMat;Print Analysis");
AppendMenu(inputmenu,"\pRandom;Design;(-
;1;2;3;4;5;6;7;8;9;10;11;12;13;14;15");
AppendMenu(learninmenu,"\pHebb;SpinGlass;Synchronus");
AppendMenu(learnoutmenu,"\pError;Dynamic");
AppendMenu(networkmenu,"\pProcesSpecs;(-;Start/R;Stop/P;Show");
AppendMenu(exitmenu,"\pSaveSpecs/S;(-;GoToStart/G;Quit/Q");
}
```

/* The design menus. */

```
filldesignmenus()
{
ClearMenuBar();
fontmenu=NewMenu(FONT_ID,"\pFonts");
sizemenu=NewMenu(SIZE_ID,"\pSize");
drawmenu=NewMenu(DRAW_ID,"\pDraw");
preprocessmenu=NewMenu(PREPROCESS_ID,"\pPreprocess");
InsertMenu(fontmenu,0);
InsertMenu(sizemenu,0);
InsertMenu(drawmenu,0);
InsertMenu(preprocessmenu,0);
DrawMenuBar();
AppendMenu (fontmenu, "\pSystem;Apple;newYork;geneva;monaco");
AppendMenu (sizemenu, "\p9;10;12;14;16;18;20;22;24;34;44");
AppendMenu(preprocessmenu,"\pRenorm;Natural");
}
```

/* **The training menus.** */

```
back_trainmenus()
{
DisposeMenu(fontmenu);
DisposeMenu(sizemenu);
DisposeMenu(drawmenu);
DisposeMenu(preprocessmenu);
ClearMenuBar();
InsertMenu(applemenu,0);
InsertMenu(filemenu,0);
InsertMenu(inputmenu,0);
InsertMenu(learninmenu,0);
InsertMenu(learnoutmenu,0);
InsertMenu(networkmenu,0);
InsertMenu(exitmenu,0);
DrawMenuBar();
}
```

/* **The testing menu bar and menus.** */

```
fill_testmenus()
{
DisposeMenu(learninmenu);
DisposeMenu(learnoutmenu);
DisposeMenu(networkmenu);
ClearMenuBar();
InsertMenu(applemenu,0);
InsertMenu(filemenu,0);
InsertMenu(inputmenu,0);
InsertMenu(neurprocmenu,0);
InsertMenu(exitmenu,0);
DrawMenuBar();
AppendMenu(filemenu,"\pLoad Specs;Load SynMat;Save SynMat;Print
                SynMat;Print Analysis");
```

```
AppendMenu(inputmenu,"\pRandom;Design;(-;1;2;3;4;5;10;15");
AppendMenu(neurprocmenu,"\pStart_Process");
AppendMenu(exitmenu,"\pSaveSpecs/S;(-;GoToStart/G;Quit/Q");
}


/* The following two routines are adjusting the menus. */

Adjust_conm()
{
if (count1==layers)
  enable(exitmenu,3,1);
else
   enable(exitmenu,3,0);
}


static
enable(menu,item,ok)
Handle menu;
{
 if(ok)
    EnableItem(menu,item);
 else
    DisableItem(menu,item);
 }


/* This routine controls the actions initiated by a menu choice from
a specific menu bar. */


 do_menu (command)
  long command;
{
  int menu_id=HiWord(command);
  int item = LoWord (command);
  Str255 item_name;
  GrafPtr savePort;
  WindowPeek frontWindow;
  register int i,j;

  switch (menu_id)
   {case APPLE_ID:
        GetPort(&savePort);
        GetItem(applemenu,item,item_name);
        OpenDeskAcc(item_name);
        SetPort(savePort);
```

```
      break;
   case START_ID:
      switch(item)
       {case DESIGN_NET:
           fill_desmenus();break;
        case TRAIN_NET:
           fill_trainmenus();SetUpNetWindow();SetUpOutWindow();
           SetUpAnalWindow();break;
        case TEST_NET:
           fill_testmenus();SetUpNetWindow();SetUpOutWindow();
           SetUpAnalWindow();break;
        case QUIT1:
           finish();break;}
        break;
  case NEURON_ID:
      switch(item)
        {case 1:
          neurons[count1]=16;hneurons[count1]=vneurons[count1]=4;
          count1++; Adjust_conm(count1);
          break;
        case  2:
          neurons[count1]=32;hneurons[count1]=4;vneurons[count1]=8;
          count1++; Adjust_conm(count1);
          break;
        case 3:
          neurons[count1]=64;hneurons[count1]=vneurons[count1]=8;
          count1++; Adjust_conm(count1);
          break;
        case 4:
          neurons[count1]=128;hneurons[count1]=8;vneurons[count1]=16;
          count1++; Adjust_conm(count1);
          break;
        case 5:
          neurons[count1]=256;hneurons[count1]=vneurons[count1]=16;
          count1++; Adjust_conm(count1);
          break;}
      break;
  case LAYER_ID:
      switch(item)
       {default:
          layers=item;break;}
      break;
  case TIME_ID:
      switch(item)
       { case 6:
          ntime=10;break;
         default:
          ntime=item;break;}
      break;
```

```
case STATES_ID:
    switch(item)
     {case 1:
      case 2:
      case 3:
        flag1=1;state=item+1;break;
      default:
        flag1=0;state=item-3;break;}
    break;
case CONTINUE_ID:
    switch(item)
     {case MORE:
       cont_menus();
       count1=0;
       Adjust_conm(count1);
       break;}
     break;
case GEOMETRY1_ID:
    flag0=item-2;
    break;
case EXIT_ID:
    switch(item)
     {case SAVE_SPECS:
          savespecs();break;
      case GOTOSTART:
          halfinit();
          break;
      case QUIT:
          finish();break;}
   break;
case FILE_ID:
    switch(item)
     {case LOAD_SPECS:
          loadspecs();break;
      case LOAD_SYNAP:
         break;
      case SAVE_SYNAP:
         break;
      case PRINT_SYNAP:
          SetPort(InpWindow);
          DoPageSetUp();
          PrintText();break;
      case PRINT_ANALYSIS:
         break;}
   break;
case INPUT_ID:
    SetPort(InpWindow);
    ShowWindow(InpWindow);
    SelectWindow(InpWindow);
```

```
      HideWindow(OutWindow);
      HideWindow(AnalWindow);
     switch (item)
      {case RANDOM:
           random_input();break;
       case DESIGN:
           filldesignmenus();break;
       default:
           bufinp=item-3;break;}
    break;
case  PREPROCESS_ID:
     switch(item)
      {case RENORM:
          scan_input(siz);
          break;
       case NATURAL:
          break;}
    break;
case LEARNIN_ID:
     switch(item)
       {case HEBB:
           corelmatrix();
           {for (j=0;j<bufinp;++j)
             for (i=0;i<neurons[0];++i)
              store[i][j]=buffer[i][j];
           }
           hebbJs();
           break;
        case SPINGLASS:
           break;
        case SYNCHRONUS:
           break;}
    break;
case LEARNOUT_ID:
     switch(item)
       {case ERROR:
           break;
        case DYNAMIC:
           break;}
    break;
case NETWORK_ID:
     switch(item)
     {case PROCES_SPECS:
        fillprocessmenu();break;
      case START:
         ShowLearn(Select());
        break;}
     break;
case NEURPROC_ID:
        switch(item)
         {case 1:
           SetPort(NetWindow);
           SelectWindow(NetWindow);
           assign(0);
           ShowNeurons();
           ShowOutput( neuprocess(0) );
          break;}
        break;
  case FONT_ID:
        switch (item)
        {fon=item;}
       break;
  case SIZE_ID:
      switch(item)
      {case 1:
        siz=9;
        break;
       case 10:
         siz=34;
         break;
       case 11:
         siz=44;
         break;
       default:
         siz=2*(3+item);
         break;}
      break;
  }
}
```

/* The following routine feeds the inputs to the network. */

