14

Artificial Neural Networks and Genetic Algorithms

Preview

In Chapters 12 and 13, we have looked at mechanisms that might lead to an emulation of human reasoning capabilities. We approached this problem from a macroscopic point of view. In this chapter, we shall approach the same problem from a microscopic point of view, i.e., we shall try to emulate learning mechanisms as they are believed to take place at the level of neurons of the human brain, and evolutionary adaptation mechanisms as they are hypothesized to have shaped our genetic code.

14.1 Introduction

In this chapter, we shall discuss how artificial neural networks can indeed emulate the human capability of *association* (remembering of similar events), and of *learning* (the autonomous organization of knowledge based on stochastic input stimuli). We shall then proceed to discuss yet another "learning" mechanism: the encoding of hereditary knowledge in our genetic code.

Artificial neural networks were invented several decades ago. Much of the early research on neural networks was accomplished by McCulloch and Pitts [14.21], and by Hebb [14.11]. This research was consolidated in the late fifties with the conceptualization of the so–called *perceptron* [14.30].

Unfortunately, a single devastating report by Minsky and Papert discredited artificial neural network research in the late sixties [14.23]. In this report, the authors demonstrated that a perceptron

cannot even learn the "behavior" of a simple exclusive–or gate. As a consequence of this publication, research in this area came to a grinding halt. Remarkably, an entire branch of research can be aborted because of a single influential individual, especially in the United States where research funds are not provided automatically, but must be requested through research proposals. For almost 20 years, funding for artificial neural network research was virtually non–existent. Only the last couple of years have seen a renaissance of this methodology due to the relentless efforts and pioneering research of a few individuals such as Grossberg [14.8,14.9,14.10], Hopfield [14.14], and Kohonen [14.17]. Meanwhile, artificial neural network research has been fully rehabilitated, and virtually thousands of research results are published every year in dozens of different journals and books. Impressive results have been obtained in many application areas, such as image processing and robot grasping systems [14.29].

Artificial neural networks still suffer somewhat from their pattern recognition heritage. Classical pattern recognition techniques were always concerned primarily with the recognition of *static images.* For this reason, most research efforts in neural networks were also concentrated around the analysis of static information. For quite some time, it was not recognized that humans more often than not base their decisions on *temporal patterns* or *cartoons*, i.e., series of sketchy static images which comprise an entire episode. Notice that graphical trajectory behavior is called a *film*, whereas graphical episodical behavior is called a *cartoon.* If I drive my car through the City of Tucson, and I suddenly see a ball rolling onto the street from behind a parked vehicle, I immediately engage the brake because I expect a child to run after the ball. I would be much less alarmed if the ball were simply lying on the street. Thus, my decision is based on an entire cartoon rather than on a single static image. The cartoon of the rolling ball is a temporal pattern which is matched with stored temporal patterns of my past. The prediction of the future event is based on the association and replay of another temporal pattern that once followed the rolling ball pattern in time, namely that of a running child.

Artificial neural networks are well suited to identify temporal patterns. All that needs to be done is to store the individual images of the cartoon below each other in a large array, and treat the entire cartoon as one pattern. Due to the inherent parallelism in artificial neural network algorithms, the size of a network layer (the length of the pattern array) does not have to increase the time needed for its processing.

Recently, artificial neural network researchers have begun to investigate temporal patterns. Promising results have been reported in the context of speech recognition systems [14.36]. Other important applications are related to dynamical systems. They include the design of adaptive controllers for non-linear systems [14.25], and the development of fault diagnosers for manufacturing systems [14.32]. However, much remains to be done in these areas.

Finally, it is important to distinguish between research in engineering design of artificial neural networks [14.28,14.34], and scientific analysis (modeling) of the mechanisms of signal processing in the human cerebellum [14.3,14.16] and information processing in the human cerebrum [14.7,14.33]. Engineering design concentrates on very simple mathematical models which provide for some learning capabilities. All artificial neurons are identical models of simple motor neurons, and the interconnections between artificial neurons are structured in the form of a well organized, totally connected, layered network. The biological analogy is of minor significance. The goal of engineering design research is to create algorithms which can be implemented in robots, and which provide these robots with some, however modest, reasoning capabilities. Scientific analysis focuses on modeling the activities of the human brain. These models are usually considerably more sophisticated. They distinguish between different types of neurons, and attempt to represent correctly the interconnections between these different neurons as they have been observed in various zones of the human brain. However, the applicability of the research results for engineering purposes is of much less importance than the scientific understanding itself.

In this chapter, we want to pursue both research directions to some extent. However, the primary accent is clearly on engineering design, and not on neurobiological modeling.

14.2 Artificial Neurons

Fig.14.1 shows a typical motor neuron of the human cerebellum. The motor neuron is the most studied and best understood neuron of the human brain. Altogether, the human brain contains on the order of 10^{11} neurons.

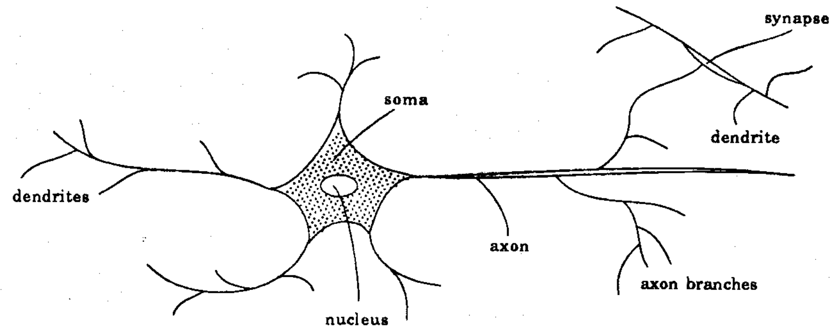

Figure 14.1. Motor neuron of the human cerebellum

When a neuron "fires", it sends out an electrical pulse from its soma through its single efferent axon. The axon branches out into several sub–axons, and passes the electrical pulse along all of its branches. Each neuron has on the order of 1000 axon branches. These branches terminate in the vicinity of dendrites of neighboring neurons. The pulse which is sent through the axon is of a fixed amplitude known as the action potential, and it travels along the axon at a high speed of approximately 20 $m\ sec^{-1}$ [14.3]. The pulse has a duration of approximately 1 *msec*. Once a neuron has fired, it needs to rest for at least 10 *msec* before it can fire again. This period is called the refractory period of the neuron [14.3].

When a pulse arrives at the end of an axon branch, it is transmitted to a dendrite of a neighboring neuron in a synaptic contact by molecules known as neurotransmitters. The synapses can be of the excitatory (positive) or inhibitory (negative) type. In the synapse, the signal is converted from a digital to an analog signal. The amplitude of the signal transmitted by the synapse depends on the strength of the synapse. In the afferent dendrites, the transmitted signal travels at a much lower speed towards the soma of its neuron. The typical dendrite attenuation time is on the order of 10 *msec* [14.3]. However, the attenuation time varies greatly from one neuron to another, because some synaptic connections are located far out on an afferent dendrite, while others are located directly at the base of the primary dendrite, or at the soma itself, or even at the axon hillock (the base of the efferent axon).

If the sum of the analog signals arriving at a soma from its various dendrites is sufficiently high, the neuron fires, and sends out a digital pulse along its axon. Consequently, the soma converts the arriving

analog signals back to digital signals using frequency modulation. In this way, signals are propagated through the brain by means of consecutive firings of neurons. During their voyage through the brain, the signals are constantly converted back and forth between a digital and an analog form.

Fig.14.2 shows a typical artificial neuron as used in today's artificial neural networks. Typical artificial neural networks contain several dozens to several thousands of individual artificial neurons.

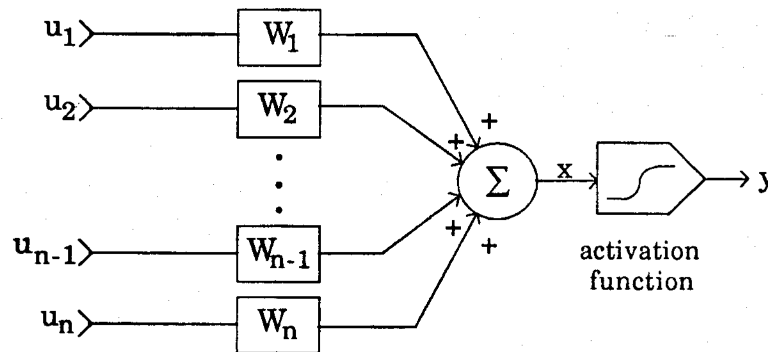

Figure 14.2. Neuron of an artificial neural network

The input signal u_i symbolizes the digital pulses arriving from the ith neuron. The gains w_{ji} are the synaptic weights associated with the digital to analog (D/A) conversion of the neurotransmission from the ith neuron to the jth neuron across the synaptic cleft. The state of the jth neuron x_j is computed as the weighted sum of its inputs:

$$x_j = \sum_{\forall i} w_{ji} \cdot u_i = \mathbf{w_j}' \cdot \mathbf{u} \tag{14.1}$$

The output of the jth neuron y_j is computed as a nonlinear function of its state:

$$y_j = f(x_j) \tag{14.2}$$

Fig.14.3 shows some typical activation functions.

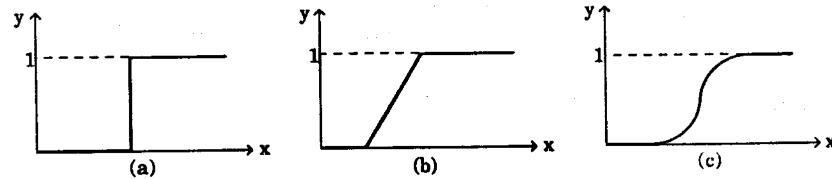

Figure 14.3. Typical activation functions of artificial neurons

Most artificial neural networks ignore the frequency modulation of real neural systems, i.e., the output corresponding to a constant input is constant rather than pulsed. They also ignore the low–pass characteristics of real neurons, i.e., the time delay of the signal transmission across the true neuron is neglected. While the former omission may be harmless, the latter causes formidable problems when feedback loops are present in the artificial neural network.

Engineering–oriented artificial neural networks use the artificial neuron as the basic building block for constructing networks. All artificial neurons are functionally identical, and all synapses are excitatory. Inhibitory synapses are simulated by allowing weighting factors to become negative. These artificial neural networks are symmetrically structured, totally connected, layered networks. Many engineering–oriented artificial neural networks are *feedforward networks* or *cascade networks*. Their response is immediate, since the time constants of the neurons are not modeled. Feedforward networks can be considered elaborate non–linear function generators. Due to the lack of feedback loops, these networks have no "memory", i.e., they don't store any signals. Information is "stored" only by means of weight adjustments. *Recurrent networks* or *reentrant networks* contain feedback loops. Feedback loops are mandatory if the network is supposed to learn cartoons directly, i.e., if the individual images of temporal patterns are not compressed in time to one instant and stored underneath each other in a long pattern array (as proposed earlier), but are to be fed sequentially into the network as they arrive. A serious problem with most recurrent networks is their tendency to become unstable as the weights are adjusted. Global stability analysis of a recurrent network is a very difficult problem due to the inherent non–linear activation functions at each node.

Neurobiologically–oriented neural network models take into consideration the fact that the human brain contains various *different* types of neurons which are interconnected in a few standard patterns,

and which form local neuronic "unit circuits". For example, Green's and Triffet's *cerebellar unit circuit* distinguishes between granule, Golgi, basket, stellate, and Purkinje neurons which are interconnected as shown in Fig.14.4 [14.7,14.33].

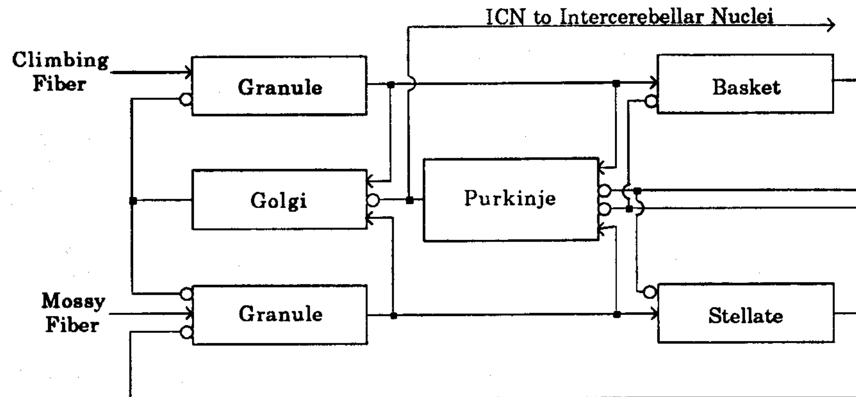

Figure 14.4. Unit circuit of the cerebellum

Arrowheads indicate excitatory synaptic connections, whereas small circles indicate inhibitory synaptic connections. Notice the local feedback loops within the unit circuit. The unit circuits cannot work properly unless the dynamics of the individual neurons are modeled. Triffet and Green do indeed model the time and frequency response characteristics of signals traveling through the cerebellum. In fact, they even take into consideration the frequency modulation of neuronic signals, i.e., they model the action potential, the refractory period, and the resting period of the neuron.

14.3 Artificial Neural Engineering Networks

While the model of an individual neuron bears a certain similarity with the real world, today's artificial neural networks as they are used in engineering are totally artificial. This is due to the fact that we must enforce a very unnatural order among interconnections between artificial neurons to guarantee decent execution times of our artificial neural network programs. Most artificial neural network programs rely on matrix manipulations for communications between neurons.

To this end, it is necessary to enforce a strict topology among the neuronic connections.

The most common artificial neural networks operate on layers of n artificial neurons. Within a given layer, none of the artificial neurons are connected to each other, but each artificial neuron is connected to every neuron of the next layer of m neurons. Fig.14.5 shows a typical network configuration.

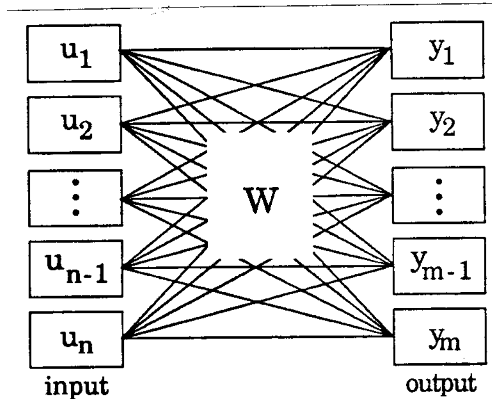

Figure 14.5. Simple artificial neural network

The computational advantage of this structure becomes evident when we apply eq(14.1) and eq(14.2) to this network. In matrix form, we can write:

$$\mathbf{x} = \mathbf{W} \cdot \mathbf{u} \tag{14.3a}$$

$$\mathbf{y} = \mathbf{f}(\mathbf{x}) \tag{14.3b}$$

where $\mathbf{x}$ and $\mathbf{y}$ are vectors of length m, and $\mathbf{u}$ is a vector of length n. $\mathbf{W}$ is the interconnection matrix between the first and the second layer. It has m rows and n columns. One single matrix multiplication and one vector function suffice to simulate the entire network with all its $n \times m$ connections.

What is this infamous problem discovered by Marvin Minsky which disgraced neural networks for more than a decade? A *perceptron* is an artificial neuron with a threshold output function as shown in Fig.14.3a. Let us look at a single perceptron with two inputs u_1 and u_2 and one output y. We wish to make this perceptron

learn an exclusive–or function, i.e., the function whose truth table is presented in Table 14.1.

Table 14.1 Truth table of exclusive–or gate

u_1	u_2	y
0	0	0
0	1	1
1	0	1
1	1	0

In our convention, '0' represents the logical state *false*, while '1' represents the logical state *true*. The output of the exclusive–or gate is *true* if the two inputs are in the opposite state, and it is *false*, if both inputs are in the same state. "Learning" in an artificial neural network sense simply means to adjust the values of the synaptic weights. Fig.14.6 depicts our perceptron.

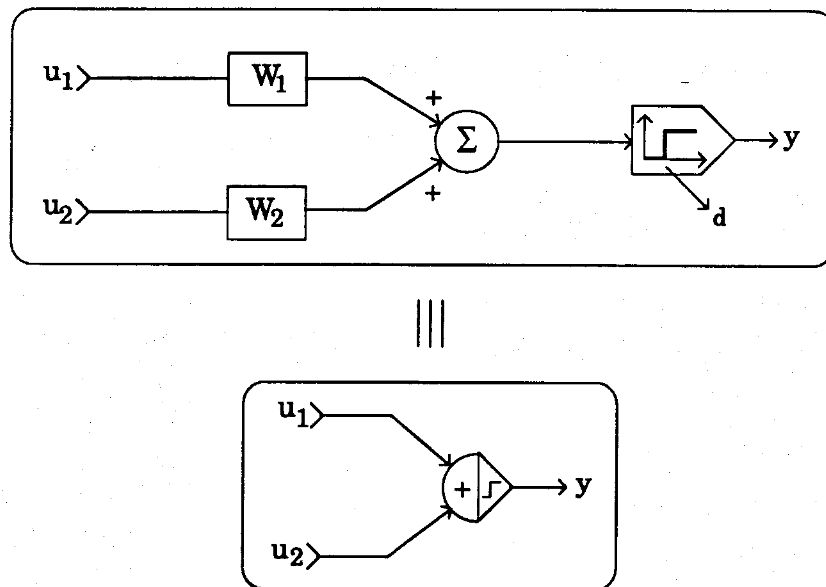

Figure 14.6. Single perceptron for exclusive–or gate

The state of this perceptron can be written as:

$$x = w_1\, u_1 + w_2\, u_2 \tag{14.4}$$

The output is '1' if the state is larger than a given threshold, d, and it is '0' otherwise. Thus, we can represent the perceptron in the following way:

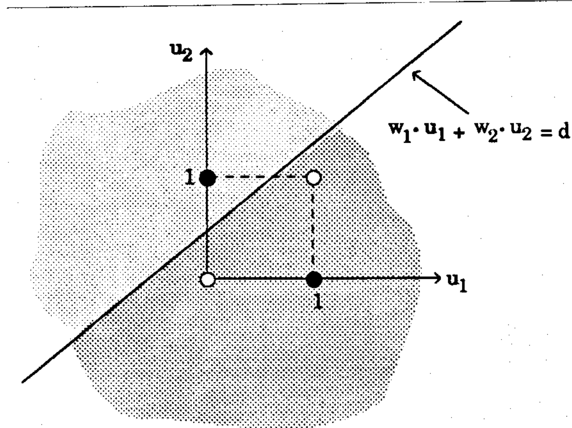

Figure 14.7. Perceptron solution of exclusive–or problem

Fig.14.7 shows the output of the perceptron in the $< u_1, u_2 >$ plane. The empty circles represent the desired '0' outputs, whereas the filled circles represent the desired '1' outputs. The slanted line separates the $< u_1, u_2 >$ plane into a half–plane for which the perceptron computes a value of '0' (below the line), and a half–plane for which it computes a value of '1' (above the line). By adjusting the two weight factors w_1 and w_2 and the threshold d, we can arbitrarily place the slanted line in the $< u_1, u_2 >$ plane. Obviously, no values of w_1, w_2, and d can be found which will place the two filled circles on one side of the line, and the two empty circles on the other. This simple truth sufficed to bring most artificial neural network research to a grinding halt.

Fig.14.8 shows a slightly more complex artificial neural network which can solve the exclusive–or problem.

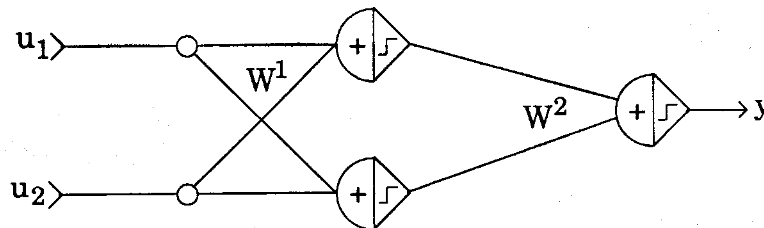

Figure 14.8. Two layer solution of exclusive–or problem

The enhanced network contains three perceptrons. Fig.14.9 shows how this network solves the exclusive–or problem.

