Learning Algorithms in Neural Networks

Francisco A. Camargo

December 12, 1990

(Draft)

The DCC Laboratory
Computer Science Department - Columbia University
New York, NY, 10027

CUCS-062-90

Abstract

Neural Network models have received increased attention in the recent years. Aimed at achieving human-like performance in tasks of the cognitive sciences domain, these models are composed of a highly interconnected mesh of nonlinear computing elements, whose structure is drawn from our current knowledge of biological neural systems. Several Neural Network Learning Algorithms have been developed in the past years. In these algorithms, a set of rules defines the evolution process undertaken by the synaptic connections of the networks, thus allowing them to learn how to perform specified tasks. In this article, several such algorithms are surveyed. They range from simple associative learning paradigms to more complex reinforcement learning systems. A detailed description of each algorithm is presented, and a discussion of their capabilities and limitations is included.

1 Introduction

The history of artificial neural networks starts with the work by McCulloch and Pitts [64] in which neurons are modeled as simple threshold-logic devices. Despite their assumed ignorance about the behavior of real neurons, their models were powerful enough to guarantee the construction of arbitrarily general computing circuits [65]. The saga for understanding neuronal behavior continues with the work by D.O. Hebb [30], who first proposes a qualitative mechanism describing the process by which synaptic connections are modified in order to reflect the learning process undertaken by interconnected neurons, when these are influenced by some environmental stimuli. The work by Rosenblatt [73] with his **perceptron**, further enhances our understanding of artificial learning devices, but the analysis by Minsky and Papert [66] demonstrated serious restrictions existing in these simplified models, thus causing a major set back in this research area. With the advent of more sophisticated models and learning algorithms proposed by Rumelhart and MacClelland [74], Hopfield [36], Kohonen [47], Grossberg [28], and others, a new wave of research was initiated. Since then, several new models have been proposed, and many applications of neural network models have been devised.

In this article, a survey of the most important neural network learning models is presented. A detailed discussion of their implementation is conducted, and a discussion of their general capabilities and limitations is included. This article follows a plan in which the various models are grouped according to their computational characteristics. In the next section, a simplified introduction to neural networks and their learning paradigms is presented. Section 3 discusses the models of networks suited for storing and retrieving association lists. Section 4 introduces the single layer perceptron and its evolution into the multi-layered feed forward networks, which are trained with the back-propagation algorithm. Section 5 discusses competitive unsupervised learning algorithms, and Section 6 provides a brief introduction to reinforcement learning algorithms. A brief discussion about some learning complexity issues closes this survey.

2 Neural Networks Overview

2.1 General Model

A **neural network** consists of a set of highly interconnected processing elements **(neurons)**, which are labeled with integers $1, 2, \ldots, N$. These neurons are connected through a set of **connection weights**, or **synaptic weights**. Every neuron i has M_i inputs, and one output y_i. The inputs are real valued quantities labeled $s_{i1}, s_{i2}, \ldots, s_{iM_i}$, representing signals coming either from other neurons in the network, or from the external world. Every neuron i has M_i **synaptic weights**, each one associated with each of the neuron inputs. These synaptic weights are labeled $w_{i1}, w_{i2}, \ldots w_{iM_i}$, and represent real valued quantities that multiply the corresponding input signal. Every neuron i has also an extra input, which is set to a fixed value I_i, and is referred to as the **threshold** of the neuron. Every neuron computes its own

internal state, or **total activation**, according to the following expression:

$$x_i = \sum_{j=1}^{M_i} w_{ij} s_{ij} + I_i = \sum_{j=0}^{M_i} w_{ij} s_{ij}, \qquad i = 1, \ldots, N. \tag{1}$$

Note that the threshold input can be hidden from the formulation by assuming that it is associated with a weight labeled $w_{i0} = I_i$, connected to a fixed input signal $s_{i0} = 1$.

The total activation is simply the inner product of the **input vector** $S_i = [s_{i0}, s_{i1}, \ldots, s_{iM_i}]^T$ by the **weight vector** $W_i = [w_{i0}, w_{i1} \ldots, w_{iM_i}]^T$. Geometrically, this inner product is proportional to the projection of S_i into W_i, i.e., the greater the agreement of directions in between S_i and W_i, the greater the total activation of the neuron. Every neuron i computes its output according to a function $y_i = f(x_i)$. The exact nature of f will depend on the neural network model under study. A few examples of the most frequently used **neuron functions** are:

$$y = f(x) = \frac{1}{1 + e^{-kx}}, \tag{2}$$

$$y = f(x) = \frac{e^{kx} - e^{-kx}}{e^{kx} + e^{-kx}} = tanh(kx), \tag{3}$$

$$y = f(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}, \tag{4}$$

$$y = f(x) = sign(x). \tag{5}$$

Figure 1 shows a schematic representation of a neuron, and a plot of its output function.

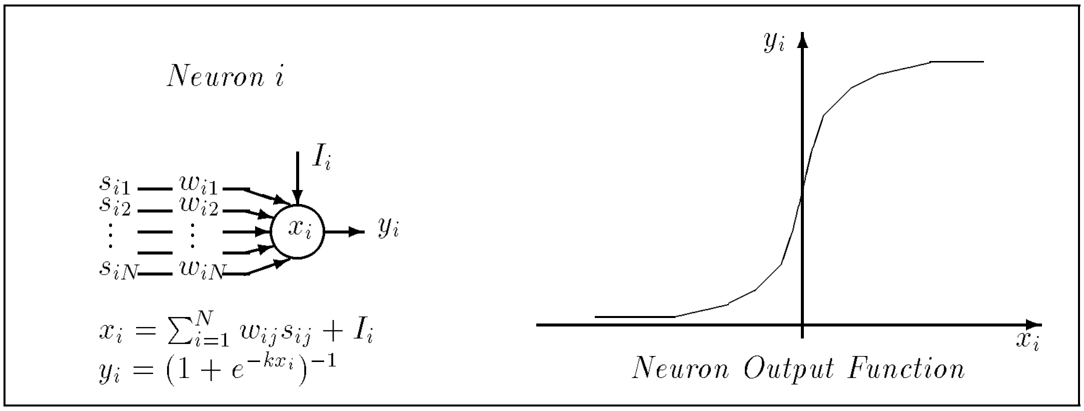

Figure 1: Neuron representation & typical output function

Equations (2) and (3) are continuous differentiable, monotonically increasing, step-like **(sigmoid)** functions. Their steepness are controlled by the **gain** term $k > 0$. In equation (2) the range of y is $(0, 1)$ for x in $(-\infty, \infty)$. In the same way, equation (3) limits the range of y

in $(-1, 1)$. By the other hand, equation (4) allows y to take only the discrete values 0, or 1. Equivalently, equation (5) allows y to assume only the discrete values -1, or 1. Note that the equations (2) and (4) are equivalent when $k \to \infty$. The same limit holds for equations (3) and (5).

Equation (1) can be re-written into its most common form, with the input signals substituted by the output signals of all the other neurons in the network [1], i.e. $s_{ij} = y_j$ for $i = 1, \ldots, N$, and $j = 1, \ldots, M_i$. With the simplifying assumption that every neuron has the same number of inputs, i.e. $M_i = N$, and that every neuron is connected to a fixed output neuron with label 0, such that $y_0 = 1$ and $w_{i0} = I_i$, the equation for the neuron total activation, without loss of generality, becomes

$$x_i = \sum_{j=1}^{N} w_{ij} y_j + I_i = \sum_{j=0}^{N} w_{ij} y_j, \qquad i = 1, \ldots, N. \tag{6}$$

Introducing $\mathbf{W} = [W_1, \ldots, W_N]^T$, the **network connection matrix**, and $\mathbf{Y} = [y_1, \ldots, y_N]^T$, the **network total state vector**, the network total activation vector, $\mathbf{X} = [x_1, \ldots, x_N]^T$, becomes

$$\boldsymbol{X} = \boldsymbol{W}.\boldsymbol{Y}. \tag{7}$$

Finally, by defining $\mathbf{F}(\mathbf{X}) = [f_1(x_1), \ldots, f_N(x_N)]^T$, where f_i represents the output function[2] of neuron i, the **network state motion equation** becomes

$$\boldsymbol{Y} = \boldsymbol{F}(\boldsymbol{W}.\boldsymbol{Y}). \tag{8}$$

It has not been mentioned to how the neurons are connected to one another, nor to the external world. Inspecting equation (8), one is tempted into believing that every neurons is connected to all the others. It turns out that different interconnection schemes allows for different classes of computations. For now, it is simply stated that with the proper choice of the connection weights, number of neurons, external inputs, outputs, and other parameters, it is possible to devise neural networks to perform virtually any computation. In fact, it is possible to devise broad classes of networks, where only the geometry of the connections is specified, and then employ some training algorithm which, by adjusting the values of each of the connection weights, allows the network to **learn** some specific computation. The most important of these learning algorithms will be covered in the next sections of this paper.

2.2 Learning paradigms

Neural Networks are useful for constructing models of arbitrary systems, here represented by time-varying stochastic processes over some vector space [74]. Therefore, it is assumed that

[1] Indeed, the external world signals can be assumed to come from neurons with no input connections, and whose outputs are either fixed, or some other function of time.

[2] In most cases all neurons will compute the same output function, with the exception of those representing signals coming from the external world.

there exists a probability distribution defining the "patterns" that, at any given time, can be presented to the inputs of the network. The outputs of the network can correspond to an action that the network is indicating should be performed (let's say for a control problem), or to a classification of what is presented to its inputs, or even to any other transformation over the input space. Whatever it is, it can be seen as a mapping from the input vector space to the output vector space. **Learning** is taken here as the means by which such mapping can be constructed. Usually, learning will require that one provides the network with enough "examples" of input-output pairs, and forces it to adapt in a way that allows it to generalize the computations necessary to reproduce such mapping, even for pairs never presented before.

One might ask why not to hardwire the network connections so that it produces the desired computation. However, one must remember that neural networks use a distributed representation of information: bits of information are spread through the connection weights. For complex tasks, requiring many neurons and many weights, there is often no possible way of telling what the connection weights should be. Besides, by using learning algorithms, one is relieved from having to explicitly determine what set of rules govern the behavior of the system being modeled. Instead, the network is trained by examples, allowing it to deduce such rules by itself. This makes it possible to use neural networks in domains for which there is not a complete understanding of the inner workings of the systems being modeled.

In the classic PDP book [74], Rumelhart and Zipser mention the existence of four common learning paradigms in neural networks. Their classification makes it easier to study the various network models available. In this paper, we expand their classification with the addition of **Reinforcement Learning** as extra paradigm. A brief overview of their distinctive characteristics is presented next. They are:

- **Auto Associator.** Under this paradigm, patterns are stored by repeatedly presenting them to a network, which learns how to internally represent them. Latter, an arbitrary pattern is presented, and the network is supposed to "recover" the stored pattern most similar to the presented one. Note that the inputs and outputs span the same vector space.

- **Pattern Associator.** Very similar to the previous case, pairs of patterns are presented and stored by the network. One of the patterns represents a **key**, and the other represents an **associated pattern**. Later presentations of complete keys, or even partially corrupted ones, should enable the network to reproduce the associated pattern.

- **Pattern Classifier.** In this case, patterns are presented to the network, which is supposed to categorize them according to some predefined set of classes. Usually, the learning process is supervised by a teacher who presents several patterns along with their correct classification. After being trained, the network should be able to correctly classify patterns different than those used during the training phase, but similar enough to the exemplars of each category used during the training phase. The algorithms that implement this kind of learning are referred to as **supervised learning algorithms**.

- **Regularity Detector.** Given an arbitrary probability distribution of patterns, each pattern is presented to the network with the probabilities in which they occur in that distribution. The network is supposed to "discover" salient features in such distribution, and thus classify each pattern according to these features. In this case, no teacher is provided, and the network has to "learn by itself" what each category should represent. The algorithms that implement this kind of learning are referred to as **unsupervised learning algorithms**.

- **Reinforcement Learning.** Under this paradigm, the network accepts input signals which are processed and transformed into the output signals. The only clue to the correctness of this transformation is an extra reinforcement signal, which acts as to reward or penalize the network, depending on the success or failure of its current computation. The network adapts in trying to minimize the penalties, and maximize the rewards, thus eventually producing only the correct transformations.

Most of the learning algorithms use some form of the mechanism proposed by D.O. Hebb [30] in 1949, which dictated how the synaptic connections among cells would evolve. In this classical work, it was proposed that any two brain cells for which there existed a synaptic connection among them, whenever they fired simultaneously due to some environmental stimulus, the efficacy of the synaptic connection among them would increase. This principle is known as the **Hebb learning rule**. Put in quantitative terms, if we denote by Y_a and Y_b the activity of these two cells, then the synaptic strength of their connection is affected by a quantity $\Delta W_{a,b} = \alpha Y_a Y_b$, for some adequate proportionality coefficient α. Although there seems to exist, it the real brain, evidence for synaptic modification caused by environmental stimulus, it is far from our purpose to study the validity of that statement. It is enough to say that for many purposes it does produce the desired effects, and that it appears naturally in many of the algorithms that are presented in this article.

3 Associative Networks

In this section we study networks used for storing and retrieving **associations**. An association is a "simple connection of abstract patterns", the simplest form of which is a **binary relation** [47]. A **pattern** is a multi-dimensional vector $R = [r_1, \ldots, r_m]^T$, with real valued coordinates. For simplicity, it is assumed that $r_i \in [-1, 1]$, for all $i = 1, \ldots, m$. There may be cases where the more restrictive condition that $r_i \in \{-1, 1\}$ will be imposed.

An association is an ordered pair of patterns (R, S), were $R = [r_1, \ldots, r_m]^T$ is the **key-pattern**, and $S = [s_1, \ldots, s_n]^T$ is the **associated pattern**. Finally, an **association list**, or **training set**, is a set of p associations $\{(R^1, S^1), \ldots, (R^p, S^p)\}$, such that all corresponding patterns have identical dimensionality, i.e., $\mid R^k \mid= m$, and $\mid S^k \mid= n$, for $k = 1, \ldots, p$.

In **Associative Networks**, the associations will be stored in the connection weights of the neurons. The representation of each association will be distributed over many connections, and every connection will be involved in storing many associations. This distributed representation will provide the system with a certain degree of robustness against minor physical damage, besides allowing it to "perceive" certain regularities in the stored training set.

The retrieval process takes place after all associations are stored, or **learned**. It is characterized by the presentation of different key-patterns to the network, which should, in turn, reproduce the corresponding associated-patterns. In most cases, the presented key-patterns will be noise-corrupted versions of those present in the stored association list. It is expected that the networks be capable of "eliminating" the noise components of these key patterns, and reproduce the stored associated pattern whose key "best matches" the presented key.

3.1 The Linear Associator

This Linear Associator was first introduced by Anderson [6], but has also been studied by many other authors [47, 18]. In fact, the computations performed by this model can be perfectly mapped into the more traditional methods found in the field of **linear regression analysis**.

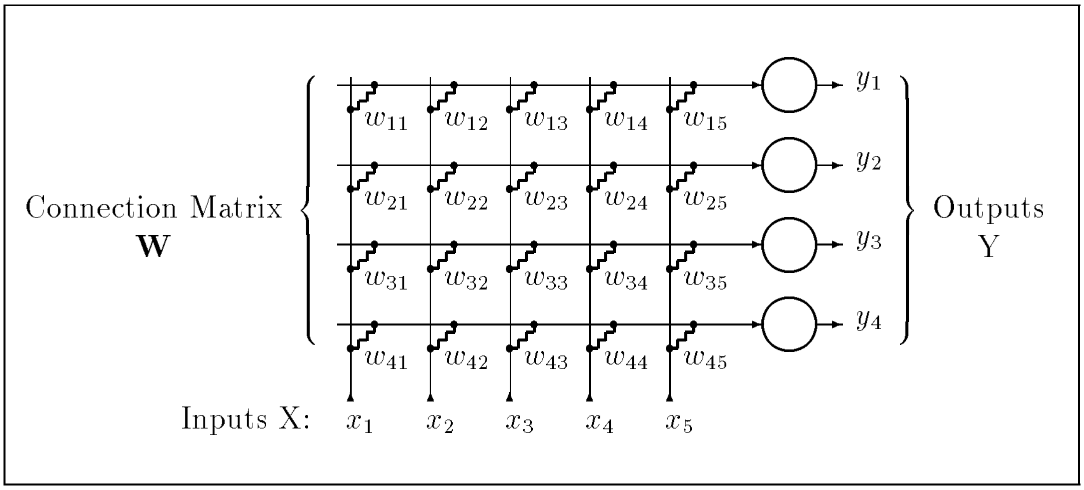

Figure 2: Linear Associator (4 neurons with 5 inputs)

The Linear Associator is a neural network suitable for storing association lists. The network has n neurons with m inputs each. It accepts the input vector $X = [x_1, \ldots, x_m]^T$ and produces the output vector $Y = [y_1, \ldots, y_n]^T$. Every neuron i receives the same input vector X, and computes its output y_i according to

$$y_i = \sum_{j=1}^{m} w_{i,j} x_j, \qquad i = 1, \ldots, n, \tag{9}$$

where $w_{i,j}$ is the connection weight from the input coordinate j to the neuron i. Figure 2 shows a typical linear associator.