```
Select()
{
int count1,count2,count3,count4=0,count5;
SetPort(NetWindow);
SelectWindow(NetWindow)·
for (count5=1;count5<=1;++count5)
{count3=0;
 while(count3-bufinp)
  {count2=ran(bufinp);
   count3=0;
   ++count4;
   events();
   for (count1=0;count1<bufinp;++count1)
      {assign((count1+count2)%bufinp);
```

```
        ShowNeurons();
        count3+=process((count1+count2)%bufinp);
        ShowNeurons();}
  }
 random_input();
 corelmatrix();
 hebbJs();
 }
return(count4);
}
```

/* The declarations in the Input.c source file. */

```
 #include "Network.h"

extern int flag1,state,neurons[MAXLAYERS],bufinp;
extern int hneurons[MAXLAYERS],vneurons[MAXLAYERS];
extern WindowPtr InpWindow,AnalWindow;

int buffer[MAXNEURONS][MAXBUFFER];
int count=1;
```

/* This routine creates random patterns as input. */

```
random_input()
{
:egister int i,j;
randSeed=Time;
if (flag1)
{if (state-3)
 {for (j=0;j<bufinp;++j)
   for(i=0;i<neurons[0];++i)
    buffer[i][j]=2*ran(state)-state+1;
 }
 else
  {for (j=0;j<bufinp;++j)
    for(i=0;i<neurons[0];++i)
     buffer[i][j]=Random()%state;
  }
 }
else
{for (j=0;j<bufinp;++j)
  for(i=0;i<neurons[0];++i)
    buffer[i][j]=ran(state);
 }
```

```
}

/*  This routine puts letters on the retina. */

do_input(c,f,s)
char c;
int f,s;
{
MoveTo(15,45);
TextFont(f);
TextFace(bold);
TextSize(s);
DrawChar(c);
}

/* This routine scans the letters on the retina. */

scan_input(s)
int s;
{
register int i,j;
int  to,lef,botto,righ;
Rect inrect;
randSeed=Time;
to=46-s;
lef=15;
botto=47;
righ=15+s-1;
SetPort(InpWindow);
please_wait();
for(i=lef;;++i)
  for(j=botto;j>=to;--j)
    if (GetPixel(i,j))
      { inrect.left=i;
         goto for2;}
for2:
for (i=righ;;--i)
   for(j=botto;j>=to;--j)
     if (GetPixel(i,j))
       { inrect.right=i+1;
          goto for3;}
for3:
for (i=to;;++i)
    for(j=lef;j<=righ;++j)
      if (GetPixel(j,i))
        {inrect.top=i;
          goto for4;}
for4:
for (i=botto;;--i)
```

```
    for(j=lef;j<=righ;++j)
      if (GetPixel(j,i))
        {inrect.bottom=i+1;
         goto act;}
act:
preproccess(inrect);
if(count-bufinp)
   ++count;
else
  {back_trainmenus();
   count=1;}
}
```

/* The following two routines renormalize the input in order to present it to the network in a suitable format. */

```
preproccess(inrect)
Rect inrect;
{
register int i,j,a,b,rc;
Rect intrect;
double a1,b1;
a1=(inrect.right-inrect.left-1)/hneurons[0];
a=INT(a1)+1;
b1=(inrect.bottom-inrect.top-1)/vneurons[0];
b=INT(b1)+1;
for(i=0;i<hneurons[0];++i)
  { rc=i*vneurons[0];
    for(j=0;j<vneurons[0];++j)
      {intrect.left=inrect.left+i*a-1;
        intrect.top=inrect.top+j*b-1;
        intrect.right=intrect.left+a;
       intrect.bottom=intrect.top+b;
      if (Valrect(intrect)>=1)
        buffer[rc+j][count-1]=1;
      else
        buffer[rc+j][count-1]=-1;}
  }
}


Valrect(intrect)
Rect intrect;
{
register int s=0,k,j;
 for(k=intrect.left;k<intrect.right;++k)
```

```
  for(j=intrect.top;j<intrect.bottom;++j)
    s+=GetPixel(k,j);
 return s;
 }
```

/* The calculation of the average correlation of the input patterns. * /

```
corelmatrix()
{
int q[15][15];
register int i,j,s,l,q1;
Str255 str;
q1=0;
for(i=0;i<bufinp;++i)
  for(j=i+1;j<bufinp;++j)
   {s=0;
    for(l=0;l<neurons[0];++l)
      s+=buffer[l][i]*buffer[l][j];
   q[i][j]=s;
   q[j][i]=s;
   q1=q1+s;}
SetPort(AnalWindow);
ShowWindow(AnalWindow);
SelectWindow(AnalWindow);
MoveTo(6,25);
TextFont(1);
TextFace(bold);
TextSize(12);
NumToString(q1,&str);
DrawString(str);
}
```

/* A function that returns the integer part of its argument. */

```
INT(x)
double x;
{
 int i;
 i=x;
 return (i);
 }
```

/* A function that returns a positive random integer less than its argument. */

```
ran(x)
int x;
```

```
{
int k;
k=Random()%x;
if(k<0) k=-k;
return(k);
}
```

/* A function that returns the sign of an integer. */

```
sign(x)
int x;
{
if (x>=0)
  return(1);
else
  return(-1);
}
```

/* The declarations in the Files.c source file. */

```
#include "Network.h"

extern int neurons[MAXLAYERS];
extern WindowPtr OutWindow;

Str255 theFileName;
static int theVRefNum;
```

/* Setting up a file.*/

```
SetUpFiles()
{
pStrCopy("\p",theFileName);
theVRefNum=0;
}
```

/* Saving a network's specifications. */

```
savespecs()
{
int vRef;
Str255 fn;
if(theFileName[0]==0)
  {fn[0]=0;
   if(SaveAs(fn,&vRef))
```

```
      {pStrCopy(fn,theFileName);
      theVRefNum=vRef;}
    }
  else
    SaveFile(theFileName,theVRefNum);
  }


static Point SFPwhere = {106,104};
static SFReply reply;
static Point SFGwhere={90,82};
long size=sizeof(neurons[0]);
```

/* Saving as a different file the network's specifications. */

```
SaveAs(fn,vRef)
Str255 fn;
int   *vRef;
{
 int refNum;
 if (NewFile(fn,vRef))
   if (CreateFile(fn,vRef,&refNum))
     {WriteFile(refNum,&neurons,size);
      FSClose(refNum);
      return(1);}
    else
     {FileError("\pError creating file ",fn);}
 return(0);
 }
```

/* Saving the file with the network's specifications. */

```
SaveFile(fn,vrn)
Str255 fn;
int vrn;
{
int refNum;
if (FSOpen(fn,vrn,&refNum)==noErr)
   {WriteFile(refNum,&neurons,size);
    FSClose(refNum);
    return(1);}
else FileError("\pError opening file ",fn);
return(0);
}
```

/* Opening a new file. */