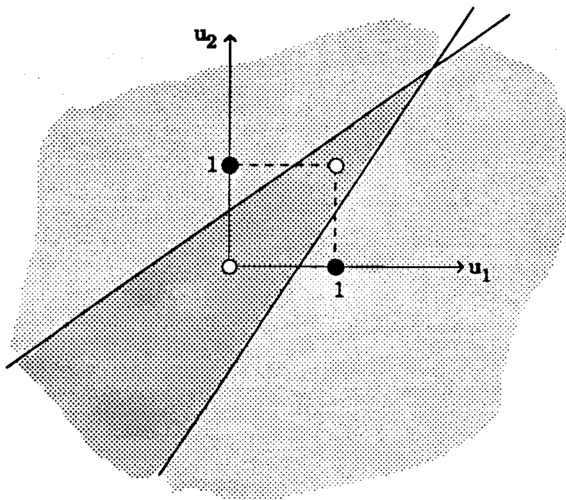

Figure 14.9. Two layer solution of exclusive–or problem

The darker shaded area of the $< u_1, u_2 >$ plane is the area for which the network computes a '1'. Thus, while the single layer network failed to solve the exclusive–or problem, a simple two layer network will do the trick. Multi–layer networks can mask arbitrary subspaces in the input space. These subspaces can be concave as well as convex.

Complex architectures of arbitrarily connected networks are still fairly seldom used. This is because, until recently, we didn't understand how to adjust the weighting coefficients of an arbitrarily connected network to learn a particular pattern (such algorithms will be introduced towards the end of this chapter). Backpropagation and counterpropagation are two training algorithms which have become popular since they provide systematic ways to train particular types of multi–layered feedforward networks. However, before we discuss artificial neural network learning in more detail, let me explain why artificial neural networks are attractive as tools for pattern recognition.

14.4 The Pattern Memorizing Power of Highly Rank–Deficient Matrices

We have seen that a one layer perceptron network multiplies the input vector $\mathbf{u}$ from the left with a weighting matrix $\mathbf{W}$ to determine the state $\mathbf{x}$ of the network. Let us assume that the input of our network is the ASCII code for a character, say Z:

$$\mathbf{u} = ASCII(Z) = (0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0)' \tag{14.5}$$

We choose our weighting matrix as the outer product of the input with itself except for the diagonal elements which are set equal to '0' [14.28]:

$$\mathbf{W} = \mathbf{u}\cdot\mathbf{u}' - \mathrm{DIAG}(\mathrm{DIAG}(\mathbf{u}\cdot\mathbf{u}')) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{14.6}$$

$\mathbf{W}$ is a dilute (sparsely populated) symmetric matrix of rank 4. If we feed this network with the correct input, the output is obviously the same as the input, since:

$$\mathbf{x} = \mathbf{W}\cdot\mathbf{u} \approx (\mathbf{u}\cdot\mathbf{u}')\cdot\mathbf{u} = \mathbf{u}\cdot(\mathbf{u}'\cdot\mathbf{u}) = k\cdot\mathbf{u} \tag{14.7}$$

The output threshold eliminates the k factor. The elimination of the diagonal elements is unimportant except for input vectors which contain one single '1' element only.

Now, let us perform a different experiment. We feed into the network a somewhat different input, say:

$$\mathbf{u} = (0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1)' \tag{14.8}$$

A comparison with the previous input shows that three bits are different. Amazingly, the network will still produce the same output as before. This network is able to recognize "similar" inputs, or equivalently, is able to filter out digital errors in a digital input signal. Hopfield has shown that this is a general property of such highly

rank–deficient sparsely populated matrices [14.14]. He showed that the technique works for input signals which have a smaller or equal number of '1' elements than '0' elements. Otherwise, we simply invert every bit of the input.

One and the same network can be used to recognize several different inputs u_i with and without distortions. For this purpose, we simply add the $\mathbf{W}_i$ matrices. Hopfield [14.14] showed that a matrix of size $n \times n$ can store up to $0.15 \cdot n$ different symbols. Thus, in order to store and recognize the entire upper–case alphabet, each letter should be represented by a vector of at least 174 bits in length. The ASCII code is completely inadequate for this purpose. It does not contain enough redundancy. Notice that the code sequence of eq(14.8) was recognized as the character Z, while in fact, it is the ASCII representation of another character, namely the character I.

A good choice may be a pixel representation of a 14×14 pixel field as shown in Fig.14.10.

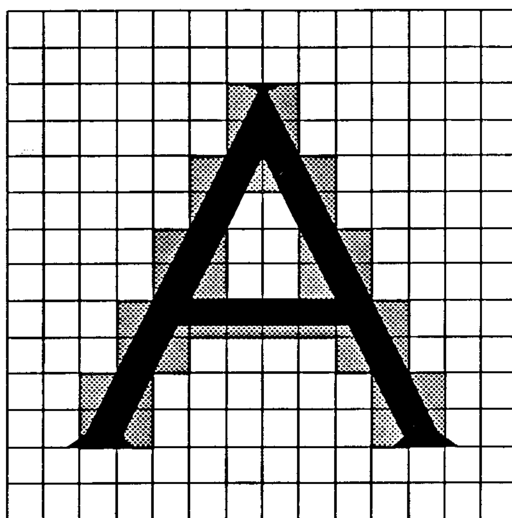

Figure 14.10. Pixel representation of a written character

We simply number the pixels from left to right, top to bottom. White pixels are represented as '0', while grey pixels are represented as '1'. We then write all the pixels into a vector of length 196. Minor variations in the writing of characters can be filtered out by the artificial neural network. This technique can be used to recognize hand–written characters. Some preprocessing will be necessary to

center the character in the pixel field, to normalize its size, and to correct for slanting.

The same method also works if the desired output is symbolically *different* from the given input. For example, it may be desirable to map the pixel representation of a hand–written, upper–case character into its ASCII code. In this case, we use the following weighting matrix:

$$\mathbf{W} = \sum_{\forall i} (\mathbf{y} \cdot \mathbf{u}') \tag{14.9}$$

where $\mathbf{u}$ is the 196 bit long pixel vector, and $\mathbf{y}$ is the corresponding eight bit long ASCII code vector. Thus, $\mathbf{W}$ is now a rectangular matrix of the size 8×196. This is how we believe the brain to map visual pixel information into symbols (not necessarily in ASCII format, of course).

Auditory information can be treated similarly. Here, we could sample and digitize the audio signal, and store an entire series of values sampled at successive sampling intervals as an input vector. Each time value is represented by a number of bits, perhaps eight. If we assume that the minimum required band width of the ear is 10 kHz, and if we assume that one spoken word lasts 1 sec, we need to store $8 \times 10,000$ bits in the input vector. According to Hopfield [14.14], 80,000 bits allow us to recognize 12,000 different words which should be sufficient for most purposes. We can then map the input vector of length 80,000 into an output vector of length 14, since 14 bits suffice to store 12,000 words. In this way, we should be able to comfortably recognize the spoken language. Of course, we need some additional preprocessing to "center" the spoken word in the time window, i.e., we need to determine when one word ends and when the next begins. We should also normalize the spoken word in length and altitude, i.e., compensate for fast *vs* slow speakers as well as for female *vs* male voices. Also, it may be better to operate on frequency signals instead of time signals. Between different speakers, the FFT of the spoken word varies less than the corresponding time signal.

Much more has been written about this topic. Several authors have discussed these types of weight matrices from a statistical point of view. They showed that optimal separation between different patterns is achieved if the input patterns are orthogonal to each other. The optimal $\mathbf{W}$ matrix turns out to be the input correlation matrix. A good overview is given in [14.12].

14.5 Supervised and Unsupervised Learning

We are now ready to discuss mechanisms of artificial neural network learning. In the last section, we *knew* how to set the synaptic weights. In a more general situation, this will not be the case. The question thus is: How do we modify the weights of our interconnection matrices such that the artificial neural network learns to recognize particular patterns?

Traditionally, two modes of neural network learning have been distinguished: supervised learning and unsupervised learning. In *supervised learning*, the artificial neural network "attends school". A "teacher" provides the network with an input, and asks the network for the appropriate output. The teacher then provides the network with the correct output, and uses either "positive reinforcement" or "punishment" to enhance the chance of a correct answer next time around.

Supervised (interactional) learning has been observed within humans and other living organisms at various levels. At the lowest level of learning, we notice *behavior modification schemes* (Skinnerism). These schemes do not assume that the person or animal has any *insight* regarding what s/he is taught. Behavior modification programs are prevalent in animal training, and also in the rehabilitation of the severely mentally retarded. In normal human education, behavior modification plays a minor role, yet this is the only form of learning that is being imitated by today's artificial neural networks.

At the next higher level, we should mention the mechanisms of *social learning* [14.1]. Social learning operates on concepts such as *social modeling* and *shaping*. The child learns through mechanisms of *vicarious learning* and *imitation*. Learning occurs in a feedback loop. The child reacts to external stimuli, but the teacher also modifies his or her own behavior on the basis of the reactions s/he observes in the child, and the child, in return, will notice the effects that his or her behavior has on the teacher's behavior, and so on. The problems and patterns which are presented to the child are initially very simple, and become more and more intricate as the child develops. This is a fruitful concept which could (and should) be adapted for use in artificial neural networks. Today's artificial neural network research focuses on network training, and ignores the importance of the training pairs. Optimization of training pairs for accelerated network learning would be a worthwhile research topic. Maybe, we can

create a "teacher network" which accepts arbitrary training pairs and preprocesses them for learning by a "student network". Perhaps, one type of "student network" could be trained to become a next generation "teacher network". Notice that social learning (like the previously discussed Skinnerism) focuses on a purely phenomenological view of learning. Social learning analyzes the interaction between the teacher and the student, but ignores what happens inside the student and/or the teacher.

At an even higher level, psychologists focus on the mechanisms of cognition. They discuss the role of *cognitive functions* (language, attention span, learning abilities, reasoning, and memorization) [14.26]. At this level, we no longer focus on *what* is learned, but rather on *how* learning occurs, and how we "learn to learn". In analogy to artificial neural network terminology, this corresponds to using a second artificial neural network in conjunction with the original network. The output of the second network is the algorithm that the first network uses to adapt its weights. We then use a third network on top of the second one, and so forth, until we come to a point where the highest level network has become so abstract and general that it doesn't require any training at all, but can be used to bootstrap itself. Presently, our state–of–the–art neural networks are far from such degrees of complexity. All that we have achieved with our artificial neural networks is the capability to emulate only the lowest level of behavioral modification schemes.

Supervised training of artificial neural networks is attractive because it is fairly easy to implement. It is unattractive because the network is kept in an unproductive learning phase for a long time before the learned knowledge can be applied to solve real problems. In a very simple analogy, our children have to grow up and attend school for many years before they can be integrated into the work force and "make money" on their own. However, the analogy is not truly appropriate since the child is *extremely* productive during his or her training period except for one particular aspect: the economic one.

In *unsupervised learning*, the network can be used immediately, but it will produce increasingly better results as time passes. Learning is accomplished by comparing the current input/target pair with previous similar input/target pairs. The network is trained to produce consistent results [14.34]. Many biologists insist that at least low level learning occurs in a basically unsupervised mode [14.34]. I am not so sure that this is correct. Even motoric functions are

learned through mechanisms of reinforcement and punishment (behavior modification) and through mechanisms of imitation (social learning). Walking is learned by falling many times, and by trying to imitate walking adults. Yet, the child indeed does not learn to walk in a "dry run" mode, and walks only after mastering the problem in theory. The dichotomy between supervised and unsupervised learning is an artifact. Usually, artificial neural networks are studied in isolation. Humans do not learn in isolation. They live in an environment to which they react, and which reacts to them in return. Real learning is similar to *adaptive control.* The network is constantly provided with real inputs from which it computes real outputs, and it is also provided with desired outputs to compare them with. It then tries to modify its behavior until the real outputs resemble the desired outputs. However, while the network learns, it is constantly "in use". Thus, it would make sense to replace the term *unsupervised learning* by the term *adaptive learning.*

The real problem with artificial neural network learning lies somewhere else. Artificial neural networks can observe only the inputs and outputs of the real system, but not its internal states. The question thus is: How does the network modify the weights of internal hidden layers? This dilemma is at the origin of the (artificial) distinction between supervised and unsupervised learning.

The answer to this question is quite simple: Artificial neural network training is an extremely slow and tedious process. Modifying the weights of the internal network layers is accomplished either by trial and error, or by gradient propagation. The real question thus is: How do we guarantee the convergence of the weight adaptation algorithm in use?

We shall address this question in greater generality towards the end of this chapter. For now, we shall restrict ourselves to the analysis of a few simple feedforward network topologies, and develop learning algorithms for those networks only.

Single Layer Networks

Let us compute the difference between the jth desired output $\hat{y}_j$ and the jth real output y_j:

$$\delta_j = \hat{y}_j - y_j \tag{14.10}$$

In the case of a perceptron network, δ_j will be either -1, 0, or $+1$. In the case of other networks, δ_j can be a continuous variable. We

then compute the change in the weight from the ith input to the jth output as follows:

$$\Delta_{ji} = g \cdot \delta_j \cdot u_i \tag{14.11}$$

The rationale for this rule is simple. If the jth real output is too small, δ_j is positive. In this case, we need to reinforce those inputs which are strong by making them even stronger. This will raise the output level. However, if the jth real output is too large, δ_j is negative. In this case, we need to weaken those inputs which are strong. This will reduce the output level.

The constant multiplier g is the *gain value* of the algorithm. For larger values of g, the algorithm converges faster if it does indeed converge, but it is less likely to converge. For smaller values of g, the chance of convergence is greater, but convergence will take longer.

Finally, we update the weight w_{ji} in the following manner:

$$w_{ji_{new}} = w_{ji_{old}} + \Delta_{ji_{old}} \tag{14.12}$$

In a matrix form, we can summarize eq(14.10) to eq(14.12) as follows:

$$\mathbf{W_{k+1}} = \mathbf{W_k} + g \cdot (\hat{\mathbf{y}} - \mathbf{y_k}) \cdot \mathbf{u_k}' \tag{14.13}$$

Eq(14.13) is commonly referred to as the *delta rule* [14.35].

Notice the following special case. If we choose the initial weighting matrix as zero, $\mathbf{W_0} = 0$, then the initial output will also be zero, $\mathbf{y_0} = 0$. If we furthermore choose a gain factor of $g = 1.0$, we can compute the weighting matrix after one single iteration as follows:

$$\mathbf{W_1} = \hat{\mathbf{y}} \cdot \mathbf{u_0}' \tag{14.14}$$

which is identical with the explicit weight assignment formula, eq(14.9), that we used in the last section.

For most applications, we shall choose a considerably smaller gain value such as: $g = 0.01$, and we shall start with a small random weighting matrix:

$$\mathbf{W_0} = 0.01 * \text{RAND}(m, n) \tag{14.15}$$

Backpropagation Networks

Backpropagation networks are multi–layer networks in which the various layers are cascaded. Fig.14.11 shows a typical three–layer backpropagation network.

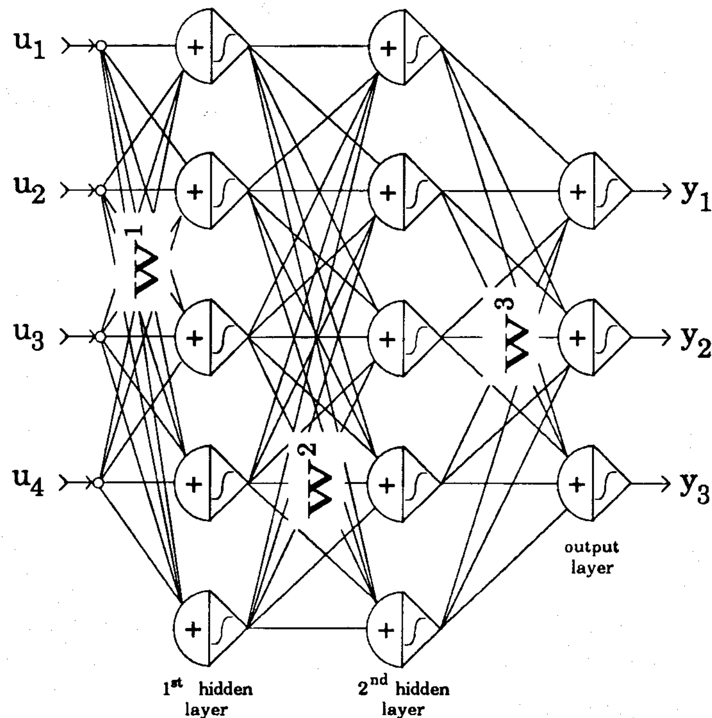

Figure 14.11. Three–layer backpropagation network

In order for the backpropagation learning algorithm to work, we must assume that the activation function of each artificial neuron is differentiable over its entire input range. Thus, perceptrons cannot be used in backpropagation networks.

The most commonly used activation function in a backpropagation network is the *sigmoid* function:

$$y = \text{sigmoid}(x) = \frac{1.0}{1.0 + \exp(-x)} \tag{14.16}$$

The shape of the sigmoid function is graphically shown in Fig.14.12.

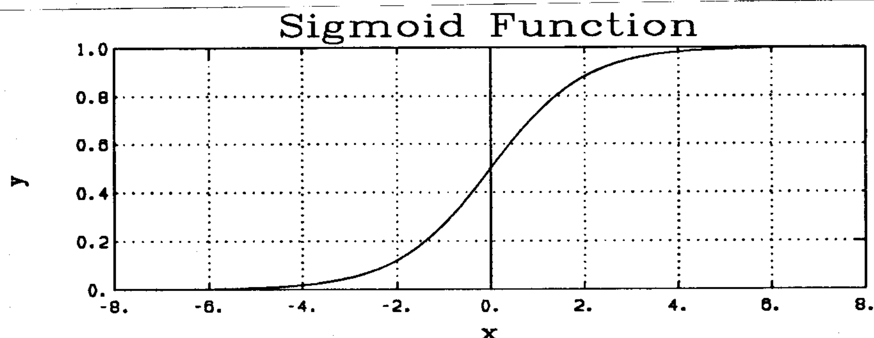

Figure 14.12. Sigmoid function

The sigmoid function is particularly convenient because of its simple partial derivative:

$$\frac{\partial y}{\partial x} = y \cdot (1.0 - y) = \text{logistic}(y) \tag{14.17}$$

The partial derivative of the output y with respect to state x does not depend on x explicitly. It can be written as a *logistic* function of the output y.