To store an association list $\{(R^1, S^1), \ldots, (R^p, S^p)\}$, a simple form of the Hebb learning rule is prescribed [47, 5, 21]. In this case, the weights are computed as follows:

$$w_{i,j} = \sum_{k=1}^{p} s_i^k r_j^k, \qquad i = 1, \ldots, n;\ j = 1, \ldots, m. \tag{10}$$

Denoting by $\mathbf{R}$ the $m \times p$ matrix $[R^1, \ldots, R^p]$, and by $\mathbf{S}$ the $n \times p$ matrix $[S^1, \ldots, S^p]$, and using matrix notation, the $n \times m$ weight matrix $\mathbf{W}$ can be computed by $\mathbf{W} = \mathbf{SR^T}$.

After computing the matrix $\mathbf{W}$, the associative network with an input vector R, will produce an output $Y = \mathbf{W}.R = \mathbf{S}.\mathbf{R^T}.R = \sum_{k=1}^{p}(R^k.R).S^k$, i.e., a linear combination of the vectors S^k with coeficientes determined by the inner products $(R^k.R)$. In particular, If the input vectors are orthonormal, then the Linear Associator with connection weights computed according to equation (10) will exhibit **perfect recall** [47].

This result is trivial to derive. It also implies that the maximum number of associations that can be perfectly stored in the Linear Associator is at most m, i.e., the dimensionality of the input vectors. Unfortunately, the orthonormality condition is seldom satisfied. Nevertheless, the Linear Associator shows other processing capabilities worth investigating, even under weaker constraints on the input vectors.

For instance, if there are p different input vectors R^k to be learned, all associated with the same output vector S^0, and if the vectors R^k have a mean value V, such that $R^k = V + D^k$, where D^k is stochastic noise vector, with $\|D^k\| \ll \|V\|$ for all $k = 1, \ldots, p$, then the Linear Associator behaves as if it had repeatedly learned only the association (V, S^0) [5]. The output of the network will be an amplified version of the vector S^0, plus a small error dependent on the noise components of the input vectors. Anderson calls this property as **prototype learning**; the network extracts the common features of the input vectors and builds an internal representation for a "prototype" association (V, S) never present in the training set.

If the input vectors are not orthonormal, then the Hebb rule is no longer optimal. We have to introduce an error measure in order to estimate the "performance" of the linear associator. The sum of square errors is the measure most frequently used:

$$E = \sum_{k=1}^{p} \sum_{i=1}^{m} (s_i^k - y_i^k)^2. \tag{11}$$

Kohonen [47] has shown that the matrix W which minimizes equation (11) is given by

$$\boldsymbol{W} = \boldsymbol{S}.\boldsymbol{R}^*, \tag{12}$$

where $\mathbf{R}^*$ is the Moore-Penrose generalized pseudoinverse of $\mathbf{R}$. For the cases where the columns of R are linearly independent, the formula $\mathbf{R}^* = (\mathbf{R^T R})^{-1}\mathbf{R^T}$ can be used. If the rows are linearly independent, then $\mathbf{R}^* = \mathbf{R^T}(\mathbf{RR^T})^{-1}$ is another solution. Otherwise, a general, but expensive algorithm for computing the pseudoinverse can be found in [21]. Another alternative is to use any gradient technique which attempts to "move" the weights in the directions that reduce the error. One of these techniques, the Widrow-Hoff LMS procedure [82], will be studied later in this paper.

Anderson also notes that if the input and output vectors are made the same, the network implements an **auto-associator** capable of reconstructing partially corrupted vectors. In fact, he demonstrates that if the output vector is fed back to the inputs of the network, many bits of a corrupted input vector can be restored at every iteration. He goes one step further, devising a model where the neurons have a linear response within certain limits of their total excitation, but "saturate" if their total excitation fall outside a specified range $[-a, b]$, for $a, b > 0$. He demonstrates how this model, which he call the **Brain-State-in-a-Box**, attempts to minimize the "energy function" $E = -\frac{1}{2}X^T\mathbf{W}X$ as the state of the network evolves in time [6]. More general models than the Brain-State-in-a-Box will be discussed in the next section.

3.2 The Little/Hopfield Model

An important model of neural networks capable of storing and recollecting abstract association lists is that of fully connected sets of neurons. In this model, the inputs of a neuron are fed by the outputs of all the other neurons in the network. This arrangement creates a recurrent dynamic system capable of storing and retrieving any kind of association lists. Popularized by Little and Hopfield, this model has its origin in the fields of spin-glasses, Lyapunov functions, and dynamical systems [8, 17, 62, 71, 36, 37, 44, 21]. Figure 3 shows a typical representation of this type of network.

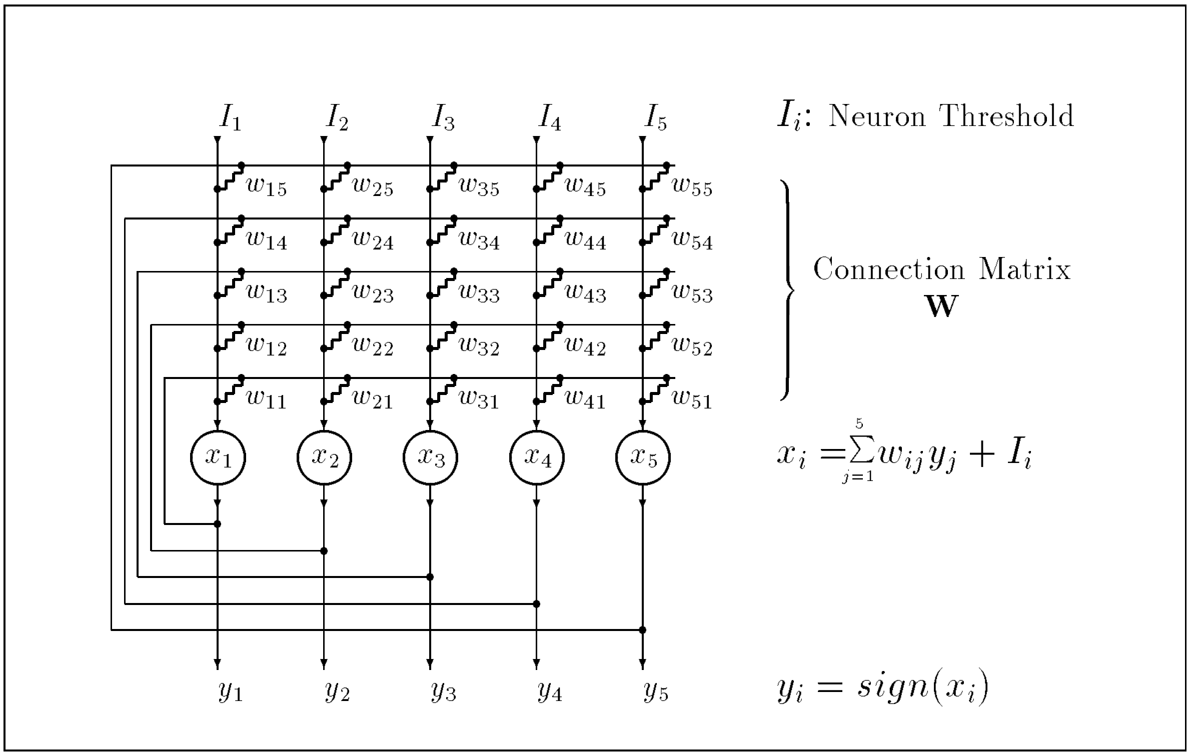

Figure 3: Fully connected network with 5 neurons.

In this model, a set of equations plus an update policy defines the dynamics of the system. The total activity of the neuron is defined by equation (6). The neuron outputs are computed according to equation (5), producing discrete outputs -1 or 1. If the output values should be 0 or 1, then the equation (4) can be used instead.

In order to fully describe the dynamics of the system, we must characterize how the state of the network evolves in time. There are four general **update policies:**

- **synchronous or parallel:** All neurons update their state and output simultaneously, with a unique clock defining the pace of the computation.
- **asynchronous:** The neurons update their states randomly, independently from one another, but all with the same mean frequency, and standard deviation. However, it can be assumed that, at any given moment, only one neuron updates its state.
- **sequential:** The neurons update their states one after the other, according to some predefined order.
- **block-sequential:** Groups of neurons are updated synchronously, while each group is updated one after the other, according to some predefined order.

It is desirable to have some single quantity that can be associated with the network state, thus allowing one to make predictions about the evolution of the network dynamics. Hopfield defines an "energy function" [36], or Hamiltonian [33, 1] which serves that purpose. Following his notation, we write

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{i,j} y_i y_j - \sum_{i}^{N} I_i y_i \tag{13}$$

Theorem 3.1 (Convergence) *For a neural network with dynamics as those described by equations (6) and (5), where* $\mathbf{W}$ *is a zero-diagonal symmetric connection matrix, every change* Δy_i *causes a change* $\Delta E \leq 0$ *in the energy of the network [36], no matter which update policy is used. The system converges to a stationary regime were* $\Delta E = 0$*. So,* E *is a Lyapunov function associated with the network, i.e., a strictly decreasing operator in the transient phase of the dynamic evolution, and constant in the steady state.*

The proof of the above theorem, for the asynchronous and sequential update policies, is straightforward: compute ΔE_i for a change Δy_i. For a zero-diagonal symmetric matrix $\mathbf{W}$, $\Delta E_i = -\Delta y_i[\sum_i w_{i,j} y_j + I_i]$. Note that if $\Delta y_i > 0$ then $\Delta E_i \leq 0$, and if $\Delta y_i < 0$ then $\Delta E_i < 0$. So, the system has to converge to a stationary regime where $\Delta E_i = 0$. The proofs for the other update policies are similarly simple, and can be found in [21].

An important corollary of the above theorem, and yet another theorem are all that is needed to guarantee the use of these fully connected networks as associative memories.

Corollary 3.2 (Limiting Cycles 1) *Every system as describe above, evolving under any update policy, will reach a fixed state Y, where no more changes can occur. In other words, there will be no cycles. The final state Y is called a* **ground state** *[21].*

Theorem 3.3 (Limiting Cycles 2) *A sufficient condition for any neural network to have cycles of length at most 2 is that the connection matrix* $\mathbf{W}$ *be non-negative, definite. Under these conditions, if the asynchronous or sequential update policies are used, then there will be no cycles at all [23].*

These two results are valid for networks with neurons using the equation (5) as their output function. For networks using equation (4), and running under a synchronous or block-sequential update policy, there may exist cycles of length at most 2. If the asynchronous or sequential update policies are used, then there will exist no cycles at all. This was proved independently by Frumkin and Goles. Note that it is always possible to devise different neuron functions with special connection matrix that will cause the system to have cycles of length greater than 2. In general, this will require that the energy function be not strictly decreasing, i.e, different contiguous state transitions would exist for which $\Delta E = 0$. Refer to [21, 23] for a more detailed discussion.

The importance of the above results should be clear by now: one can devise a recurrent dynamic system, and predict its stability. Such system may be started at any random state, and it will evolve, converging to a **ground state**. A quadratic energy function can be associated to the network states, and it is guaranteed that the ground states will correspond to the local minima of this energy function.

A procedure for sculpting the shape of the energy function, thus allowing one to "specify" which ground states the system will have, is the final ingredient for making these networks capable of implementing associative memories. The first of such procedures is again derived from the Hebb learning rule. It is identical to that of the linear-associator, and is called by many author as the **sum of outer products** procedure [36, 47, 21].

Algorithm 3.1 (Sum of Outer Products Procedure) .
Given M patterns $S^p = [s_1^p, \ldots, s_N^p]$ with $s_i \in \{-1, 1\}$ for $i = 1, \ldots, N$, and $p = 1, \ldots, M$, and if $M \ll N$, then the $N \times N$ connection matrix $\mathbf{W}$ *built according to equation*

$$w_{i,j} = \begin{cases} \sum_{p=1}^{M} s_i^p s_j^p, & \text{for } i \neq j, \text{ and } i, j = 1, 2, \ldots, N, \\ 0, & \text{for } i = j = 1, 2, \ldots, N. \end{cases} \tag{14}$$

will store all S^p as its ground states. As for the terms I_i, they are all set to zero.

One problem with this storage procedure is the fact that if Y is a ground state, then so is $-Y$. This can be immediately seen from equation (13). Another problem occurs if the stored patterns are not linearly independent; there will exist cross-talk [35] among the units involved in deciding the possible ground states, originating what has been called **spurious**

states. In fact, Hopfield [36] states that a neural network using this storage procedure will be able to store only about $M = 0.15N$ memory patterns before its "recollection" ability degenerates. He achieved this result through extensive computer simulation, but stronger bounds have been obtained.

Hopfield conjectured that the networks would almost always converge to the closest ground state, in terms of the relative Hamming distance from the starting state to all stored patterns [37]. He proposed that these networks could be used to implement **k-nearest neighbor classifiers**, which has been proved false through computer simulations developed by Montgomery and Kumar [67]. The main reason for this result is that, although the stored patterns create "valleys" in the quadratic energy surface, the shape of the basins of attraction are not symmetric in all dimensions of the state space. It may even happen that a single wrong bit in a ground state may cause the network to converge to a totally different state, far away from that starting point. In fact, Montgomery and Kumar proved that **"two vectors which differ only in one component cannot be made stable vectors for any diagonally zero matrix W"**.

Working towards increasing the memory capacity of his networks, Hopfield also conjectured that "unlearning" could help to improve the network capacity [38]. This conjecture was fully supported by the experimental work developed by Kleinfeld et al[45]. They argued that spurious states could be "diluted" through a reduction in their corresponding connection weights by a constant term ε. The idea is to adjust the synaptic weights by the amount $\Delta w_{i,j} = -\varepsilon \sum_{k=1}^{s} y_i^k y_j^k$ for all spurious states $Y^k, k = 1, \ldots, s$, that can be identified. They used values ranging from 0 to 0.2 for the 'unlearning rate" ε and concluded that indeed the storage capacity of the network increased, and that the minimal Hamming distance in between stored patterns could also be reduced. Other results where that the Hamming radius at which a random starting state converged to the desired memory remained constant, and the accessibility of stored patterns became indeed more evenly distributed after the unlearning was applied.

Using the methods found in the field of statistical mechanics, other researchers have improved our understanding of the real capabilities of these fully connected networks. Specifically, the work developed in the field of spin-glasses has proved extremely fruitful. In these lines, Amit et al [4] proved the following theorem:

Theorem 3.4 (Phase Transitions 1) *A neural network with dynamics as those of equations (1) and (5), and using the storage procedure prescribed by equation (14), will undergo two phase transitions as the ratio* $\alpha = M/N$ *increases. The first phase transition corresponds to the memory states becoming of lower energy than the spurious states, and happens with* $\alpha_1^c > 0.051$*. The second phase transition corresponds to the spurious states becoming again of lower energy than the stored memories, and happens with* $\alpha_2^c > 0.138$*. Beyond this point, no more patterns can be safely stored.*

Bruce et al [10] extended the above results for networks using the neuron equation (4). They computed $\alpha_1^c = 0.025$ and $\alpha_2^c = 0.069$. They also proposed a new storage procedure guaranteed to construct a weight matrix which encoded up to N specified ground states.

Diederich et al [17] proposed a similar storage procedure and, in yet another procedure, demonstrated how the ground states could be discriminated from spurious states. Both algorithms are presented next.

Algorithm 3.2 (Storage Procedure 2) *Given N patterns $S^p = [s_1^p, s_2^p, \ldots, s_N^p]^T$, with $s_i \in \{-1, 1\}$, for $i = 1, \ldots, N$, and $p = 1, 2, \ldots, N$, this algorithm builds a neural network that will have* **up to N** *patterns S^p as ground states, provided that a solution exists. Start with a "tabula rasa", i.e., $\mathbf{W}^k = \mathbf{0}$. Next, set an iteration counter $k = 0$ and run once through all patterns p, computing $w_{i,j}^{k+1}, i \neq j$ as follows:*

$$w_{i,j}^{k+1} = w_{i,j}^k + \frac{1}{N-1} s_i^p s_j^p H(1 - s_i^p \sum_{j=0}^{N} w_{i,j}^k s_j^p), \qquad i \neq j, \text{ and } i, j = 1, \ldots, N, \tag{15}$$

where $H(\cdot)$ is a Heaviside function just like equation (4). The iteration counter is incremented and the process is repeated. If a solution exists, the algorithm is guaranteed to converge, in a finite number of steps, to a point where the terms $w_{i,j}$ will no longer change. The resulting connection matrix will store all the N patterns.

The other algorithm due to Diederich [17], also builds a matrix $\mathbf{W}$ that stores up to N **linearly independent** patterns, but with the added property that, for all stored patterns, the following equality will hold:

$$s_i^p \sum_{j=1}^{N} w_{i,j} s_j^p = 1, \qquad \text{for } i = 1, 2, \ldots, N. \tag{16}$$

This property is useful in discriminating spurious states from the stored patterns. The price to pay for using this procedure is that the patterns have to be linearly independent.

Algorithm 3.3 (Storage Procedure 3) *Identical to the algorithm (3.2), but the weight update equation is replaced by the following expression:*

$$w_{i,j}^{k+1} = w_{i,j}^k + \frac{1}{N} s_i^p s_j^p (1 - s_i^p \sum_{j=0}^{N} w_{i,j}^k s_j^p), \qquad i, j = 1, \ldots, N. \tag{17}$$

Note that in this case the terms $w_{i,i} \neq 0$, and they are not excluded from the above iteration. What is important in these algorithms is that they attempt to change the weights only in the cases in which the corresponding association between neurons is not yet learned. This produces a more evenly distributed connection matrix and results in a more uniform energy surface [17].