```
NewFile(fn,vRef)
Str255 fn;
int *vRef;
{
SFPutFile(SFPwhere,"\p",fn,0L,&reply);
if(reply.good)
  {pStrCopy(reply.fName,fn);
   *vRef=reply.vRefNum;
   return(1);
  }
 else
 return(0);
}
```

/* Creating a new file. */

```
CreateFile(fn,vRef,theRef)
Str255 fn;
int *vRef;
int *theRef;
{
 int io;
 io=Create(fn,*vRef,'CEM8','TEXT');
 if ((io==noErr) || (io==dupFNErr)) io=FSOpen(fn,*vRef,theRef);
 return((io==noErr) || (io=dupFNErr));
 }
```

/* Writing in a file. */

```
WriteFile (refNum,p,num)
int  refNum;
int *p;
long num;
{
int io;
SetFPos(refNum,fsFromStart,0);
io=FSWrite (refNum,&num,p);
}
```

/* Reading a network's specifications. */

```
loadspe   ()
 {
 int vRef,refNum;
 Str255 fn,ns;
 if(OldFile(fn,&vRef))
```

```
    if(FSOpen(fn,vRef,&refNum)==noErr)
      {if(ReadFile(refNum,&neurons))
        {pStrCopy(fn,theFileName);
         theVRefNum=vRef;}
       if(FSClose(refNum)==noErr);
       SetPort(OutWindow);
       MoveTo (20,20);
       NumToString(-neurons[0],&ns);
       DrawString(ns);
      }
    else FileError("\pError opening",fn);
}
```

/* Opening an old file. */

```
OldFile(fn,vRef)
Str255 fn;
int *vRef;
{
SFTypeList myTypes;
myTypes[0]='TEXT';
SFGetFile(SFGwhere,"\p",0L,1,myTypes,0L,&reply);
if(reply.good)
{pStrCopy(reply.fName,fn);
*vRef=reply.vRefNum;
return(1);}
else return(0);
}
```

/* Reading from an old file. */

```
ReadFile(refNum,pc)
int refNum;
int *pc;
{
char Buffer[256];
long count;
int io;
do
{count=256;
 SetFPos(refNum,fsFromStart,0);
 io=FSRead(refNum,&count,&Buffer);
 *pc=Buffer[1];
 }
while (io==noErr);
return(io==eofErr);
}
```

/* Copy one string on another. */

```
pStrCopy(p1,p2)
register char *p1,*p2;
{
register int len;
len=*p2++=*p1++;
while(--len>=0) *p2++=*p1++;
}
```

/* Error alert when opening a file. */

```
FileError(s,f)
Str255 s,f;
{
ParamText(s,f,"\p","\p");
Alert(2,0L);
}
```

/* THE NEXT PART IS A SET OF TRANSPORTABLE ROUTINES. THESE ROUTINES ARE MACHINE INDEPENDENT AND CAN RUN ON A MAINFRAME. */

/* The declarations in the Transinput.c source file. */

```
#include "Network.h"
#include "stdio.h"

extern int flag1,state,neurons[MAXLAYERS],bufinp;

int buffer[MAXNEURONS][MAXBUFFER];
int count=1;
```

/* This routine creates random patterns as input. */

```
random_input()
{
register int i,j;
randSeed=Time;
if (flag1)
{if (state-3)
 {for (j=0;j<bufinp;++j)
   for(i=0;i<neurons[0];++i)
```

```
    buffer[i][j]=2*ran(state)-state+1;
  }
 else
  {for (j=0;j<bufinp;++j)
    for(i=0;i<neurons[0];++i)
     buffer[i][j]=Random()%state;
  }
 }
else
{for (j=0;j<bufinp;++j)
  for(i=0;i<neurons[0];++i)
    buffer[i][j]=ran(state);
 }
}
```

/* The calculation of the average correlation of the input patterns. * /

```
corelmatrix()
{
int q[15][15];
register int i,j,s,l,q1;
q1=0;
for(i=0;i<bufinp;++i)
  for(j=i+1;j<bufinp;++j)
    {s=0;
     for(l=0;l<neurons[0];++l)
       s+=buffer[l][i]*buffer[l][j];
    q[i][j]=s;
    q[j][i]=s;
    q1=q1+s;}
 printf("\n The correlation is %d",s);
}
```

/* A function that returns the integer part of its argument. */

```
INT(x)
double x;
{
 int i;
 i=x;
 return (i);
 }
```

/* A function that returns a positive random integer less than its argument. */

```
ran(x)
int x;
{
int k;
k=Random()%x;
if(k<0) k=-k;
return(k);
}
```

/* A function that returns the sign of an integer. */

```
sign(x)
int x;
{
if (x>=0)
  return(1);
else
  return(-1);
}
```

/* The declarations in the TransSpesif.c source file. */

```
#include "Network.h"
#include "stdio.h"


int      fon=3,siz=22,neurons[MAXLAYERS]={81},bufinp=15;
int      flag1=1,state=2,flag2[MAXLAYERS]={0};
int      flag0=-1,layers=1,ntime=1,count1,count6;
```

/* A routine that reads the specifications of a network from the keyboard. */

```
readvalues()
{
int first,choose1,choo2,choo3;
FILE *fopen(), *fp;
printf ("Sellect Action: DESIGN_NET =1\n");
printf ("                TRAIN_NET  =2\n");
printf ("                TEST_NET   =3\n");
printf ("                QUIT1      =5\n");
scanf("%d",&first);
```

```
if (first != 5)
 {switch(first)
  {case 1:
    printf("LAYERS=");
    scanf("%d",&layers);
    printf("\nTIME=");
    scanf("%d",&ntime);
     printf("\nSPIN MODEL (2 for spin 1/2)");
    printf("\n         (3 for spin  1 )");
    printf("\n          (4 for spin 3/2)");
    printf("\n         (-2 for Potts 2)");
    printf("\n         (-3 for Potts 3)\n");
    scanf("%d",&state);
     {flag1 = (state<0) ? 0 : 1;
      state=abs(state);}
    printf("\nTYPE OF GEOMETRY BETWEEN THE LAYERS");
    printf("\n         ( -1 for totally connected layers)");
     printf("\n         ( 0 for dilute connections among the layers)\n");
    scanf("%d",&flag0);
    count1=0;
    while (count1 < layers)
      {printf("\nNO OF NEURONS IN %2d LAYER =",layers);
       scanf("%d",&neurons[count1]);
       printf("\nGEOMETRY IN %2d LAYER",layers);
       printf("\n       ( 0 for totally connected neurons)");
        printf("\n       ( 1 for  diluted connections among neurons)\n");
       scanf("%d",&flag2[count1]);
       count1++;}
     printf("\nSave = 1");
     printf("\nGoToStart = 2");
     printf("\nQuit = 3\n");
    scanf("%d",&choose1);
    if (choose1==1)
        {fp=fopen("SPECS","a");
          fprintf(fp,"%4d %4d %4d %4d %4d\n",layers,ntime,flag1,state,flag0);
         count1=0;
         while (count1 < layers)
          {fprintf(fp,"%4d %4d\n",neurons[count1],flag2[count1]);
          count1++;}
         fclose(fp);
         readvalues();}
    else if (choose1== 2)
          readvalues();
    else
          finish();
  case 2:
       printf("\nLoad Specs =1");
       printf("\nLoad SynMatrix =2");
       printf("\nSave SynMatrix =3");
```