We shall train the output layer in basically the same manner as in the case of the single layer network, but we shall modify the formula for δ_j. Instead of simply using the difference between the desired output $\hat{y}_j$ and the true output y_j, we multiply this difference by the activation gradient:

$$\delta_j = \frac{\partial y}{\partial x} \cdot (\hat{y}_j - y_j) = y_j \cdot (1.0 - y_j) \cdot (\hat{y}_j - y_j) \tag{14.10^{alt}}$$

Therefore, the matrix version of the learning algorithm for the output layer can now be written as:

$$\mathbf{W}^{\mathrm{n}}_{\mathrm{k+1}} = \mathbf{W}^{\mathrm{n}}_{\mathrm{k}} + g * \left(\mathbf{y}^{\mathrm{n}}_{\mathrm{k}} \;.\!*\; (\mathrm{ONES}(\mathbf{y}^{\mathrm{n}}_{\mathrm{k}}) - \mathbf{y}^{\mathrm{n}}_{\mathrm{k}}) \;.\!*\; (\hat{\mathbf{y}} - \mathbf{y}^{\mathrm{n}}_{\mathrm{k}})\right) * \mathbf{u}^{\mathrm{n}\prime}_{\mathrm{k}} \tag{14.13^{alt}}$$

The subscript k denotes the kth iteration, whereas the superscript n denotes the nth stage (layer) of the multi-layer network. I assume that the network has exactly n stages. Eq(14.13^{alt}) is written in a pseudo CTRL-C (MATLAB) style. The '$*$' operator denotes a regular matrix multiplication, whereas the '$.*$' operator denotes an

elementwise multiplication. The vector $\mathbf{u}_k^n$ is obviously identical with $\mathbf{y}_k^{n-1}$. Let:

$$\vec{\delta}_k^n = \mathbf{y}_k^n \;.*\; (\mathrm{ONES}(\mathbf{y}_k^n) - \mathbf{y}_k^n) \;.*\; (\hat{\mathbf{y}} - \mathbf{y}_k^n) \tag{14.18}$$

denote the kth iteration of the $\vec{\delta}$ vector for the nth (output) stage of the multi–layer network. Using eq(14.18), we can rewrite eq(14.13^{alt}) as follows:

$$\mathbf{W}_{k+1}^n = \mathbf{W}_k^n + g \;*\; \vec{\delta}_k^n \;*\; \mathbf{u}_k^{n\prime} \tag{14.19}$$

Unfortunately, this algorithm will work for the output stage of the multi–layer network only. We cannot train the hidden layers in the same fashion since we don't have a "desired output" for these stages. Therefore, we replace the gradient by another (unsupervised) updating function. The $\vec{\delta}$ vector of the ℓth hidden layer is computed as follows:

$$\vec{\delta}_k^\ell = \mathbf{y}_k^\ell \;.*\; (\mathrm{ONES}(\mathbf{y}_k^\ell) - \mathbf{y}_k^\ell) \;.*\; (\mathbf{W}_k^{\ell+1\prime} \;*\; \vec{\delta}_k^{\ell+1}) \tag{14.20}$$

Instead of weighing the $\vec{\delta}$ vector with the (unavailable) difference between the desired and the true output of that stage, we propagate the weighted $\vec{\delta}$ vector of the subsequent stage back through the network. We then compute the next iteration of the weighting matrix of this hidden layer using eq(14.19) applied to the ℓth stage, i.e.:

$$\mathbf{W}_{k+1}^\ell = \mathbf{W}_k^\ell + g \;*\; \vec{\delta}_k^\ell \;*\; \mathbf{u}_k^{\ell\prime} \tag{14.21}$$

In this fashion, we proceed backwards through the entire network.

The algorithm starts by setting all weighting matrices to small random matrices. We apply the true input to the network. we propagate the true input *forward* to the true output, generating the first iteration on all signals in the network. We then propagate the gradients *backward* through the network to obtain the first iteration on all the weighting matrices. We then use these weighting matrices to propagate the same true input once more *forward* through the network to obtain the second iteration on the signals, and then propagate the modified gradients *backward* through the entire network to obtain the second iteration on the weighting matrices. Consequently, the $\mathbf{u}^\ell$ and $\mathbf{y}^\ell$ vectors of the ℓth stage are updated on the forward path, while the $\vec{\delta}^\ell$ vector and the $\mathbf{W}^\ell$ matrix are updated on the backward path. Each iteration consists of one forward path followed by one backward path.

The backpropagation algorithm was made popular by Rumelhart *et al.* [14.31]. It presented the artificial neural network research community with the first systematic (although still heuristic) algorithm for training multi–layer networks. The backpropagation algorithm has a fairly benign stability behavior. It will converge on many problems provided the gain g has been properly selected. Unfortunately, its convergence speed is usually very slow. Typically, a backpropagation training session may require several hundred thousand iterations for convergence.

Several enhancements of the algorithm have been proposed. Frequently, a bias vector is added, i.e. the state of an artificial neuron is no longer the weighted sum of its inputs alone, but is computed using the formula:

$$\mathbf{x} = \mathbf{W} \cdot \mathbf{u} + \mathbf{b} \tag{14.22}$$

Conceptually, this is not a true enhancement. It simply means that the neuron has an additional input which is always '1'. Consequently, the bias term is updated as follows:

$$\mathbf{b_{k+1}} = \mathbf{b_k} + g \cdot \vec{\delta}_k \tag{14.23}$$

Also, a small "momentum" term is frequently added to the weights in order to improve the convergence speed [14.18]:

$$\mathbf{W_{k+1}} = (1.0 + m) \cdot \mathbf{W_k} + g \cdot \vec{\delta}_k \cdot \mathbf{u_k}' \tag{14.24a}$$

The momentum should obviously be added to the bias term as well:

$$\mathbf{b_{k+1}} = (1.0 + m) \cdot \mathbf{b_k} + g \cdot \vec{\delta}_k \tag{14.24b}$$

The momentum m is usually very small, $m \approx 0.01$.

Other references add a small percentage of the last change in the matrix to the weight update equation [14.12]:

$$\mathbf{\Delta W_k} = g \cdot \vec{\delta}_k \cdot \mathbf{u_k}' \tag{14.25a}$$

$$\mathbf{W_{k+1}} = \mathbf{W_k} + \mathbf{\Delta W_k} + m \cdot \mathbf{\Delta W_{k-1}} \tag{14.25b}$$

Finally, it is quite common to limit the amount by which the $\vec{\delta}$ vectors, the $\mathbf{b}$ vectors, and the $\mathbf{W}$ matrices can change in a single step. This often improves the stability behavior of the algorithm.

Counterpropagation Networks

Robert Hecht–Nielsen [14.12] introduced a two–layer network which can be trained much more quickly than the backpropagation network. In fact, counterpropagation networks can be trained *instantaneously.* We can provide an *explicit algorithm* for generating the weighting matrices of counterpropagation networks. The idea behind counterpropagation is fairly simple. The problem of teaching a two–layer network to map an arbitrary set of input vectors into another arbitrary set of output vectors can be decomposed into two simpler problems:

(1) We map the arbitrary set of inputs into an intermediate (hidden) digital layer in which the kth input/target pair is represented by the kth unit vector, $\mathbf{e_k}$:

$$\mathbf{e_k} = [\ 0\ ,\ 0\ ,\ \ldots\ ,\ 0\ ,\ 1\ ,\ 0\ ,\ \ldots\ ,\ 0\ ]' \tag{14.26}$$

The kth unit vector, $\mathbf{e_k}$ is a vector of length k which contains only '0' elements except in its kth position where it contains a '1' element.

(2) We map the intermediate (hidden) digital layer into the desired arbitrary output layer.

Obviously, the length of the hidden layer must be as large as the number of different input/target pairs that we wish to train the network with.

The map from the intermediate layer to the output layer is trivial. Since each output is driven by exactly one '1' source, and since this source doesn't drive any other output, the output layer weighting matrix consists simply of a horizontal concatenation of the desired outputs:

$$\mathbf{W^2} = [\ \mathbf{y_1^2}\ ,\ \mathbf{y_2^2}\ ,\ \ldots\ ,\ \mathbf{y_n^2}\ ] \tag{14.27}$$

where $\mathbf{y_i^2}$ denotes the ith output vector of the second stage which is driven by the ith unit input vector of the second stage which is identical with the ith output vector of the first stage, $\mathbf{y_i^1}$.

Thus, the interesting question is: Can we train a single layer perceptron network (the first stage of the counterpropagation network) to map each input into an output such that exactly one of the output elements is '1' while all other output elements are '0', and such that no two output vectors are identical if their input vectors are different?

Teuvo Kohonen [14.17] addressed this question. In a way, the two layers of the counterpropagation network are inverse to each other. The optimal weighting matrix can be written as follows:

$$\mathbf{W}^1 = [\, \mathbf{u}_1^1 \,,\, \mathbf{u}_2^1 \,,\, \ldots \,,\, \mathbf{u}_n^1 \,]' \tag{14.28}$$

The weighting matrix of the first stage is the transpose of the matrix which consists of a horizontal concatenation of the input vectors.

The threshold of the perceptrons d is automatically adjusted such that only one of the states of the first stage is larger than the threshold.

Let us apply this technique to the problem of reading hand–written characters. The purpose is to map the pixel representation of hand–written upper–case characters to their ASCII code. Counterpropagation enables us to solve the character reading problem in one single step without need for any training at all. Let us assume that each input character is resolved in a 14×14 pixel matrix. Thus, each character is represented by a pixel vector of length 196. Since the alphabet contains 26 different upper–case characters, the hidden Kohonen layer must be of length 26. Each of the outputs is of length eight. Thus, the total network contains 34 artificial neurons: 26 perceptrons and eight linear output neurons. The dimension of the weighting matrix of the first stage is 26×196. We store the pixel representation of the ideally written character A as the first row vector of $\mathbf{W}^1$. We then concatenate from below the pixel representation of the ideally written character B, etc. This will map the pixel representation of the kth character of the alphabet into the kth unit vector $\mathbf{e_k}$. The dimension of the weighting matrix of the second stage is 8×26. We store the ASCII representation of the character A as the first column vector of $\mathbf{W}^2$, and concatenate from the right the ASCII representation of the character B, etc. This will map the unit vector representation of the kth character of the alphabet into its ASCII representation.

Several unsupervised training algorithms have been devised which make the network adaptive to variations in the input vectors. A typically used updating rule is the following:

$$\mathbf{ww}_{k+1}^{1}{}' = \mathbf{ww}_k^{1'} + g \cdot (\mathbf{u}_k^{1'} - \mathbf{ww}_k^{1'}) \tag{14.29}$$

where $\mathbf{ww}^{1'}$ denotes the "winning" row vector of the $\mathbf{W}^1$ matrix. According to eq(14.29), we don't update the entire $\mathbf{W}^1$ matrix at once. Instead, we update only one row at a time, namely the row

that corresponds to the "winning" output of the Kohonen layer, i.e., the one output which is '1' while all others are '0'. Eq(14.29) is frequently referred to as the *Kohonen learning rule.* Notice that Kohonen learning does not always work. Improved learning techniques have been proposed. However, we shall refrain from discussing these updating algorithms here in more detail. Additional information is provided in Wasserman [14.34].

In general, *counterpropagation networks* work fairly well for digital systems, i.e., mappings of binary input vectors into binary output vectors. They don't work very well for continuous systems since the hidden layer must be digital and must enumerate all possible system states. *Backpropagation networks* work fairly well for continuous systems. They don't work well for systems with digital output, because their activation functions must be continuously differentiable.

Notice that the counterpropagation networks introduced in this section are somewhat different from those traditionally found in the artificial neural network literature. While the general idea behind the counterpropagation architecture is the same, the explicit algorithm for generating the weighting matrices is more appealing. Classical counterpropagation networks randomize the first (Kohonen) layer, and then use eq(14.29) to train the Kohonen layer. They use a so–called *Grossberg outstar* as the second layer. However, my algorithm is more attractive since it doesn't require any iterative learning, and since it works reliably for all types of digital systems, whereas eq(14.29) often leads to convergence problems.

14.6 Neural Network Software

Until now, we have written down all our formulae in this chapter in a pseudo MATLAB (CTRL–C) format. This was convenient since the reader should be familiar with this nomenclature by now.

Let us use this approach to solve an example problem. We wish to design a counterpropagation network to solve the infamous exclusive–or problem. Since this is a digital system, we expect the counterpropagation network to work well. Fig.14.13 depicts the counterpropagation network for this problem.

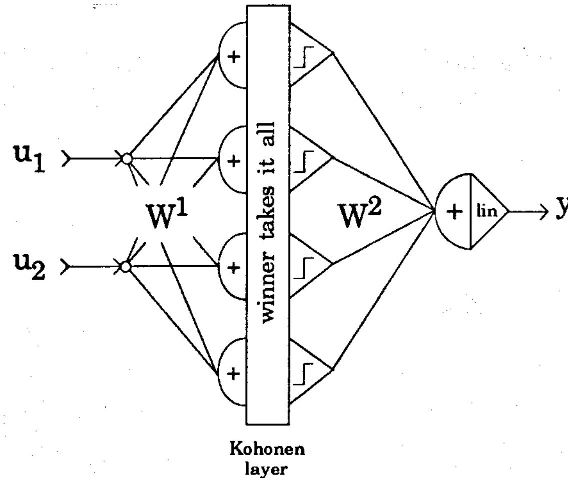

Figure 14.13. Counterpropagation network for XOR

This system has two inputs and one output. Since the truth table contains four different states, the hidden layer must be of length four. Until now, we have always used the state '0' to denote *false*, and the state '1' to denote *true*. This is not really practical for most artificial neural networks. Therefore, we shall use the real value -1.0 to denote the logical state *false*, and the real value $+1.0$ to denote the logical state *true* in both the input vector and the output. The hidden layer will still use '0' and '1'.

With this exception, the counterpropagation network functions as described in the previous section. The code for this network is shown below.

```
// This procedure designs a counterpropagation network for XOR
//
DEFF winner -c
//
// Define the input and target vectors
//
inpt = [ -1  -1   1   1
         -1   1  -1   1 ];
target = [ -1   1   1  -1 ];
//
// Set the weighting matrices
//
W1 = inpt';
W2 = target;
```

```
// Apply the network to evaluate the truth table
//
y = ZROW(target);
FOR nbr = 1:4, ...
   u1 = inpt(:, nbr); ...
   x1 = W1 * u1; ...
   y1 = WINNER(x1); ...
   u2 = y1; ...
   x2 = W2 * u2; ...
   y2 = x2; ...
   y(nbr) = y2; ...
END
//
// Display the results
//
y
//
RETURN
```

This procedure is fairly self–explanatory. The function *WINNER* determines the largest of the perceptron states, and assigns a value of +1.0 to that particular output in a "winner–takes–it–all" fashion. The function *WINNER* is shown below.

```
// [y] = WINNER(x)
//
// This procedure computes the winner function
//
[n, m] = SIZE(x);
ind = SORT(x);
y = ZROW(n, m);
y(ind(n)) = 1;
//
RETURN
```

The function *WINNER* sorts the input vector x in increasing order. The vector *ind* is not the sorted array, but an index vector that shows the position of the various elements in the original vector. Thus, $x(ind(1))$ is the smallest element of the x vector, and $x(ind(n))$ is the largest element of the x vector. We then set the output vector y to 0.0, except for the winning element which is set equal to +1.0.

The performance of the counterpropagation network is flawless. However, the network is not error tolerant, since the input is digital and does not contain any redundancy. Consequently, a single bit error converts the desired input vector into another undesired, yet equally legal, input vector. The *Hamming distance* between two neighboring legal input states is exactly one bit.

Let us now try to solve the same problem using a backpropagation network. We expect problems since the system is digital. Fig.14.14 shows the resulting backpropagation network.

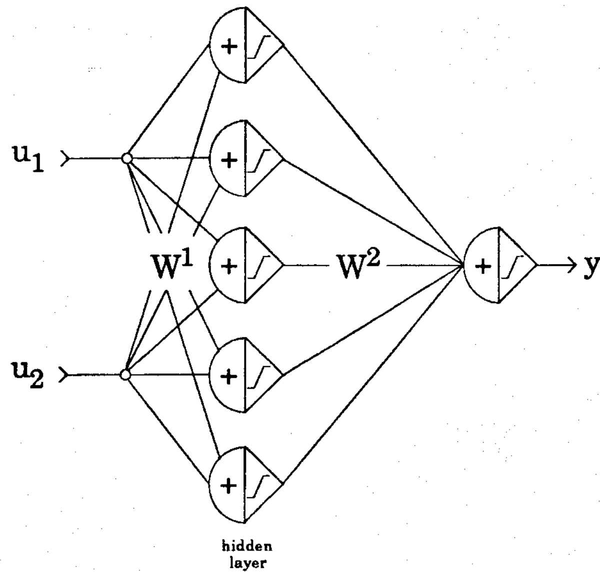

Figure 14.14. Backpropagation network for XOR

The length of the hidden layer is arbitrary. In our program, we made this a parameter, *lhid*, which can be chosen at will. The program is shown below.

```
// This procedure designs a backpropagation network for XOR
// Select the length of the hidden layer (LHID) first
//
DEFF limit -c
DEFF tri -c
//
// Define the input and target vectors
//
inpt = [ -1  - 1   1   1
         -1    1 - 1   1 ];
target = [ -1   1   1 - 1 ];
```

```
// Set the weighting matrices and biases
//
W1 = 0.1 * (2.0*RAND(lhid, 2)- ONES(lhid, 2));
W2 = 0.1 * (2.0*RAND(1, lhid)- ONES(1, lhid));
b1 = ZROW(lhid, 1);   b2 = ZROW(1);
WW1 = ZROW(lhid, 2);   WW2 = ZROW(1, lhid);
bb1 = ZROW(lhid, 1);   bb2 = ZROW(1);
//
// Set the gains and momentums
//
g1 = 0.6;   g2 = 0.3;
m1 = 0.06;   m2 = 0.03;
//
// Set the termination condition
//
crit = 0.025;   error = 1.0;   count = 0;
//
// Learn the weights and biases
//
WHILE error > crit, ...
  count = count + 1; ...
  ... //
  ... // Loop over all input/target pairs
  ... //
  error = 0; ...
  FOR nbr = 1:4, ...
    u1 = inpt(:, nbr); ...
    y2h = target(nbr); ...
    ... //
    ... // Forward pass
    ... //
    x1 = WW1 * u1 + bb1; ...
    y1 = LIMIT(x1); ...
    u2 = y1; ...
    x2 = WW2 * u2 + bb2; ...
    y2 = LIMIT(x2); ...
    ... //
    ... // Backward pass
    ... //
    e = y2h - y2; ...
    delta2 = TRI(y2) .* e; ...
    W2 = W2 + g2 * delta2 * (u2') + m2 * WW2; ...
    b2 = b2 + g2 * delta2 + m2 * bb2; ...
    delta1 = TRI(y1) .* ((WW2') * delta2); ...
    W1 = W1 + g1 * delta1 * (u1') + m1 * WW1; ...
    b1 = b1 + g1 * delta1 + m1 * bb1; ...
    error = error+ NORM(e); ...
  END, ...
```

```
      ... // Update the momentum matrices and vectors
      ... //
      WW1 = W1;   WW2 = W2; ...
      bb1 = b1;   bb2 = b2; ...
   END
   //
   // Apply the learned network to evaluate the truth table
   //
   y = ZROW(target);
   FOR nbr = 1:4, ...
      u1 = inpt(:, nbr); ...
      x1 = WW1 * u1 + bb1; ...
      y1 = LIMIT(x1); ...
      u2 = y1; ...
      x2 = WW2 * u2 + bb2; ...
      y2 = LIMIT(x2); ...
      y(nbr) = y2; ...
   END
   //
   // Display the results
   //
   y
   //
   RETURN
```

It took some persuasion to get this program to work. The first difficulty was with the activation functions. The *sigmoid* function is no longer adequate since the output varies between -1.0 and $+1.0$, and not between 0.0 and 1.0. In this case, the *sigmoid* function is frequently replaced by:

$$y = \frac{2}{\pi} \cdot \tan^{-1}(x) \tag{14.30}$$

which has also a very convenient partial derivative:

$$\frac{\partial y}{\partial x} = \frac{2}{\pi} \cdot \frac{1.0}{1.0 + x^2} \tag{14.31}$$

However, also this function won't converge for our application. Since we wish to obtain outputs of exactly $+1.0$ and -1.0, we would need infinitely large states, and therefore infinitely large weights.

Without the $\frac{2}{\pi}$ term, the network does learn, but converges very slowly. Therefore, we decided to eliminate the requirement of a continuous derivative, and used a *limit* function as the activation function:

```
// [y] = LIMIT(x)
//
// This procedure computes the limit function
//
[n, m] = SIZE(x);
FOR i = 1:n, ...
   y(i) = MIN([MAX([x(i), -1.0]), 1.0]); ...
END
//
RETURN
```

In this case, we cannot backpropagate the gradient. Instead, we make use of the fact that we *know* that all outputs must converge to either $+1.0$ or -1.0. We therefore punish the distance of the true output from either of these two points using the *tri* function [14.18]:

```
// [y] = TRI(x)
//
// This procedure computes the tri function
//
[n, m] = SIZE(x);
y = ONES(n, m)- ABS(x);
//
RETURN
```

We call this type of network a *pseudo-backpropagation network.*

In addition to the weighting matrices, we needed biases and momentums. The optimization starts with a zero weight matrix, but adds small random momentums to the weights and to the biases. After each iteration, the momentums are updated to point more towards the optimum solution.

The program converges fairly quickly. It usually takes less than 20 iterations to converge to the correct solution. The program is also fairly insensitive to the length of the hidden layer. The convergence is equally fast with $lhid = 8$, $lhid = 16$, and $lhid = 32$.