3.3 Higher Order Models

A generalization of the Little/Hopfield model has been proposed by several authors [1, 10, 63]. It is based on a "non-physical" synaptic connection that correlates up to p signals, thus having the potential to resolve linear-dependency situations, and increased storage capacity. For a general $\boldsymbol{p}^{th}$-order network, the total neuron activation is computed as follows:

$$x_i = A_i^{(0)} + \sum_{k=1}^{p} \sum_{j_1=1}^{N} \cdots \sum_{j_k=1}^{N} \left(A^{(k)}_{i,j_1,\ldots,j_k} \prod_{l=1}^{k} y_{j_l} \right) \tag{18}$$

The output of each neuron is still computed according to equations (5) or (4), depending on whether the $[-1,1]$ or $[0,1]$ neuron is desired. The corresponding energy function can also be generalized as follows:

$$E = -A_0 - \sum_{i=1}^{N} A_i^{(0)} y_i - \sum_{k=1}^{p} \sum_{i=1}^{N} \sum_{j_1=1}^{N} \cdots \sum_{j_k=1}^{N} \left(y_i A^{(k)}_{i,j_1,\ldots,j_k} \prod_{l=1}^{k} y_{j_l} \right) \tag{19}$$

Although it would be preferable, from a formulation stand point, to have the degree of the energy function associated with the order of the network, a different convention is adopted here: the Little/Hopfield model is a 1^{st} order model, i.e., the neuron weights reflect first order correlations on the bits of stored patterns; its corresponding energy function is of degree 2. Thus, a $\boldsymbol{p}^{th}$ order network will have an energy function of degree $(\boldsymbol{p}+\mathbf{1})$.

This model allows for the neurons in the network to compute higher order correlations among the various bits of the stored patterns. However, it is rather difficult to establish the convergence conditions for the general case. The symmetry of the connection weights is no longer a sufficient condition for guaranteeing the monotonicity of the free energy of the system. Some simplified high order models have been studied by Lee et al [54], and, in particular, it was demonstrated that a system with an energy function described by equation (19) will follow a path that monotonically decreases its "free energy", and converges to a regime for which $\Delta E = 0$ for every change ΔY [54] if the following conditions are satisfied:

- Energy function of **even** degree $(\boldsymbol{p}+\mathbf{1}) \geq 2$.
- $A_0 = 0$, and $A_i^{(0)} = 0$, for $i = 1, \ldots, N$;
- $A^{(k)}_{j_0,\ldots,j_k} = 0$, for $j_i = 1, \ldots, N$; $i = 0, \ldots, k$; and $k = 1, \ldots, p-1$;
- $[A^{(p)}_{j_0,\ldots,j_p}]_{N\times\ldots\times N}$ is symmetric over any permutation of indexes $j_0, \ldots, j_p$.

Given m patterns $S^l = [s^l_1, \ldots, s^l_N]$, for $l = 1, \ldots, m$, the **generalized sum of outer products** is a storage procedure that enforces the above conditions. The synaptic weights are computed as follows:

$$A^{(p)}_{j_0,\ldots,j_p} = \sum_{l=1}^{m} s^l_{j_0} \ldots s^l_{j_p}; \qquad \text{for } j_i = 1, \ldots, N; \text{ i=1,...,p.} \tag{20}$$

The ground states of the energy function will correspond to the stored patterns S^l. Networks with such energy functions will have the ability to discriminate even among linearly dependent patterns, provided that the degree of the energy function is high enough to allow such discrimination, and that the network storage capacity have not yet been exceeded.

Several results have been developed for the computation of bounds on the storage capacity of generalized neural networks of higher orders [33, 8, 10]. The following theorem is due to Baldi and Venkatesh [8]:

Theorem 3.5 (Storage Capacity) *For neural networks with energy functions of fixed degree* $(\boldsymbol{p}+\mathbf{1})$*, the maximal number of programmable states is essentially* $O(N^p)$ *[8]. More precise bounds are:*

- *For arbitrary storage procedure: If* m *is the largest integer such that for almost any set of* m *vectors* $S^1, S^2, \ldots, S^m$*, we can find a nontrivial form of degree* $(p+1)$ *having* $S^1, S^2, \ldots, S^m$ *among its stable points, then:*

$$(N+1)^{-1}\sum_{j=0}^{p+1}\left(\begin{array}{c} N \\ j \end{array}\right) \leq m \leq \sum_{j=0}^{p}\left(\begin{array}{c} N-1 \\ j \end{array}\right) \tag{21}$$

- *For the generalized sum of outer products, the largest allowable rate of growth of* m *with respect to* N*, such that* m *vectors are stable with probability approaching 1, as* $N \to \infty$*, is given by:*

$$m = \frac{N^p}{2(p+1)!\ln N} \tag{22}$$

Storage procedures more general than the above mentioned "generalized sum of outer products" can be found in [54], but there hasn't been widespread acceptance of these models yet. One reason for that, is the fact that the increased capacity of these higher order networks is achieved at the expense of a significant increase in the number of connection weights. Indeed, for the simplified case of networks using the generalized sum of outer products storage procedure, a network with N neurons of order p will require exactly N^p weights.

3.4 Boltzmann Machines

Another way to look at higher order networks is to imagine an "equivalent" Hopfield network where not all neurons have their states specified by the stored patterns. These unspecified neurons, and the connections to them, act as to "encode" higher order relationships among the specified signals, and allow the network to store correlated patterns, besides increasing its storage capacity. Unfortunately, there is no way of telling what the states of the so called **hidden neurons** should be, and therefore this is not a practical approach. However, there exists a more sophisticated model for which this approach can be employed. The **Boltzmann machines** can be trained to produce arbitrary mappings from an input space to an output space, without the need to fully specify the states of all neurons in the network. Besides, these "machines" follow the same underlying principle of energy minimization present in the Hopfield networks, thus making it possible to use them as associative memories that can restore patterns from incomplete or corrupted keys.

Many times referred to as a generalization of the Hopfield networks [74, 75], **Boltzmann Machines** are an important neural network model suited for solving **optimization and constraint satisfaction problems** [43, 39]. The overall structure of the network is essentially the same of the Hopfield networks, i.e., a fully connected sets of neurons, with the connection weights defining an energy landscape whose minimum is being sought. The main difference is the behavior of the neurons, which are no longer deterministic: they fire with a probability that is function of their total activation. Indeed, the neurons are given two possible states, either 0, or 1. The probability that a neuron will "fire", ie., the probability that its output y will be 1, is given by

$$P(y = 1) = \frac{1}{1 + e^{-\frac{x}{T}}}, \tag{23}$$

where x is the total activation of the neuron, computed according to equation (1), and T is the current **temperature**. It has been proved that, if the neurons change states according to the above rules, the network will reach a dynamic "thermal equilibrium", i.e., the neurons will continue to change their states, but the probability of finding the network in any given global state, for a fixed temperature T, remains constant, and obeys a Boltzmann distribution. In such distributions, the ratio of the probabilities of two possible states A and B is function only of the energy difference between the two. The network energy is given by equation (13). The probability ratio is computed by

$$\frac{P_A}{P_B} = e^{\frac{-(E_A - E_B)}{T}}. \tag{24}$$

From the above equation, it is easy to see that, if the temperature T is low, and a thermal equilibrium has been reached, then "lower energy states" are favored over "higher energy states". That is not to say that, for low temperatures, the network will only be in low energy states, but, if one samples the states of the network long enough, chances are that he/she will find lower energy states happening more frequently.

The Boltzmann machines have been used in a variety of optimization problems. In fact, any optimization problem that can be cast into a quadratic equation of the form of equation (13) is a strong candidate. Their advantage over Hopfield's network is their ability to escape local minima states, which is achieved by the non-deterministic neuron update function. The optimization process usually is started at some initial state with the temperature set to some "high value", and the network is left alone, following its update rules and eventually reaching a thermal equilibrium, most likely at some high energy local minima. The temperature is then gradually lowered, enhancing the chances of lower energy states. The process continues until a satisfactory final state is reached at a very low temperature. This process is usually referred to as **simulated annealing** [43], due to its resemblance to the annealing processes used to strength materials in metallurgy.

As mentioned before, of particular interest is a learning algorithm developed by Ackley et al [2] which allows the network to be trained to store association lists of vector patterns.

Furthermore, the algorithm allows that the **hidden neurons** in the network be trained to compute higher order correlations among the signals present in the patterns being stored, without the need of specifying their "desirable" state for every single pattern.

For the purpose of explaining the algorithm, we assume a network where some neurons correspond to the input vector R, and other neurons correspond to the output vector S. The training process starts with a network where the weights are set at random to small values. A pair of vectors to be stored is presented to the network by clamping the input neurons to the values specified by the vector R, and by clamping the output neurons to the values specified by the vector S. Starting at some high temperature T^i, the network is allowed to reach its thermal equilibrium, with the temperature being slowly reduced up to the value of T^f. It then runs for a fixed amount of time at the temperature T^f and each weight "records" the amount of time during which the interconnecting units are both firing. This process is repeated for all pairs of patterns being stored, so that the weights can compute the quantity $\langle y_i y_j \rangle^+$, i.e. the expected probability that the units i and j are both firing, averaged over all patterns being stored. Next, the same process is repeated, but at this time only the input units are clamped, and the output units are set free to assume whatever state they can. Again, after thermal equilibrium is reached for every pattern, every weight "records" the amount of time during which the interconnecting units are both firing. The result of this phase is the computation of the quantity $\langle y_i y_j \rangle^-$, i.e. the expected probability that the units i and j are both firing, despite the fact that the outputs are not clamped. Finally, the weights are updated by a small quantity, proportional to the difference of the measurements in both phases, i.e,

$$\Delta w_{i,j} = \frac{\varepsilon}{T^f}(\langle y_i y_j \rangle^+ - \langle y_i y_j \rangle^-), \tag{25}$$

where ε is a small positive constant. For small enough ε, the above weight update rule causes the system to follow a gradient descent in an information theoretic measure of the distance between the clamped and free-running global network state probability distributions [2]. This measure, due to Kullback [52], is given by

$$G = \sum_{\alpha,\beta} P^+(R^\alpha \wedge S^\beta) \ln \frac{P^+(S^\beta|R^\alpha)}{P^-(S^\beta|R^\alpha)} \tag{26}$$

where $P^+(R^\alpha \wedge S^\beta)$ is the global state probability during the clamped phase, with the input vector R^α and the output vector S^β; $P^+(S^\beta|R^\alpha)$ is the conditional probability during the clamped phase that the output is S^β for the input R^α; and $P^-(S^\beta|R^\alpha)$ is the conditional probability during the free-running phase that the output is S^β for the input R^α.

It can be shown [2, 74] that the partial derivatives of the above quantity with respect to the connection weights has always the form of equation (25) (divided by $-\varepsilon$), no matter whether the interconnecting neurons are input, output or hidden units. Moreover, G is never negative, and is only zero if the distributions in the clamped and free-running phases are identical. Thus, the process should find the proper set of weights, provided that it does not get stuck in local minima points of G. This problem can happen whenever not enough samplings for the computations of the expected probabilities in the clamped and free-running phases is taken, or they are taken when the network is not yet at its thermal equilibrium. This represents a major drawback in the performance of the algorithm. It is not clear how fast one should allow the temperature to be reduced, nor it is clear how to determine that the network is already at its thermal equilibrium. In practice, only trial and error can provide the answers for these questions, and this can be a very time consuming process.

4 Feed Forward Networks

The network models covered in the previous sections were mostly used for storing and retrieving association lists. However, the same models could be used for training networks to perform classifications of vectors drawn from some input space. In order to do so, one would have to prepare an association list with the associated vectors corresponding to a classification of some sort on the key vectors. If enough samples of each class were stored, a "prototipical" representation of each class would be internally constructed in the connection weights of the network. Therefore, whenever the network were started at a state corresponding to a key vector to be classified, the network would converge and settle in the state corresponding to the prototipical class. However, the problems with storage capacity, and cross-talk among the various neurons in the network makes such approach not always feasible. For one thing, it cannot be guaranteed that the network will always converge to the "closest" state from its starting state.

In this section, a class of networks better suited for classification problems is introduced. The networks in this class will be mostly made of layers of neurons stacked one on top of the other, with the outputs of one layer feeding the inputs of the following layer. The inputs to the network are the vectors to be classified, and the outputs are the corresponding classes. The training procedures used here fall under the **classification paradigm**, and are mostly of the type of **supervised learning** algorithms. The reasons that makes these networks more appealing for classification problems will become evident from the discussion that follows.

4.1 The Single Layer Perceptron

A single layer perceptron is the simplest of all neural classifiers. In fact, it is implemented with a single neuron, which receives an input vector $X = [x_1, \ldots, x_N] \in R^N$, and produces an output $+1$, if the input vector belongs to a certain class A^+, or -1, if the input vector belongs to the class A^-. In practice, many units are used, organized in a linear array, each one being the recognizer of a different class. As usual, the neurons compute their total activation x_i according to equation (1), which is applied to equation (5) in order to produce the output y_i.

It is easy to see that the weight vector $W = [w_0, \ldots, w_N]$ defines a hyperplane dividing the input space in two. The output of the neuron will be 1 if the direction of the input vector X agrees with the direction of $\boldsymbol{W}$, and -1 otherwise. Rosenblatt [73] proposed a supervised learning algorithm that can be used to "train" the perceptron, and to find the "proper"

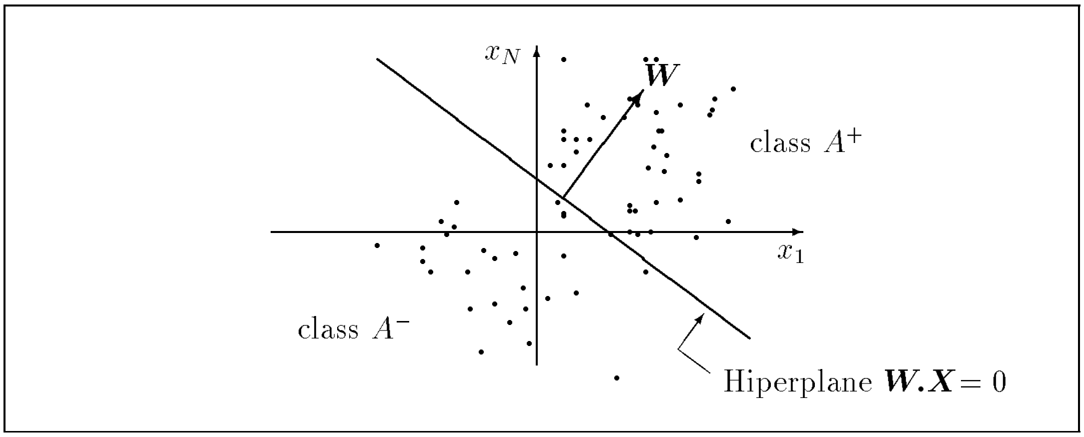

Figure 4: Perceptron pattern classification

weight vector that separates the two classes A^+ and A^- for the classification problem at hand.

Algorithm 4.1 (The Perceptron Learning Algorithm) *Starting the perceptron with a random set of connection weights, present all vectors from the input set, in order, one after the other. For every presentation, compare the output produced by the perceptron with the corresponding class of the input vector, and adjust the weights as follows:*

$$\Delta w_i = \left\{ \begin{array}{cl} 0 & \textit{if the output is correct,} \\ +\varepsilon x_i & \textit{if } X \in A^+ \textit{ and } y = -1, \\ -\varepsilon x_i & \textit{if } X \in A^- \textit{ and } y = +1. \end{array} \right\} \qquad \textit{for } i = 1, \ldots, N. \tag{27}$$

The process is repeated for all input vectors, until the weights no longer change.

Rosenblatt [73] proved that, if the classes A^+ and A^- are **linearly separable**, i.e., there exists a hyperplane which separates them, then the above algorithm will converge, and properly classify all input vectors in a finite number of steps. It is only required that a sufficiently small $\varepsilon > 0$ is used. However, if such hyperplane does not exit, the algorithm will oscillate for ever.

Gallant [26] proposed a modification to the above algorithm aimed at making it useful even when the classes could not be separated. Indeed, his algorithm will approximate the set of weights that will maximize the number of correct classifications performed by the perceptron. It can be seen as "drawing" the best possible hyperplane that separates the two classes, leaving the smallest number of members of each class in the "wrong sides". The **Pocket Algorithm** simply consists of keeping aside (in your pocket) the set of weights that has correctly classified the largest subset of the training data. Misclassifications are handled exactly like in the perceptron algorithm, but, correct classifications may change the "pocketed" set weights whenever the current set outperforms the last one saved. Gallant proved that, as the number of trials increase, the probability that the pocketed weights are optimal approaches 1. However, the number of trials may have to be very large for an optimal set of weights to be produced.

Minsky and Papert [66] have thoroughly studied the perceptron learning algorithm. They used this procedure in conjunction with a slightly different model: their "machine" accepted inputs from a rectangular retina array where different pixels could be set to "dark" or "white", i.e., 1, or 0. A preprocessing stage was in charge of computing arbitrary partial predicates on subsets of this retina, i.e., they could "see" an arbitrary subset of the retina, and output either "true" or "false", i.e., 1 or 0, depending on the values of the pixels in these subsets. These partial predicates could be any arbitrary functions chosen as they were needed. Several of such partial predicates were computed, and their outputs fed into a perceptron-like device, which was then subject to the perceptron learning algorithm. Minsky and Papert studied the classes of predicates that could be learned by this model, as function of the partial predicates. They hoped that the there could be some "gain" of power provided by the preprocessing state. Their main conclusion was that only predicates that were linearly separable could be efficiently learned. This result caused a major set back in the ongoing research of neural networks. Only after it was understood that the lack of processing power in their model was not intrinsic to neural networks in general, but to the fixed nature of the preprocessing stage, that researchers came back to the field. As we shall see later in this section, these difficulties can be overcome with more sophisticated learning procedures, coupled together with more general devices. For now, we resume our study of the single layer perceptron.