```
        printf("\nInput =4\n");
        scanf("%d",&choo2);
        switch (choo2)
         { case 1:
             break;
           case 2:
             break;
           case 3:
             break;
           case 4:
             printf("\nBUFFER INPUT=");
             scanf("%d",&bufinp);
             printf("%4d\n", flag0);
             random_input();
             break;}
        printf("\nSelect Learning Procedure");
        printf("\n        Hebb=1");
        printf("\n        Spinglass=2");
        printf("\n        Synchronous=3\n");
        scanf("%d",&choo3);
        switch (choo3)
         { case 1:
             corelmatrix();
             hebbJs();
             break;
           case 2:
             break;
           case 3:
             break;}
       printf("\nTHE NETWORK IS PROCESSING NOW");
       printf("\nThe number of iterations was %4d \n",Select());
       readvalues();
   case 3:
        break;
     }
   }
finish();
}
```

/* **The following routine feeds the inputs to the network.** */

```
Select()
{
int count1,count2,count3,count4=0,count5;
for (count5=1;count5<=20;++count5)
{count3=0;
 while(count3-bufinp)
  {count2=ran(bufinp);
  count3=0;
  ++count4;
  for (count1=0;count1<bufinp;++count1)
      {assign((count1+count2)%bufinp);
       count3+=process((count1+count2)%bufinp);
     }
  }
 random_input();
 count6=0;
 corelmatrix();
 hebbJs();
 }
return(count4);
}
```

/* The declarations in the TransNetwork.c source file. */

```
#include "Network.h"
#include "stdio.h"
#include "MemoryMgr.h"


extern int      neurons[MAXLAYERS],bufinp,fon,siz,count6;
extern int      buffer[MAXNEURONS][MAXBUFFER];


struct neuron mynet[MAXNEURONS];

net        thenet;
```

/* The main routine in the transportable program. */

```
main()
{
initbuffer();
InitAdress();
readvalues();
}
```

/* The buffer initialization in the transportable program. */

```
initbuffer()
{
register int i,j;
for (i=0;i<MAXNEURONS;++i)
  for(j=0;j<MAXBUFFER;++j)
    buffer[i][j]=-1;
}
```

/* The memory initialization in the transportable program. */

```
InitAdress()
{
long int i,j,c=MAXNEURONS*MAXNEURONS*MAXLAYERS;
char *malloc();
Str255 s;
thenet= (net) malloc(MAXMEMORY);
for (i=0;i<MAXNEURONS;++i)
 {for (j=0;j<MAXLAYERS;++j)
mynet[i].instar[j]=thenet+50000+MAXNEURONS*(MAXLAYERS*i+MAXNEURO
                                                       NS*j);
  mynet[i].values=thenet+c+MAXTIME*i;}
}
```

/* The unsupervised assignment for the synaptic strengths. */

```
hebbJs()
{
register int s,k,j,i,tf;
tf=9000/neurons[0];
for (i=0;i<neurons[0];++i)
  {for (j=i+1;j<neurons[0];++j)
   {s=0;
    for (k=0;k<bufinp;++k)
      s+=buffer[i][k]*buffer[j][k];
    *(mynet[i].instar[0]+j)=tf*s;
    *(mynet[j].instar[0]+i)=tf*s;}
  *(mynet[i].instar[0]+i)=0;}
}
```

/* The assignment of a pattern to the network. */

```
assign(c1)
register int c1;
{
register int i;
for (i=0;i<neurons[0];++i)
```

```
 *(mynet[i].values)=buffer[i][c1];
}

/*  The relaxation process. */

process(c2)
int c2;
{
sequencial();
ranchoice();
sequencial();
return(compare(c2));
}
```

/* The following two routines perform sequential tests on the neurons in a network with or without self interactions. * /

```
sequencial()
{
register int s,i,j;
for (i=0;i<neurons[0];++i)
 {s=0;
  for (j=0;j<i;++j)
     {s+=*(mynet[i].instar[0]+j)*(*(mynet[j].values));}
   for (j=i+1;j<neurons[0];++j)
     {s+=*(mynet[i].instar[0]+j)*(*(mynet[j].values));}
   *(mynet[i].values) = ((s+*(mynet[i].instar[0]+i))>0) ? 1 : -1;}
}

Wsequencial()
{
register int s,i,j;
for (i=0;i<neurons[0];++i)
 {s=0;
   for (j=0;j<neurons[0];++j)
     {s+=*(mynet[i].instar[0]+j)*(*(mynet[j].values));}
   *(mynet[i].values) = (s>0) ? 1 : -1;}
}
```

/* The following two routines perform random tests on the neurons in a network with or without self interactions. */

```
ranchoice()
{
register int s,i,j,k;
for (i=0;i<neurons[0];++i)
{k=ran(neurons[0]);
 s=0;
  for (j=0;j<i;++j)
    {s+=*(mynet[k].instar[0]+j)*(*(mynet[j].values));}
   for (j=i+1;j<neurons[0];++j)
    {s+=*(mynet[k].instar[0]+j)*(*(mynet[j].values));}
  *(mynet[k].values) = ((s+*(mynet[i].instar[0]+i))>0) ? 1 : -1;}
}


Wranchoice()
{
register int s,i,j,k;
for (i=0;i<neurons[0];++i)
{k=ran(neurons[0]);
 s=0;
  for (j=0;j<neurons[0];++j)
   {s+=*(mynet[k].instar[0]+j)*(*(mynet[j].values));}
  *(mynet[k].values) = (s>0) ? 1 : -1;}
}
```

/* This routine compares the output of the network with the desired output of the network during training. */

```
compare(c2)
register int c2;
{
register int s,i;
s=0;
for (i=0;i<neurons[0];++i)
  s+=buffer[i][c2]-*(mynet[i].values);
if(s==0)
  return(1);
else
  {dynamic(c2);
   return(0);}
}
```

/* The following two routines perform the generalized learning and growth algorithms. */

```
Gdynamic(c2)
register int c2;
{
register int l=1,i,j;
count6++;
for (i=0;i<neurons[0];++i)
  {for (j=0;j<neurons[0];++j)
```

```
    *(mynet[i].instar[0]+j)-=l*(*(mynet[i].values)*(*(mynet[j].values))-
buffer[i][c2]*buffer[j][c2]);
  }
for (i=0;i<7;++i)
       {if (count6 < 4)
  *(mynet[i].instar[0]+i)-=1000*(*mynet[i].values-buffer[i][c2]);
    else
  *(mynet[i].instar[0]+i)-=l*(*mynet[i].values-buffer[i][c2]);}
 }

dynamic(c2)
register int c2;
{
register int l=1,i,j,tf;
tf=9000/neurons[0];
for (i=0;i<neurons[0];++i)
  {for (j=0;j<neurons[0];++j)
    *(mynet[i].instar[0]+j)-=l*(*(mynet[i].values)*(*(mynet[j].values))-
buffer[i][c2]*buffer[j][c2]);
   *(mynet[i].instar[0]+i)=3*tf*bufinp;}
 }
```

/* **This routine performs the testing of the network.** */

```
neuprocess(c2)
int c2;
{
sequencial();
ranchoice();
sequencial();
return(compare1(c2));
}
```

/* **This routine compares the output of the network with the desired output of the network during testing.** */

```
compare1(c2)
register int c2;
{
register int s,i;
s=0;
for (i=0;i<neurons[0];++i)
  if (buffer[i][c2]-*(mynet[i].values))
    ++s;
if(s<5)
   return(1);
else
   return(0);
 }
```

The FORTRAN programs

/* This program decodes the output of a digitizer for a page of handwritten and typed letters. */