This discussion teaches us another lesson. The design of neural networks is still more an art than a science. We usually start with one of the classical textbook algorithm ... and discover that it doesn't work. We then modify the algorithm until it converges in a satisfactory manner for our application. However, there is little generality in this procedure. A technique that works in one case, may fail when applied to a slightly different problem. The backpropagation algorithm, as presented in this section, was taken from [14.18]. Korn's new book contains a wealth of little tricks and ideas how the convergence speed of neural network algorithms can be improved.

Notice the gross difference between backpropagation and the functioning of our brain. Backpropagation learning is a *gradient technique.* Such optimization techniques have a tendency to either diverge or converge on a local minimum. In comparison, our brain learns slowly but reliably, and doesn't exhibit any such convergence problems. Thus, gradient techniques, beside from the fact that they are not biologically plausible, may not be the best of all learning techniques. In an artificial neural network, the *robustness* of the optimization technique is much more important than the *convergence speed.* Unfortunately, these two performance parameters are always in competition with each other, as I shall demonstrate in the second volume of this text.

I have explained earlier that most artificial neural network programs (such as backpropagation networks) will need many iterations for convergence. However, one thing that we certainly *don't* want to do is to rerun a MATLAB or CTRL–C procedure several hundred thousand times. The efficiency of such a program would by incredibly poor. Thus, MATLAB and CTRL–C are useful for *documenting* neural network algorithms, but not for using them in an actual implementation. (Because the previously demonstrated exclusive–or problem is trivial from a computational point of view, CTRL–C was able to solve this problem acceptably fast.)

Special artificial neural network hardware is currently under development. Neural network algorithms lend themselves to massive parallel processing, thus, a hardware solution is clearly indicated. However, today's neural network chips are still exorbitantly expensive and not sufficiently flexible. For the time being, we must therefore rely on software simulation tools.

Granino Korn recently developed a new DESIRE dialect, called DESIRE/NEUNET [14.18]. This code has been specially designed for the simulation of artificial neural networks. It has been optimized for fast compilation and fast execution. The DESIRE/NEUNET solution of the exclusive–or backpropagation network is given below.

```
------------------------------------
-- ARTIFICIAL NEURAL NETWORK
-- Exclusive-Or Backpropagation
------------------------------------
-- Constants
lhid = 8 | g1 = 0.6 | g2 = 0.3
m1 = 0.06 | m2 = 0.03 | crit = 0.025
-- Declarations
ARRAY u1[2], y1[lhid], y2[1]
ARRAY W1[lhid, 2], W2[1, lhid], b1[lhid], b2[1]
ARRAY WW1[lhid, 2], WW2[1, lhid], bb1[lhid], bb2[1]
ARRAY delta1[lhid], delta2[1], e[1]
ARRAY inpt[4, 2], target[4, 1]
-- Read Constant Arrays
data -1, -1; -1, 1; 1, -1; 1, 1 | read inpt
data -1; 1; 1; -1 | read target
-- Initial conditions
for i = 1 to lhid
   W1[i, 1] = 0.1*ran(0) | W1[i, 2] = 0.1*ran(0)
   W2[1, i] = 0.1*ran(0)
   WW1[i, 1] = 0.0 | WW1[i, 2] = 0.0
   WW2[1, i] = 0.0
   b1[i] = 0.0 | bb1[i] = 0.0
next
b2 = 0.0 | bb2 = 0.0 | error = 0.0
min = -1.0 | max = 1.0 | sr = 4
------------------------------------
TMAX = 20.0 | t = 1 | NN = 20
------------------------------------
-- scaling
scale = 5
drun TEACH
drun RECALL
------------------------------------
DYNAMIC
------------------------------------
label TEACH
iRow = t | VECTOR u1 = inpt#
VECTOR y1 = WW1 * u1 + bb1; min, max
VECTOR y2 = WW2 * y1 + bb2; min, max
VECTOR e = target# - y2
VECTOR delta2 = e * tri(y2)
VECTOR delta1 = WW2% * delta2*tri(y1)
LEARN W1 = g1 * delta1 * u1 + m1 * WW1
LEARN W2 = g2 * delta2 * y1 + m2 * WW2
UPDATE b1 = g1 * delta1 + m1 * bb1
UPDATE b2 = g2 * delta2 + m2 * bb2
DOT e2 = e * e | error = error + e2
```

```
--------------------------------------------
SAMPLE sr
term crit - error
error = 0.0
MATRIX WW1 = W1  | MATRIX WW2 = W2
VECTOR bb1 = b1  | VECTOR bb2 = b2
dispt error
--------------------------------------------
label RECALL
iRow = t  | VECTOR u1 = inpt#
VECTOR y1 = WW1 * u1 + bb1; min, max
VECTOR y2 = WW2 * y1 + bb2; min, max
in1 = u1[1]  |  in2 = u1[2]  |  out1 = y2[1]
type in1, in2, out1
--------------------------------------------
/--
/PIC 'xor.prc'
/--
```

This DESIRE program exhibits a number of new features that we never before met. DESIRE is able to handle vectors and matrices in a somewhat unflexible but extremely efficient manner. Contrary to CTRL–C (or MATLAB), DESIRE does not allow us to easily manipulate individual elements within matrices or vectors. DESIRE's matrix and vector operators deal with the whole data structure at once. Thus, some algorithms, such as the genetic algorithms described later in this chapter, can be elegantly programmed in CTRL–C, while they are almost impossible to implement in DESIRE. Yet, DESIRE works very well for many classical neural network algorithms such as backpropagation, and counterpropagation.

In DESIRE, all vectors and matrices must be *declared* using an *ARRAY* statement. *ARRAY* declarations can make use of previously defined constants.

Within the DYNAMIC block, vector assignments can be made using the *VECTOR* statement. The '*' operator in a vector assignment denotes either the multiplication of a matrix with a vector, or the multiplication of a scalar with a vector, or the elementwise multiplication of two vectors. Notice that the *VECTOR* statement has been explicitly designed for the simulation of artificial neural networks. The statement:

$$\textbf{VECTOR}\ y1 = WW1 * u1 + bb1;\ min,\ max \tag{14.32}$$

computes the state vector of a neural network, and simultaneously its output using a hard limiter as the activation function. DESIRE/NEUNET also offers most other commonly used activation functions and their derivatives as system defined functions, such as the *tri* function used in the above program. Remember that DESIRE is case sensitive. The *VECTOR* assignment:

$$\textbf{VECTOR}\ y = y + x \tag{14.33}$$

can be abbreviated as:

$$\textbf{UPDATE}\ y = x \tag{14.34}$$

Matrix assignments can be made using the *MATRIX* statement. The '*' operator in a *MATRIX* statement denotes the outer (Hadamard) product of two vectors, or the multiplication of a scalar with a matrix. The *MATRIX* assignment:

$$\textbf{MATRIX}\ A = A + B \tag{14.35}$$

can be abbreviated as:

$$\textbf{LEARN}\ A = B \tag{14.36}$$

The *DOT* statement computes the inner product of two vectors. The result is a scalar.

The '%' operator denotes a matrix transpose. *WW2%* is the transpose of matrix *WW2*. The '#' operator is a special row vector extraction operator. In the above program, *inpt* is a matrix. *inpt#* extracts one particular row from that matrix, namely the row indicated by the system variable *iRow*. '#' is a modulo operator. If *iRow* is larger than the number of rows of *inpt*, '#' starts counting the rows anew. During the first step, $t = 1$, and therefore $iRow = 1$. Thus, *inpt#* extracts the first row vector from the *inpt* matrix. During the second step, $iRow = 2$, and therefore, the second row vector is used. In this way, the network gets to use all four input/target pairs. During the fifth step, $iRow = 5$, and since *inpt* has only four rows, the first row is extracted again. This language construct is much less elegant than CTRL–C's (and MATLAB's) *wild card* feature. It is the price that we pay for DESIRE's ultrafast compilation and execution. Remember that DESIRE was designed for optimal efficiency, and not for optimal flexibility.

The SAMPLE block is similar to a DISCRETE block in ACSL which contains an *INTERVAL* statement. It is normally used to

model difference equations. The argument of the *SAMPLE* statement denotes the frequency of execution of the SAMPLE block. In our example, the SAMPLE block will be executed once every four communication intervals. This construct is somewhat awkward, because it isn't necessarily meaningful to link the sampling rate with the communication interval. It may be desirable to sample a signal much more frequently than we wish to store results for output. The argument of the SAMPLE block should therefore refer to an arbitrary time interval rather than a multiple of the communication interval. Moreover, we may wish to simulate several discrete blocks sampled at different frequencies. Unfortunately, DESIRE allows only one SAMPLE block to be specified in every program.

Finally, we notice that several different DYNAMIC blocks can be coded in a single DESIRE program. Labels can precede sections of DYNAMIC code. These labels can be referenced in the *drun* statement.

Although I am quite critical of some of the details of the DESIRE language specification, DESIRE/NEUNET is clearly the best tool currently available for neural network simulations. On a 386–class machine, the exclusive–or backpropagation program compiles and executes in considerably less than 1 *sec*.

ACSL also offers matrix manipulation capabilities, but they are not useful for neural network simulation. As DESIRE, also ACSL does not provide easy access to individual matrix or vector elements. Yet, the reason is different. In ACSL, matrices were only an afterthought. They were implemented as *generic macros* (ACSL's "macro macro"). Consequently, matrix operations must be coded in *inverse polish notation*. For example, the statement:

$$\mathbf{x} = \mathbf{W} \cdot \mathbf{u} + \mathbf{b} \tag{14.37}$$

would have to be coded in ACSL as:

$$\mathrm{MADD}(x = \mathrm{MMUL}(W, u)\ ,\ b) \tag{14.38}$$

and their compilation is fairly slow. ACSL does not offer any non–linear vector functions as they are needed to describe the non–linear activation function of artificial neurons.

14.7 Neural Networks for Dynamical Systems

Artificial neural feedforward networks are basically non–linear multivariable function generators. They statically map a set of inputs into a set of outputs.

For the state–space representation:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \tag{14.39}$$

at any given time t, eq(14.39) maps the state vector $\mathbf{x}$ and the input vector $\mathbf{u}$ statically into the state derivative vector $\dot{\mathbf{x}}$. Thus, an artificial neural feedforward network should be able to "identify" the state–space model, i.e., to learn the system behavior. Unlike classical identification techniques, we need not provide the neural network with the explicit structure of the state–space model. In this respect, the neural network operates like the inductive reasoners discussed in Chapter 13.

Fig.14.15 shows a typical configuration of an *adaptive (self–tuning) controller* of a plant. The fast inner loop controls the inputs of a plant using a controller. The optimal controller parameters $\mathbf{p_c}$ depend on the current model parameters $\mathbf{p_m}$. In the slow outer loop, the model parameters $\mathbf{p_m}$ are identified from measurements of the plant input u and the plant output y. The optimal controller parameters $\mathbf{p_c}$ are then computed as a non–linear function of the model parameters $\mathbf{p_m}$.

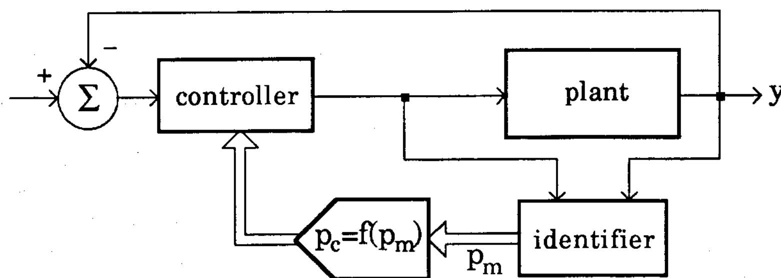

Figure 14.15. Plant with self–tuning regulator

Classical adaptive controllers work only if the model can be parameterized, i.e., if the uncertainties of the model can be represented through a model parameter vector $\mathbf{p_m}$. Moreover, most of the known

parameter identification techniques require the plant itself to be linear. This is true for both self–tuning regulators and model–reference adaptive controllers.

It is feasible to replace both the identification stage and the map from the model parameters to the controller parameters by neural networks. An excellent current review of research in this area was recently published by Narendra and Parthasarathy [14.25]. They show how the linear plant requirement can be eliminated. In this chapter, we shall not pursue the identification of adaptive controllers any further. Instead, we shall restrict our discussion to a very simple example of the identification of a dynamical system. Given the system:

$$\begin{aligned}\dot{\mathbf{x}} &= \mathbf{A}\cdot\mathbf{x} + \mathbf{b}\cdot u \\ &= \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -2 & -3 & -4 \end{pmatrix}\cdot\mathbf{x} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}\cdot u \end{aligned} \qquad (14.40a)$$

$$\begin{aligned} y &= \mathbf{C}\cdot\mathbf{x} + \mathbf{d}\cdot u \\ &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\cdot\mathbf{x} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}\cdot u \end{aligned} \qquad (14.40b)$$

We wish to train an artificial neural network to behave like the state–space model of this system. We shall use a backpropagation network similar to the one used before to replace the state equations, eq(14.40a). For this problem, we simulated the linear system in CTRL–C, and stored the resulting *inpt* and *target* matrices for off–line (supervised) learning. The CTRL–C code for the simulation is given below.

```
// Define the system
//
a = [  0   1   0
       0   0   1
      -2  -3  -4 ];
b = [ 0 ; 0 ; 1 ];
c = EYE(a);
d = ZROW(b);
```

```
// Simulate the system in CTRL - C
//
t = 0:3:900;
u = ROUND(RAND(t));
x0 = ZROW(3,1);
SIMU('ic', x0)
[y, x] = SIMU(a, b, c, d, u, t);
//
// Postprocess the results
//
xdot = ZROW(x);
FOR i = 1:301, ...
   xdot(:, i) = a * x(:, i) + b * u(i); ...
END
//
inpt = [ u' , x' ];
target = xdot';
//
// Save the results
//
SAVE inpt target   > linear.dat
//
RETURN
```

The system to be learned contains four inputs (u and $\mathbf{x}$) and three targets ($\dot{\mathbf{x}}$). 301 different input/target pairs are available for training. The CTRL–C version of the backpropagation network is given below.

```
// This procedure designs a backpropagation network for
// learning the behavior of a linear system
// Select the length of the hidden layer (LHID) first
//
DEFF fatan -c
//
// Load the input and target vectors
//
LOAD inpt target   < linear.dat
[npair, ninpt] = SIZE(inpt);
[npair, ntarg] = SIZE(target);
```

```
// Initialize the weighting matrices and biases
//
W1 = 0.01 * (2.0*RAND(lhid, ninpt)- ONES(lhid, ninpt));
W2 = 0.01 * (2.0*RAND(ntarg, lhid)- ONES(ntarg, lhid));
b1 = ZROW(lhid, 1);    b2 = ZROW(ntarg, 1);
WW1 = ZROW(lhid, ninpt);    WW2 = ZROW(ntarg, lhid);
bb1 = ZROW(lhid, 1);    bb2 = ZROW(ntarg, 1);
//
// Set the gains
//
g1 = 0.1;    g2 = 0.005;
//
// Set the termination condition
//
crit = 3.0;    error = 10.0;    count = 0;
//
// Learn the weights and biases
//
WHILE error > crit, ...
  count = count + 1; ...
  ... //
  ... // Loop over all input/target pairs
  ... //
  error = 0; ...
  FOR nbr = 1:npair, ...
    u1 = inpt(nbr, :)'; ...
    y2h = target(nbr, :)'; ...
    ... //
    ... // Forward pass
    ... //
    x1 = WW1 * u1 + bb1; ...
    y1 = (2.0/pi)*ATAN(x1); ...
    u2 = y1; ...
    x2 = WW2 * u2 + bb2; ...
    y2 = (2.0/pi)*ATAN(x2); ...
    ... //
    ... // Backward pass
    ... //
    e = y2h - y2; ...
    delta2 = (2.0/pi)*FATAN(y2) .* e; ...
    W2 = W2 + g2 * delta2 * (u2'); ...
    b2 = b2 + g2 * delta2; ...
    delta1 = (2.0/pi)*FATAN(y1) .* ((WW2') * delta2); ...
    W1 = W1 + g1 * delta1 * (u1'); ...
    b1 = b1 + g1 * delta1; ...
    error = error+ NORM(e); ...
  END, ...
```

```
    ... // Update the momentum matrices and vectors
    ... //
    WW1 = W1;   WW2 = W2; ...
    bb1 = b1;   bb2 = b2; ...
    err(count,1) = error; ...
END
//
// Save the learned network weights and biases for later
//
SAVE WW1 WW2 bb1 bb2 err   > linear_2.dat
//
RETURN
```

Since the system is continuous, we replaced the *LIMIT/TRI* function pair by an *ATAN/FATAN* function pair, where *FATAN* is the partial derivative of *ATAN*. To obtain convergence, the gains had to be considerably smaller than those used for the exclusive–or problem. The momentum terms had to be eliminated for this network. We looped over all available input/target pairs before we updated the weighting matrices and bias vectors. Thus, each iteration of the weight learning algorithm contained $npair = 301$ forward and backward passes through the network.

All desired target variables are approximately 0.1 units in amplitude. Thus, if our backpropagation network computes a real target of similar amplitude, but with arbitrary direction, the error is also approximately 0.1. Since we accumulate the errors of all input/target pairs, we expect an initial total error of about 30. This error should be reduced by one order of magnitude, before we can claim that our network has "learned" the system. Thus, we set $crit = 3.0$.

The learning required 774 iterations, and consumed more than 7 *hours* of CPU time on a VAX–11/8700. Obviously, this is not practical. If you wish to know how much time DESIRE/NEUNET requires to train the same network, solve hw(H14.2). It turns out that DESIRE/NEUNET solves this problem within a few minutes on a 386–class machine. Moreover, notice that I gave you good gain values to start with. In most cases, we shall need to rerun the same optimization many times in order to determine decent gain values. CTRL–C (or MATLAB) are obviously not useful for practical neural network computations. However, these languages are excellent tools for *documenting* neural network algorithms.

Fig.14.16 displays the total error as a function of the iteration count.

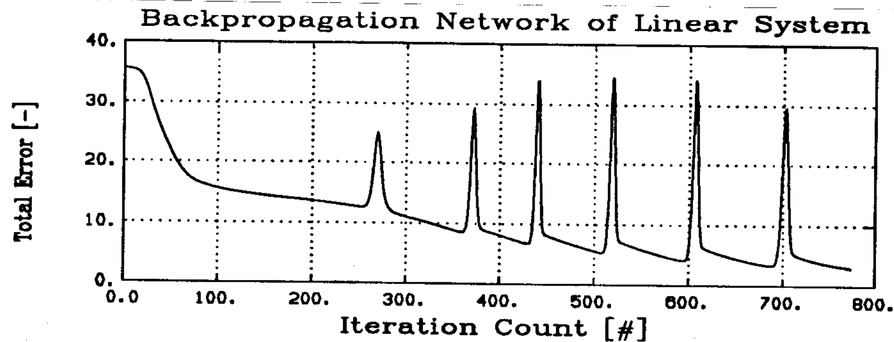

Figure 14.16. Learning a backpropagation network for a linear system

The results are rather interesting. The weight learning problem can be interpreted as a discrete–time dynamical system, i.e., the updating of the weights is like solving a highly non–linear high–dimensional set of difference equations. This "dynamical system" exhibits an oscillatory behavior. About once every 95 iterations, the network goes through a short phase in which the error temporarily grows. However, the network settles on a lower error level after each successive temporary instability. Notice that this "dynamical system" is *chaotic*. Each peak is slightly different from every other. The system behaves similarly to the Gilpin model for a competition factor of 1.0.

Let us analyze the behavior of our linear system at a particular point in time, $t = 813$ *sec*. The input and the state vector at that time have values of:

$$u = 0.0; \quad \mathbf{x} = \begin{pmatrix} -0.0464 \\ -0.0646 \\ 0.0756 \end{pmatrix}$$

and the true state derivative vector is:

$$\dot{\mathbf{x}}_{\mathbf{true}} = \begin{pmatrix} -0.0646 \\ 0.0756 \\ -0.0157 \end{pmatrix}$$

The approximated state derivative vector computed by the backpropagation network is:

$$\dot{\mathbf{x}}_{\mathbf{A.N.N.}} = \begin{pmatrix} -0.0606 \\ 0.0728 \\ -0.0170 \end{pmatrix}$$

Thus, the backpropagation network has indeed successfully learned the multivariable function. The approximated state derivative values are only slightly smaller than the true values.