4.2 The Widrow-Hoff LMS Algorithm

A variation of the above algorithm is the **LMS** or **Widrow-Hoff Algorithm** [82, 83]. Here, an error measure is introduced in order to estimate the performance of the perceptron. Originally developed for the **Adaline**, a device similar to the perceptron but with a linear output function, this procedure can be used by the highly non-linear perceptron. The algorithm simply defines the error for each presentation as the difference between the desired output response and the total activation of the neuron. The LMS procedure is proved to minimize the sum of the squares of these errors, over the entire training set. Therefore, its name: **Least Mean Square**. The main advantage over the perceptron learning algorithm is that a solution will be found, even if the classes overlap, or are not linearly separable. However, in such cases, the error is not guaranteed to be small.

Algorithm 4.2 (Widrow-Hoff LMS) *Given m input vectors $X^p = [x_1^p, \ldots, x_N^p] \in R^N$, for $p = 1, \ldots, m$, train a single layer perceptron so that it "learns" to separate these vectors into two distinct classes A^+ and A^-. Use one neuron with N inputs, all starting with random weights. Present all vectors to the perceptron, in order, one after the other. At presentation k, the weight vector is W^k, the input vector is X^p, where $p = 1 + k \bmod m$, and the "error" is $E^k = T^p - (W^k)^T X^p$, where T^p is the target output for the input X^p, i.e. $+1$ if $X^p \in A^+$,* *or -1 if $X^p \notin A^i$. At every presentation, the weights are adjusted as follows:*

$$W^{k+1} = W^k + \frac{\alpha E^k}{\|X^p\|^2} X^p, \tag{28}$$

were $1 > \alpha > 0$ is the learning rate. This process is repeated until the changes in the weights are less than a prescribed precision.

Lippmann [60] showed how a single layer perceptrons can perform identically to a Maximum Likelihood Gaussian Classifiers like those in [18]. Indeed, if μ_i^- and σ_i^- are the mean and standard deviation of the values of the i^{th} coordinate of X^-, for all $X^- \in A^-$, and μ_i^+ and σ_i^+ the mean and standard deviation of the values of the i^{th} coordinate of X^+, for all $X^+ \in A^+$, and $\sigma_i = \sigma_i^+ = \sigma_i^-$, then it is enough to "set" $w_i = 2(\mu_i^+ - \mu i^-)/\sigma_i^2$, with $w_0 = \sum_{i=1}^m ((\mu_i^+)^2 - (\mu i^-)^2)/\sigma_i^2$. The perceptron with the above weights performs identically to a maximum likelihood Gaussian classifier for the two given classes. If the distributions of vector of the classes A^+ and A^- are indeed Gaussian, then the perceptron learning algorithm will "approximate" the above weight vector.

But, in truth, the perceptron learning algorithm and the LMS algorithm should perform better than Gaussian classifiers. The lack of any underlying assumption about the distribution of the patterns in each class, unlike the Gaussian classifiers, allows the perceptron learning algorithm to "discover" irregularities in these distributions, and therefore adapt to them. This hypothesis was first raised by Lippmann, and fully supported by the results of his experimental work [61]. In addition, Lippmann [60] also suggested that M neurons can be trained to discriminate patterns from M distinct classes. The idea is to use the total activation of the neurons as a measure of the likelihood that an input pattern belongs to a specific class. The best "match"can be identified by another set of M neurons with thresholding outputs, each one of these fed by the total activation of each one of the previous neurons, and also by negative feedback connections from all the other $M-1$ thresholding neurons [3]. With the proper selection of the feedback weights, only one neuron should come up active, thus identifying the class that the input vector most likely belongs to.

4.3 The Multi-Layer Perceptron

The problems with the LMS and the perceptron learning algorithms are not due to the algorithms themselves, but due to the machines' simplicity and lack of power as computing devices; they are limited to "separate" the input space in two halves, and nothing else. Although many problems can be characterized that way, most problems require that more complex decision regions be identified. The most famous example, the two-input XOR problem, requires at least two lines cutting the two-dimensional state space, with the points at the coordinates $(0,1)$ and $(1,0)$ falling in the region between these two lines, and the points at coordinates $(0,0)$ and $(1,1)$ falling outside that region [66]. Such decision region can never be learned by a single layer perceptron.

[3] This final stage configuration is popularized by the name of "winner-takes-all".

The solution is, simply enough, to stack several layers of these devices, connecting the outputs of one layer to the inputs of the following layer, and adjusting the connection weights in such a way that more complex regions can be approximated, in the higher layers, by unions and intersections of the spaces defined by the units in the lower layers. This arrangement, known as **multi-layer feed-forward network**, has been proved successful in many problems of signal processing [3, 13, 16], pattern recognition [20, 40], and time series analysis [19, 53, 80]. Indeed, it is easy to show that a two-layer perceptron, i.e., a device with one **hidden layer** and one **output layer**, can identify any convex region, provided that "enough" units are present in the hidden layer, and that the weights are properly adjusted. If one more hidden layer is added, we have a three-layer perceptron, potentially capable of identifying any arbitrary region in the state space [74, 82, 60].

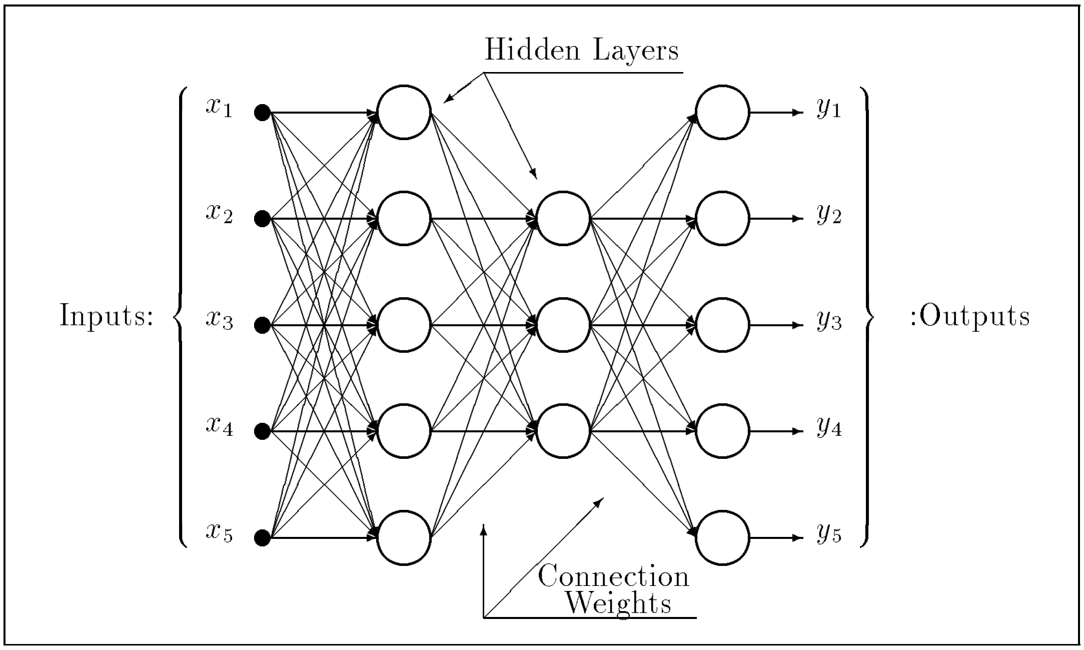

Figure 5: Typical 3-layer feed-forward network

In order to use these networks, for any given classification task, one has to face two problems: First, one has to determine how many units should be present in every hidden layer. To date, there is no exact solution to this problem. It can only be said that, for a three layer perceptron with N inputs and 1 output, the number of neurons in the second hidden layer, i.e., the layer closest to the output, need not be larger than the number of disconnected regions in the distributions of the classes being identified. If the distribution is convex and connected, then the second layer can be eliminated entirely. The first hidden layer will need $N + 1$ neurons to "enclose" any single "input vector" in a convex region. Given that not all regions are necessarily convex, this is usually an over-estimation, but may also not be enough. If these regions are complex, with many edges, it may be required many more units per region. Lippmann[60] suggests that, in practice, "the first layer should provide 3 or more edges for each convex region identified by every second layer unit". Therefore, the number of units it the first layer should be more than 3 times the number of units in the second hidden layer. These figures are heuristics only. Note, however, that the number of neurons in each layer should not be under-estimated, otherwise the network will not be able to build a complete representation of the distributions of the input vectors. By contrast, it should not be over-estimated either, otherwise the network may not be able to build a compact representation of the relationships among the input patterns, and may end-up behaving as a simple look-up table. In reality, only trial and error can determine the best configuration.

The second problem is how to adjust the weights of a multi-layer perceptron, and "program" it the for task at hand. This is the **credit assignment problem.** Given a pair of vectors, one representing the input to the network, the other representing the target output, it is very easy to compute an error measure between the actual network output and the target output. In order to reduce this error, the connection weights have to be modified. However, it is unknown which units in the hidden layers should have their weights modified; and it is unknown by how much such weights should be modified. In short, there is no way of telling which units should be "blamed" for the errors in the output layer.

Widrow proposed an algorithm aimed at solving this problem. His algorithm embodies what he calls the "least disturbance principle", which essentially says to modify the units whose actual response is most undecided. Working from the input layer and moving towards the output layer, all units whose total activity were closest to zero should have their weights modified "a little bit", producing a small reduction in the output error. The algorithm states that units should then be modified in pairs, triples, and so on, until a minimal error is produced. Although it is stated that the algorithm shows promising results in the domains for which it has been applied, it has not been formally described by the author, and, as stated, will clearly lead to combinatorial explosion. Refer to [83] for a more detailed discussion.

It must be stressed that the power of the multi-layer perceptron is essentially due to the non-linear behavior of its neurons. If a linear output function were used, e.g. adalines, then there would be no advantage in using more layers, for one could always find an equivalent single-layer network to perform the same task as the multi-layer linear net. However, the problem of adjusting the weights of threshold-like units has an added difficulty: it is not amenable to differentiation; therefore, it is impossible to use standard minimization techniques for solving it. A solution, accepted by the entire research community, is to use a non-linear, smooth, monotonically increasing **(sigmoid)** neuron output function, like equations (2) and (3).

Finally, it was assumed in the previous discussion that a multi-layer perceptron had only one output unit. In truth, this is not a restriction. In the general case these networks produce a mapping from one multi-dimensional vector space into another multi-dimensional space space. More specifically, p pairs of vectors (R^k, S^k), with $R^k = [r_1^k, \ldots, r_m^k]^T$, and $S^k = [s_1^k, \ldots, s_n^k]^T$, $k = 1, \ldots, p$ are given. The vectors R^k are the inputs to the network, which in turn produces, for each input R^k, a corresponding output $\boldsymbol{Y}(\boldsymbol{W}, R^k) = [y_1^k, \ldots, y_n^k]^T$, where $\boldsymbol{Y}(\boldsymbol{W}, \cdot)$ is the output of the network parametrized by its connection matrix $\boldsymbol{W}$. Observe that the number of layers in the network, and the number of neurons per layer, are implied by the dimensions and structure of W. If the error incurred by the network is defined as the mean square error E, then there exist a multi-layer fedd-forward neural network **W** such that

$$E = \min_W \frac{1}{2} \sum_{k=1}^{p} [\boldsymbol{Y}(\boldsymbol{W}, R^k) - S^k]^T [\boldsymbol{Y}(\boldsymbol{W}, R^k) - S^k]. \tag{29}$$

The **Back-Propagation** algorithm, the most popular procedure to compute the the non-zero elements of **W** which minimize the above error, was invented by Paul Werbos [79] in 1974, and re-invented by David Parker [68] in 1982, being popularized by Hinton and Rumelhart [74]. It is a gradient technique that slowly moves the weights towards their "correct" value.

4.3.1 Back-Propagation Learning Algorithm

Assume that the multi-layer network is composed of M layers, such that the outputs of one layer are connected to the inputs of the following layer. Each layer contains N_l neurons, for $l = 1, \ldots, M$. The neurons in the first hidden layer contains $N_0 = m$ inputs, and every such neuron receives the same input vector R. The output layer contains $N_M = n$ neurons. The total activation of the i^{th} neuron in layer l, denoted by $x_{l,i}$, is compute as follows:

$$x_{l,i} = \sum_{j=1}^{N_{l-1}} w_{l,i,j} y_{l-1,j} + w_{l,i,0}, \qquad i = 1, \ldots, N_l,\ l = 1, \ldots, M, \tag{30}$$

were $w_{l,i,j}$ is the connection weight from neuron j in layer $l-1$ to neuron i in layer l, and $w_{l,i,0}$ is the threshold of the neuron. The output of the neuron will be a continuous, differentiable, monotonically increasing (sigmoid) function of its total activation. With neuron outputs in the range $[0, 1]$, this function usually is:

$$y_{l,i} = f(x_{l,i}) = \frac{1}{1 + \exp^{-kx_{l,i}}}, \qquad k > 0. \tag{31}$$

At the input layer, the network is fed with the input patterns. Therefore, when pattern k is presented to the network, $y_{0,i}^k = r_i^k$, for $i = 1, \ldots, N_0$. The neurons in the first hidden layer will compute their output and propagate their outputs to the layers above. The output of the network will be given by the neurons at the very last layer, i.e, $y_{M,i}$, for $i = 1, \ldots, N_M$. The goal is to find the best set of weights **W** such that the following error measure is minimized:

$$E = \min_W \sum_{k=1}^{p} E^k = \min_W \frac{1}{2} \sum_{k=1}^{p} \sum_{i=1}^{n} (y_{M,i}^k - s_i^k)^2. \tag{32}$$

The partial derivative of the error with respect to the weights of the neurons in the output layer is

$$\frac{\partial E}{\partial w_{M,i,j}} = \sum_{k=1}^{p} \frac{\partial E^k}{\partial y_{M,i}} \frac{dy_{M,i}^k}{dx_{M,i}} \frac{\partial x_{M,i}^k}{\partial w_{M,i,j}} = \sum_{k=1}^{p} \left(y_{M,i}^k - s_i^k\right) \frac{dy_{M,i}^k}{dx_{M,i}} y_{M-1,j}^k. \tag{33}$$

Note that $dy_{l,i}^k / dx_{l,i} = f'(x_{l,i}^k)$. Next, define $\delta_{M,i}^k$ as

$$\delta_{M,i}^k = \left(y_{M,i}^k - s_i^k\right) f'\left(x_{M,i}^k\right), \qquad \text{for } i = 1, \ldots, N_M. \tag{34}$$

Therefore, the partial derivative of the error with respect to the weights in the output layer can be written as

$$\frac{\partial E}{\partial w_{M,i,j}} = \sum_{k=1}^{p} \delta_{M,i}^{k} y_{M-1,j}^{k}. \tag{35}$$

If the partial derivative $\partial E^k / \partial y_{l,i}$ could be computed for the hidden layer, then an equation similar to (33) would be derived for every weight in the network. For that matter, observe that one neuron in a layer l can only affect the error in the output layer via its effects on the layer $l+1$ immediately above. Therefore,

$$\frac{\partial E^k}{\partial y_{l,i}} = \sum_{j=1}^{N_{l+1}} \frac{\partial E^k}{\partial y_{l+1,j}} \frac{dy_{l+1,j}}{dx_{l+1,j}} \frac{dx_{l+1,j}}{dy_{l,i}} = \sum_{j=1}^{N_{l+1}} \frac{\partial E^k}{\partial y_{l+1,j}} \frac{dy_{l+1,j}}{dx_{l+1,j}} w_{l+1,j,i}. \tag{36}$$

Inspecting equations (33),(36) and (34), $\delta_{M,i}^{k}$ is generalized for the other layers by defining $\delta_{l,i}^{k}$ as

$$\delta_{l,i}^{k} = f'\left(x_{l,i}^{k}\right) \sum_{j=1}^{N_{l+1}} \delta_{l+1,j}^{k} w_{l+1,j,i}, \qquad \text{for } l = M-1, \ldots, 1;\ i = 1, \ldots, N_l. \tag{37}$$

Here is exactly where the name **back-propagation** becomes clear. The errors at one layer are propagated backwards into the previous layer according to equation (37). Therefore, combining equations (33) and (37), the partial derivative of the error with respect to any of the weights in the hidden layers can be written as follows:

$$\frac{\partial E}{\partial w_{l,i,j}} = \sum_{k=1}^{p} \delta_{l,i}^{k} y_{l-1,j}^{k}. \tag{38}$$

Finally, the last step is to "change" the weights by a small amount in the direction which causes the error to be reduced, i.e., the direction opposite to the partial derivative defined by equation (38). The following update procedure is introduced:

$$w_{l,i,j}^{(t+1)} = w_{l,i,j}^{(t)} - \alpha_t \frac{\partial E^{(t)}}{\partial w_{l,i,j}}. \tag{39}$$

The above procedure is shown to converge provided that α_t, the **learning rates** are chosen such that $\sum_{t=1}^{\infty} \alpha_t = +\infty$ and $\sum_{t=0}^{\infty} \alpha_t^2 < \infty$. These conditions are necessary for dumping oscillations that may occur around the optimal solution. In particular, they are satisfied by the choice of $\alpha_t = \alpha_0 / t$, for some $\alpha_0 > 0$, but convergence may become very slow [21, 47, 18].

4.3.2 Increasing the rates of convergence

The previous formulation is known as the **batch version** of the back-propagation. The weights are adjusted following the steepest descent path in the error surface defined by equation (32). However, as pointed out before, the convergence rate of the algorithm may be very slow.