```
        PROGRAM DECOD(INPUT,OUTPUT)
   CHARACTER C(120,50)*1,C1(120,50)*2,C2(77)*1,A*2,A1*1
   INTEGER K(120,50),K1(12,50,33)
   OPEN (UNIT=2,FILE='SC10',STATUS='OLD')
   REWIND (2)
8  FORMAT (120A1)
   READ (2,8) ((C(I,J),I=1,120),J=1,50)
   CLOSE (UNIT=2)
   DO 20 J=1,50
   DO 20 I=1,120
   C1(I,J)='??'
20 CONTINUE
   DO 10 J=1,50
   I1=0
   DO 10 I=1,120
   I1=I1+1
   A1=C(I1,J)
   IF (I1-120) 12,12,10
12 IF (A1.EQ.'?') GO TO 1
   C1(I,J)=C(I1,J)//'?'
   GO TO 10
1  C1(I,J)=C(I1+1,J)//C(I1+2,J)
   I1=I1+2
10 CONTINUE
11 C2(1)='A'
   C2(2)='B'
   C2(3)='C'
   C2(4)='D'
   C2(5)='E'
   C2(6)='F'
   C2(7)='G'
   C2(8)='?'
   C2(9)='?'
   C2(10)='H'
   C2(11)='I'
   C2(12)='J'
   C2(13)='K'
   C2(14)='L'
   C2(15)='M'
   C2(16)='N'
   C2(17)='O'
   C2(18)='?'
```

```
C2(19)='?'
C2(20)='P'
C2(21)='Q'
C2(22)='R'
C2(23)='S'
C2(24)='T'
C2(25)='U'
C2(26)='V'
C2(27)='W'
C2(28)='?'
C2(29)='?'
C2(30)='X'
C2(31)='Y'
C2(32)='Z'
C2(33)='['
C2(34)='\'
C2(35)=']'
C2(36)='
C2(37)='_'
C2(38)='?'
C2(39)='?'
C2(40)=' '
C2(41)='!'
C2(42)='"'
C2(43)='#'
C2(44)='$'
C2(45)='%'
C2(46)='&'
C2(47)='
C2(48)='?'
C2(49)='?'
C2(50)='('
C2(51)=')'
C2(52)='*'
C2(53)='+'
C2(54)=','
C2(55)='-'
C2(56)='.'
C2(57)='/'
C2(58)='?'
C2(59)='?'
C2(60)='0'
C2(61)='1'
C2(62)='2'
C2(63)='3'
C2(64)='4'
C2(65)='5'
C2(66)='6'
C2(67)='7'
```

```
      C2(68)='?'
      C2(69)='?'
      C2(70)='8'
      C2(71)='9'
      C2(72)=':'
      C2(73)=';'
      C2(74)='<'
      C2(75)='='
      C2(76)='>'
      C2(77)='?'
      DO 30 J=1,50
      DO 30 I=1,120
      A=C1(I,J)
      IF (A.EQ.'??') THEN
       K(I,J)=0
       GO TO 30
      ELSE
      IF (A(2:2).EQ.'?') THEN
       DO 40 L=1,76
       IF (A(1:1).EQ.C2(L)) THEN
         K(I,J)=(L/10)*8+(L-(L/10)*10)
       GO TO 30
       ENDIF
40       CONTINUE
      ELSE
       DO 50 L=1,76
       DO 50 N=1,76
      IF (A(1:1).EQ.C2(L) .AND. A(2:2).EQ.C2(N)) THEN
         K(I,J)=(L/10)*512+(L-(L/10)*10)*64+(N/10)*8+(N-(N/10)*10)
      GO TO 30
      ENDIF
50       CONTINUE
      ENDIF
      ENDIF
30     CONTINUE
      PRINT*, (K(I,9),I=1,120)
      DO 80 I1=1,12
      DO 80 I2=1,50
      DO 80 I3=1,33
      K1(I1,I2,I3)=0
80     CONTINUE
      DO 100 J=1,50
      L1=1
      I1=0
      L=0
      DO 70 I=1,120
      L=L+K(I,J)
       IF (L.GT.35) THEN
        I1=I1+1
```

```
          IF (K1(1,J,L1).NE.0) THEN
           L2=35+INT(L1*51.4)
            IF (L.GT.L2) THEN
             IF (L.LT.1668) THEN
               K1(I1,J,L1)=L2-L+K(I,J)
                N1=INT((L-L2)/51.4)
               IF (N1) 74,73,74
74               DO 72 I3=L1+1,L1+N1
               K1(1,J,I3)=51
72               CONTINUE
               L1=L1+N1
73               L1=L1+1
              I1=1
               K1(1,J,L1)=L-L2
              GO TO 70
             ELSE
               K1(I1,J,L1)=L2-L+K(I,J)
                N1=INT((1668-L2)/51.4)
               IF (N1) 81,82,81
81               DO 83 I3=L1+1,L1+N1
               K1(1,J,I3)=51
83               CONTINUE
               L1=L1+N1
82               L1=L1+1
                K1(1,J,L1)=1668-L2
              GO TO 100
             ENDIF
            ELSE
             K1(I1,J,L1)=K(I,J)
            GO TO 70
            ENDIF
          ELSE
           L1=INT((L-35)/51.4)
           IF (L1) 75,76,75
75          DO 71 I3=1,L1
           K1(1,J,I3)=51
71          CONTINUE
76          L1=L1+1
           K1(1,J,L1)=L-35-INT((L1-1)*51.4)
          GO TO 70
          ENDIF
         ELSE
          GO TO 70
         ENDIF
70    CONTINUE
100   CONTINUE
      PRINT*, (K1(I,19,29),I=1,10)
      OPEN (UNIT=2,FILE='LET1',STATUS='NEW')
      REWIND (2)
      DO 110 I=1,33
      WRITE (2,*) ((K1(I1,J,I),I1=1,12),J=1,50)
 110  CONTINUE
      CLOSE (UNIT=2)
      STOP
      END
```

/* To read and write a pattern in a file. */

```
      PROGRAM PATPAR(INPUT,OUTPUT)
      INTEGER XI(121,6)
      OPEN (UNIT=2,FILE='PAT1',STATUS='OLD')
      REWIND (2)
      DO 3 J1=1,6
      DO 3 J2=1,121
      READ (2,*) XI(J2,J1)
 3    CONTINUE
      CLOSE (UNIT=2)
      OPEN (UNIT=2,FILE='PAT7',STATUS='OLD')
      DO 4 J=1,121
      WRITE (2,*) XI(J,1)
 4    CONTINUE
      CLOSE (UNIT=2)
      STOP
      END
```

/* The main program for the recognition of patterns. */

```
      PROGRAM RECOGN(INPUT,OUTPUT)
      COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6),A(9)
      INTEGER X,XI,H,ND,C
      INTEGER IM(6)
      PLAM=0.3
      CM=0.1
 100  CALL COMM(M6,M1,M2)
      IF (M6) 9,20,20
 20   IF (M2) 1,2,1
 1    CALL TEACHER(PLAM,M1,CM,M6)
 2    IF (M1) 14,15,15
 15   CALL READ1
 14   CALL READ4
      PRINT *,'DO YOU WANT TO INPUT A LETTER?(1/0)'
      READ *,M3
      IF (M3) 11,12,11
 11   CALL INPUT
```

