Neural Network Architectures

An Introduction

Judith E. Dayhoff

VNR VAN NOSTRAND REINHOLD
New York

To the memory of Margaret Oakley Dayhoff, pioneer in evolutionary biology.

Library of Congress Catalog Number 89-78147
ISBN 0-442-20744-1

Printed in the United States of America

Van Nostrand Reinhold
115 Fifth Avenue
New York, New York 10003

Van Nostrand Reinhold International Company Limited
11 New Fetter Lane
London EC4P 4EE, England

Van Nostrand Reinhold
480 La Trobe Street
Melbourne, Victoria 3000, Australia

Nelson Canada
1120 Birchmount Road
Scarborough, Ontario M1K 5G4, Canada

16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data

Dayhoff, Judith E.
Neural network architectures : an introduction / by Judith E. Dayhoff.
p. cm.
ISBN 0-442-20744-1
1. Computer architecture. 2. Neural computers. 3. Computer networks. I. Title.
QA76.9.A73D39 1990
006.3—dc20 89-78147
CIP

4 Back-Error Propagation

Back-error propagation is the most widely used of the neural network paradigms and has been applied successfully in applications studies in a broad range of areas. Applications studies have spanned tasks from military pattern recognition to medical diagnosis, and from speech recognition and synthesis to robot and autonomous vehicle control. Back-propagation has been applied to character recognition, sonar target recognition, image classification, signal encoding, knowledge processing, and a variety of other pattern-analysis problems. Back-propagation can attack any problem that requires pattern mapping: Given an input pattern, the network produces an associated output pattern.

Back-propagation is one of the easiest networks to understand. Its learning and update procedure is intuitively appealing because it is based on a relatively simple concept: If the network gives the wrong answer, then the weights are corrected so that the error is lessened and as a result future responses of the network are more likely to be correct.

Back-propagation networks are usually layered, with each layer fully connected to the layers below and above. When the network is given an input, the updating of activation values propagates forward from the input layer of processing units, through each internal layer, to the output layer of processing units. The output units then provide the network's response. When the network corrects its internal parameters, the correction mechanism starts with the output units and back-propagates backward through each internal layer to the input layer — hence the term back-error propagation, or back-propagation.

The conceptual basis of back-propagation was first presented in 1974 by Paul Werbos, then independently reinvented by David Parker in 1982, and

presented to a wide readership in 1986 by Rumelhart and McClelland. Their book, *Parallel Distributed Processing* introduced the broad potential of the neural network approach and widespread interest in back-propagation followed.

Notable early applications of back-propagation were done by Terry Sejnowski and his colleagues at Johns Hopkins University. Particularly striking is the NETTalk program, by Sejnowski and Rosenberg, in which a back-propagating neural network was trained in the rules of phonetics in just two weeks of CPU time on a VAX. The network, which produced sounds as it was learning phonetic rules, sounded uncannily like a child learning to read aloud.

Back-propagation is a tremendous step forward compared to its predecessor, the perceptron. The perceptron was limited to only two layers of processing units, with only a single layer of adaptable weights. This key limitation meant that the perceptron could only classify patterns that were linearly separable. Back-propagation overcomes this limitation because it can adapt two or more layers of weights, and uses a more sophisticated learning rule. The power of back-propagation lies in its ability to train hidden layers and thereby escape the restricted capabilities of single-layer networks.

When two or more layers of weights are adjusted, the network has middle — or hidden — layers of processing units. Each hidden layer acts as a layer of "feature detectors" — units that respond to specific features in the input pattern. These feature detectors organize as learning takes place, and are developed in such a way that they accomplish the specific learning task presented to the network.

For years it has been known that the key to pattern-recognition problems and many learning tasks has been in the choice of salient features to consider. Knowing the correct distinguishing features allows one to classify different patterns. Back-error propagation can identify an effective set of features automatically. Thus, a fundamental step toward solving pattern recognition problems has been taken with back-error propagation.

AN OVERVIEW

Typically, back-propagation employs three or more layers of processing units. Figure 4-1 shows the topology for a typical three-layer back-propagation network. The bottom layer of units is the input layer — the only units in the network that receive external input. The layer above is the hidden layer, in which the processing units are interconnected to layers above and below. The top layer is the output layer.