In order to significantly improve performance of the algorithmm, two modifications are introduced in equation (39). The first modification is the use of a constant learning rate α_t.

If a small learning rate, usually in the range $0.1 < \alpha < 0.9$, is used, the algorithm still approximates the steepest descent path, and should produce a reasonably good solution. Considering that the algorithm should run only for a finite number of steps, and that only an approximation of the solution is being sought, this modification can be safely used. The other modification is aimed at reducing the effects of a very shallow error surface. When the weights are close to the solution, but not quite there yet, the term $\partial E/\partial w_{l,i,j}$ becomes very small, causing only insignificant changes in the weights. The solution is to add a momentum term $\beta\Delta w$ to the equation (39), which then becomes

$$\Delta w_{l,i,j}^{(t+1)} = -\alpha(1-\beta)\frac{\partial E^{(t)}}{\partial w_{l,i,j}} + \beta\Delta w_{l,i,j}^{(t)}, \tag{40}$$

where β is the amount of momentum that should be added. In practice, values of β should range from 0 to 0.4, to even larger values if the error surfaces are very shallow, and convergence becomes very slow. The effect of this momentum term can be best understood by rewriting equation (40) as follows:

$$\Delta w_{l,i,j}^{(t+1)} = -\alpha(1-\beta)\sum_{k=0}^{t}\beta^k\frac{\partial E^{(t-k)}}{\partial w_{l,i,j}}. \tag{41}$$

The change to the weights becomes the exponentially weighted sum of the current and past error's partial derivative with respect to the weight being changed. The momentum term β is the base and t is the exponent. When consecutive derivatives possess the same sign, the exponentially weighted sum becomes large in magnitude, and the weight is adjusted by a large amount. Similarly, when consecutive derivatives possess opposite signs, this sum becomes small and the weight is adjusted by a small amount.

It is stressed that the above modifications are widely used by those running simulations with backpropagation, but they are only based on heuristics. They are not problem free; One limitation is that the amount of weight adjustment allowed becomes bounded. For instance, if the error derivative with respect to some weight is constant equal to 1, the exponential sum will be limited by $1/(1-\beta)$. The weight will never be allowed to change by more than this amount. Another limitation is that the sign of the exponential weighted sum may be different than the sign of the error derivative, thus causing the weight to change in the wrong direction. Furthermore, experimentation is the only method available to determine the best values for these parameters, and thse values will be different for different problems.

The motivations for these modifications, as well as their performance improvements have been fully studied by Jacobs [41]. Indeed, he introduces another heuristic aimed at solving the above limitations. This heuristic will cause the learning rate to increase by a constant term η, whenever the current derivative of the error has the same sign of the exponential average of the current and past derivatives. If the signs become different, then the learning rate is decremented by an amount proportional to its size. In mathematical terms, the weights are changed according to equation (39), but α_t undergoes the following adaption process:

$$\Delta\alpha(t) = \begin{cases} \eta & \text{if } \overline{\delta}(t-1)\delta(t) > 0); \\ -\phi\alpha(t) & \text{if } \overline{\delta}(t-1)\delta(t) < 0); \\ 0 & \text{otherwise.} \end{cases} \tag{42}$$

where

$$\delta(t) = \frac{\partial E^{(t)}}{\partial w_{l,i,j}}, \tag{43}$$

and

$$\overline{\delta}(t) = (1 - \theta)\delta(t) + \theta\overline{\delta}(t-1). \tag{44}$$

where η, ϕ, and θ are positive small constants. Note that $\overline{\delta}$ is the exponential average of the current and past error's partial derivative, with base θ and time for its exponent. The effect of this heuristic is to increase the learning rate by linear increments η, but to decrease it exponentially by fractions ϕ of its current size. This prevents the learning rate to increase too fast, but decreases it fast enough whenever the error derivative changes sign. Also, the learning rate is kept always positive, thus ensuring that the weights are changed always in the correct direction. This heuristic is very appealing to whoever has tried to use the original back-propagation algorithm and experienced the frustration of watching the changes in the weights becoming smaller, and smaller as the solution is approximated. The price to pay for using it, is the increased time and storage requirements. As it should be clear by now, every weight will have its own learning rate, thus twice as much memory will be required. Refer to [41, 68, 69] for a more detailed discussion on the subject.

Yet another modification finds its motivation in the Widrow-Hoff LMS algorithm presented earlier. Known as the **on-line version** of backpropagation, it consists of dropping the summation over all patterns in the error derivative of equation (38), and changing the weights at every single pattern presentation. Although appealing at first, care should be taken when adopting this modification. If the training set is not regular enough, or if some patterns are "strong exceptions" to some hidden formation rule in the training set, then the on-line version of backpropagation may easily oscillate, and never find a good solution. For small problems requiring small number of units and a small training set, it is viable and faster alternative.

4.3.3 Avoiding local minima

The generality of the back-propagation algorithm has encouraged its use in a wide variety of domains. Indeed, it has been successfully applied to problems in Speech Synthesis [75], Speech Recognition [61, 20], Image Data Compression [86, 16], Semantic Networks [74], Natural Language Processing [3] and many others. However, its good performance is somewhat surprising, considering that it is a gradient technique, plagued by the possibility of getting stuck at local minima. Indeed, for some time, it was thought that the local minima didn't exist in multi-layered perceptrons. Baldi and Hornik [7] proved that these networks, when using linear units, had only one global minima, and that all other critical points were just saddle points. Almost simultaneously, in 1988, Hecht-Nielsen [32] proved the existence of local minima in networks with sigmoid units. Nevertheless, the problems cause by these local minima seem not to be of much significance, since that the initial conditions for the network can be changed, and almost invariably a set of working parameters can be found. Nevertheless, the trial and error process for finding a good set of parameters can be very time consuming.

A more critical problem is the poor scaling performance of the algorithm. Whenever the training set is small, and the network doesn't have many units, the algorithm performs well. Increasing the size of the network, and/or the training set, causes the convergence time to increase dramatically. Hinton [74] states that the learning time for a network simulated on serial hardware is $O(M^3)$, where it is assumed that the network has $O(M)$ weights, and the training set is also of size $O(M)$. In parallel hardware, this time could be reduced to $O(M^2)$, if one processor where given to each connection in the network. Although these are empirical observations, it seems unlikely that much better improvements will be found. Whenever the training set has many disconnected regions, the amount of data and time required to train the network will be large. Whenever dealing with large problems, the best approach is to decompose the problem into smaller sub-problems, and then implement each part in separate sub-networks that can be trained separately. This reduces the complexity of the distribution of the data used to train each piece, and the overall training time. After training, the pieces are put together and the whole system can be used for whatever task it was designed.

4.4 Computing with Feed-Forward Networks

Multi-layer feed-forward networks have the ability to produce arbitrary mappings from one input space to the output space. This can be interpreted, in light of the back-propagation algorithm, as a generalization of traditional **regression analysis** [18, 79]: the main goal is to minimize an error measure of an aproximation for a mapping from one vector space into another. However, their ability to "create" a much richer and compact internal representation of the relationships hidden in the data provided by the training set deserves credit. This is accomplished by the units in the hidden layers, which strive to "find" the best way to organize their weights, and, sharing responsibility, encode the input data in ways that can be decoded by the units in the layers above, up to the output layer which then produces the desired output.

As an example, consider a network with 8 inputs, one hidden layer with 3 units, and 8 outputs. The input data consists of 8 different tuples, each with 8 binary digits, such that in every tuple only one digit is 1, and all the others are zero. The output data is forced identical to the input, so that the network has to "learn" the identity function. If the units are given high gain, they will behave as if they were "almost" binary switches, producing outputs either very close to zero, or very close to one. After training the network with the back-propagation algorithm, the hidden units will have to find the "best" way to encode the input, which is then decoded by the output units. It is easy to see that the hidden units can only "find" the binary encoding of a 3-bit binary number, since that they will not be allowed (due to the high gain) to assume many values in between 0 and 1. If the high gain restriction is dropped, a richer representation may be "created", with the units assuming values other than just 0 and 1, thus allowing many more tuples to be learned.

Hinton [34] studied this property with a network trained to establish "family relationships" using data drawn from two typical families. After the training was over, analysis of the behavior of the hidden units demonstrated their ability to "generalize" for relationships never explicitly present in the training set, but embedded in the various data samples. Zipser

[20] also exploited this property with a network trained to uncover the hidden structure of speech signals. He demonstrated how the hidden units could become sensitive to sound units such as vowels and consonants, without being provided with any clues about them. Other examples are the work by Allen [3], Lippmann [61], and the PDP group in general [74]. Of particular theoretical interest is the work developed by Baldy and Hornik [7]. They proved that the hidden units in networks with linear units, and only one hidden layer, when trained to perform the identity function over some multi-dimensional training set, learned to compute the the coordinates of the input vectors along the first eigenvectors of the auto-correlation matrix of the training set. If some transfer function other than identity were to be learned, the hidden units activities would correspond to the projection of the input vectors over the **principal components** of the output vectors distribution. Although it is unknow how this result can be generalized to networks with non-linear output functions, as a concluding remark, the recent work by Funahashi [25] proves that any continuous mapping can be approximately realized by multi-layer neural networks with at least one hidden layer of units with sigmoid output functions.

5 Unsupervised & Competitive Learning

One very important limitation of multi-layer networks, not mentioned in the previous section, is the requirement that the training set contains both input and output data. Indeed, the previous models belong to the class of **supervised learning** algorithms. For multi-dimensional regression problems this is hardly a limitation. But for many other problems, the output data set may be not known in advance. This is particularly true for classification problems where the goal is to determine the best way to partition some input space into a finite number of unknown classes. The models and algorithms in this section are aimed at solving this problem. For that reason, they are called **unsupervised learning algorithms**, and, in many instances, **competitive learning algorithms**.

5.1 Unsupervised Back-Propagation

Zipser and Cottrell [20, 16] proposed a simple scheme that allows a multi-layer network to be trained with the back-propagation algorithm, and to create its internal data representation without the need of a supervising output data set. A multi-layer network with equal number of inputs and outputs is used. The hidden layers should have fewer units than the size of the training set, and usually they will have fewer units than the output layer. The networks are trained to perform the identity function, reproducing in the output layer whatever signals are presented to the inputs. As in the 8-3-8 encoder of section 4.5, the hidden units will have to find a compact way of encoding the input vectors, since that each one of them will be involved in representing many input data vectors. As mentioned before, if the units where linear, this scheme would be equivalent to performing a reduction in the dimensionality of the input space, similar to that of principal component analysis [7]. But, since the units are non-linear, much richer representation will emerge. Zipser demonstrated that such networks can be used to label pre-segmented tokens of speech with accuracies of up to 95%. These networks trained to produce the identity mapping of its input signals were able to develop *"rich internal representation of speech signals, including traditional distinctions such as vowels and consonants, as well as other novel and non-standard features [20]."* In many situations, the output layer can be removed after the training process is completed, and the resulting network will provide a self-supervised classification of the input patterns.

5.2 G-Maximization Algorithm

A rather frequent problem in signal processing is the determination of whether several data channels are independent of one another, or whether they obey some hidden mechanism that causes the data flowing through them to be some how correlated. Based on the principle that a neuron should respond to patterns that occur more often than would be expected if the activities of the individual input lines were assumed independent, Pearlmutter and Hinton [70] proposed an algorithm to adapt the weights in such a way that causes the neuron to detect higher order correlations among its signals. As usual, the neuron computes its total activity x as the inner product of the input vector R and the weight vector W. Again, the output will be a sigmoid function of its total activity, i.e., $f(x) = 1/(1 + e^{-x})$. Assuming that the input patterns follow the distribution P, the expected output $\langle y \rangle$ of the neuron is

$$\langle y \rangle = \sum_{\alpha} P(\alpha) f(x^{\alpha}) \tag{45}$$

By recording the first order statistics p_i of each input line i, and assuming independence among the lines, one would be led to believe that the distribution of the input vectors followed the distribution P', given by

$$P'(\alpha) = \prod_i \begin{cases} p_i & \text{when } r_i^{\alpha} = 1 \\ 1 - p_i & \text{when } r_i^{\alpha} = 0 \end{cases} \tag{46}$$

Assuming that the input patterns follow the distribution P', the expected output of the neuron then becomes $\langle y \rangle'$, which is given by

$$\langle y \rangle' = \sum_{\alpha} P'(\alpha) f(x^{\alpha}) \tag{47}$$

The discrepancy among $\langle y \rangle$ and $\langle y \rangle'$ is indicative of the different assumptions about the independence of the input signals. By assuming that the expected output was $\langle y \rangle'$, and then realizing that it actually is $\langle y \rangle$, one gains information about the real distribution P of the input vectors. This gain of information [52] can be measured by the quantity

$$G = \langle y \rangle \log \frac{\langle y \rangle}{\langle y \rangle'} + (1 - \langle y \rangle) \log \frac{(1 - \langle y \rangle)}{(1 - \langle y \rangle')} \tag{48}$$

The goal is to make the neuron as good a detector as possible. So, its weights are updated in a way to maximize the information gain about the real distribution of the input patterns.

In order to do that, one computes the partial derivatives

$$\begin{aligned}\frac{\partial G}{\partial w_i} &= \frac{\partial G}{\partial \langle y\rangle}\frac{\partial \langle y\rangle}{\partial w_i} + \frac{\partial G}{\partial \langle y\rangle'}\frac{\partial \langle y\rangle'}{\partial w_i} \\ &= \left(\log\frac{\langle y\rangle}{\langle y\rangle'} - \log\frac{1-\langle y\rangle}{1-\langle y\rangle'}\right)\sum_\alpha P(\alpha)f'(x^\alpha)r_i^\alpha + \\ &\qquad \left(\frac{1-\langle y\rangle}{1-\langle y\rangle'} - \frac{\langle y\rangle}{\langle y\rangle'}\right)\sum_\alpha P'(\alpha)f'(x^\alpha)r_i^\alpha \end{aligned} \tag{49}$$

and then uses any hill climbing method (e.g. $w_i^{new} = w_i^{old} + \varepsilon\partial G/\partial w_i$) to maximize the quantity G. Note that the algorithm requires the sampling of the distributions in two phases; In the first phase, which is called the "structured phase", the inputs are sampled in order to compute $\langle y\rangle$, which assumes the high order correlations are present in the input patterns; In the second phase, the "unstructured phase", the input lines are assumed to be statistically independent, and the sampling is used to compute $\langle y\rangle'$ for each one of the unit's weights.

The power of the algorithm lies in its ability to adapt the units without having to keep track of the entire history of the input sampling. Instead, only the expectations $\langle y\rangle'$ are kept for each weight. Through computer simulations, Pearlmutter and Hinton demonstrated that the above algorithm will cause neurons to develop into on-center off-surround [29, 56] detectors, whenever the training data consists of uncorrelated white noise. Linsker [56] has shown how this property can be used to develop efficient "line orientation detectors" that could be used in artificial vision machinery. However, the algorithm, as presented, deals only with one single unit, and if many units are used, it is possible that they will develop into detectors for the same features in the input space. Pearlmutter and Hinton proposed the use of lateral inhibition as a way to prevent the units to become identical detectors. Simple to implement, this approach also ensures that the outputs of the units will be uncorrelated.

5.3 Order Preserving Feature Maps

It has been established that in many areas of the brain neocortex the neurons are organized in ways that reflect some physical characteristic of the signals stimulating them [55]. For instance, in the visual and somatossensory cortex, the response signals are obtained in the same topographical order in which they were received at the sensory organs, although the fibers transporting these signals do not maintain this ordering. Inspired by that, Kohonen [47] developed a self-supervised competitive learning algorithm capable of producing order preserving maps of signals coming from multi-dimensional vector spaces. By order preserving, it must be understood that the neurons in Kohonen's model will become "specialized" to detect arbitrary features existing in the set of input vectors and, at the same time, maintain the same topological ordering which is present in the metric relations of the input vectors. In simpler terms, assuming some predefined spatial neuron distribution, the neurons located close to one another will become sensitive to patterns "similar" to one another, and neurons far apart will become sensitive to patterns "distant" from one another.

Since the neurons are usually organized in two-dimensional arrays, Kohonen's model will develop mappings that preserve topological relations, while performing a dimensionality reduction of the input space. The computations performed by the network can be compared to that of a vector quantizer with as many distinct representatives vectors as the number of neurons in the network.

The model exploits the fact that, within certain areas of the cerebral neocortex, the neurons follow a very well defined interconnection pattern: in a planar distribution, the neurons will have short-range strong lateral excitory connections reaching up to a radius of 50 to 100μm, and, beyond that distance, inhibitory lateral connections reaching up to a radius of 200 to 500μm. Kohonen describes the strength of the connections varying with distance, following a function with the form of a Mexican hat [47]. Besides, all neurons are fed by a bundle of fibers coming perpendicular to the plane defined by lateral connections, thus providing the network with the input signals from the external world. This structure will cause "clustered" activities among the neurons, each cluster representing a different class of input signals. A simple synaptic adaption process is in charge of "discovering" which features in the input signals should be represented in the activity clusters, while "preserving" the order relationships of these features along the spatial distribution of the neurons.