Notice that the true system itself may be interpreted as an artificial neural network (A.N.N.):

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} + \mathbf{b} \cdot u = (\mathbf{A} \quad \mathbf{b}) \cdot \begin{pmatrix} \mathbf{x} \\ u \end{pmatrix} = \mathbf{W} \cdot \mathbf{u_1} \tag{14.41}$$

Thus, we could have identified a single layer network with four inputs and three outputs using a linear activation function. We could have "trained" this network much more easily, and the results would have been even better ... but this would have been no fun. The purpose of the exercise was to show that we can train an arbitrary artificial neural network to blindly learn the behavior of an arbitrary multivariable function.

We chose to learn the network off–line in a supervised learning mode. However, we could have easily learned the network on–line in an adaptive learning mode. We would simply have used input/target pairs as they arrive to train our $\mathbf{W}^i$ matrices and $\mathbf{b}^i$ vectors, and occasionally update the $\mathbf{WW}^i$ matrices and $\mathbf{bb}^i$ vectors.

Encouraged by these nice results, let us now close the loop. Fig.14.17a shows the true system, which we shall now replace by the approximated system of Fig.14.17b.

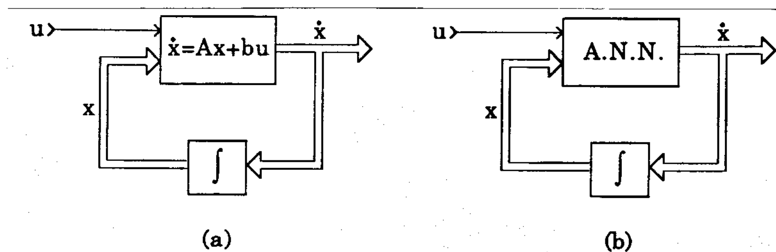

Figure 14.17. Approximation of a linear system by an A.N.N.

We close the loop around the backpropagation network using vector integration. As in Chapter 13, we shall assume that forecasting begins at time $t = 813$ *sec*. We shall predict the future state trajectories of the system using the A.N.N. Fig.14.18 shows the results.

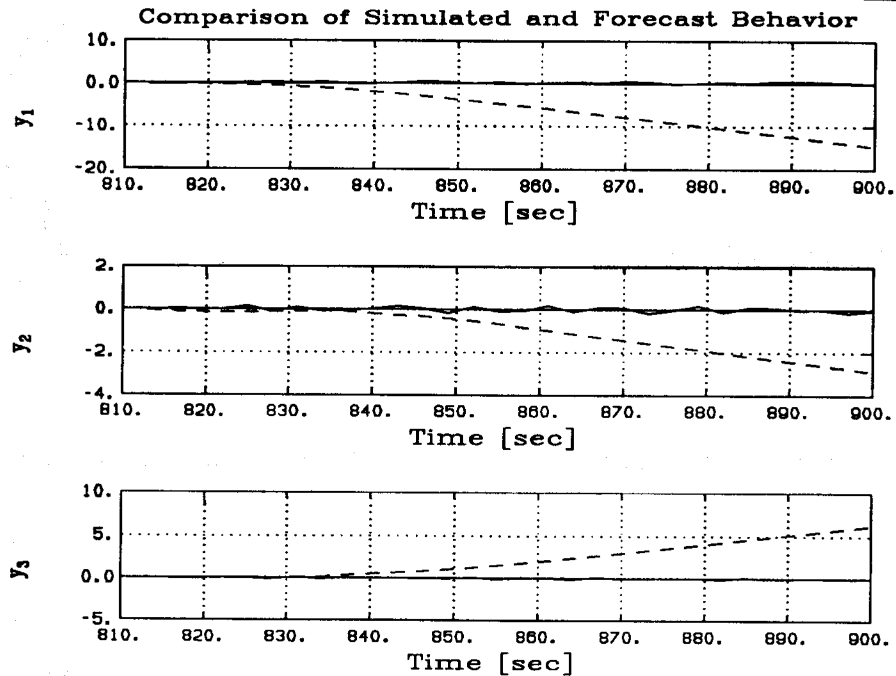

Figure 14.18. Comparison of true and approximated trajectories

The solid lines represent the true trajectory behavior, while the dashed lines represent the approximated trajectory behavior. This was obviously not such a brilliant idea after all. The A.N.N. approximation and the **true** trajectory behavior vary greatly. The A.N.N. systematically *underestimates* the first and second state derivatives, whereas it systematically *overestimates* the third state derivative. The errors in each individual step are fairly small, but these errors accumulate, and the approximated solution quickly drifts away from the true solution. Unlike the results obtained in Chapter 13, errors accumulate if we close an integration loop around a neural feedforward network trained to approximate the behavior of an open–loop system.

Let us now try another approach. Remember that, in Chapter 13, we chose a mask depth of three, i.e.:

$$\mathbf{x_{k+1}} = \mathbf{f}(u_{k-1}, \mathbf{x_{k-1}}, u_k, \mathbf{x_k}, u_{k+1}) \tag{14.42}$$

Thus, we could try to train an A.N.N. to approximate the closed–loop behavior of our dynamical system using eq(14.42). This time, we have nine inputs and three outputs. Since I have become impatient, and don't want to spend several more hours training yet another backpropagation network, I shall use a counterpropagation network instead. Since counterpropagation networks work much better for digital systems, let us try the idea shown in Fig.14.19.

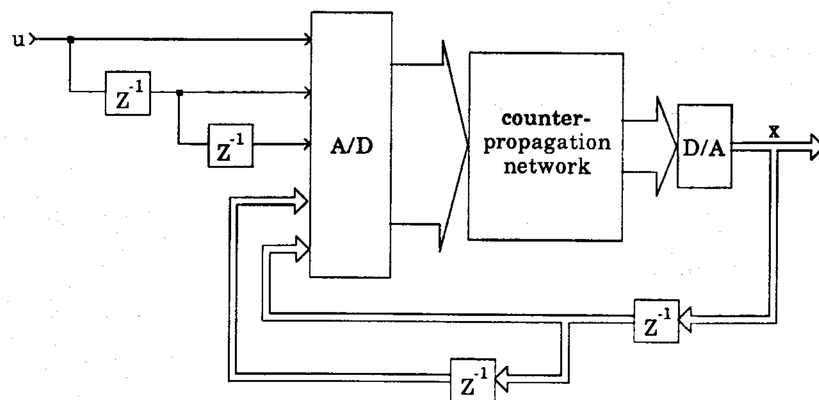

Figure 14.19. Counterpropagation network for linear system

We generate the weighting matrices of the feedforward counterpropagation network by converting the nine analog inputs to 90 digital inputs, and the three analog targets to 30 digital targets using 10 *bit* analog to digital (A/D) converters. We set up the counterpropagation network using the first 269 input/target pairs. Thus, the counterpropagation network has 90 inputs, a hidden layer of length 269, and an output layer of length 30. The *off* or *false* state of the digital inputs and targets is represented by −1.0, whereas it is represented by '0' in the hidden layer. The *on* or *true* state of all digital signals is represented by +1.0.

During recall, we convert the analog input vectors of length nine to digital input vectors of length 90 using the same 10 *bit* A/D converters. For each digital input vector, the counterpropagation network predicts a digital target vector of length 30. We then convert the resulting 30 digital targets back to three analog targets using 10 *bit* digital to analog (D/A) converters.

In our example, we use this configuration to predict the trajectory behavior of the linear dynamical system over the last 87.0 time units or 30 input/target pairs. Fig.14.20 shows the results of this effort.

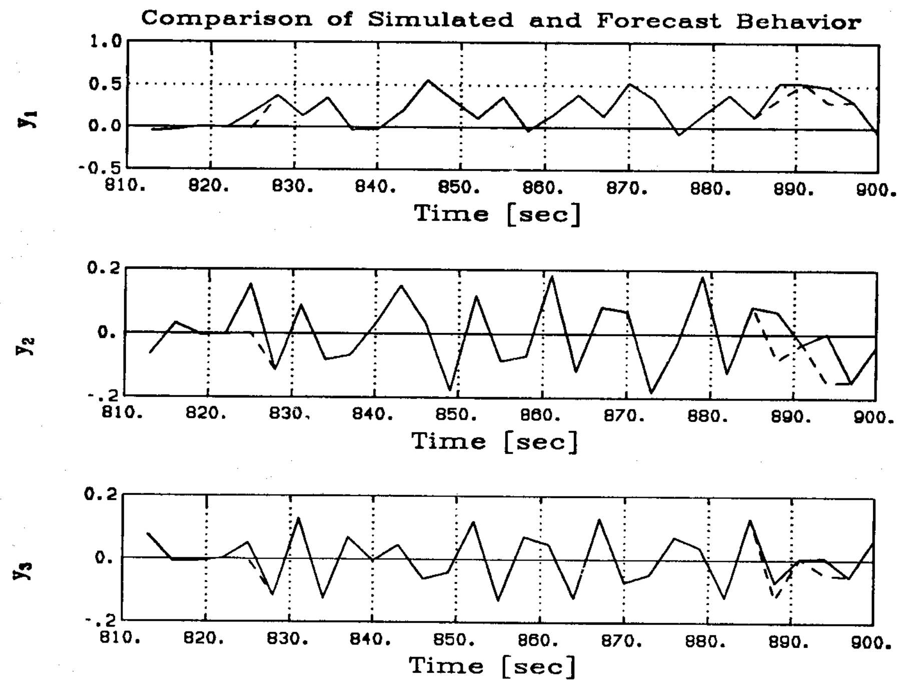

Figure 14.20. Comparison of true and approximated trajectories

This time, we were right on target. Within the forecasting sequence, only a few predictions were incorrect. There are two possible sources of errors:

(1) Since the input/target map is no longer strictly deterministic, the same input may produce several different targets. In that case, we may subsequently recall the wrong target.

(2) A particular input may never have been observed during the setup period. In this case, the prediction will be arbitrary, and probably incorrect.

As in the solution presented in Chapter 13, errors don't accumulate. An error which occurs once is not propagated through the network. Compared to the SAPS method, the counterpropagation network allows us to operate with much better resolution. In SAPS, we had to discretize our continuous signals into no more than five

different levels. In the counterpropagation network, we discretized our continuous signals into 1024 different levels. The price that we pay for this luxury is an increased memory requirement. The SAPS masks store their knowledge about the system in a much more compact fashion. The weighting matrices of the counterpropagation network have dimensions 269×90, and 30×269, and thus, they are fairly large. If you wish to learn how Fig.14.20 was produced, solve hw(H14.4).

Notice that also this algorithm can be implemented in an adaptive learning mode. To do this, we begin with a hidden layer of length one, and store the first input/target pair as the two weighting matrices $\mathbf{W}^1$ and $\mathbf{W}^2$. We then process the second input. The second input will probably differ from the first. In this case, the counterpropagation network will forecast the wrong target. If this happens, we concatenate the new input as an additional row to the $\mathbf{W}^1$ matrix from below, and we concatenate the true target as an additional column to the $\mathbf{W}^2$ matrix from the right, thereby incrementing the length of the hidden layer. If the target is predicted correctly, we leave the A.N.N. as is. After many iterations, most of the inputs will have been seen earlier, and eventually, the network will stabilize. If you wish to study this algorithm in more detail, solve hw(H14.5).

Yet another technique to emulate dynamical systems is direct identification of a *recurrent network*, i.e., identification of an A.N.N. with built in feedback loops. This approach was first proposed by Hopfield [14.14]. The most popular and widely studied recurrent network is therefore referred to as a *Hopfield net.* Hopfield nets have only one layer, and require a much smaller number of neurons than the previously proposed counterpropagation networks. For our (trivial) example, three artificial neurons would suffice. The Hopfield net would simply be the sampled–data version of our continuous system. Other commonly used variations of recurrent networks are the multi–layered *bidirectional associative memories*, and Grossberg's *adaptive resonance circuits (ART's)*. Recurrent networks can be problematic with respect to their stability behavior. We won't pursue this avenue any further in this text. However, notice that the explicitly dimensioned "counterpropagation network" of Fig.14.19 is in fact also a recurrent network. It does not exhibit stability problems.

14.8 Global Feedback through Inverse Networks

As I mentioned earlier, today's A.N.N.'s bear little resemblance to actual neural networks found in mammals. Let us now return to the question of how our brain works, how it processes information, and how it "learns". Obviously, my remarks must become a little more philosophical at this point, and I won't be able to support my hypotheses with short CTRL–C or DESIRE/NEUNET programs.

We wish to discuss how our brain is believed to processes various types of sensory input signals. Let us pursue the processing of information related to the animal *DOG*. Its symbol in my brain will be denoted as $\mathcal{DOG}$. The sound "dog" will be written as $\approx dog \approx$, and a picture of a dog will be coded as $\overset{\circ\!\prec}{\wedge\wedge}$.

My brain has a *visual neural network* which maps $\overset{\circ\!\prec}{\wedge\wedge}$ into $\mathcal{DOG}$, and an *auditory neural network* which maps $\approx dog \approx$ into $\mathcal{DOG}$. Thus, the brain can map different sensory input signals into the same symbol.

If somebody says $\approx dog \approx$ to me, an image of my dog, *my*–$\overset{\circ\!\prec}{\wedge\wedge}$, appears before my *inner eye*. How is this possible, and what is an "inner eye"? It seems that the brain is not only capable of mapping pictures into symbols, but it can also map symbols back into pictures. Somehow, these pictures can superpose real images captured by the eye. We call this internal seeing the "inner eye". I shall denominate the neural network which maps symbols back into pictures the *inverse visual neural network*.

Similarly, my brain also has an *inverse auditory neural network*. In fact, my brain has an inverse neural network for each of my senses. If I watch old slides from one of my trips to the Middle East, I suddenly "smell" the characteristic spices in the bazaars (such as *sumak*), and I "hear" the muezzin call the believers to prayer. These sensations are produced by the inverse neural networks, i.e., by global feedback loops which connect the symbols right back to to the senses from which they originated. Usually, these feedback loops are fairly rigorously suppressed. We have been taught to ignore these feedback signals most of the time. They are constantly present, however, and play an important role in our lives.

Fig.14.21 depicts these feedback loops for the visual and auditory networks.

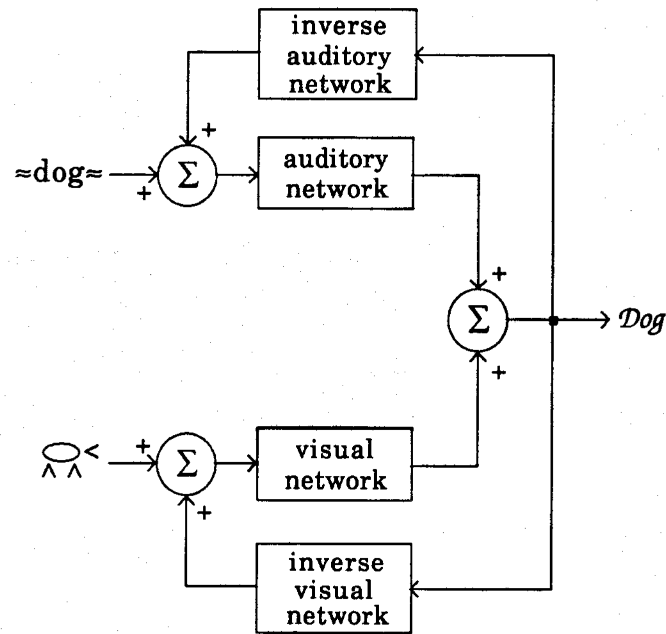

Figure 14.21. Global feedback through inverse neural networks

I am convinced that this global feedback mechanism is essential to our brain's ability to learn. Without such feedback, the brain could mindlessly react to stimuli, but it could never achieve higher levels of cognition. We would not be able to truly "think". Let me elaborate on this idea a little further in the next section.

14.9 Chaos and Dreams

It turns out that, while the neural networks themselves are massive parallel processors, at the symbolic or conscious level, we can "think" only one thought at a time. This means that, at the output summer of Fig.14.21, where the symbols are formed and ultimately rise to the level of consciousness, we have a "winner–takes–it–all" situation. We are faced with *competition*.

Remember our lessons from Chapter 10:

$$\begin{aligned} &\textit{high-dimensional system} \\ &\quad + \textit{feedback loops} \\ &\quad\quad + \textit{competition} \\ &\quad\quad\quad \Longrightarrow \textit{chaos} \end{aligned}$$

If we take a high–dimensional system like our brain, introduce some feedback loops (hopefully without making the system unstable), and introduce competition, the result is almost invariably *chaos.* I therefore argue that, due to the existence of global feedback loops, our brain operates permanently under conditions of a *chaotic steady–state.*

Normally, the external inputs are much more powerful than the feedback loops. Thus, they will usually win the competitive battle at the output summer. However, during the night, when the external inputs are reduced to a minimum, the feedback loops take over. The behavior of our brain is then dictated by its own chaotic steady–state behavior. We call this mental state: *dreaming.* In our dreams, colorful sceneries appear before our "inner eyes". Since the visual input is usually dominant, this is what we most likely will remember in the morning. However, with our "inner ears", we also hear people speak to us, and I am convinced that we also smell with our "inner nose", taste with our "inner taste buds", and touch with our "inner skin".

Recall another lesson from Chapter 10. For self–organization to occur in a system, we need an *innovator* and an *organizer.* In our brain, the *innovator* is the chaotic feedback. Without this feedback, we would never create new and original ideas. Instead, we would react to our environment like mindless machines [14.22]. The *organizer* is the reward mechanism which is responsible for the survival of the "good ideas". Thus, without chaos, no learning and no self–organization could occur. No wonder that today's robots are "mindless". We haven't figured out yet how to introduce chaos into their "brains". This certainly is a worthwhile research topic.

14.10 Internalization Processes and Control Mechanisms

In our brains, the relative weights of the external *vs* internal inputs are controlled by various mechanisms. The control is exerted by

two higher level functions which have been coined our *ego* and our *superego*. The intensity of the feedback loops is controlled by mechanisms usually referred to as our *will* (attributed to the *ego*), and by *societal taboos* (attributed to the *superego*). The latter mechanism prevents us from pursuing particular thoughts beyond a "danger level" which is a threshold coded into our *superego*. When a "dangerous" idea pops up, the *superego* will automatically reduce the weights of the feedback loops, and we divert our attention to another topic [14.22]. Moreover, we all want to be "liked". Since strangely behaving people are "not liked", our *ego* reduces the impact of the internal feedback loops to an extent where we react to external stimuli reliably and coherently.

In order to understand these mechanisms better, we need to expand the block diagram of Fig.14.22 to that in Fig.14.22.