The layers in Figure 4-1 are fully interconnected — each processing unit is connected to every unit in the layer above and in the layer below. Units are not

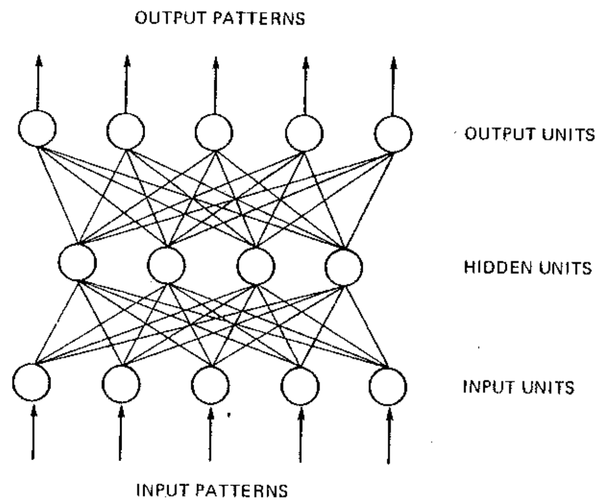


Figure 4-1. A three-layered back-propagation network, fully interconnected.

connected to other units in the same layer. A back-propagation network must have at least two layers. Figure 4-2 shows such a network with five layers, all fully interconnected. The three internal layers are hidden layers (with hidden units). Back-propagation networks do not have to be fully interconnected,

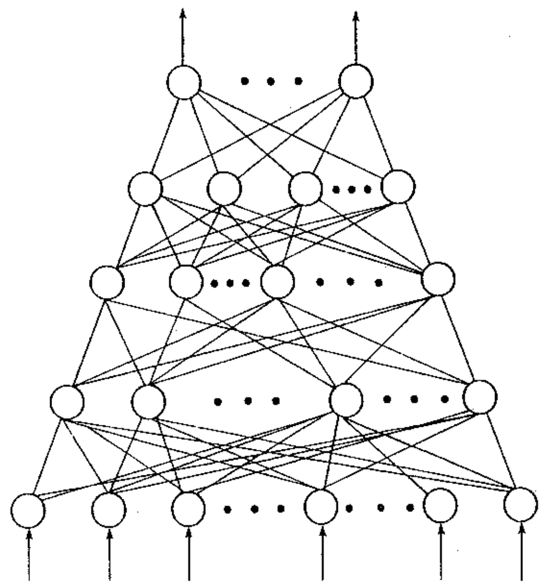


Figure 4-2. A five-layered back-propagation network, fully interconnected.

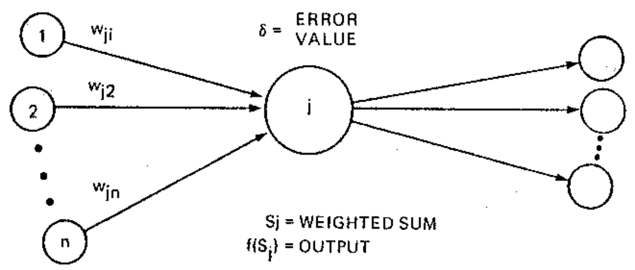
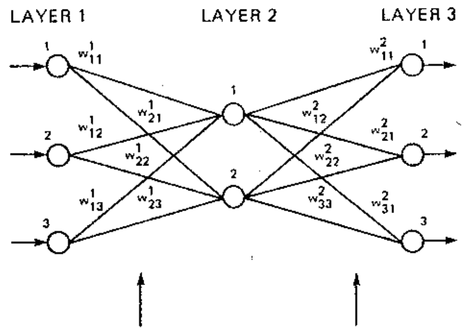


Figure 4-3. The basic back-propagation processing unit.

although most applications work has been done with fully interconnected layers.

In Figure 4-3, a basic back-propagation processing unit, inputs are shown at the left, and at the right are units that receive outputs from the processing unit at the center. The processing unit has a weighted sum of inputs (S_j), an output value (a_j), and an associated error value (δ_j) that is used during weight adjustments.