```
   GO TO 13
12  CALL READ3
13  CALL RECOG1(M1,CM)
    CALL COMPAR1(FLG,NUI)
    CALL OUTPUT
    IF (FLG) 3,4,3
3   PRINT *,'FAILURE'
4   PRINT *,NUI
5   CONTINUE
    PRINT *,'PUT THE PROPER PATTERN NUMBER'
    READ *,NNU
    IF (NNU-NUI) 8,99,8
8   CALL NJJK(PLAM,NNU)
    CALL LEARN(PLAM,M1,CM)
    CALL WRITE
99  GO TO 100
9   CONTINUE
    STOP
    END
```

/* Communications with the user. */

```
    SUBROUTINE COMM(M6,M1,M2)
    PRINT *,'WEIGHTS ?(1/0)'
    READ *,M6
    PRINT *,'MAGNETIC FIELD ?(1/0/-1)'
    READ *,M1
    PRINT *,'DO YOU WANT ME TO START A TEACHING SESSION?(1/0)'
    READ *,M2
    RETURN
    END
```

/* The main training routine. */

```
    SUBROUTINE TEACHER(PLAM,M1,CM,M6)
    COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6),A(9)
    INTEGER X,XI,H,C,ND
    PRINT *,'DO YOU WANT GEOMETRICAL FIGURES ?(1/0)'
    READ *,M5
    IF (M5) 5,6,5
5   CALL PATT1
    GO TO 13
6   PRINT *,'DO YOU WANT TO INPUT ANOTHER SET OF PATTERNS?(1/0)'
    READ *,M4
    IF (M4) 11,12,11
11  CALL PATT2
    GO TO 13
12  CALL READ4
```

```
13  IF (M6) 20,21,20
20  CALL WEIGHT
21  CALL CORR(C)
    DO 1 J=1,9
    G(J,J)=0.0
 1   CONTINUE
    CALL JJK
    CALL LEARN(PLAM,M1,CM)
    CALL WRITE
    RETURN
    END
```

/* This routine assigns different firing rates to the neurons. */

```
    SUBROUTINE WEIGHT
    COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6),A(9)
    INTEGER X,H,XI
    A(1)=1
    A(2)=1
    A(3)=1
    A(4)=1
    A(5)=-1
    A(6)=4.5
    A(7)=1
    A(8)=1
    A(9)=1
    RETURN
    END
```

/* The initial assignment of the synaptic matrix. */

```
    SUBROUTINE JJK
    COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6),A(9)
    INTEGER XI,X,H,C,ND
    DO 1 J=1,8
    DO 1 K=J+1,9
    SNU=0.0
    DO 2 NU=1,6
     SNU=SNU+XI(J,NU)*XI(K,NU)*A(J)*A(K)
 2   CONTINUE
     G(J,K)=SNU/9.0
    G(K,J)=G(J,K)
 1   CONTINUE
    RETURN
    END
```

/* The change in the synaptic matrix due to learning. */

```
      SUBROUTINE NJJK(PLAM,NUI)
      COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6),A(9)
      INTEGER XI,X,H,C,ND
      DO 1 J=1,8
      DO 1 K=J+1.9
       G(J,K)=G(J,K`-PLAM*A(J)*A(K)*(X(J)*X(K)-XI(J,NUI)*XI(K,NUI))/9.0
      G(K,J)=G(J,K)
 1     CONTINUE
      RETURN
      END
```

/* The following three routines calculate the energy in the network.*/

```
      SUBROUTINE HAM(HM)
      COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6),A(9)
      INTEGER H,X,XI,ND
      HM=0.0
      DO 1 J=1,8
      DO 1 K=J+1,9
      HM=HM-G(J,K)*H(J)*H(K)
 1     CONTINUE
      RETURN
      END


      SUBROUTINE HAM1(HM,CM)
      COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6)
      INTEGER H,X,XI,C,ND
      HM=0.0
      DO 6 L=1,6
      DO 6 M=1,6
       MM1=4*ND(L)-C(L,M)-3*9
       HM=HM-CM*SIGN(1,MM1)*ABS(ND(L))
 6     CONTINUE
      RETURN
      END


      SUBROUTINE HAM2(HM,CM)
      COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6)
      INTEGER H,X,XI,C,ND
      CALL HAM(HM)
      HM1=HM
      CALL HAM1(HM,CM)
      HM=HM+HM1
      RETURN
      END
```

```
/* The following three routines calculate the energy change. */

      SUBROUTINE DH2(K,DE,CM)
      COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6)
      INTEGER XI,X,H,K,C,ND
      CALL DH(K,DE)
      DE2=DE
      CALL DH1(K,DE,CM)
      DE=DE2+DE
      RETURN
      END

      SUBROUTINE DH1(K,DE,CM)
      COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6)
      INTEGER XI,X,H,K,C,ND
      DE=0.0
      DO 1 L=1,6
      LL=ND(L)
      ND(L)=ND(L)+2*H(K)*XI(K,L)
      DO 1 M=1,6
      M1=4*ND(L)-C(L,M)-3*9
      M2=4*LL-C(L,M)-3*9
       DE=DE-CM*(SIGN(1,M1)*ABS(ND(L))-SIGN(1,M2)*ABS(LL))
 1    CONTINUE
      RETURN
      END

      SUBROUTINE DH(K,DE)
      COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6),A(9)
      INTEGER XI,X,H,K
      HO=0.0
      DO 2 L1=1,9
      HO=HO-G(K,L1)*H(L1)
 2    CONTINUE
      DE=2.0*H(K)*HO
      RETURN
      END

/* The next six routines check the neuron values . */

      SUBROUTINE RELAX2
      COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6),A(9)
      INTEGER H,X,XI
      DO 1 J=1,9
      H(J)=H(J)*(-1)
 6    CALL DH(J,DE)
 8    IF (DE) 2,4,4
```

```
4  H(J)=H(J)*(-1)
   GO TO 1
2  CONTINUE
1  CONTINUE
   RETURN
   END

   SUBROUTINE RELAX1
   COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6),A(9)
   INTEGER H,X,XI
   DO 1 MU=1,9
   J=INT(RANF()*9.0)+1
   H(J)=H(J)*(-1)
6  CALL DH(J,DE)
8  IF (DE) 2,4,4
4  H(J)=H(J)*(-1)
   GO TO 1
2  CONTINUE
1  CONTINUE
   RETURN
   END

   SUBROUTINE RELAX4(CM)
   COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6)
   INTEGER H,X,XI,C,ND
   DO 1 J=1,9
   H(J)=H(J)*(-1)
   CALL DH1(J,DE,CM)
8  IF (DE) 2,4,4
4  H(J)=H(J)*(-1)
   DO 3 L=1,6
   ND(L)=ND(L)+2*H(J)*XI(J,L)
3  CONTINUE
   GO TO 1
2  CONTINUE
1  CONTINUE
   RETURN
   END

   SUBROUTINE RELAX3(CM)
   COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6)
   INTEGER H,X,XI,C,ND
   DO 1 MU=1,9
   J=INT(RANF()*9.0)+1
   H(J)=H(J)*(-1)
   CALL DH1(J,DE,CM)
```

```
8  IF (DE) 2,4,4
4  H(J)=H(J)*(-1)
   DO 3 L=1,6
   ND(L)=ND(L)+2*H(J)*XI(J,L)
3  CONTINUE
   GO TO 1
2  CONTINUE
1  CONTINUE
   RETURN
   END