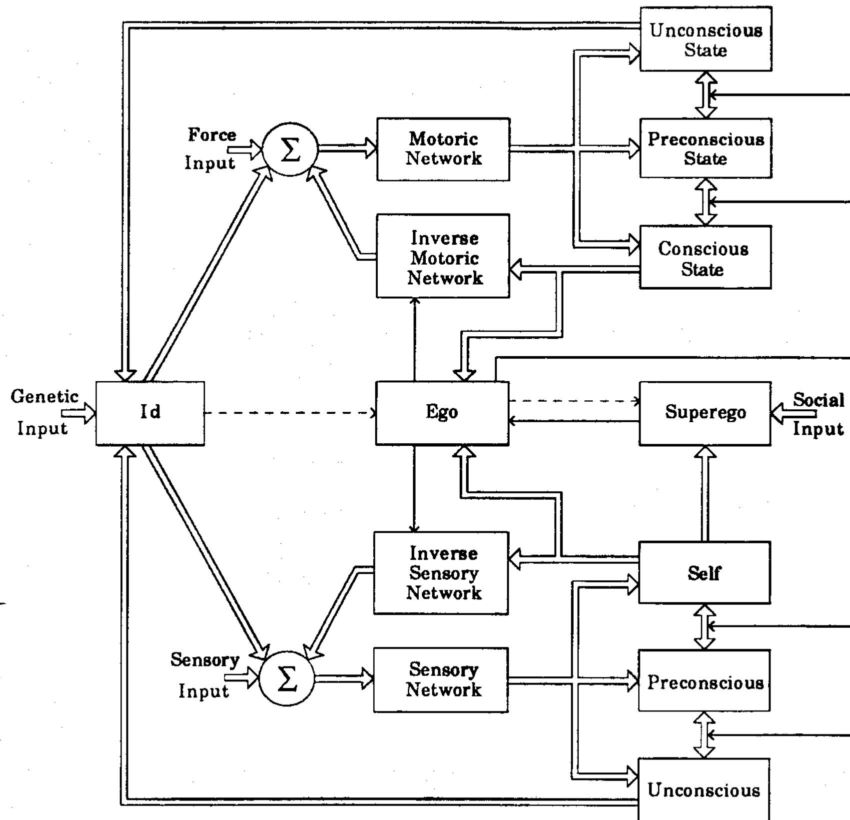

Figure 14.22. Block diagram of human mental functions

From a psychodynamical point of view, we can distinguish between three major *subjective functions* (we could call them programs), the *id*, the *ego*, and the *superego*. They operate on three major *objective functions* (we could call them data bases), the *unconscious*, the *preconscious*, and the *conscious self*. The *id* is the earliest of our "programs". It is purely hereditary, i.e., we are born with it. Initially, the *id* performs all of the control functions. During the first three years of our lives, the *ego* develops and subsequently assumes most of the control functions. However, since the *ego* is flexible and comparatively easy to re–program (since it is predominantly conscious), we need yet another mechanism which ensures the overall stability of the "system". This is called the *superego*. It sets the limits for the *ego*. The *superego* develops last among the three "programs", around age four or five. The *superego* is much more difficult to re–program than the *ego* since it is mostly unconscious. The *superego* holds our unconscious beliefs and commands. It knows what is inherently "right" and "wrong". It is the basis of our conscience, of our superstitions, and all sorts of societal and familial taboos.

From the perspective of a control engineer, our brain contains two major *model–reference adaptive control (MRAC) loops*. On the sensory side, the innermost automatic feedback control loop contains the sensory network (forward path) which feeds the conscious self, the preconscious, and the unconscious. It also contains the inverse sensory network (feedback path). The innermost control loop is supervised by the *ego*, our major adaptive controller. It receives information mostly from the conscious self, but also from the preconscious and unconscious. The *ego* decides whether the system performs adequately. If the system reacts too slowly, the feedback gains of the innermost control loop are increased. If it starts to become unstable, the feedback gains are decreased. The *id* can perform similar functions. It receives information mostly from the unconscious, but also from the preconscious and from the conscious self. However, it is mostly content to let the *ego* do its job. It has more direct means to intervene. It can provide strong input signals directly to our sensory system. If the *id* decides that I should eat, it provides my stomach sensors with the strong sensation of hunger. Notice that the *id* does not directly tamper with my data bases. It does not make me fall upon the idea that I might wish to eat. It stimulates my sensory input, and lets the sensory network process the sensation of hunger which will ultimately arrive at my *self*, will then be forwarded as useful information to my *ego*, which then instructs the second (motoric) control loop to do something about. The *superego*

is the external model of the MRAC. It provides the *ego* with the needed set values, and ensures the overall stability of the system. The *ego* can be re–programmed, but only to the extent that the *superego* permits. The motoric MRAC is a mirror image of the sensory MRAC. The innermost loop contains the motoric network (forward path) which feeds the three objective functions or data bases which maintain positional and velocity information. The inverse motoric network feeds back the perception of my current position, velocity, and force vectors. The *ego* is again the adaptive controller which puts *purpose* into my motions. The *id* can perform similar functions, but it usually doesn't. It has more direct means to influence my motoric behavior, through reflexes. The *superego* knows "my place in the world". It prevents the *ego* from requesting motoric actions which are considered undecent.

From the perspective of a software engineer, we usually distinguish between *data*, and *programs* that act on data. We may classify the *conscious self*, the *preconscious*, the *unconscious*, and their motoric counterparts as data, and the *id*, the *ego*, and the *superego* as programs that act upon these data. The four "network" boxes have not been named in the psychological literature. They are also "programs", but hardwired ones. They are basically data filters. However, as software engineers, we know that the distinction between programs and data is not very crisp. An ACSL "program" is a data filter which maps input trajectories (input "data") into output trajectories (output "data"). However, the ACSL preprocessor is yet another data filter which maps the ACSL program (input "data") into a FORTRAN program (output "data"). Thus, what constitutes "data" and what constitutes "programs" depends on our perspective. In LISP, we notice this fusion even more clearly. There is really no difference at all between a data item and an instruction operating on a data item. In an expert system, we usually distinguish between the *knowledge base* which contains the facts, and the *rule base* which contains instructions for how to process the facts. Yet, both are data bases which are syntactically identical. This similarity applies to our brain as well. We possess only one brain which stores both the *facts* (objective functions) and the *rules* (subjective functions). When we look at our brain under a microscope, we cannot distinguish between them, because the microscope shows us only the *syntax*, and not the *semantics* of our brain.

In Fig.14.22, data flow is shown as double lines. Control signals are shown as single lines. The basic "programs" of our brain are the four network boxes. The *ego* acts as an *incremental compiler* which

constantly modifies the four basic programs, but it tampers more with the feedback networks than with the feedforward networks. The *ego* constantly modifies parameters in our feedback networks. The dashed lines in Fig.14.22 represent control signals of a deeper type. Initially, there is no *ego*. The *id* represents the "desire" for an *ego* to develop. Thus, the *id* controls the creation of the *ego* "program". After the *ego* is fully developed, the *id* even delegates this function to the *ego*, i.e., the *ego* starts re–programming itself.

Even this capability is not unheard of in software engineering. ELLPACK [14.27] is a simulation language for solving elliptic partial differential equations. Like in ACSL, a *preprocessor* translates ELLPACK programs into FORTRAN. However, the ELLPACK language is not static. It is meant to be user modifiable. Therefore, the preprocessor is not hand–coded. Instead, the system comes with a *compiler generator* which can automatically generate a new version of the preprocessor from a *data template file* which contains an abstract description of the ELLPACK *syntax* (the grammar) and of the ELLPACK *semantics* (the code to be generated). Then, the software designers decided that this data template file was too difficult to create manually. Consequently, they designed a *template processor* which generates the required data template file from a yet more abstract description. Naturally, they didn't want to manually code the template processor either. Instead, they described the syntax and semantics of the template processor, and generated the template processor using the same compiler generator that they had used before to generate the ELLPACK preprocessor. Finally, they described the syntax and the semantics of the compiler generator itself in terms of the syntax and semantics of the compiler generator, and they now can feed this description into one version of the compiler generator, and use it to generate the next version of itself. They actually started out with a very simple compiler generator, and bootstrapped it by iteratively processing it through itself.

The *ego* and the *superego* together perform the function of a highly effective MRAC. If the feedback gains are set too low, we lose our creativity and inspiration. If they are set too high, the "system" becomes unstable, and we end up in a mental institution. Yet, the "set values" of the adaptive controller are highly individual.

People who effectively suppress their feedback loops are perceived as reliable and predictable, but not very imaginative, possibly as compulsive, and in extreme cases as *neurotic*. These people are the "bureaucrats" of our society. They desire predictable, routine lives.

They feel uncomfortable changing jobs. They usually prefer the apparent "safety" of an existing situation, even if it is unbearable, to the uncertainty of change. Neurotic people have a small *ego* and an overpowering *superego* which leaves the *ego* little latitute for improvement. The feedback loops are heavily suppressed. Much of the conscious self is intentionally repressed into the preconscious and unconscious. Threatening thoughts are swept under the carpet rather than confronted.

People who allow their feedback gains to be at a higher level, are considered inspirational and imaginative, but also somewhat impulsive and incalculable. These people are the artists and Bohemians of our society. They are considered somewhat egotistic (which simply means that they have a strong *ego*). They will change a bad situation rather than suffer. Since their *ego* is strong, they don't need an overpowering *superego* to keep the system stable. Artists allow themselves to be influenced by their unconscious. They allow information to flow from their unconscious into their conscious self. They consider their unconscious their best friend rather than a threatening enemy. In psychodynamic terms, this is called the ability to experience a "controlled regression".

Psychotic people, finally, can be described as people whose *ego* is disintegrating. The *ego* is not stable, the *ego* boundaries dissolve, and the feedback loops are out of control. Whatever exists of the *superego* contributes to the disintegration of the *ego*. The weakened *ego* perceives the flow from the unconscious as extremely threatening, but cannot stop it. The feedback loops take over. The psychotic person reacts incoherently. S/he cannot concentrate on any one topic, but changes the subject frequently during the course of a conversation. S/he suddenly "hears voices" (through the inverse auditory network). S/he is in a constant state of panic because s/he is overwhelmed by the flow of incomprehensible undigested information from the unconscious into the conscious self, and back to the sensory level through the inverse networks.

Some psychedelic drugs, such as LSD, have the tendency to increase both the external and the internal input gains. LSD users have described that they "see the colors much more intensively". Since the feedback gains are increased simultaneously, the "system" becomes less stable. Consequently, some drug users have been described as exhibiting "psychotic behavior". Since drug use disables the control mechanisms of the *ego*, this is a very dangerous proposition, even independently of the chemical side effects.

Until now, we have only discussed the "end product", the fully developed adult person. We have described the control mechanisms of the strong and healthy. We have pointed out how some mental disabilities, neurosis and psychosis, reflect upon functional deficiencies of some of the control mechanisms of our personality. We have not analyzed yet how and why these deficiencies have occurred in the first place.

From a phenomenological perspective, it may be observed that some of us grow up under adverse conditions. The adults around us don't provide us with sufficient affection (psychologists call this phenomenon *neglect*), or they suffocate us with too much of it (psychologists call this phenomenon *overprotection*). Input signals may be "scrambled" [14.19], i.e., words imply something, but mean something else (psychologists call this the *double bind*).

Children who grow up under such adverse conditions may eventually lose their "interest" in the external input, because it is either too painful (neglect or overprotection), or too unreliable (double bind). They turn into *day dreamers* ... because they consider their internal input less threatening than input from the environment [14.4]. Notice that day dreaming is not a negative phenomenon *per se*. It is the sole source of our creativity and inspiration. It is the wood from which our geniuses are carved. Yet, under more severe circumstances, such children may become *psychotic*. They "hear voices", they react incoherently to their environment, and external objects, such as their parents or partners, may no longer be cathected [14.5]. They are caught in a world in which they are the "only actors on a stage which encompasses the entire world". All other people can only be perceived as either threats or properties. Finally, if the trauma is experienced sufficiently early in life (within the first few months after birth), and if it is sufficiently strong, they simply "switch off" the external input altogether and become *autistic*.

However, a much more comprehensive picture has been painted by Otto Kernberg [14.15]. He identifies several pre-oedipal development phases. The first phase is called the *primary undifferentiated autistic phase* which lasts for a few months after birth. During this phase, the child slowly develops a symbiotic relationship with his or her mother. The child cannot yet distinguish between itself and the mother. If this symbiotic relationship is traumatized, the child remains autistic. When a strong and stable symbiotic relationship has been established, the child enters the second phase which is called the *phase of primary undifferentiated self-object images*. An *all-good mother-self image* enables the child to slowly learn to differentiate

between the *self–images* and the *object–images.* This is the time when every child plays the "peekaboo" game. The child covers its head with a blanket and is "gone" ... but of course, the mother is gone as well. Within a short time span, the tension becomes unbearable, and the blanket is removed. The child is "back", ... and of course, so is the mother. If the mother is not able to maintain an embracing, allowing, and loving relationship with her child during this critical developmental phase, the child experiences an extreme *aggression* against the loved object, which can only be overcome by *re–fusing* the self–images with the object–images. The child cannot properly define its ego boundaries, and this is the seed which will eventually lead to a psychosis, because a strong *ego* is needed for the development of a healthy *superego.* In the last pre–oedipal phase, the child learns to integrate libidinally determined and aggressively determined self–images and object–images. It learns to integrate love and hate, and it learns to acknowledge the co–existence of both in the self and in the others. The child learns that nobody is all–good or all–evil, and yet, it is integrated enough to accept this fact without being threatened.

Notice that Kernberg's description does not contradict the phenomenological description given earlier. It only provides us with a more profound analysis of the internal mechanisms of the higher level mental functions of the human personality.

Most neurophysiologists tend to attribute psychoses to chemical problems with neurotransmitters. Several hypotheses have been formulated relating various types of chemical substances to the occurrence of psychoses, none of which has yet been proven. Although this approach seems quite different from the psychodynamic explanations given above, these hypotheses do not contradict each other. Just as muscles in our body shrink when they are not in use, so are the synaptic strengths between neighboring neurons believed to weaken when the involved axon is not frequently fired. This is precisely the biological hypothesis behind our weight adjustment algorithms as expressed in today's artificial neural networks. Thus, the "disconnection of inputs" will ultimately be electrochemically implemented in our brain in the form of weakened synaptic strengths. This may be what neurophysiologists attempt to confirm with their measurements. The only remaining question is: What came first, the chicken or the egg?

14.11 Genetic Learning

Let us now return to the mechanisms of learning. I had mentioned earlier that gradient techniques are dangerous because of potential stability problems, beside from the fact that they are not biologically plausible.

In this section, I shall introduce another optimization technique which does not exhibit the stability problems characteristic of gradient techniques, and while also this approach is not biologically plausible in the context of neural learning, it has at least been inspired by biology. Genetic algorithms were first developed by John Holland in the late sixties [14.13]. As with the neural networks, the basic idea behind genetic algorithms encompasses an entire methodology. Thus, many different algorithms can be devised which are all variations of the same basic scheme.

The idea behind genetic algorithms is fairly simple. Let me describe the methodology by means of a particular dialect of the genetic algorithms applied to the previously introduced linear system backpropagation network. In that problem, we started out with initializing the weighting matrices and bias vectors to small random numbers. The randomization was necessary in order to avoid stagnation effects during startup. Yet, we have no reason to believe that the initial choice is close to optimal, or even, that the weights remain small during optimization. Thus, the initial weights (parameters) may differ greatly from the optimal weights, causing the optimization to require many iterations. Also, since backpropagation learning is basically a gradient technique, the solution may converge on a local rather than a global minimum, although this didn't happen in this particular example.

Genetic algorithms provide us with a means to determine optimal parameter values more reliably even in a "rough terrain", i.e., when applied to systems with a cost function that has many "hills" and "valleys" in the parameter space.

Let us assume that we already know approximate ranges for the optimal weights. In our case, the optimal weights belonging to the $\mathbf{W}^1$ matrix assume values between -2.0 and $+2.0$, those belonging to the $\mathbf{W}^2$ matrix assume values in the range -0.5 to 0.5, those from the $\mathbf{b}^1$ vector are between -0.05 and 0.05, and those from $\mathbf{b}^2$ are bounded by -0.005 and 0.005. I am cheating a little bit. Since I solved the backpropagation problem already, I know the expected

outcome. The more we can restrict the parameter ranges, the faster the genetic algorithm will converge.

We can *categorize* the parameter values by classifying them as *very small*, *small*, *large*, and *very large*, respectively. In terms of the terminology used in Chapter 13, we transform the formerly *quantitative* parameter vector into a *qualitative* parameter vector. A semi–quantitative meaning can be associated with the qualitative parameters using fuzzy membership functions as shown in Fig.14.23 for the parameters stored in $\mathbf{W}^1$.

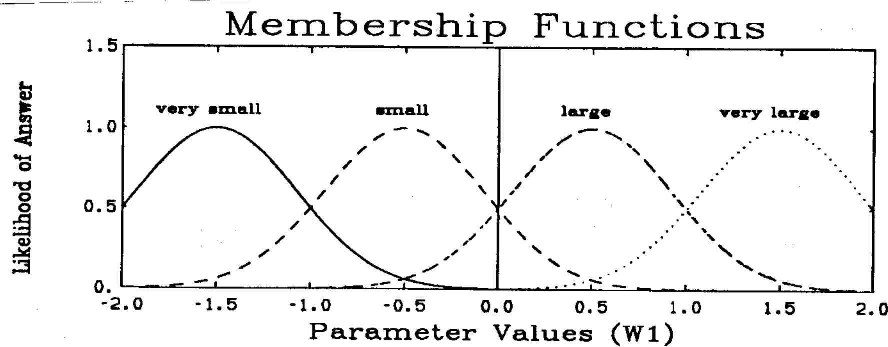

Figure 14.23. Fuzzy membership functions for parameter values

The number of levels can, of course, be chosen freely. In our example, we decided to use four levels, $nlev = 4$. Let us now denote each class by a single upper–case character:

$$A \Leftrightarrow \textit{very small}$$
$$B \Leftrightarrow \textit{small}$$
$$C \Leftrightarrow \textit{large}$$
$$D \Leftrightarrow \textit{very large}$$

Thus, each qualitative parameter can be represented through a single character. We may now write all qualitative parameter values into a long character string such as:

$$ABACCBDADBCBBADCA$$

where the position in the string denotes the particular parameter, and the character denotes its class. The length of the string is identical with the number of parameters of the problem. This is our qualitative parameter vector.

Somehow, this string bears a mild resemblance with our genetic code. The individual parameters mimic the amino acids as they alternate within the DNA helix. Of course, this is an extremely simplified version of a "genetic code".

In our example, let us choose a hidden layer of length $lhid = 8$. Consequently, the size of $\mathbf{W}^1$ is 8×4, since our system has four inputs, and the size of $\mathbf{b}^1$ is 8. The size of $\mathbf{W}^2$ is 3×8, since the system has three targets, and the size of $\mathbf{b}^2$ is 3. Therefore, the total number of parameters of our problem $npar$ is 67. Thus, the parameter string must be of length 67 as well.

The algorithm starts out with a "genetic pool". We arbitrarily generate $nGenSt = 100$ different "genetic strings", and write them into a matrix of size 100×67. In CTRL-C (or MATLAB), it may be more convenient to represent the "genes" by integer numbers than by characters. The genetic pool can be created as follows:

$$GenPool = ROUND(nlev * RAND(nGenSt, npar) + 0.5 * ONES(nGenSt, npar))$$

Initially, we pick 10 arbitrary genetic strings (row vectors) from our genetic pool. We assign quantitative parameter values to them using their respective fuzzy membership functions by drawing random numbers using the fuzzy membership functions as our distribution functions. Next, we generate weighting matrices and bias vectors from them, by storing the quantitative parameters back into the weighting matrices in their appropriate positions. Finally, we evaluate our feedforward network 301 times using the available input/target pairs for each of these 10 parameter sets. The result will be 10 different *figures of merit* which are the total errors, the sums of the individual errors for each training pair, found for the given weighting matrices. We sort the 10 performance indices and store them in an array. This gives us a vague first estimate of network performance.

We then arbitrarily pick two genetic strings (the parents) from our pool, draw an integer random number k from a uniform distribution between 1 and 67, and simulate a *crossover*. We pick the first k characters of one parent string (the head), and combine them with the remainder (the tail) of the other parent string. In this way, we obtain a new qualitative genetic string called the child. We then generate quantitative parameter values for the child using the fuzzy membership functions, and simulate again. If the resulting performance of the child is worse than the fifth of the 10 currently stored

performance indices, we simply throw the child away. If it is better than the fifth string in the performance array, but worse than the fourth, we arbitrarily replace one genetic string in the pool by the child, and place the newly found performance index in the performance array. The worst performance index is dropped from the performance array. If it is better than the fourth, but worse than the third stored performance, we duplicate the child once, and replace *two* genetic strings in the pool by the two copies of the child. Now, two of the 100 strings in the pool are (qualitatively) identical twins. If it is better than the third but worse than the second performance, we replace *four* genetic strings in the pool by the child. If it is better than the second but worse than the first, we replace *six* genetic strings in the pool by the child. Finally, if the child is the all time champion, we replace 10 arbitrary genetic strings in the genetic pool by copies of our genius.

We repeat this algorithm many times, deleting poor genetic material while duplicating good material. As time passes, the quality of our genetic pool hopefully improves.