Weights associated with each interconnection are adjusted during learning. The weight to unit j from unit i is denoted here as w_{ji}. After learning is completed, the weights are fixed. These final values are then used during "recall" sessions. Figure 4-3 illustrates the weights along the incoming connections to the processing unit at the center. There is a matrix of weight values



$$\begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \end{bmatrix} \quad \begin{bmatrix} w^2_{11} & w^2_{12} \\ w^2_{21} & w^2_{22} \\ w^2_{31} & w^2_{32} \end{bmatrix}$$

Figure 4-4. Weight matrices of a three-layered back-propagation system. Units are indexed starting with 1 in each layer. Superscripts have been added to distinguish weights in different layers.

that corresponds to each layer of interconnections (Figure 4-4); these matrices are indexed with superscripts to distinguish weights in different layers.

The *activation level* parameter associated with each processing unit in the back-propagation network is also the output value of each processing unit. An animated simulation of a back-propagation neural network depicts the activation level for each unit by using colors, gray levels, or different sizes of units.

Figure 4-5 graphically shows both the activation levels and weight levels in a back-propagation network. Squares of different shades represent processing units with different activation levels. The two grids represent two layers of weights, with the shade of each square determined by the weight value for each interconnection. The numeric scale for determining the shade is different for the activation levels and the weights, as each type of parameter has a different range.

A back-propagating neural network is trained by supervised learning. The network is presented with pairs of patterns — an input pattern paired with a target output. Upon each presentation, weights are adjusted to decrease the difference between the network's output and the target output. A training set — a set of input/target pattern pairs — is used for training, and is presented to the network many times. After training is stopped, the performance of the network is tested.

The back-propagation learning algorithm involves a forward-propagating step followed by a backward-propagating step. Both the forward- and back-propagation steps are done for each pattern presentation during training. The forward-propagation step begins with the presentation of an input pattern to the input layer of the network, and continues as activation level calculations propagate forward through the hidden layers. In each successive layer, every processing unit sums its inputs and then applies a sigmoid function to compute its output. The output layer of units then produces the output of the network.

The backward-propagation step begins with the comparison of the network's output pattern to the target vector, when the difference, or "error," is calculated. The backward-propagation step then calculates error values for hidden units and changes for their incoming weights, starting with the output layer and moving backward through the successive hidden layers. In this back-propagating step the network corrects its weights in such a way as to decrease the observed error.

The error value (δ) associated with each processing unit reflects the amount of error associated with that unit. This parameter is used during the weight-correction procedure, while learning is taking place. A larger value for δ indicates that a larger correction should be made to the incoming weights, and its sign reflects the direction in which the weights should be changed.