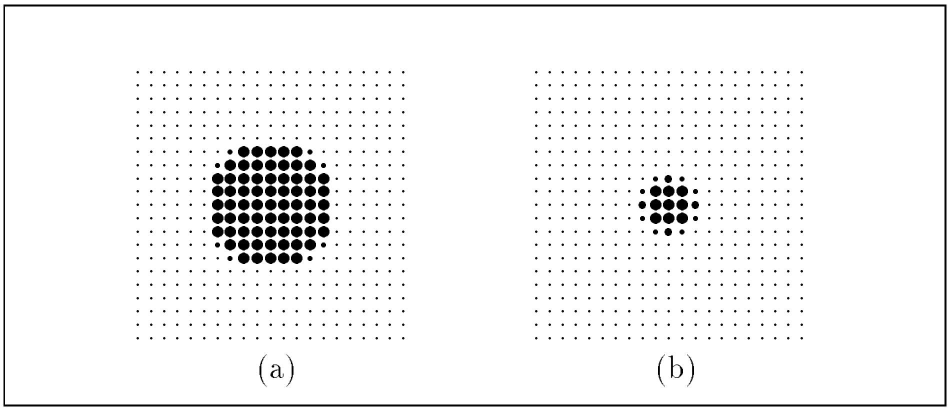

Figure 6: Effects of bubble activity formation in a two dimensional grid of neurons. (a) Stronger excitory lateral connections. (b) Stronger inhibitory lateral connections.

Throughout the development of his algorithm, Kohonen makes several simplifications in the model describing the interactions among the neurons, thus making it amenable to computer simulations, while maintaining the important "topology preserving" features. One of such simplifications is the assumption that "bubbles" of activities, or clusters, will indeed be formed. This eliminates the need for simulating the effects of the lateral interconnections. It is enough to define a neighborhood around each neuron, and assume that the clusters of activity will always happen within the limits of these neighborhoods. Several alternatives are proposed for the shape of the neighborhoods, which can be rectangular, hexagonal, circular, and so on. However, for best convergence results, during the training process, they should slowly decrease with time.

During the training process, patterns are presented to all neurons in the network. A search for the neuron with the largest activity is performed and its neighborhood is identified.

Next, all neurons within this neighborhood have their weights adjusted by a small fraction proportional to their distance from the input pattern. This adjustment is done in order to further increase their activity, thus their sensitivity to the pattern just being presented, but it should be clear that the neuron with maximum activity will still be the "winner" after the adaption takes place. This process is repeated for all patterns until no more changes in the weights takes place. In order to guarantee convergence, a gain term multiplies the fraction by which the weights are adjusted. This term must be a decreasing function of time, and usually is taken as a linearly decreasing function, such that the "learning" process stops when it this function reaches zero. What follows is a more explicit, but informal, description of Kohonen's algorithm.

Algorithm 5.1 (Feature Maps) . *Assume N neurons organized in a rectangular grid. All neurons have M inputs, with associated weights $W_i = [w_{i1}, \ldots, w_{iM}]^T$, for $i = 1, \ldots, N$; all neurons receive the same input vector $R^k = [r_1^k, \ldots, r_M^k]^T$, for $k = 1, \ldots, p$.*

- **Step 0:** *Initialize all weights to small random values. Initialize the set $Ne(i)$, i.e., the neighborhood around each unit $i = 1, \ldots, N$.*
- **Step 1:** *Present new input vector R^t to all neurons. Compute distances d_i between the input vector, and each neuron's weight vector, according to*

$$d_i = \sum_{j=1}^{M} \left(r_j^{(t)} - w_{i,j}^{(t)} \right)^2, \qquad i = 1, \ldots, N \tag{50}$$

- **Step 2:** *Select neuron i with the smallest distance d_i.*
- **Step 3:** *Update weights of the neuron i and of all neurons k, such that $k \in Ne(i)$, i.e., the neighborhood around neuron i. The weights are updated using the following expression:*

$$w_{k,j}^{(t+1)} = w_{k,j}^{(t)} + \eta^{(t)} \left(x_j^{(t)} - w_{k,j}^{(t)} \right), \qquad \text{for all } k \in Ne(i), \text{ and } j = 1, \ldots, N, \tag{51}$$

  where $\eta^{(t)}$ is a decreasing gain term, such that $0 < \eta^{(t)} < 1$.
- **Step 4:** *Update $Ne(i)$, for $i = 1, \ldots, N$, such that the neighborhoods around each neuron slowly shrink with time.*
- **Step 5:** *Repeat steps 1 to 4 for all input patterns, until the weights no longer change.*

Note that the algorithm does not specify how the neighborhoods around each neuron should "shrink" with time, nor does it specify the best choice for the decreasing function $\alpha(t)$. Indeed, Kohonen states that the proper choice for these parameters can only be determined by experience, and that it is desirable to perform the training in two phases: initial formation of correct order, and final convergence of the map. In the first phase, larger neighborhoods are allowed, as well as larger values values for the gain term. The second phase, which may take from 10 to 100 times as many steps as the first, should employ smaller neighborhoods and smaller values for the gain term.

Kohonen shows that after the training process is completed, the weights specify the centers of clusters which sample the input space, and that the point density function of these centers tend to approximate the probability density function of the input space [47]. He also developed a variety of applications for the algorithm. A rather interesting one, the **phonetic typewriter** [48, 49], demonstrates the ability of the network to efficiently map spectral speech information directly into phonemes. Other applications includes the **Feeler Mechanism** in which a "environment map" is constructed from the angle sensors coming from two arms used to explore the "world". This example is rather important in that it sets ground for the development of networks capable of using multi-modal information to produce invariant maps. In fact, this result is extended by the work of Ritter et al [72] in which a topology preserving map is used for training a simulated robot arm in the task of visuo-motor coordination. Another not so typical application is the **Counterpropagation Network** developed by Hecht-Nielsen [31], which uses a combination of Kohonen's learning algorithm and Grossberg's **outstar units** in order to produce a statistically optimal self-programming lookup table.

5.4 Adaptive Resonance Theory

The attempt to understand how a learning system can be designed to remain adaptive in response to new significant events, and yet remain stable in response to irrelevant events, has motivated Grossberg and Carpenter in the development of the Adaptive Resonance Theory [28, 11, 12], or **ART** for short. The above problem, named the **stability-plasticity dilemma**, asks for how can a system preserve its previously learned knowledge while continuing to learn new things, and for what prevents the new knowledge from erasing previously learned things. Moreover, it asks for the mechanism that causes the system to switch in between the two modes, i.e., how the systems decides when to learn from new events, and when to ignore irrelevant information.

The Adaptive Resonance Theory introduces the framework under which systems that embody "solutions" to the above dilemma have been constructed [28, 29, 11, 12, 15, 14]. ART models employ a competitive learning paradigm similar to Kohonen's feature maps, but have added mechanisms for controlling the "orientation" and "vigilance" of the learning process. The models are described by a system of parameterized differential equations [11] whose study is beyond the scope of this article. Instead, the intuition for its behavior is now presented.

Two sets of neurons F_1 and F_2 are interconnected through a set of adaptive weights. Bottom-up connection weights exist from the outputs of neurons in F_1 to the inputs of neurons in F_2, and are called the long-term memory (LTM) traces. Top-down connection weights exist from the outputs of the neurons in F_2 to the inputs of the neurons in F_1, and are called the learned expectation memory traces. The activity of the neurons in F_1 is the result of the environmental stimulus coming from the outside of the system, and the signals coming top-

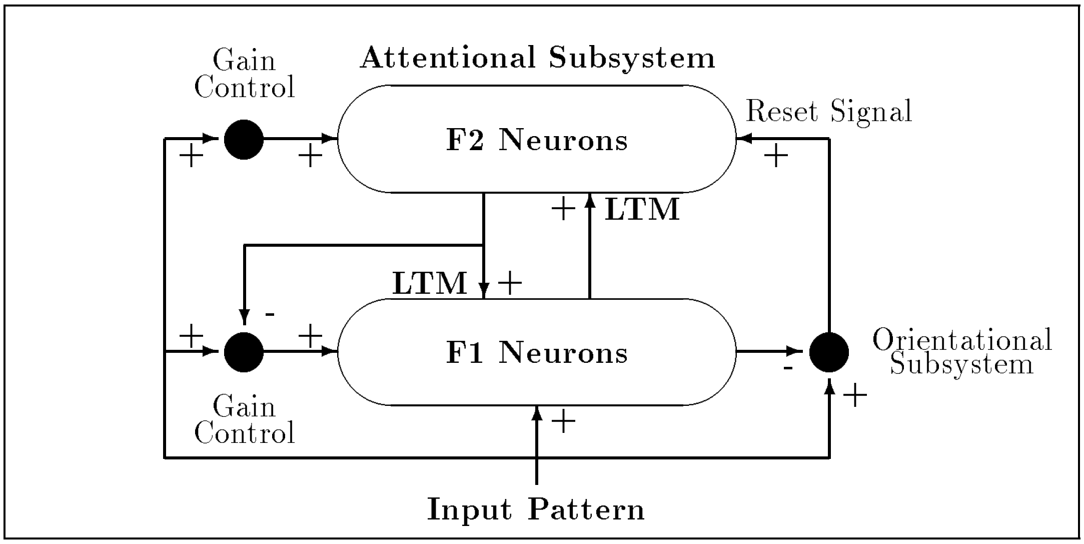

Figure 7: Anatomy of the ART architecture

down from the outputs of the neurons in F_2. This activity is called the short-term memory trace in F_1, and is responsible for stimulating "an orientational subsystem". Moreover, this orientational subsystem together with "gain control units" is responsible for controlling the stability-plasticity of the system, selectively turning off the activity of each set of neuron whenever a change in strategy is required. The activity of the neurons in F_2 is the result of the stimulus coming from the neurons in F_1 through the bottom-up connection weights and the activity of the orientational subsystem, plus the competition existing among themselves, so that the units with strongest response shut down those with weaker response.

It is stressed that the neurons in F_2 implement a categorization process for the signals coming to the neurons in F_1. Therefore, the system is able to classify its inputs in as many classes as the number of neurons in F_2.

If the environmental stimulus presented to the neurons in F_1 "matches" the signals coming through the top-down weights from the outputs of F_2, the input signal is said to match the "expectation" of the system, and the orientational subsystem shuts off. This causes the bottom-up signals from F_1 to F_2 to resonate with the top-down signals coming from F_2 to F_1, and the system locks up in this activity pattern. At this point, any minor differences in between the top-down and bottom-up signals is diminished through an adaption process that causes the weights in both paths to adjust so that the match is further enhanced.

By contrast, if the stimulus presented to the neurons in F_1 does not match the signals coming through the top-down weights from the outputs of F_2, then the orientational subsystem is stimulated and causes the neurons in F_2 to temporarily shut down their activity level. After a while, the neurons in F_2 will produce a "different" activity pattern because the previous winners were inhibited longer than the other units. This new activity pattern will propagate top-down and produce a new expectation for the system. Again, if this new expectation matches the environmental stimulus, the system locks itself up in this activity pattern. And, if no match occurs, a new "search cycle" begins. However, if a match is never found, and there are still units in F_2 that were never committed to any class of input signals, the gain control subsystem will cause the system to lock itself in the activity pattern involving some uncommitted unit, thus causing it to learn to produce the expectation of this new input. Therefore, the system alternates in between the learning and query modes automatically without the need of any supervisor.

Note that the adaption of the weights is a much slower process than the bottom-up and top-down interactions among the units in both sets F_1 and F_2, and this is exactly what enables the system to switch in between the learning and searching modes. A valid criticism is that the search process may be time consuming, besides being very sensitive to noisy input signals. Yet another criticism is that the storage capacity of the system is limited to the number of units in F_2. The strong points of the system are its ability to learn in real-time, self-regulatory behavior, and controlled sensitivity to noise through the adjustment of the "vigilance" parameter.

In any event, Grossberg and his team of collaborators have published a vast number of articles describing different cognitive process capable of being simulated with ART-models. The references mentioned before should constitute a starting point for the interested reader. Many other can be found through them. For now, a discrete version of an ART-model capable of implementing a sequential leader clustering algorithm is introduced. This simplified ART-model was introduced by Lippmann [60] and uses the ART option of **fast-learning**, in which the inputs and outputs of the units take only the values 0 and 1. It assumes two sets of neurons, the bottom set with N neurons and the top set with M neurons. The bottom-up connections are represented by the terms $b_{i,j}$, and the top-down connections are represented by the terms $t_{j,i}$. A "vigilance" parameter ρ defines the sensitivity of the network to noise, as well as its ability to decide for when to create a new class for the inputs applied to the bottom set of neurons.

Algorithm 5.2 (ART Classifier) .

- **1. Initialization.** *Set all* $t_{i,j} = 1$ *and* $b_{i,j} = 1/(1+N)$ *for* $i = 1, \ldots, N$ *and* $j = 1, \ldots, M$. *Set the vigilance threshold to some small value such that* $0 \leq \rho < 1$.
- **2. Apply new input vector** I.
- **3. Compute Activities.**

$$y_j = \sum_{i=1}^{N} b_{i,j} x_i \qquad j = 1, \ldots, M, \tag{52}$$

  where y_j *is the activity of the neuron* j *in the top set of neurons, and* x_i *is the input to the* i *neuron the the bottom set, i.e.,* $x_i = I_i$, *for* $i = 1, \ldots, N$.
- **4. Select Best Matching Unit.** *Find* k *such that* $y_k = \max_j \{y_j\}$.

- **5. Vigilance Test.** *Compute the similarity match μ give by*

$$\mu = \frac{\sum_{i=1}^{N} t_{i,k} x_i}{\sum_{i=1}^{N} x_i}. \tag{53}$$

  If $\mu > \rho$ then go to step 7. Otherwise, go to step 6.

- **6. Disable Matching Unit.** *The output of the highest matching unit is temporarily set to zero and no longer takes part in the maximization process of step 4. Resume at step 3.*

- **7. Adapt Best Matching Unit** *The top-down and bottom-up connection weights are updated as follows:*

$$t_{i,k}^{(t+1)} = t_{i,k}^{(t)} x_i \tag{54}$$

$$b_{i,k}^{(t+1)} = \frac{t_{i,k}^{(t)} x_i}{0.5 + \sum_{l=1}^{N} t_{l,k}^{(t)} x_l} \tag{55}$$

- **8. Repeat.** *Enable units disabled at step 6 and go to step 2.*

Lippmann [60] notes that the above algorithm works well in a noise free environment. However, if the input patterns are slightly corrupted by noise, the "vigilance" threshold must be set to a smaller value, otherwise two similar patterns will be considered different and the network will put them in different classes. Nevertheless, this is just a simplification of the whole model. The model has provisions for automatic gain control and varying vigilance by the orientational subsystem, thus making it less noise sensitive. The price to pay is that a more complex system of differential equations must be simulated. No longer the fast learning mode can be used.

5.5 Hebbian Learning Rule

As mentioned before, much of the motivation for the research in Neural Networks came after the proposition by D. O. Hebb [30] in which the synaptic efficacy among any two interconnected brain cells would increase, whenever both cells were simultaneously firing due to some environmental stimulus. In mathematical terms, if y_i and y_j represents the activity of two such cells, such that cell i's input has already a synaptic connection with cell j's output, then $\Delta w_{i,j} = \alpha y_i y_j$ represents the increase in the efficacy of the existing connection, where α is some positive "learning rate" constant. Using this principle, or some variations, as the central idea, several models to explain the behavior of various brain structures and learning machines were proposed [6, 22, 27, 46].

A valid criticism to this approach is that instead of stating the goals of some learning process, and then pursuing the rules under which such goals would be attained, this approach first states the rules for the modifications that the system undergoes, and then tries to analyze their consequences. It should be clear that many such modification rules could be proposed, and the work of analyzing the consequences of each one of them would be infeasible. However, the observation of a general underlying principle being fulfilled by the Hebb learning rule has brought attention back to this kind of learning process.

Indeed, a rather striking result due to Linsker [56, 57, 58] is the observation of the progressive maturation process of a system composed of a multi-layer feed-forward network in which the units are subject to a constrained type of the Hebb learning rule. The experiment starts with *"a network comprising several two-dimensional layers of cells, with connections of unspecified strength to each cell of one layer from a neighborhood of cells of the preceding layer"* [56]. Furthermore, in these neighborhoods the connections distribution is assumed to be Gaussian, and their positions are not allowed to change with time. The network is not presented any meaningful inputs, but the cells in the first layer are allowed to show spontaneous uncorrelated random activity which is propagated to the layers above. The maturation process takes place by allowing the connection strengths to develop very slowly according to a parameterized Hebb learning rule, but with constraints imposed on the maximum excitory and minimum inhibitory strength value. Linsker presents a highly parameterized set of equations to describe the rules governing the system. The analysis of these equations is beyond the scope of this article, but a very important result is the observation of the development of biologically suggestive receptive fields which are dependent only on a few generic parameters in each layer. In addition, the values of these parameters are allowed to vary over a wide range. In fact, Linsker demonstrated that such networks with just a few layers, and the addition of lateral interconnections within cells in the same layer, can develop into a system whose response to environmental stimulus bears striking similarity to the "orientation columns" found in the mammalian visual cortex.

Although the above observation is important per se, Linsker also demonstrated that the system as described above will undergo an optimization process in which the variance of the output activity of each cell is maximized. He goes one step further, and proposes the **INFORMAX** principle: the development of neural networks with lateral and forward connections should obey rules that cause the information contained in the signals being transported from layer to layer to be maximally preserved [59]. Indeed, Linsker demonstrates that his network of multiple layers of neurons, with contrained connections from layer to layer, will try to maximize the rate of information per message that a cell passes to its output from stimulus received at the inputs. Denoting by L the input vector, and by M the output vector, this rate is defined as

$$R = \langle \ln[\frac{P(L|M)}{P(L)}] \rangle, \tag{56}$$

where $P(L|M)$ is the conditional probability that the input vector is L given that the output is M, and $\langle\cdot\rangle$ denotes average over the entire training set. Learning paradigms that obey the INFORMAX principle should attempt to maximize the above quantity, for it denotes the amount of information that knowing the output vector M conveys about the input vector L. Although the INFORMAX principle is satisfied by the rules governing the system described above, it could be satisfied by other learning algorithms as well. In fact, the G-maximization algorithm presented earlier maximizes a quantity whose expression is very similar to the information measure adopted by Linsker.