It could happen that the very best combination cannot be generated in this way. For example, the very best genetic string may require an A in position 15. If (by chance) none of the randomly generated 100 genetic strings had an A in that position, or if those genes that had an A initially got purged before they could prove themselves, we will never produce a child with an A in position 15. For this reason, we add yet another rule to the genetic game. Once every $nmuta = 50$ iterations, we arbitrarily replace one of the characters in the combined string by a randomly chosen new value, simulating a *mutation*. Eventually, this mutation will generate an A in position 15.

Obviously, this algorithm can be applied in an adaptive learning mode. Our "genetic pool" will hopefully become better and better, and with it, our forecasting power will increase.

Of course, this algorithm can be improved. For instance, if we notice that a particular parameter stabilizes into one class, we can re–categorize the parameter by taking the given class for granted and by selecting new subclasses within the given class. In our example, we might notice that the parameter 27 which belongs to $\mathbf{W}^1$ always assumes a qualitative value of C, i.e., its quantitative value is in the range between 0.0 and 1.0. In this case, we can subdivide this range. We now call values between 0.0 and 0.25 *very small*, and assign a character of A to them. Values between 0.25 and 0.5 are now called

small, and obtain a character value of B, etc. I decided to check for re–categorization once every 50 iterations, whenever I simulated a mutation. I decided that a re–categorization was justified whenever 90% of the genes in one column of *GenPool* had assumed the same value, $nperc = 0.9$.

I ran my genetic algorithm over 800 iterations which required roughly 2 *hours* of CPU time on our VAX–11/8700. The execution time was less than that of the backpropagation program since each iteration contains only the forward pass and no backward pass, and since the length of the hidden layer was reduced from 16 to eight. Thus, for a fair comparison between the two techniques, I should have allowed the genetic algorithm to iterate 2800 times. The results of this optimization are shown in Fig.14.24a.

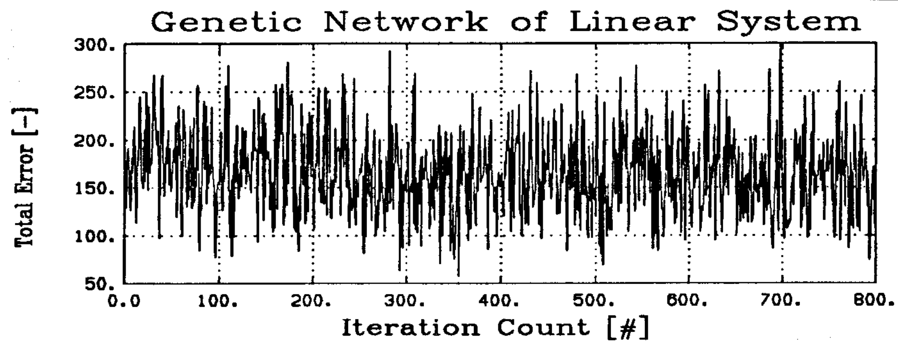

Figure 14.24a. Optimization of linear system with genetic algorithm

Obviously, this optimization didn't work too well. Fig.14.24b shows a *moving average* computed over 100 iterations. The first value on Fig.14.24b is the average of the first 100 values of Fig.14.24a, the second value is the average of values 2 through 101 of Fig.14.24a, etc. I computed the moving average using the *AVERAGE* function of SAPS-II.

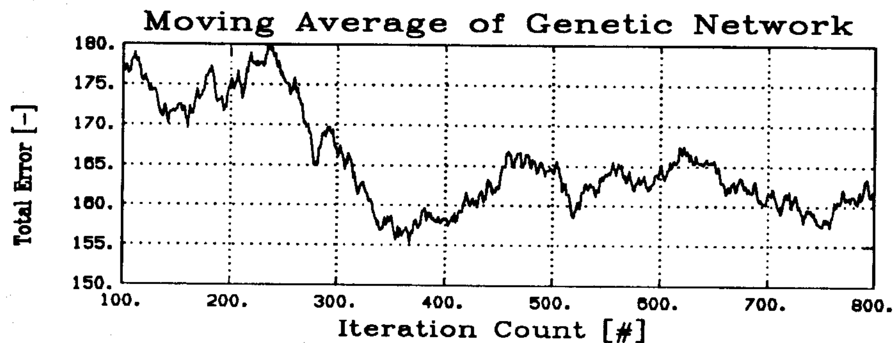

Figure 14.24b. Moving average of linear system

The genetic algorithm does learn indeed. However, the progress is painfully slow. My interpretation of these results is as follows: The terrain (in the parameter space) is very rough. Therefore, since we decided to use four levels only, each level contains both high mountains and deep gulches. Since we only retain the class values but not the quantitative values themselves, we throw away too much information. Consequently, I decided to rerun the optimization with $nlev = 16$. Since there are now more possible outcomes, I decided to consider 30% a solid "majority vote", and thus, I reduced *nperc* to 0.3. Another 1.2 *CPU-hours* later, I obtained the results for the modified algorithm. The simulation required less time because the optimization was terminated after 445 iterations. The results are shown in Fig.14.25.

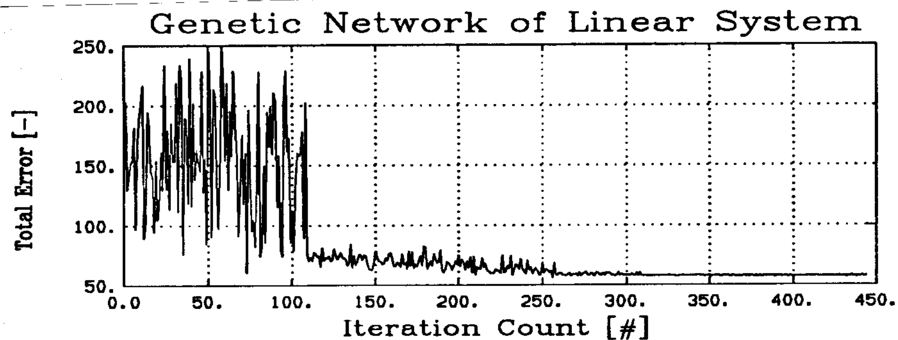

Figure 14.25. Optimization of linear system with genetic algorithm

This time, the genetic algorithm learned the weights much faster. Unfortunately, good genetic material was weeded out too quickly, and the algorithm ended up in a ditch.

Montana and Davis designed another genetic algorithm specifically for the purpose of training neural feedforward networks [14.24]. They argue against eliminating useful information by coding fuzzy information into our genetic strings. Indeed, both the *crossover* operator and the *mutation* operator can be applied to both quantitative (real) parameters and qualitative (fuzzy) parameters. In addition, they designed a set of interesting more advanced "genetic operators". They claim that networks function due to the synergism between weights associated with individual nodes. Thus, instead of applying the crossover algorithm blindly, they keep all the incoming weights of a node intact, and use either those of the father or those of the mother. Also, they consider multiple crossovers. Each node with all its incoming weights is arbitrarily taken from either the father or the mother. Thus, they simulate multiple crossovers of entire *features*. This makes a lot of sense. Montana and Davis also developed a very interesting concept of node assessment. They evaluate the quality (error) of a network in exactly the same manner that I use, i.e., they add the errors of the network over all training pairs. Then, they remove an individual node from the network, i.e., they lobotomize all incoming and outgoing connections of that node by setting the corresponding weights equal to zero, and recompute the quality of the modified network. They repeat the same procedure over and over, each time lobotomizing exactly one node. Using this information, they define the node whose presence has the least effect on the overall quality as the *weakest node*. Their mutation algorithm influences all incoming and outgoing weights of the weakest node in the hope to thereby improve the quality of the overall network. Again, this algorithm makes a lot of sense from an engineering point of view. They use a different distribution function for randomizing the initial weights of the network. Initially, they evaluate the quality of the entire genetic pool. However, in each generation, they pair up only one couple (as I do), and produce only one child which replaces the worst genetic string in the genetic pool (unless it is even worse). The parents are chosen randomly, but with a distribution function such that the second best genetic string is chosen 0.9 times as often as the best, and the third best string is chosen 0.9 times as often as the second best, etc. This rule makes also a lot of sense. Fig.14.26 shows the results of a simulation of the same problem that was discussed earlier, but now using Montana's and Davis' algorithm [14.24].

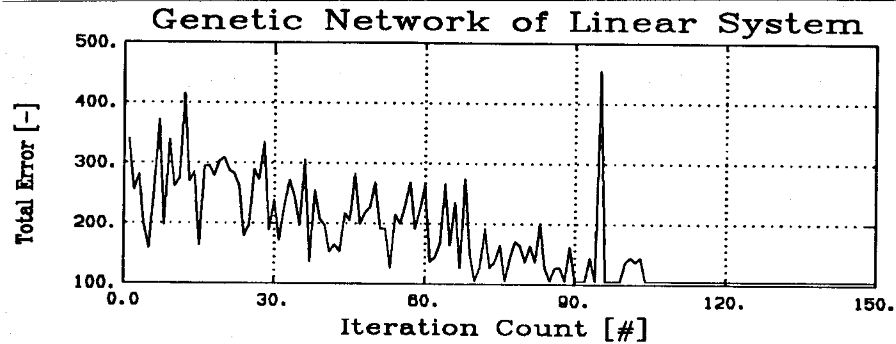

Figure 14.26. Montana/Davis optimization of linear system

The algorithm is very efficient. The error is reduced quickly, and the system learns fast. Unfortunately, it stagnates. Because the real parameter values are stored, no new information is entered into the system except through mutation. The system finds the smallest error among all the combinations of parameters present in the initial genetic pool reliably and quickly, but then it is stuck. The local superman wipes out his competition effectively and efficiently, and becomes a tyrant ... unfortunately, he is but a midget in global terms.

The algorithm suffers from the same disease as mine. In both algorithms, we were greedy, and tried to retain as much "good" genetic material as possible. We never let a "good" genetic string die. This is the seed to stagnation.

Even the fittest among us must die for progress to survive.

Goldberg suggested using a genetic algorithm closer to a biological model [14.6]. He proposed the following genetic dialect: We start out with a randomly chosen qualitative genetic pool (as in my algorithm). We evaluate the quality of the entire genetic pool (as in the case of Montana's and Davis' algorithm). We rank the genetic strings according to their quality. We define the *fitness* of a genetic string as:

$$fitness = \frac{1.0}{total\ error} \tag{14.43}$$

We then add up the fitnesses of all genetic strings in the genetic pool, and define the *relative fitness* of a genetic string as:

$$relative\ fitness = \frac{fitness}{sum\ over\ all\ fitnesses} \tag{14.44}$$

We then replace the entire genetic pool by a new pool in which each genetic string is represented never, once, or multiple times proportional to its relative fitness. Poor genetic strings are removed, while excellent genetic strings are duplicated many times. We then pair the genetic strings up arbitrarily. Each pair produces exactly two offsprings, one consisting of the head of the first string concatenated with the tail of the second, and the other consisting of the head of the second string concatenated with the tail of the first. We then let the old generation "die", and replace the *entire* genetic pool by the new generation. The algorithm is repeated until convergence.

This algorithm grants fit adults many children with varying sex partners potentially including twin siblings, and deprives unfit adults of the right to reproduction. The algorithm enforces strict birth control.

An obvious disadvantage of this genetic dialect is the need to evaluate the fitness of the entire genetic pool once per generation. Thus, we can optimize this algorithm over 16 generations only if we wish to compare it with the previously advocated dialect. However, I decided to compute 100 iterations anyway. Fig.14.27a shows the results of this optimization. I plotted the mean value of the total errors of all genetic strings in the genetic pool.

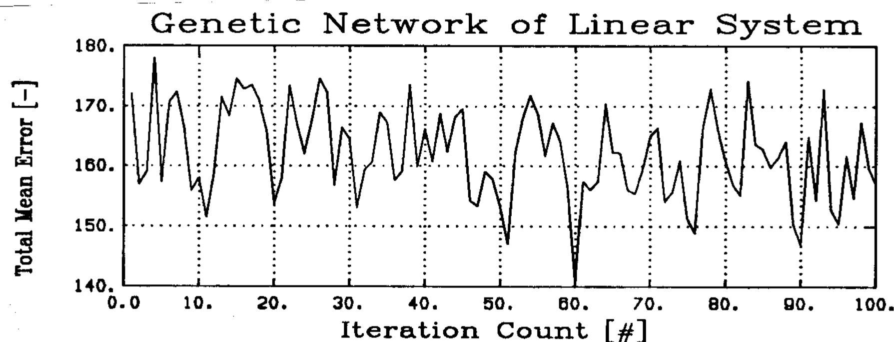

Figure 14.27a. Optimization of linear system with Goldberg's algorithm

The results are disappointing. If the algorithm has learned anything, the improvement is lost in the noise. I then computed a moving average of the previously displayed mean values over 50 generations. The results are shown in Fig.14.27b.

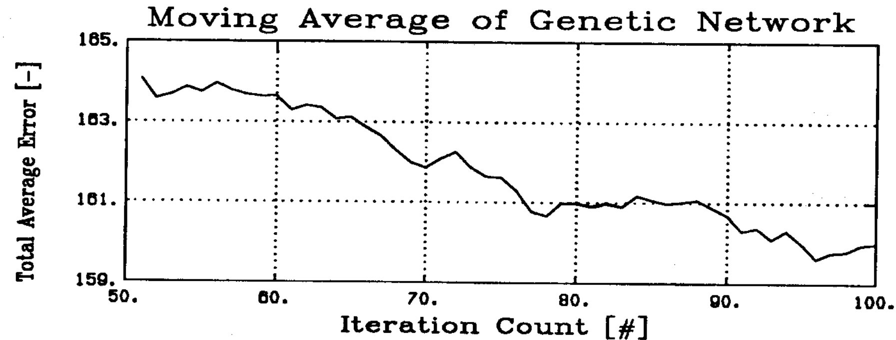

Figure 14.27b. Moving average of Goldberg's algorithm

Notice that the algorithm does indeed learn. However, the progress is unbelievably slow. I ran this program in batch. It required just over 12 *CPU–hours*. Obviously, I cannot determine whether this algorithm will stagnate or find the true minimum, but I believe that it will eventually find the true minimum.

The problem with Goldberg's algorithm is the following. The entire idea of the genetic crossover operator bases on the naïve belief that the child of two fit parents is, at least in a statistical sense, a fit child. This belief is justified in nature since the genetic parameters reflect *features*, and since the child will inherit an entire feature either from the father or from the mother. The overall fitness of a person is defined as the cumulative quality of all of his or her features. Thus, by inheriting features from both parents, fit parents will indeed have fit children. However, in our case, the individual parameters don't represent features. Each parameter influences all features, and each feature is influenced by all parameters. There is no compelling reason to believe that the crossover child of two fit parents is more fit than the average genetic string. Amazingly, the simulation results showed that such a child is indeed statistically more fit than the average genetic string ... but only by a narrow margin. This is why progress was so incredibly slow. It might have been worthwhile to combine Goldberg's algorithm with the previously proposed algorithm by Montana and Davis by combining the

genetic operators of the latter (*crossover of features* and *mutation of the weakest node*) with the social behavior of the former (replacement of the entire population once per generation), but I was afraid that the director of our computer center would knock me over my head if I continued in this way.

We have just demonstrated the power of evolutionary development. We learned a lesson: For evolution to work, we must permit all individual genetic strings to die irrespective of their quality. Retaining any individual string invariably leads to stagnation, and the evolutionary process comes to a halt. It is the power of ever–changing, non–repetitive variations — we call this phenomenon a chaotic steady–state — which enables the evolutionary process to continue.

In the beginning, there was Chaos.
Chaos nurtures Progress.
Progress enhances Order.
Order tries to defy Chaos at all cost.
... But the day Order wins the final battle
against Chaos, there will be mourning.
'Cause Progress is dead.

Genetic algorithms are a class of simple stochastic optimization techniques. Their behavior was demonstrated here by means of a neural network learning problem. However, no direct relationship exists between the two. Genetic algorithms can be interpreted as one particular implementation of a *Monte Carlo* optimization technique, and can be applied to arbitrary optimization problems. We shall return to this discussion in the second volume of this text in the context of general parameter estimation methods. It made sense to introduce the genetic algorithms here due to their inspirational biological foundation.

In the context of artificial neural networks, the genetic algorithm provides us with a *systematic and stable* technique to optimize arbitrarily constructed networks. This idea is fairly new, and it hasn't yet been exploited to its full potential. The idea is fruitful, because it removes configuration constraints on artificial neural networks. For instance, it allows us to optimize arbitrarily connected perceptron networks in a general, systematic, and robust (though fairly inefficient) way.

14.12 Neurobiological Learning

We have discussed various techniques for learning the weights of a neural network. The explicit counterpropagation network, both in its original feedforward form and in its derived recurrent form, is very attractive since it does not require any training at all. Yet, the algorithm requires that the coded (symbolic) targets be known *a priori*, and this is certainly not how our brain works. A freshly born human child does not have a notion of a $\mathcal{DOG}$. It learns the symbol itself from observation. In this sense, human learning is indeed "unsupervised". It is not even obvious that different human beings use the same symbolic representation for $\mathcal{DOG}$. If we were able to connect the *ego* of one person to the *self* of another (what an atrocious idea!), we might discover that the *ego* doesn't understand a word of what it reads in the *self* ... since the *self* uses a foreign alphabet.

The same difficulty holds true for the backpropagation algorithm, aside from the fact that the backpropagation algorithm learns far too slowly to be a realistic model of what happens in our brain.

Genetic algorithms are not plausible at all in the context of neurobiological learning, but at least, they add a stochastic component to the learning process. I am convinced that this stochastic component exists in our brain, but it is introduced through the mechanism of chaotic feedback loops.

How does the brain learn? We don't really know yet. All we can say is that the brain learns very *reliably* (no stability problems), and with amazing *efficiency*.

However, first attempts to shed light on this mystery have been made. Green and Triffet [14.7,14.33] have modeled "unit circuits" of the human brain (the allocortex, the cerebellum, and the cerebrum). Unit circuits are themselves organized in mini–zones (columns) which are connected to macro–zones (rows), comprising a matrix structure. The unit circuits are interconnected in various ways. For example, in the cerebellum, the granule cells connect to the Purkinje cells of all four neighboring unit circuits (excitatory synapses), and also the basket cells connect to the Purkinje cells of all four neighboring unit circuits (inhibitory synapses).

Triffet and Green represent their artificial neurons in a way which resembles the biological reality much more closely than any other

artificial neural networks. Their artificial neurons simulate the frequency modulation of real neurons. The potential of each neuron can assume one of 10 discrete levels. Levels '−6' to '−1' represent refractory states, level '0' represents the resting state, levels '+1' and '+2' represent excited states, and at level '+3', the neuron fires, thereby returning to level '−6' for the next refractory period. The neuron may immediately return to a higher refractory level. This will happen if the sum of the incoming weights of the firing neuron is sufficiently large.

The proposed model is a discrete–time model. Each clock impulse represents 2 *msec*. During the refractory period, the potential is incremented by one level once every clock impulse. Consequently, it will take 12 *msec* for a neuron to return from the lowest refractory level to the resting level. During the refractory period, all other input is ignored. This procedure simulates the compulsory refractory period of biological neurons.

Once a neuron has reached its resting level or an excited level, it is susceptible to both synaptic and extracellular input. Whether or not the potential of a resting or excited neuron will change during one clock impulse depends on three factors:

(1) Neurons have a natural tendency to return to their resting state. Thus, if no input is applied to the neuron, it will gradually return to its resting state. This is called the *potential relaxation* mechanism.

(2) Neurons are influenced by firings of neighboring neurons located both within the same unit circuit and also within neighboring unit circuits. The synaptic weights of the (partly excitatory and partly inhibitory) inputs will influence the potential of the neuron. If the neuron is exposed to a strong inhibitory input, the potential will be decremented. If it is exposed to a strong excitatory input, the potential will be incremented.

(3) Neurons are influenced by global electromagnetic wave transmissions (extracellular excitation or inhibition). Neurobiological measurements confirm that "intention potentials" sweep across specific regions of animal brains in advance of any motoric action.