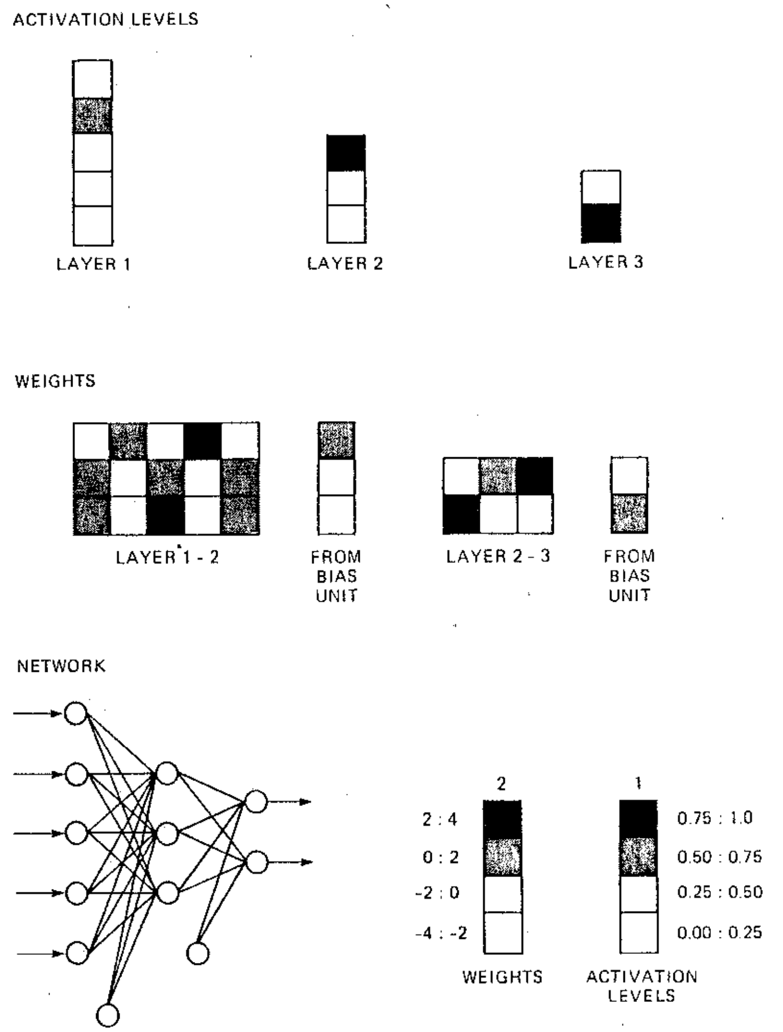


Figure 4-5. A graphical representation of activation levels and weights.

FORWARD-PROPAGATION

The forward-propagation step is initiated when an input pattern is presented to the network. Each input unit corresponds to an entry in the input pattern vector, and each unit takes on the value of this entry. After the activation levels for the first layer of units is set, the remaining layers perform a forward-propagation step, which determines the activation levels of the other layers of units.

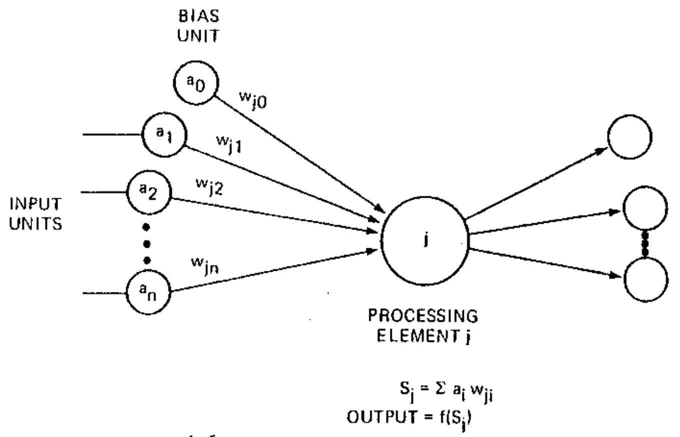


Figure 4-6. The forward-propagation step.

Figure 4-6 illustrates the specifics of the forward-propagation step. Incoming connections to unit j are at the left and originate at units in the layer below. Output values from these units arrive at unit j and are summed by

$$S_j = \sum_i a_i w_{ji} \tag{4-1}$$

where a_i = the activation level of unit i, and w_{ji} = the weight from unit i to unit j (unit i is one layer below unit j). After the incoming sum S_j is computed, a function f is used to compute $f(S_j)$. The function f, a sigmoid curve, is illustrated in Figure 4-7a.

The sigmoid curve is relatively flat at both ends, and has a rapid rise in the middle. When x is less than -3, $f(x)$ is close to 0; when x is greater than 3, $f(x)$ is close to 1. In fact, $f(x)$ approaches 1 asymptotically as x gets larger, and $f(x)$ approaches 0 asymptotically as x becomes a greater negative value. There is a transition from 0 to 1 that takes place when x is approximately 0 ($-3 < x < 3$). The sigmoid function performs a sort of "soft" threshold that is rounded (and differentiable) compared to the step function (Figure 4-7b).