   SUBROUTINE RELAX6(CM)
   COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6)
   INTEGER H,X,XI,C,ND
   DO 1 J=1,9
   H(J)=H(J)*(-1)
   CALL DH2(J,DE,CM)
8  IF (DE) 2,4,4
4  H(J)=H(J)*(-1)
   DO 3 L=1,6
   ND(L)=ND(L)+2*H(J)*XI(J,L)
3  CONTINUE
   GO TO 1
2  CONTINUE
1  CONTINUE
   RETURN
   END


   SUBROUTINE RELAX5(CM)
   COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6)
   INTEGER H,X,XI,C,ND
   DO 1 MU=1,9
   J=INT(RANF()*9.0)+1
   H(J)=H(J)*(-1)
   CALL DH2(J,DE,CM)
8  IF (DE) 2,4,4
4  H(J)=H(J)*(-1)
   DO 3 L=1,6
   ND(L)=ND(L)+2*H(J)*XI(J,L)
3  CONTINUE
   GO TO 1
2  CONTINUE
1  CONTINUE
   RETURN
   END
```

/* The relaxation process. */

```
      SUBROUTINE RECOG1(M1,CM)
      COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6),A(9)
      INTEGER H,X,XI,ND,C
      IF (M1) 1,2,3
 2    DO 5 I=1,3
      CALL RELAX1
      CALL RELAX2
      CALL RELAX1
 5    CONTINUE
      GO TO 7
 1    CALL DOT(ND)
      DO 6 L=1,3
      CALL RELAX3(CM)
      CALL RELAX4(CM)
      CALL RELAX3(CM)
 6    CONTINUE
      GO TO 7
 3    CALL DOT(ND)
      DO 8 L=1,3
      CALL RELAX5(CM)
      CALL RELAX6(CM)
      CALL RELAX5(CM)
 8    CONTINUE
 7    DO 20 M=1,9
      X(M)=H(M)
 20   CONTINUE
      RETURN
      END
```

/* Permuting the patterns. */

```
      SUBROUTINE PERM(IM)
      INTEGER IM(6)
      DO 3 L=1,3
      I=INT(RANF()*6.0)+1
 2    J=INT(RANF()*6.0)+1
      IF(I-J) 1,2,1
 1    K=IM(I)
      IM(I)=IM(J)
      IM(J)=K
 3    CONTINUE
      RETURN
      END
```

/* The next two routines compare the output of the relaxation with the desired output. */

```
      SUBROUTINE COMPARE(FLG,NUI)
      COMMON X(9),H(9),XI(9,6)
      INTEGER XI,X,H,FLG
      ISUM=0
      FLG=0
      DO 3 I=1,9
      ISUM=XI(I,NUI)-X(I)
      IF(ISUM) 2,3,2
3     CONTINUE
      GOTO 4
2     FLG=1
4     CONTINUE
      RETURN
      END


      SUBROUTINE COMPAR1(FLG,NUI)
      COMMON X(9),H(9),XI(9,6)
      INTEGER XI,X,H,FLG
      NUI=0
      ID=0
      ID1=0
      FLG=0
      DO 3 I1=1,6
      DO 4 I=1,9
      ID=ABS(XI(I,I1)-X(I))
4     CONTINUE
      IF (ID) 6,5,6
6     IF (ID-ID1) 7,3,3
7     ID1=ID
      NUI=I1
3     CONTINUE
      GO TO 8
5     FLG=1
      NUI=I1
8     CONTINUE
      RETURN
      END
```

/* The following two routines implement the generalized learning.*/

```
      SUBROUTINE MODIFY(NUI,M1)
      COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6)
      INTEGER XI,H,X,C,ND
      DO 1 I=1,9
      H(I)=XI(I,NUI)
1     CONTINUE
      K=INT(RANF()*9.0)+1
      H(K)=-H(K)
      IF (M1) 2,3,2
3     CALL DOT(ND)
2     CONTINUE
      RETURN
      END


      SUBROUTINE LEARN(PLAM,M1,CM)
      COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6),A(9)
      INTEGER XI,X,H,FLG
      INTEGER IM(6)
      DO 23 IJK=1,6
      IM(IJK)=IJK
23    CONTINUE
1     ID=0
      CALL PERM(IM)
      DO 2 L=1,6
      NUI=IM(L)
      CALL MODIFY(NUI,M1)
      CALL RECOG1(M1,CM)
      CALL COMPARE(FLG,NUI)
      IF (FLG) 3,4,3
4     ID=ID+1
      IF (M1) 6,7,8
6     CALL HAM1(HMIN,CM)
      GO TO 9
7     CALL HAM(HMIN)
      GO TO 9
8     CALL HAM2(HMIN,CM)
9     PRINT *,HMIN
      GO TO 2
3     CALL NJJK(PLAM,NUI)
2     CONTINUE
      PRINT *,ID
      IF (ID-6) 1,5,5
5     CONTINUE
      RETURN
      END
```

/* The next two routines store patterns in a file. */

```
      SUBROUTINE PATT1
      COMMON X(9),H(9),XI(9,6)
      INTEGER XI,X,H
      DO 1 I=1,6
      DO 1 J=1,9
      XI(J,I)=-1
1     CONTINUE
```

```
      DO 2 I=6,9,3
      XI(I,1)=1
 2    CONTINUE
      DO 3 I=4,6
      XI(I,2)=1
 3    CONTINUE
      DO 4 I=1,9,4
      XI(I,3)=1
 4    CONTINUE
      DO 5 I=1,9,5
      XI(I,4)=1
 5    CONTINUE
      DO 6 I=3,9,4
      XI(I,5)=1
 6    CONTINUE
      DO 7 I=3,9,5
      XI(I,6)=1
 7    CONTINUE
      DO 8 I=7,8
      XI(I,6)=1
 8    CONTINUE
      RETURN
      END


      SUBROUTINE PATT2
      COMMON X(9),H(9),XI(9,6)
      INTEGER X,H,XI
      OPEN (UNIT=2,FILE='PAT8',STATUS='NEW')
      REWIND (2)
      DO 2 J=1,6
      DO 1 I=1,9
      XI(I,J)=-1
 1    CONTINUE
 2    CONTINUE
 5    READ *,K,M
      IF (K) 3,3,4
 4    XI(K,M)=1
      GO TO 5
 3    CONTINUE
      WRITE (2,*) ((XI(I,J),I=1,9),J=1,6)
      CLOSE (UNIT=2)
      RETURN
      END
```

/* Presentation of the input pattern.*/

```
      SUBROUTINE INPUT
      COMMON X(9),H(9),XI(9,6)
      INTEGER X,H,XI
      OPEN (UNIT=2,FILE='PAT9',STATUS='NEW')
      REWIND (2)
      DO 1 I=1,9
      H(I)=-1
 1    CONTINUE
 5    READ *,K
      IF (K) 3,3,4
 4    H(K)=1
      GO TO 5
 3    CONTINUE
      WRITE (2,*) (H(I1),I1=1,9)
      CLOSE (UNIT=2)
      RETURN
      END
```

/* Read the stored synaptic strengths.*/