In terms of our standard nomenclature, we can describe the potential (state) updating algorithm in the following way:

$$x^{\ell}_{t+\delta t} = x^{\ell}_t + \text{relax}(x^{\ell}_t) + \left(1 - \text{relax}(x^{\ell}_t)\right) \cdot \left(\sum_{\forall j} w^{\ell j}_t \cdot y^j_t + b^{\ell}_t - \text{relax}(-x^{\ell}_t)\right) \tag{14.45}$$

where y^j_t is the output of the jth neuron at time t. *relax(.)* is the relaxation function. It is '1' if its argument is negative, and it is '0' otherwise. Consequently, the (negative) state of a refractory neuron is simply incremented by '1' once every clock impulse. The non–negative) state of a resting neuron is exposed to synaptic input from all incoming neurons. The synaptic strength of the connection from the jth to the ℓth neuron at time t is represented by the weight $w^{\ell j}_t$. The extracellular input to the ℓth neuron at time t is represented through the bias b^{ℓ}_t. The third term ensures that excited neurons relax to their resting state when left alone.

The output function is simply:

$$y^{\ell}_{t+\delta t} = \text{firing}(x^{\ell}_{t+\delta t}) \tag{14.46}$$

The *firing* function is '1' if its argument is '+3' or larger. It is '0' otherwise. That is: the output of the neuron simply registers the fact the the neuron just fired.

Finally, we need to return the fired neuron to a refractory state. The proposed rule is simply:

$$\text{if } \; y^{\ell}_{t+\delta t} = 1 \; \text{ then } \; x^{\ell}_{t+\delta t} = \min(x^{\ell}_{t+\delta t} - 9, -4) \tag{14.47}$$

Thus, if the state is at the lowest firing level (level $+3$), it returns immediately to the lowest refractory level (level -6). If the state is elevated one level beyond the lowest firing level, it returns to the refractory level -5. If it is elevated two levels beyond the lowest firing level, it returns to the refractory level -4, but this is the highest level that we allow a fired neuron to return to. Such a neuron will require 8 *msec* to relax to its resting level.

Both the notation and the precise logic of the described algorithm deviate slightly from those used by Triffet and Green [14.7,14.33] in order to fit smoothly into the framework of this chapter.

The neural network learns both temporal and static patterns using three separate mechanisms:

(1) The synaptic weights, which are represented as integers, are incremented or decremented using a modified Hebb rule. However, unlike the traditional artificial neural networks, the optimal weights of a temporal pattern are themselves functions of

time. They may constantly be modified (even after learning is completed) while a temporal pattern is processed by the brain. Green and Triffet [14.7,14.33] call these temporal sequences "programs".

(2) The circuit learns particular features of temporal patterns simply as a consequence of the memorizing power of the feedback loops in the circuit.

(3) The circuit is prepared for learning by the electromagnetic "intention wave" which precedes the actual neuronic signals.

The weight updating algorithm is straightforward. Whenever firing of one neuron leads to firing of another, the weight connecting these two neurons is incremented or decremented depending on the nature of the synaptic connection. If the connection is excitatory, the (positive) weight is incremented, if it is inhibitory, the (negative) weight is decremented. In both cases, the weight becomes stronger. If a weight has not been modified for a given period of time, it can be automatically weakened, i.e., positive weights are decremented while negative weights are incremented. This mechanism is called *weight relaxation*. Weight relaxation simulates the process of forgetting. Notice that this simple algorithm solves the problem of updating the "hidden layers" of traditional neural networks. Implementation of the frequency modulation of biological neurons provides us with an algorithm to train neurons arbitrarily located anywhere in the network.

Biases are updated by a discrete implementation of the (distributed) wave equation. The input to the wave equation is the (distributed) firing pattern of the Purkinje cells. Whenever a Purkinje cell fires, the bias of that unit circuit is incremented. The time constant of the wave equation is chosen such that the electromagnetic wave travels slightly faster through the network than the neuronic signals.

Green and Triffet [14.7,14.33] claim that their networks can learn patterns much more quickly than traditional networks. All weights are learned in parallel. No backpropagation is necessary. While the authors haven't convinced me yet that they really have modeled true neurobiological learning, their conceptual framework is revolutionary, and it certainly deserves further pursuit.

14.13 Summary

In this chapter, we have discussed how biological research can inspire us in solving engineering problems, and how engineering methodologies can help us understand biological processes better. It has helped us gain a better understanding of how our brain and body functions achieve optimal performance, and it has brought us a step closer to answering the question how engineering systems, such as robots, can be equipped with a modest amount of decision making capability and *responsibility*. This entire research area is very new, and many problems still await a solution.

References

[14.1] Albert Bandura (1977), *Social Learning Theory*, Prentice–Hall, Englewood Cliffs, N.J.

[14.2] Rodney A. Brooks (1986), "A Robust Layered Control System for a Mobile Robot", *IEEE J. of Robotics and Automation*, Vol. RA2, pp.14–23.

[14.3] Rodney M. J. Cotterill, ed. (1988), *Computer Simulation in Brain Science*, Cambridge University Press, Cambridge, MA.

[14.4] William R. D. Fairbairn (1954), *An Object–Relations Theory of the Personality*, Basic Books, New York.

[14.5] Paul Federn (1952), *Ego Psychology and the Psychoses*, Basic Books, New York.

[14.6] David E. Goldberg (1989), *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison–Wesley, Reading, MA.

[14.7] Herbert S. Green, and Terry Triffet (1989), "A Zonal Model of Cortical Functions", *J. Theoretical Biology*, **136**, pp. 87–116.

[14.8] Stephen Grossberg (1982), *Studies of Mind and Brain: Neural Principles of Learning, Perception, Development, Cognition, and Motor Control*, D. Reidel Publishing, Hingham, MA.

[14.9] Stephen Grossberg (1987), *The Adaptive Brain*, North–Holland Publishing, Amsterdam, The Netherlands.

[14.10] Stephen Grossberg (1988), *Neural Networks and Natural Intelligence*, M.I.T. Press, Cambridge, MA.

[14.11] Donald O. Hebb (1949), *The Organization of Behavior: A Neuropsychological Theory*, John Wiley & Sons, New York.

[14.12] Robert Hecht–Nielsen (1990), ***Neurocomputers***, Addison–Wesley, Reading, MA.

[14.13] John Holland (1975), *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, MI.

[14.14] John J. Hopfield (1982), "Neural Networks and Physical Systems With Emergent Collective Computational Abilities", *Proceedings of the National Academy of Sciences, USA*, **79**, pp. 2554–2558, National Academy of Sciences, Washington, D.C.

[14.15] Otto Kernberg (1975), *Borderline Conditions and Pathological Narcissism*, Aronson, New York.

[14.16] Christof Koch, and Idan Segev, eds. (1989), *Methods in Neuronal Modeling: From Synapses to Networks*, A Bradford Book, M.I.T. Press, Cambridge, MA.

[14.17] Teuvo Kohonen (1989), *Self–Organization and Associative Memory*, Third Edition, Springer Verlag, Series in Information Sciences, **8**, Berlin.

[14.18] Granino A. Korn (1991), *Neural–Network Experiments on Personal Computers*, M.I.T. Press, Cambridge, MA.

[14.19] Ronald D. Laing (1969), *The Divided Self*, Pantheon Books, New York.

[14.20] Richard P. Lippmann (1987), "An Introduction to Computing with Neural Nets", *IEEE ASSP Magazine*, April, pp. 4–22.

[14.21] Warren S. McCulloch, and Walter Pitts (1943), "A Logical Calculus of the Ideas Immanent in Nervous Activity", *Bulletin of Mathematical Biophysics*, **5**, pp. 115–133.

[14.22] Marvin Minsky (1985), *The Society of Mind*, Simon and Schuster, New York.

[14.23] Marvin L. Minsky, and Seymour Papert (1969), *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, MA, Expanded Edition: 1988.

[14.24] David J. Montana, and Lawrence Davis (1989), "Training Feedforward Neural Networks Using Genetic Algorithms", *Proceedings International Joint Conference on Artificial Intelligence, IJCAI–89*, Vol. 1, Morgan Kaufmann, Palo Alto, CA, pp. 762–767.

[14.25] Kumpati S. Narendra, and Kannan Parthasarathy (1990), "Identification and Control of Dynamical Systems Using Neural Networks", *IEEE Transactions on Neural Networks*, **1**(1), pp. 4–27.

[14.26] Jean Piaget, and Barbel Inhelder (1967), *The Child's Conception of Space*, Norton, New York.

[14.27] John R. Rice, and Ronald E. Boisvert (1985), *Solving Elliptic Problems Using ELLPACK*, Springer Verlag, New York.

[14.28] Edward Rietman (1989), *Exploring the Geometry of Nature: Computer Modeling of Chaos, Fractals, Cellular Automata, and Neural Networks*, Windcrest Publishing, Blue Ridge Summit, PA.

[14.29] Helge Ritter, Thomas Martinetz, and Klaus Schulten (1990), *Neuronale Netze*, Addison–Wesley, Munich, FRG, English translation currently under preparation.

[14.30] Frank Rosenblatt (1962), *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Spartan Books, Washington, D.C.

[14.31] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams (1986), "Learning Internal Representations by Error Propagation", **in:** *Parallel Distributed Processing: Explorations in the Microstructure of Cognitions, Vol.1: Foundations*, (D.E. Rumelhart, and J.L. McClelland, eds.), M.I.T. Press, Cambridge, MA, pp. 318–362.

[14.32] Timo Sorsa, Heikki N. Koivo, and Hannu Koivisto (1990), "Neural Networks in Process Fault Diagnosis", submitted to *IEEE Trans. Systems, Man, Cybernetics.*

[14.33] Terry Triffet, and Herbert S. Green (1990), "Structured Neurobiological Networks", submitted to *J. Theoretical Biology.*

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

[14.35] Bernard Widrow, and M. E. Hoff (1960), "Adaptive Switching Circuits", *IRE WESCON Convention Record*, Fourth Part, Institute of Radio Engineers, New York, pp. 96–104.

[14.36] Ben P. Yuhas, Moise H. Goldstein, Jr., and Terrence J. Sejnowski (1989), "Integration of Acoustic and Visual Speech Signals Using Neural Networks", *IEEE Communication Magazine*, **27**(11), pp. 65–71.

Bibliography

[B14.1] Casimir C. Klimasauskas, ed. (1989) *The 1989 Neuro–Computing Bibliography*, M.I.T. Press, Cambridge, MA.

[B14.2] Philip D. Wasserman, and Roberta M. Oetzel (1990) *Neural Source, The Bibliographic Guide to Artificial Neural Networks*, Van Nostrand Reinhold, New York.

Homework Problems

[H14.1] Perceptron Network for Exclusive–Or Problem

Implement a perceptron network as shown in Fig.14.8 which solves the exclusive–or problem. For this purpose, compute the weights and thresholds of the two hidden perceptrons such that each one of them implements one of the slanted lines of Fig.14.9. Manually evaluate the truth table between the two inputs u_1 and u_2 and the two outputs of the hidden layer y_1 and y_2. Since you know what is the desired output value y for each combination of the hidden layer outputs, you can also generate a truth table which maps the hidden layer into the output layer. Draw a picture in the $< y_1, y_2 >$ plane similar to Fig.14.7 which represents this truth table. Draw a new slanted line into this picture which separates the '1' outputs from the '0' outputs. Design the weights and thresholds of the output perceptron such that it implements the new slanted line.

Program the perceptron network in CTRL–C (MATLAB), and check its correct performance by looping over all four input combinations.

[H14.2] Backpropagation Network for Linear System

Reimplement the backpropagation network of the linear system (with four inputs and three outputs) in DESIRE/NEUNET. Compare the DESIRE/NEUNET solution of the exclusive–or problem (presented in this chapter) with the CTRL–C version of the same problem (also shown). This should give you sufficient help to make the transcription of the CTRL–C solution to the backpropagation network for the linear system (shown in this chapter) to its DESIRE/NEUNET equivalent an easy task.

Run the program over 774 iterations (preferably on a 386– or 486–class machine) and compute the speed ratio between the DESIRE/386 solution and the CTRL–C/VAX solution. Let the DESIRE/NEUNET program run further and find which is the smallest value of the performance index that you can obtain.

[H14.3] Adaptive Backpropagation Network for Linear System

In hw(H14.2), we simulated the continuous system once over 900 time units, and stored the results away for training the backpropagation network. Modify the program of hw(H14.2) such that you constantly integrate the continuous system in parallel with training the backpropagation network. Set the communication interval to 3 *sec*. The backpropagation network is now placed in the *OUT* block which is executed once per communication interval. This is necessary since you will require an integration step size

which is considerably smaller than three time units to obtain decent simulation results. The updating of the $\mathbf{WW}^1$ and $\mathbf{WW}^2$ matrices and of the $\mathbf{bb}^1$ and $\mathbf{bb}^2$ vectors occurs in a *SAMPLE* block which is executed once every 300 communication intervals.

[H14.4] Counterpropagation Network for Linear System

Find a CTRL–C (or MATLAB) solution to generate Fig.14.20. Start by designing two small procedures implementing 10 *bit* A/D and D/A converters. The A/D converter takes a real number between -1.0 and $+1.0$, and generates a vector of length 10 containing only -1.0 and $+1.0$ elements. The D/A converter accepts a binary vector of length 10, and generates a real number between -1.0 and $+1.0$. Test the correctness of the two routines by computing:

$$xx = \mathrm{DtoA}(\mathrm{AtoD}(x))$$

for various numbers x between -1.0 and $+1.0$.

Use the *inpt* matrix from before, and generate the new analog input matrix *ainpt* by concatenating u_{k-1} with $\mathbf{x_{k-1}}$, then with u_k, further with $\mathbf{x_k}$, and finally with u_{k+1}. This is accomplished by concatenating columns of the former *inpt* matrix shifted down by one or two elements. Notice that you lose two rows in this process. Compute the new analog target matrix *atarg* from the old *inpt* matrix in the same way. *ainpt* should be a matrix with 299 rows and nine columns, while *atarg* should be a matrix with 299 rows and three columns.

Convert the two analog matrices to digital matrices using the previously defined *AtoD* routine. The digital input matrix *dinpt* should have 299 rows and 90 columns, while the digital target matrix *dtarg* should have 299 rows and 30 columns.

Set up the weight matrices of the counterpropagation network by using the first 269 rows of each of the digital matrices.

Recall the counterpropagation network by applying the remaining 30 rows of the digital input matrix to the counterpropagation network. Compute the resulting digital target vectors for each of the digital input vectors. Convert the resulting digital target vectors back to analog target vectors using the previously designed *DtoA* routine, and plot the resulting values together with the original values.

[H14.5] Adaptive Counterpropagation Network for Linear System

This program consists of a big loop which is executed 299 times. Construct digital input and target vectors one at a time from the previously computed *inpt* matrix. During the first iteration, simply place the two vectors in the

weight matrices. The hidden layer has a length of one. In subsequent iterations, apply the digital input vector to the existing network, and compare the resulting digital output vector with the correct digital output vector. If the vector is correct, do nothing, otherwise add the correct input/target pair to the weight matrices, thereby incrementing the length of the hidden layer by one.

Generate three different graphs. The first graph compares the true analog outputs to the computed analog outputs for rows 10 to 39, the second compares the same values for rows 150 to 179, and the third compares the same values for rows 270 to 299. Discuss the results! Determine the length of the hidden layer after 299 iterations!

[H14.6]* Reinventing the Binary Code

Design a two–layer pseudo–backpropagation network (i.e., a backpropagation network using LIMIT/TRI activation function pairs) which map binary unit vectors of length 16 through a hidden layer of length four back into the original unit vector representation. I suggest to code the kth unit vector as:

$$\mathbf{e_k} = [\,-0.9\;,\;\ldots\;,\;-0.9\;,\;+0.9\;,\;-0.9\;,\;\ldots\;,\;-0.9\,]'$$

That is, the logical *true* state is represented by the real value +0.9, while the logical *false* state is represented by the real value −0.9. This applies both to the input and to the target vectors.

Use gain values of $g_1 = 0.05$, and $g_2 = 0.07$, and use relaxation momentums of $m_1 = -0.06$, and $m_2 = -0.08$. Initialize the weight matrices to $0.1 \cdot RAND$. Simulate the network over 20,000 iterations, applying sequentially one input pair after the other. The weights can be updated once per iteration, i.e., it is not necessary to loop over all input/target pairs before updating the weights. Display the mean square error.

Create a DESIRE/NEUNET program implementing this algorithm. Don't try CTRL–C or MATLAB on this problem. One simulation run in CTRL–C or MATLAB will require several hours of CPU time, while DESIRE/NEUNET will execute the same problem in less than 1 *min* on a 486–class machine.

Recall the learned network once for every input/target pair, and display the resulting hidden layer output vectors. Look at the signs of the vector, and interpret the results. You should notice that this neural network just "reinvented" the binary code. Repeat the simulation several times. Notice that each result is different. Each simulation reinvents the binary code, but generates a different *sequence* of binary numbers. This is what I meant when I wrote that each human brain probably uses a different alphabet to encode the same symbolic knowledge.

Projects

[P14.1] Intelligent Autopilot

Apply the recurrent counterpropagation network presented in Section 14.7 to the continuous simulation of a Boeing 747 jetliner in high altitude horizontal flight, i.e., to the simulation program designed in pr(P4.1). The purpose of this study is to check whether we can apply the proposed technique as reliably to a highly non–linear system as to a linear system.

Enhance the ACSL program designed in pr(P4.1) by a set of faulty operational modes such as heavy ice on the wings or loss of one of the four engines.

For each of the fault modes, identify a set of weighting matrices that characterize the fault.

Modify the ACSL program once more. This time, one of the faults should be chosen arbitrarily at a randomly selected point in time during the simulation.

Use the counterpropagation network of the undamaged aircraft to identify *when* the accident has happened, and discriminate the correct fault by comparing the recoded continuous data after the accident occurred and after the transients resulting from the accident have died out with forecasts obtained from the neural network using all of the stored weighting matrices. The forecast with the smallest deviation identifies (most likely) the type of fault that has occurred.

Research

[R14.1] Simulating an Ant Brain

Rodney Brooks [14.2] developed a series of cellular automata which cooperate in simulating ant behavior. He designed and built a very small robot using solar batteries as its only source of energy, with an on–board chip implementing his cellular automata programs. He produced a beautiful and very impressive video showing the behavior of his robot as it walks over phone books and gravel, and reacts in various other ways to its environment.

This robot clearly exhibits insect behavior. Therefore, we can say that Brooks' cellular automata represent a good working hypothesis of how a relatively small series of independent unintelligent agencies [14.22] can cooperate to perform amazing tasks. While the cellular automata demonstrate how autonomous agencies can cooperate in task solving, it does not tell us anything about the physical configuration of an ant brain.

It should be perfectly feasible to simulate a true ant brain since an ant brain contains only about 20,000 neurons, and yet, ants exhibit a highly interesting and fairly complex social behavior. The problem is that we don't yet truly understand how brains are configured, and how they process information. If we are able to reproduce ant behavior using a neural network, we might learn something about the potential physical configuration of a very simple *id*.

I propose to take Green's and Triffet's neural network approach [14.7, 14.33], and try to implement Brooks' [14.2] cellular automata with it. Try to re–utilize the same unit circuits as much as possible for different related tasks. Determine what is the minimum number of unit circuits necessary to reproduce Brooks' ant behavior using a neural network.

[R14.2] Teacher Network

Investigate the potential of neural networks for data preprocessing. A "teacher network" accepts arbitrary input/target pairs and prepares them for training a "student network".

Investigate the possibilities for training a future teacher network, i.e., find out whether it is possible to have a teacher network train a student network to be become a better teacher network than the original teacher network was.

[R14.3] Chaos and Learning

Build a neural network system with a forward network and an inverse network. Both networks are of the same type. To start with, they might be feedforward networks. However, feedforward networks are not useful for learning temporal patterns which are fed sequentially into the network. On the longer run, you may wish to replace the feedforward networks by recurrent counterpropagation networks as proposed in Fig.14.19. Introduce global feedback loops as proposed in Fig.14.21. Implement an adaptive controller as suggested in Fig.14.22, i.e., a simple *ego*, to ensure global stability of the overall system. Design the adaptive controller such that the overall system is either deterministic or mildly chaotic. Expose both versions of the network system to various stimuli, and investigate the role of chaos in neural network learning.