5.6 Differential Hebbian Learning

The fact that the Hebbian Learning Rule is not enough to capture "concomitant variations" of stimulus which are central to the extraction of temporal relationships of "cause and effect", has motivated the development of yet another variation of that rule. The **Differential Hebbian Learning** proposed by Klopf [46] and studied by Kosko [50] is aimed at solving this problem. Put in simple terms, this new learning rule states that concomitant variations of neuronal activity are reflected in the increase of the synaptic strengths. In other words,

$$\dot{w_{i,j}} = -\alpha w_{i,j} + \beta \dot{y}_i \dot{y}_j, \tag{57}$$

where the term $-\alpha w_{i,j}$ represents a "forgetting" decay. Under the assumption that the expectation of the information about the environment carried by any stimulus is independent from the information carried by past stimuli, the connection process represented by the solution of equation (57) is a sub-martingale, i.e., the connection strengths are supposed to increase. But, in fact, the learning process can be decomposed into the sum of a martingale and a **novelty** process. The martingale process represents what stays the same in the connection strengths and the novelty process represents whatever new information is contained in the environmental stimulus. In Kosko own words, *"the expected connection residual is conditioned on exactly what it should be, i.e., all the information accumulated up to the present"*.

Indeed, the computer simulations carried out by Klopf [46] demonstrates the accuracy of the differential Hebb learning in modeling most of the basic animal learning phenomena that have been observed, and provides reason to believe that this learning mechanism *"offers a new building block for constructing neural network-like computer architectures for artificial intelligence"*.

6 Reinforcement Learning

The question of whether unsupervised learning procedures are better than equivalent supervised ones, has no easy answer. In fact, for many tasks, none of them are practical, since there will exist no "exemplar/response" training set to be used by any supervised learning procedure, and every unsupervised learning procedure would take enormous amount of time to find the proper classes for the data at hand. As an example, consider the problem of teaching a student how to bike-ride: Since the instructor is not receiving all the feedback that the student is getting from his trials, he cannot tell exactly what the student should do. Instead, he can simply say "balance to the right", or "balance to the left", and hope that the student finds out by himself which are the best moves. Nevertheless, his "commands" are useful, since that a person with no experience in bike-riding may have difficulty figuring out exactly what he should do next. The environment also plays a important role in this kind of task: gravity will tell the student exactly when he is moving too much in either direction, thus providing him with a "critical signal" about his performance. This is the central idea behind **reinforcement learning procedures**. They are based on a critical

scalar signal that correlates with the systems performance. Actions taken by the system in a "correct direction" are positively reinforced, and actions in the "wrong direction" are negatively reinforced. The reinforcement signal acts as a critic, in that it tells the system how good or bad its performance is.

How can a learning algorithm be designed to make use of such "non-specific reinforcement signals" ?. It turns out that the answer to this question have to be worked out in a case by case basis. In general, as pointed out by Hinton [34], most reinforcement learning procedures work by assigning credit to decisions based on a set of local variables. The correlation of these decisions with the reinforcement signal tells the system whether those decisions were accurate or not, and guides future choices, thus making the system to act as a predictor of future reinforcement signals. The learning process requires that the correlation of these local decisions with positive reinforcement signals be maximized, while the correlation of local decisions with negative reinforcement signals be minimized, thus making it more likely that the network will produce good decisions, and also reduce the impact of eventual bad decisions. Since that the training process will require many trials, an averaging process takes place in every decision, thus reducing the effects of noise in the observations of the local variables, making the learning process robust and stable.

The main advantage of this approach is that is is relatively simple to implement. The local variables may represent possible states of the system while the decisions may refer to what action to take once a specific state has been reached. In the bike-riding example, the state could be the angle of the bike with respect to the horizon, and the action would certainly be "move to the left", or "move to the right". Of course, the learning process would favor decisions like "move to right" whenever the bike were inclined to the left, and vice versa. The reinforcement signal could be something as simple as "bike is still up", or "bike felt down". Of course, this is a simplistic approach, but not far from reality. As we shall see, a very similar approach was used by Barto et al [9] in a not so different domain.

The main disadvantage of this approach is that the whole process may be very time consuming. As the number of local variables increase, so does the number of trials needed to infer the inter-relationships among them. Therefore, the number of trials needed to discover the "good" combinations may become extremely large. Another disadvantage is that the learning process may get locked into producing decisions that maximize the reinforcement signals without ever solving the problem that it is being trained for. Considering that the network is attempting a gradient ascent in the distribution of the expected reinforcement signal, this is equivalent to say that it got stuck in a local maxima. There must exist some mechanism to allow the learning process to try different combinations from time to time, thus ensuring that a not so good combination of state variables does not overshadows the best solution.

In any event, the number of publications about reinforcement learning procedures is large, making a complete coverage of the subject beyond the scope of this article. Good references are the work by Sutton [77], Barto [9], and Williams [84]. Indeed, the later brings a complete mathematical discussion on the subject, including methods for converting reinforcement learning procedures into equivalent supervised learning algorithms. Instead of trying to cover the whole issue of reinforcement learning, a detailed example of this kind of learning procedure will be presented next.

6.1 The Pole Balancing System

Presented in a very enlightening paper by Barto, Sutton and Anderson [9], the pole balancing system consists of a cart free to move within the bounds of a one-dimensional track with a pole hinged to the cart, and free to move in the vertical plane of the cart and track. Sensors measure the angle θ of the pole with the perpendicular to the cart, as well as its rate of change $\dot{\theta}$. The position x of the cart, and its velocity $\dot{x}$ are measured as well. A controller can apply "impulsive" forces F to the left or to the right of the cart, thus moving it to either direction. The objective is to have the controller attempting to generate controlling forces that keep the pole balancing in its vertical position for as long as possible, thus avoiding a "failure signal" that occurs whenever the pole falls down. An additional difficulty is that such failure signal occurs only after a long series of control decisions have been made, thus making it harder to determine when the first bad decision was made.

The approach taken by Barto et al for solving this problem makes it a exemplar case of reinforcement learning algorithm. The controller is made up of only two "neuron-like" elements whose weights are adjusted throughout the learning process. The performance of the system was proved superior to more traditional methods, and in general the system learned to balance the pole, before it experienced 100 failures. Refer to [9] for the details of the simulations. Three basic principles are involved in the construction of the controller: a **state space partitioning** through a special decoder provides the controller with a discrete description of the state space; an **eligibility function** enables the system to decide which past actions where responsible for a current failure; a **predictive reinforcement learning procedure** allows the system to maximize the elapsed time before a failure.

The state of the cart-pole system, for the purpose of this problem, is fully characterized by the four state variables $\theta, \dot{\theta}, x, \dot{x}$. A partition of the state space is performed by a special decoder which accepts those variables as input, and produces the vector $B = [b_1, \ldots, b_n]$ as output. The values of the coordinates of B are such that only one coordinate is $b_i = 1$, and all the other are equal to zero at all times, of course, i depending on the values of the state coordinates. The details of implementation of the decoder and the exact range of the partitioning of the state space are here considered unimportant. However, it is important to stress that the decoder offers enough resolution for characterizing the state space as small "boxes" of the state variables. Whenever the state variables fall within the coordinates of such boxes, the corresponding decoder line will output the value 1, and all the others will output the value 0.

The controller uses two neuron-like elements whose inputs are the output lines of the decoder described above. The first neuron, called the **Associative Search Element**, or **ASE**, computes the "decision" for moving the cart, either to the left or to the right, as follows:

$$y(t) = f\left[\sum_{i=1}^{n} w_i(t)b_i(t) + noise(t)\right], \tag{58}$$

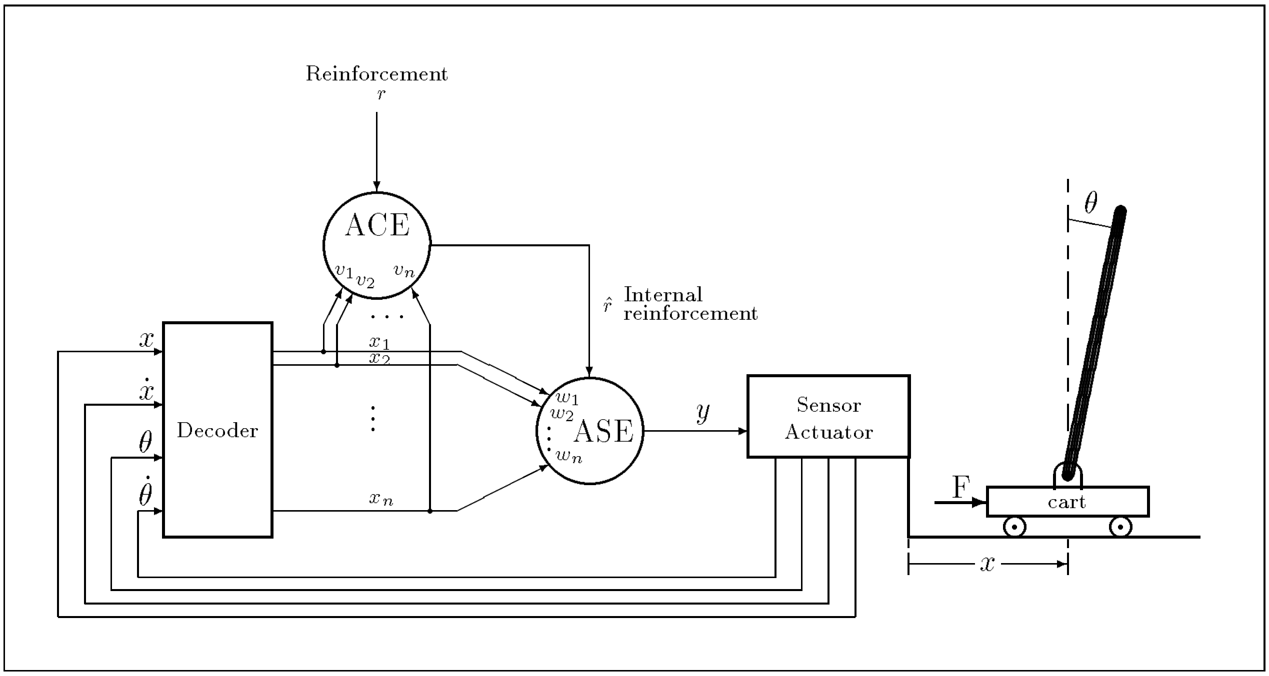

Figure 8: The Pole Balancing System

where the vector $W = [w_1, \ldots, w_n]$ is the set of adaptive weights of this element, and $noise(t)$ is a real random variable with a Gaussian distribution of mean zero, and standard deviation σ. The function $f(\cdot)$ is a threshold, or sigmoid function with high gain, such that

$$f(x) = \begin{cases} +1, & \text{if } x \geq 0 \text{ (move to right)} \\ \\ -1, & \text{if } x < 0 \text{ (move to left)} \end{cases} \tag{59}$$

Note that the ASE generates an action even if no signals are applied to its inputs. Given that the noise term has zero mean, the action produced will be either move to left, or move to right, with equal probabilities. In fact, in the event of a failure, these variables are all forced to zero, since that the cart-pole system is outside its normal operation state space. This will become important when analyzing the effects of the learning procedure. The fact that only one line b_i is 1 at all times, causes the ASE to generate a biased decision for whenever the state variables fall in the box corresponding to that line. The bias is affected by the weight w_i, which undergoes an adaption process aimed at making the system to produce the correct action for every combination of state variables. Indeed, the weights are updated as follows:

$$w_i(t+1) = w_i(t) + \alpha r(t) e_i(t), \tag{60}$$

where α denotes a positive "learning rate" constant, and $r(t)$ is a real-valued reinforcement signal. The function $e_i(t)$ denotes the **eligibility** of this synaptic weight for the **blame** or **reward** of the current reinforcement signal. In fact, $e_i(t)$ is the heart of the solution to the **credit assignment problem** existent in this task. The question of which past action allowed to current success or failure of the system is answered by a proper choice of such eligibility function. In their example, Barto et al chose to make $e_i(t)$ a exponentially decaying function, such that the eligibility of the connection weight w_i is a trace of the product $y(\tau)b_i(\tau)$ for $\tau < t$. Mathematically,

$$e_i(t+1) = \delta e_i(t) + (1-\delta)y(t)b_i(t), \tag{61}$$

where $0 < \delta < 1$ is the trace decay rate. Note that $e_i(t)$ can have positive and negative values, thus remembering not only how long ago a given state was entered, but also what was the action taken when it was entered. If the reinforcement signal could take only values 0 for success, and -1 for failures, then the product $r(t)e_i(t)$, indicates precisely how much of the blame should be assigned to weight w_i for an occurring failure. As time goes by, $e_i(t)$ approaches zero, thus making whatever decision taken when the system entered the corresponding state less likely to have caused the failure.

However, the reinforcement signal alone is not enough to train the ASE in its decisions for avoiding punishment. Since a failure will eventually occur, it is not clear that the last actions taken deserve all the blame regardless of how long it took for the failure to occur since the trial begun. It is important to have a procedure that guarantees that those actions that were taken because they ensure the longest elapsed time between failures will not be severely blamed when a failure occurs. In other words, the longer the system succeeds, the smaller the blame for an occasional failure should be. This can be accomplished by a system that attempts to predict the reinforcement signal, and punishes the ASE with values proportional to its failure rate. In Barto's system, this is accomplished by a neuron-like element called the **Adaptive Critic Element**, or **ACE** for short.

The ACE receives the same vector B as input, and produces a signal $\hat{r}(t)$ which is used by the ASE during the weight adaption process. The ACE also has a weight vector $V = [v_1, \ldots, v_n]$ which undergoes another adaption process in order to ensure that the computation performed by the ACE approximates a "prediction" of the external reinforcement signal $r(t)$. A exponentially decaying function of these prediction is then passed along to the ASE as the improved success-time-dependent reinforcement signal. In mathematical terms, the prediction computed by the ACE is given by:

$$p(t) = \sum_{i=1}^{n} v_i(t)b_i(t) \tag{62}$$

Note that the response of the ACE is fully deterministic, in that no noise is injected in the process. The improved reinforcement signal $\hat{r}$ is computed by

$$\hat{r}(t) = r(t) + \gamma p(t) - p(t-1), \tag{63}$$

where γ is a prediction decaying rate. In other words, if there is no external punishment, any predicted punishment becomes weaker as time goes by. In the presence of a external punishment, the prediction is further reinforced (or even overridden) by the term $r(t)$ in equation (63).

The final step is the adaption process that the weights in the ACE have to undergo in order to guarantee that $p(t-1)$ approximates $r(t)$. In fact, Barto et al introduces an adaption rule which causes the prediction $p(t)$ to approximate $r(t+1)+\gamma p(t+1)$, thus ensuring that the predictions $p(t)$ will be the earliest possible indications of external reinforcement signals. This update rule correlates the "improved reinforcement signal" $\hat{r}(t)$ with the decaying trace of the signal $b_i(t)$ as follows

$$v_i(t+1) = v_i(t) + \beta\hat{r}(t)\bar{b}_i(t), \tag{64}$$

where β is the positive "learning rate" constant, and $\bar{b}_i(t)$ is the exponentially decaying trace of the signal $b_i(t)$ given by

$$\bar{b}_i(t+1) = \lambda\bar{b}_i(t+1) + (1-\lambda)b_i(t), \tag{65}$$

where $0 < \lambda < 1$ is the decaying rate. Note that this decaying trace plays the same role as the eligibility trace of equation (61). According to Barto et al, this adaption rule ensures that $p(t-1)$ approximates the quantity $r(t+1)+\gamma p(t+1)$. In order to understand what this means, we imagine a failure that has just occurred: the external reinforcement signal will be -1, and the predicted reinforcement will be zero, since that the state variables are outside the normal values of the car-pole system. Therefore, the signal passed to the ASE is a strong -1, which causes it to penalize the most recent actions that probably caused the failure. At the same time, the prediction of failure for that state is increased through the adaption process in the weights of the ACE. When a new trial is started, the system will reward actions that move the system to states of lower probability of failure, thus ensuring that it learns which are the "safe state boxes" that it can be in.

6.2 Some extra remarks

This simplified description of a reinforcement learning neural-network algorithm, together with the high performance of the algorithm, should be enough to convince the reader of the power of this kind of approach. The algorithm can be easily adapted to other domains where a state space can be well characterized, and a reinforcement signal is present. It is important to note that the above control system constitutes a departure from the previous algorithms presented earlier. Nevertheless, it is easy to imagine architectures where many types of networks can be combined and produce powerful "intelligent" systems, whose capabilities have yet to be demonstrated. For instance, the above decoder could be easily implemented by a Kohonen's feature map network, or by a multi-layer perceptron trained with the back-propagation algorithm. We believe that, as more applications of neural networks are devised, more frequently combinations of different learning paradigms will be employed.

7 Discussion

The previous sections covered the most traditional Neural Networks models and their learning algorithms. However, many others are available. The **Learning Matrix** proposed by Steinbuch [76], and the **Associative Nets** proposed by Willshaw [85] represent variations on Anderson's work. The **Autocorrelation Matrix** discussed by Kohonen [47], and the subsequent **Novelty Filter** are intimately related to the auto-associative networks of Hopfield. The **Neocognitron** by Fukushima [24] employs concepts of multi-layer networks, and competitive learning in order to accomplish position invariance of learned patterns. The **Biderectional Associative Memory** introduced by Kosko [51] is an attempt to generalize the ART architectures of Grossberg an Cohen. Several other models had to be omitted, not because they are unimportant, but because it would be infeasible to cover them all. In reality, many of the existing models employ variations on the same themes described here, and they would all fall under one of the learning paradigms introduced earlier. Instead of attempting to cover all models, a discussion on some learning complexity issues is presented next.