```
      SUBROUTINE READ1
      COMMON X(9),H(9),XI(9,6),G(9,9)
      INTEGER X,XI,H
      DO 1 I=1,9
      G(I,I)=0.0
 1    CONTINUE
      OPEN (UNIT=2,FILE='GMAT',STATUS='OLD')
      REWIND (2)
      DO 2 J1=1,8
      DO 2 J2=J1+1,9
      READ (2,*) G(J1,J2)
      G(J2,J1)=G(J1,J2)
 2    CONTINUE
      CLOSE (UNIT=2)
      RETURN
      END
```

/* Find the average correlation of the input patterns. */

```
      SUBROUTINE CORR(C)
      COMMON X(9),H(9),XI(9,6)
      INTEGER C(6,6)
      INTEGER X,XI,H
      DO 1 I=1,6
      DO 1 I1=1,6
      C(I,I1)=0
 1    CONTINUE
      AC=0.0
      DO 2 L=1,5
      C(L,L)=9
      DO 2 L1=L+1,6
      DO 2 L2=1,9
       C(L,L1)=C(L,L1)+XI(L2,L)*XI(L2,L1)
       C(L1,L)=C(L,L1)
 2     CONTINUE
      C(6,6)=9
      DO 3 K=1,5
      DO 3 K1=K+1,6
      AC=AC+C(K,K1)
 3     CONTINUE
       AC=AC/15.0
      PRINT *,AC
      RETURN
      END
```

/* The average Hamming distance of the input patterns. */

```
      SUBROUTINE DIST(C,D)
      INTEGER C(6,6)
      DIMENSION D(6,6)
      DO 1 K=1,6
      DO 1 K1=1,6
      D(K,K1)=0.0
 1     CONTINUE
      DO 2 I=1,5
      DO 2 I1=I+1,6
       D(I,I1)=9.0/2.0-C(K,K1)/2.0
 2     CONTINUE
      RETURN
      END
```

/* The inner products of vectors. */

```
      SUBROUTINE DOT(ND)
      COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6)
      INTEGER X,H,XI,C
      INTEGER ND(6)
      DO 1 I=1,6
      ND(I)=0
 1     CONTINUE
      DO 2 L=1,6
      DO 2 J=1,9
      ND(L)=ND(L)+XI(J,L)*H(J)
 2     CONTINUE
      RETURN
      END
```

/* The next 2 routines are reading patterns from a file. */

```
      SUBROUTINE READ3
      COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6)
      INTEGER X,XI,H,ND,C
       OPEN (UNIT=2,FILE='PAT2',STATUS='OLD')
      REWIND (2)
      READ (2,*) (H(K),K=1,9)
      CLOSE (UNIT=2)
      RETURN
      END


      SUBROUTINE READ4
      COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6)
      INTEGER X,XI,H,C,ND
       OPEN (UNIT=2,FILE='PAT8',STATUS='OLD')
      REWIND (2)
      READ (2,*) ((XI(J2,J1),J2=1,9),J1=1,6)
      CLOSE (UNIT=2)
      RETURN
      END
```

/* This routine writes the values of the synaptic matrix in a file. */

```
      SUBROUTINE WRITE
      COMMON X(9),H(9),XI(9,6),G(9,9),C(6,6),ND(6)
      INTEGER X,XI,H,ND,C
       OPEN (UNIT=2,FILE='GMAT',STATUS='OLD')
      REWIND (2)
      DO 10 J=1,8
      DO 10 J1=J+1,9
      WRITE (2,*) G(J,J1)
 10   CONTINUE
      CLOSE (UNIT=2)
      RETURN
      END
```

/* The presentation of the network's output to the user. */

```
      SUBROUTINE OUTPUT
      COMMON X(9)
      INTEGER X
      PRINT 10
 10   FORMAT (3(/))
      DO 2 L=1,9
      IF (X(L)-1) 1,2,1
 1    X(L)=0
 2    CONTINUE
      DO 30 L=1,11
      PRINT *,(X(I),I=3*L-2,3*L)
 30   CONTINUE
      RETURN
      END
```

The Pascal programs

```
(*OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOOOOOOOOOO*)

 PROGRAM UNPAK (INPUT,OUTPUT);

(*OOOOOOOOOOOOOOOOOOOOOOOOOOOOOO

 THIS PROGRAM UNPACKS DIGITIZED PATTERNS WHICH HAVE BEEN
      STORED IN A PACKED  FORMAT.

 INPUT:  FILE CONTAINS ONE LINE PER PATTERN AS FOLLOV 'S:
 -----
              - THE FIRST 2 INTEGERS ARE SOME IDENTIFICATION NUMBERS
              - THE THIRD INTEGER IS THE IDENTITY OF THE DIGIT (0 TO 9)
              - THE 4TH AND 5TH INTEGERS ARE THE ROW AND COLUMN
                    NUMBERS
              - ALL THE FOLLOWING INTEGERS ARE GIVING THE LENGTH OF
                THE ALTERNATING SEQUENCES OF BACKGROUND AND
                    OBJECT PIXELS;
                NOTE THAT THE PATTERNS ALWAYS HAVE 2 EMPTY ROWS AT
              THE TOP AND ONE EMPTY COLUMN AT THE LEFT (THIS LAST
              "RULE"  MEANS THE FIRST SEQUENCE OF EACH LINE IS
              ALWAYS FOR BACKGROUND). OBVIOUSLY, WHEN THE SUM OF
              THE LENGTHS OF SEQUENCES IS EQUAL TO THE NUMBER OF
              COLUMNS, WE HAVE  COMPLETED A LINE ETC...

 OUTPUT: FOR EACH PATTERN:
 ------
              - FIRST LINE IS GIVING ROW AND COLUMN NUMBERS, AS WELL
                AS IDENTITY OF CHARACTER AND COUNT (4 INTEGERS IN
              ALL)
              - ALL OTHER LINES REPRESENT A ROW OF THE PATTERN
              USING
               "." FOR BACKGROUND AND "M" FOR OBJECT REGIONS;
               NOTE THAT THE FIRST 2 EMPTY ROWS AND THE FIRST EMPTY
               COLUMN ARE REMOVED.
                                        *)


 VAR

   FILENUM, ZIPCODE, IDEN, ROW, COL: INTEGER;
   I, J, COUNT, LENG, TOTLENG, K   : INTEGER;
   CHOICE                          : ARRAY[0..1] OF CHAR;
```

```
  CH                        : CHAR;

 BEGIN
  LINELIMIT (OUTPUT,-1);
  CHOICE[0]:= '.';  CHOICE[1]:= 'M';
  FOR COUNT:=1 TO 100 DO
    BEGIN
      READ (INPUT,FILENUM,ZIPCODE,IDEN,ROW,COL);
      READ (INPUT,LENG); READ (INPUT,LENG);
      ROW:= ROW - 2;
      WRITELN (OUTPUT,ROW:5,COL-1:5,IDEN:5,COUNT:5);
      FOR I:=1 TO ROW DO
        BEGIN
          TOTLENG:= 0;
          READ (INPUT,LENG);
          CH:= CHOICE[0];
          FOR J:=1 TO LENG-1 DO WRITE (OUTPUT,CH);
          TOTLENG:= TOTLENG + LENG;
          K:= 0;
          WHILE TOTLENG < COL DO
            BEGIN
              K:= 1 - K;
              CH:= CHOICE[K];
              READ (INPUT,LENG);
              FOR J:=TOTLENG+1 TO TOTLENG+LENG  DO WRITE
(OUTPUT,CH);
              TOTLENG:= TOTLENG + LENG
            END;
          WRITELN (OUTPUT)
        END;
      READLN (INPUT)
    END
 END.
```