7.1 The Complexity of Learning

Throughout this paper, and also in most of the literature about Neural Networks, "learning" is taken as the ability of a system to capture and use the information present in its environment, without ever being explicitly "hardwired" to do so. This is a somewhat loose definition of "learning". For one thing, it does not imply any measure for the efficiency of the learning task at hand, nor does it say if such tasks can, or cannot be resolved through learning. Instead, the learning systems are said to have learned if they demonstrate that they can reproduce whatever behavior they were trained for, within a certain degree of accuracy. However, this degree of accuracy is set by the "trainers" own expectation of what an adequate degree of accuracy should be, and usually his expectation will be based on his own knowledge about the domain for which he picked the task to be learned. And although this definition can be employed by those interested only in developing new applications for Neural Networks, which will usually be accomplished mostly through experimentation, a more formal theory of what can be amenable to learning is required. Without such theory, there is no hope that one can assert about the power, effectiveness, and scalability of any of the existing, or yet to be proposed, models.

The work developed by Valiant [78] sets the grounds for such theory. Dealing strictly with boolean functions, he asks for which classes of functions can be safely induced from examples in polynomial time, as opposed to those that require exponential time. This study brings the Complexity Theory distinction between polynomial ($\mathbf{P}$), and super-polynomial ($\mathbf{NP-hard}$) problems to the arena of learning machines. It also demonstrates that, while there are several classes of functions that can be learned in polynomial time (e.g. $\mathbf{k-CNF}$, monotone $\mathbf{DNF}$), there are others that will always require the transfer of "explicitly knowledge" from the teacher to the learner. The importance of this distinctions is two-fold: first, one must restrain himself from trying to find a general purpose learning algorithm that can be safely used in any situation, for such algorithm cannot exist; second, one must look for the classes of functions that can be safely learned by Neural Network models, in polynomial time; third, one must look for ways for transferring knowledge to the networks other than simply learning. It may well be that a combination of these efforts will produce more complex network architectures capable of learning how to solve harder problems in a reasonable amount of time.

In a recent thesis by Stephen Judd [42], some of these problems have been studied. Using multi-layer feed-forward networks to compute boolean functions, Judd shows how to cast the task of training the network to compute a specific function into the problem of recognizing a specific language using a Turing machine. With an ingenious transformation of the **3SAT** problem into that of *loading a neural network with the proper weights to solve the* **3SAT** *problem*, he carries with the proof that learning is, in the general case, a NP-complete task. However, he also demonstrates that there exist constraints, to be imposed on the architecture of the network, which make learning amenable to polynomial time algorithms.

However, this result contrasts with the work by Halber White [81], who proves that the arbitrary continuous mappings that multi-layer feed-forward networks are capable of producing are also learnable. His definition of learnable, however, calls for a zero-limit on probability of the network error exceeding any specified level, as the network accumulates experience. Note that this definition allows for infinite learning time, thus making it impossible to compare with Judd's result. As of now, one can say only that multi-layer feed-forward networks will, under certain conditions, learn the required continuous mapping, but this process may take a very long time. Nevertheless, White's work also sets the grounds for establishing confidence levels for the expected outputs of the networks, thus allowing one to stop the learning process whenever prescribed levels are reached. He also points into directions that may lead to the development of optimal algorithms for training such networks, thus leaving hope that they will be safely used in problems on the domain of non-linear regressions.

The two above results are derived in the context of multi-layer feed-forward networks only. Whether they can be transported to the other learning paradigms is still an open question. There are, however, several isolated studies on the complexity of training different networks for different tasks, all of them producing interesting performance bounds. But all of these fail in providing a cohesive theory that can be used to analyze arbitrary network models. Refer to Judd's recent book [42] for a good introduction on this important topic.

7.2 Conclusion

Neural Networks Research has been prodigal in producing models and algorithms which can be used to build models of different phenomena. The ongoing research in certainly going to push the limits of what can currently be achieved with these models. New learning procedures will be developed, as well as new network models will be proposed. It is our believe, as well as our hope, that the intrinsic rules that should govern the adaption process of these systems will be, one day, laid down in a sound theory of neuronal systems behavior, thus unleashing the power of these computing devices.

References

[1] L. F. Abbott and Yair Arian. Storage capacity of generalized networks. *Physical Review A*, 36(10):5091–5094, November 1987.

[2] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9:147–169, 1985.

[3] Robert B. Allen. Several studies on natural language and back-propagation. In *Proc. First International Conference on Neural Networks*. IEEE, 1987.

[4] D. J. Amit, H. Gutfreund, and H Sompolinsky. Spin-glass models of neural networks. *Physical Review Letters*, 55:1530, 1985.

[5] James A. Anderson. Cognitive capabilities of a parallel system. In Bienenstock et al., editors, *NATO ASI Series, Disordered Systems and Biological Organization*, volume F20, pages 209–226. Spring-Verlag, 1986.

[6] James A. Anderson, Richard M. Golden, and Gregory L. Murphy. Concepts in distributed systems. *SPIE - Optical and Hybrid Computing*, 634:260–276, 1986.

[7] Pierre Baldi and Kurt Hornik. Neural networks and principal component analysis: Learning from examples without local minima. *Neural Networks*, 2:55–58, 1989.

[8] Pierre Baldi and Santosh S. Venkatesh. Number of stable points for spin-glasses and neural networks of higher order. *Physical Review Letters*, 58(9):913–940, March 1987.

[9] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-13(5):834–846, September/October 1983.

[10] A. D. Bruce, A. Canning, B. Forrest, E. Gardner, and D.J. Wallace. Learning and memory properties in fully connected netwroks. In *A.I.P. Conference Proceedings 151, Neural Networks for Computing*. American Institute of Physics, 1986.

[11] Gail A. Carpenter and Stephen Grossberg. A massively parallel architecture for self-organizing neural pattern recognition machine. *Computer Vision, Graphics, and Image Processing*, 37:54–115, 1987.

[12] Gail A. Carpenter and Stephen Grossberg. Neural dynamics of category learning and recognition: Attention, memory consolidation, and amnesia. In J. Davis, R. Newburgh, and E. Wegman (Eds.), editor, *Brain Structure, Learning, and Memory*. AAA Symposium Series, 1987.

[13] Victor C. Chen and Pao Yoh-Han. Learning control with neural networks. In *IEEE - See with Peter Allen - Robotics Conference*, pages 1448–1453. DON'T KNOW, 1989.

[14] M. Cohen and S. Grossberg. Absolute stability of global pattern formation and parallel memory storage by competitive neural networks. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):815–825, September/October 1983.

[15] M. Cohen and S. Grossberg. Neural dynamics of speech and language coding: developmental programs,perceptual grouping, and competition for short-term memory. *Human Neurobiology*, 5:1–22, 1986.

[16] G. W. Cottrell. Analysis of image compression by back propagation. Technical Report C-105, Institute for Cognitive Science, U.C. San Diego, 1987.

[17] Sigurd Diederich and Manfred Opper. Learning of correlated patterns in spin-glass networks by local learning rule. *Physical Review Letters*, 58(9):949–952, March 1987.

[18] Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis*. John Wiley & Sons, 1973.

[19] J Elman. Finding structure in time. Technical Report 8801, UCSD Center for Research in Language - La Jolla, CA, 1988.

[20] Jeffrey L. Elman and David Zipser. Learning the hidden structure of speech. *Journal of the Acoustic Society of America*, 83(4):1615–1626, April 1988.

[21] Françoise Fogelman Soulie, Robert Yves, and Robert Tchuente, editors. *Automata networks in computer science: Theory and Applications*. Princeton University Press, 1987.

[22] Walter J. Freeman and Brian Burke. Central pattern generating and recognition in olfactory bulb: A correlation learning rule. *Neural Networks*, 1:277–288, 1988.

[23] Avital Frumkin and Elisha Moses. Physicality of the Little model. *Physical Review A*, 34(1):714–716, July 1986.

[24] Kunihiko Fukushima and Sei Miyake. Neocognitron: A new algorithms for pattern recognition tolerant of deformations and shifts in position. *Pattern Recognition*, 15(6):455–469, 1982.

[25] Ken-Ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2:183–192, 1989.

[26] Stephen Gallant. Optimal linear discriminants. In *Eighth International Conference on Pattern Recognition*, pages 849–852. IEEE, October 1986.

[27] Alan Gelperin, J. J. Hopfield, and D. W. Tank. The logic of Limax learning. In Allen I. Selverston, editor, *Model Neural Networks and Behavior*, chapter 13, pages 237–261. Plenum Publishing Corporation, 1985.

[28] Stephen Grossberg and Ennio Mingolla. Neural dynamics of perceptual grouping: Textures, boundaries, and emergent segmentations. *Perception & Psychophysics*, 38(2):141–171, 1985.

[29] Stephen Grossberg and Dejan Todorovic. Neural dynamics of 1-D and 2-D brightness perception: A unified model of classical and recent phenomena. *Perception & Psychophysics*, 43:241–277, 1988.

[30] D. O. Hebb. *The Organization of Behavior.* Wiley, New York, 1949.

[31] Robert Hecht-Nielsen. Counterpropagation networks. *Applied Optics*, 26(23):4979–4984, December 1987.

[32] Robert Hecht-Nielsen. Theory of the back-propagation neural network. In *First Annual I.N.N.S. Meeting*, page 445, 1988.

[33] J. L. Hemmen. Nonlinear neural networks near saturation. *Physical Review A*, 36(4):1959–1962, August 1987.

[34] Geoffrey E. Hinton. Connectionist learning procedures. Technical Report CMU-CS-87-115, Computer Science Department, Carnegie-Mellon University, 1987.

[35] Yuzo Hirai. A model of human associative processor. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-13(5):851–857, September/October 1983.

[36] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proc. Natl. Acad. Sci - Byophysics*, 79:2554–2558, 1982.

[37] J. J. Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Proc. Natl. Acad. Sci - Byophysics*, 81:3088–3092, 1984.

[38] J. J. Hopfield, D. I. Feinstein, and R. G. Palmer. Unlearning has a stabilizing effect in collective memories. *Nature*, 304(5922):158–159, July 1983.

[39] J. J. Hopfield and D. W. Tank. Neural computation of decisions in optimization problems. *Biological Cybernetics*, 52:141–152, 1985.

[40] Willian Y. Huang and Richard P. Lippmann. Comparisons between neural net and conventional classifiers. In *Proc. First International Conference on Neural Networks*, volume IV, pages 485–493. IEEE, 1987.

[41] Robert A. Jacobs. Increased rates of convergence through learning rate adaption. *Neural Networks*, 1:295–307, 1988.

[42] J. Stephen Judd. *Neural Network Design and the Complexity of Learning.* The MIT Press, Cambridge, MA, 1990.

[43] S. Kirkpatrick, C. D. Gellat, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[44] Scott Kirkpatrick and David Sherrington. Infinite-ranged models of spin-glasses. *Physical Review B*, 17(11):4384–4403, June 1978.

[45] D. Kleinfeld and D. B. Pendergraft. Unlearning increases the storage capacity of content addressable memories. *Biophysical Journal*, 51:47–53, January 1987.

[46] A. Harry Klopf. A drive-reinforcement model of single neuron function: An alternative to hebbian neuronal model. In *A.I.P. Conference Proceedings 151, Neural Networks for Computing.* American Institute of Physics, 1986.

[47] Teuvo Kohonen. *Self-Organization and Associative Memory*. Springer-Verlag, 2nd edition, 1987.

[48] Teuvo Kohonen. The neural phonetic typewriter. *Computer*, pages 11–22, March 1988.

[49] Teuvo Kohonen and K Makisara. Representation of sensory information in self-organizing feature maps. In *A.I.P. Conference Proceedings 151, Neural Networks for Computing*. American Institute of Physics, 1986.

[50] Bart Kosko. Differential hebbian learning. In *A.I.P. Conference Proceedings 151, Neural Networks for Computing*, pages 277–282. American Institute of Physics, 1986.

[51] Bart Kosko. Biderectional associative memory. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-18(1):49–60, January/February 1988.

[52] S Kullback. *Information Theory and Statistics*. Wiley, New York, 1959.

[53] Alan Lapedes and Robert Farber. Nonlinear signal processing using neural networks: Prediction and system modeling. Technical Report LA-UR-87-2662, Los Alamos National Laboratory - Theoretical Division, 1987.

[54] Y. C. Lee et al. Machine learning using a higher order correlation network. *Physica*, 22D:276–306, 1986.

[55] Daniel S. Levine. Neural population modeling and psychology: A Review. *Mathematical Biosciences*, 66:1–86, 1983.

[56] Ralph Linsker. From basic network principles to neural architecture: Emergence of spatial-opponent cells. In *Proc. Natl. Acad. Sci.*, volume 83, pages 7508–7512, 1986a.

[57] Ralph Linsker. From basic network principles to neural architecture: Emergence of orientation-selective cells. In *Proc. Natl. Acad. Sci.*, volume 83, pages 8390–8394, 1986b.

[58] Ralph Linsker. From basic network principles to neural architecture: Emergence of orientation columns. In *Proc. Natl. Acad. Sci.*, volume 83, pages 8779–8783, 1986c.

[59] Ralph Linsker. Self-organization in a perceptual network. *COMPUTER*, pages 105–117, March 1988.

[60] Richard P. Lippmann. An introduction to neural computing. *IEEE ASSP Magazine*, April 1987.

[61] Richard P. Lippmann and Ben Gold. Neural-net classifiers useful for speech recognition. In *Proc. First International Conference on Neural Networks*, volume IV, pages 417–425. IEEE, 1987.

[62] D. C. Mattis. Solvable spin systems with random interactions. *Physics Letters*, 56A(5):421–422, April 1976.

[63] T. Maxwell, C. L. Giles, Y. C. Lee, and H. H. Chen. Transformation invariance using high order correlations in neural net architectures. In *Proceedings of the IEEE international conference on systems, man, and cybernetics*, pages 627–632, October 1986.

[64] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[65] Marvin L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967.

[66] Marvin L. Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, extended edition, 1988.

[67] Bruce L. Montgomery and B. V. K. Vijaya Kumar. Evaluation of the use of the Hopfield neural network model as a nearest-neighbor algorithm. *Applied Optics*, 25(20):3759–3766, October 1986.

[68] David B. Parker. Learning logic. Technical Report TR-47, Center for Computational Research in Economics and Management Science - M.I.T., April 1985.

[69] David B. Parker. Comparisons of algorithms for neuron-like cells. In *A.I.P. Conference Proceedings 151, Neural Networks for Computing*, pages 327–332. American Institute of Physics, 1986.

[70] Barak A. Pearlmutter and Geoffrey E. Hinton. G-maximization: an unsupervised learning procedure for discovering regularities. In *A.I.P. Conference Proceedings 151, Neural Networks for Computing.* American Institute of Physics, 1986.

[71] J. P. Provost and G. Vallee. Ergodicity of the coupling constants and the symmetric n-replicas trick for a class of mean-field sping-glass models. *Physical Review Letters*, 50(8):598–600, February 1983.

[72] Helge J. Ritter, Thomas M. Martinetz, and Klaus J. Schulten. Topology-conserving maps for learning viso-motor-coordination. *Neural Networks*, 2:159–168, 1989.

[73] Frank Rosenblatt. Two theorems of statistical separability in the perceptron. In *Symposium on the Machanization of Thought Processes*, pages 421–456, 1959.

[74] David E. Rumelhart, James L. McClelland, and the PDP Research Group. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* The MIT Press, 1986.

[75] T. J. Sejnowski and C. R. Rosemberg. Nettalk: A parallel network that learns to read aloud. Technical Report TR-86-01, Department of Electrical Engineering, John Hopkins University, 1986.

[76] K. Steinbuch. Die lernmatrix. *Kybernetik*, 1:36–45, 1961.

[77] R. S. Sutton. *Temporal Credit Assignment in Reinforcement Learning.* PhD thesis, University of Massachusetts, Amherst, 1984.

[78] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.

[79] Paul J. Werbos. *Beyond Regression: New Tools for the prediction and analysis in the behavioral sciences.* PhD thesis, Harvard University, 1974.

[80] Paul J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1:339–356, 1988.

[81] Halber White. Coonectionist nonparametric regression: Multylayer feedforward networks can learn arbitrary mappings. *Neural Networks*, 3:535–549, 1990.

[82] Bernard Widrow and Capt. Rodney G. Winter. Neural nets for adaptive filtering and adaptive pattern recognition. *Computer*, pages 25–39, March 1988.

[83] Bernard Widrow, Capt. Rodney G. Winter, and Robert A. Baxter. Learning phenomena in layered neural networks. In *Proc. First International Conference on Neural Networks*, volume II, pages 411–429. IEEE, 1987.

[84] Ronald J. Williams. Reinforcement learning in connectionist networks: A mathematical analysis. Technical Report 8605, Institute for Cognitive Science - University of California, San Diego, June 1986.

[85] D. J. Willshaw. Holography, associative memory, and inductive generalization. In G. E. Hinton & J. A. Anderson, editor, *Parallel models of associative memory.* Erbaum, 1981.

[86] D. Zipser, G. W. Cottrell, and P. Munro. Image compression by back propagation: An example of extensional programming. Technical Report 8702, Institute for Cognitive Science, U.C. San Diego, 1987.