The equation for the sigmoid function is

$$f(x) = \frac{1}{1 + e^{-x}} \tag{4-2}$$

Since the operand is the weighted sum of unit j, we have

$$f(S_j) = \frac{1}{1 + e^{-S_j}} = \frac{1}{1 + e^{-\Sigma a_i w_{ji}}}$$

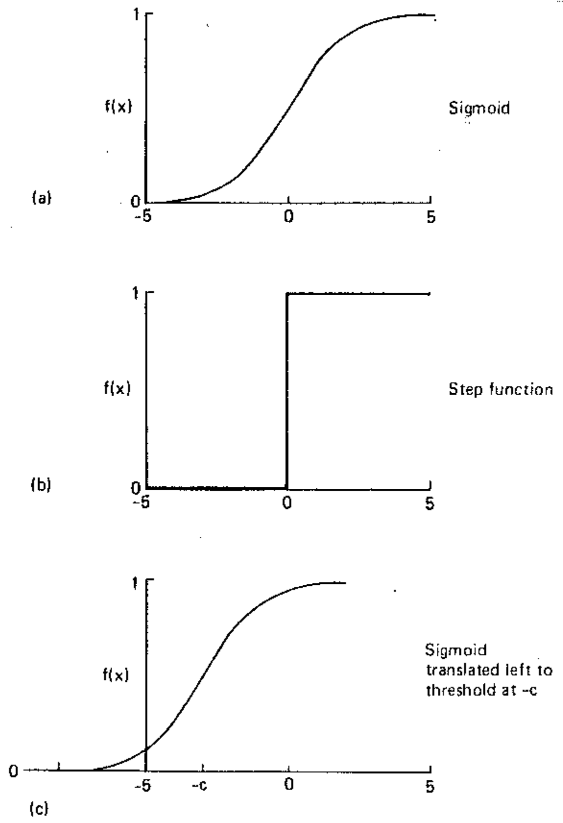


Figure 4-7. Threshold functions. (a) The sigmoid function. (b) A step function. (c) A sigmoid moved c units to the left, to threshold at $-c$.

After the sigmoid function is computed on S_j, the resulting value becomes the activation level of unit j. This value, the output of unit j, is sent along the output interconnections (on the right of Figure 4-6). The same output value is sent along all of the output interconnections.

The input layer of units is a special case. These units do not perform the weighted sum on their inputs because each input unit simply assumes the corresponding value taken from the input vector. We consider the input layer to be a layer of the network even though it does not perform the weighted sum and sigmoid calculations.

Some back-propagation networks employ a bias unit as part of every layer but the output layer. This unit has a constant activation value of 1. Each bias unit is connected to all units in the next higher layer, and its weights to them are adjusted during the back-error propagation. The bias units provide a constant term in the weighted sum of the units in the next layer. The result is sometimes an improvement on the convergence properties of the network.

The bias unit also provides a "threshold" effect on each unit it targets. It contributes a constant term in the summation S_j, which is the operand in the sigmoid function (4-2). This is equivalent to translating the sigmoid curve in Figure 4-7 to the left or to the right. For example, suppose the bias unit (a_0) in Figure 4-6 has an output value of 1.0 and a weight

$$C = w_{j0}$$

Then let

$$z = \sum_{i=1}^{n} a_i w_{ji}$$

Then z is the incoming sum from all of the units in Figure 4-6 other than the bias unit. If the bias unit contributes the constant C to the incoming sum of unit j, then this sum becomes

$$z + C$$

Now compare the graph of the sigmoid function for $f(z)$, given in Figure 4-7a, to the graph of the function $f(z + C)$, shown in Figure 4-7c. The constant C translates the graph to the left by the amount C, thus moving the threshold of the sigmoid curve from 0 to $-C$. In this way, the bias units provide an adjustable threshold for each target unit. The threshold for unit j then comes from the value of w_{j0}, the weight of the interconnection from the bias unit.

BACKWARD PROPAGATION

Figure 4-8 illustrates the backward propagation step. Here the δ values are calculated for all processing units and weight changes are calculated for all interconnections. The calculations begin at the output layer and progress backward through the network to the input layer.

The error-correction step takes place after a pattern is presented at the input layer and the forward-propagation step is complete. Each processing unit in the output layer produces a single real number for its output, which is compared to the target output specified in the training set (Figure 4-8a). Based on this difference, an error value is calculated for each unit in the output layer as in Fig. 4-8b. Then the weights are adjusted for all of the interconnections that

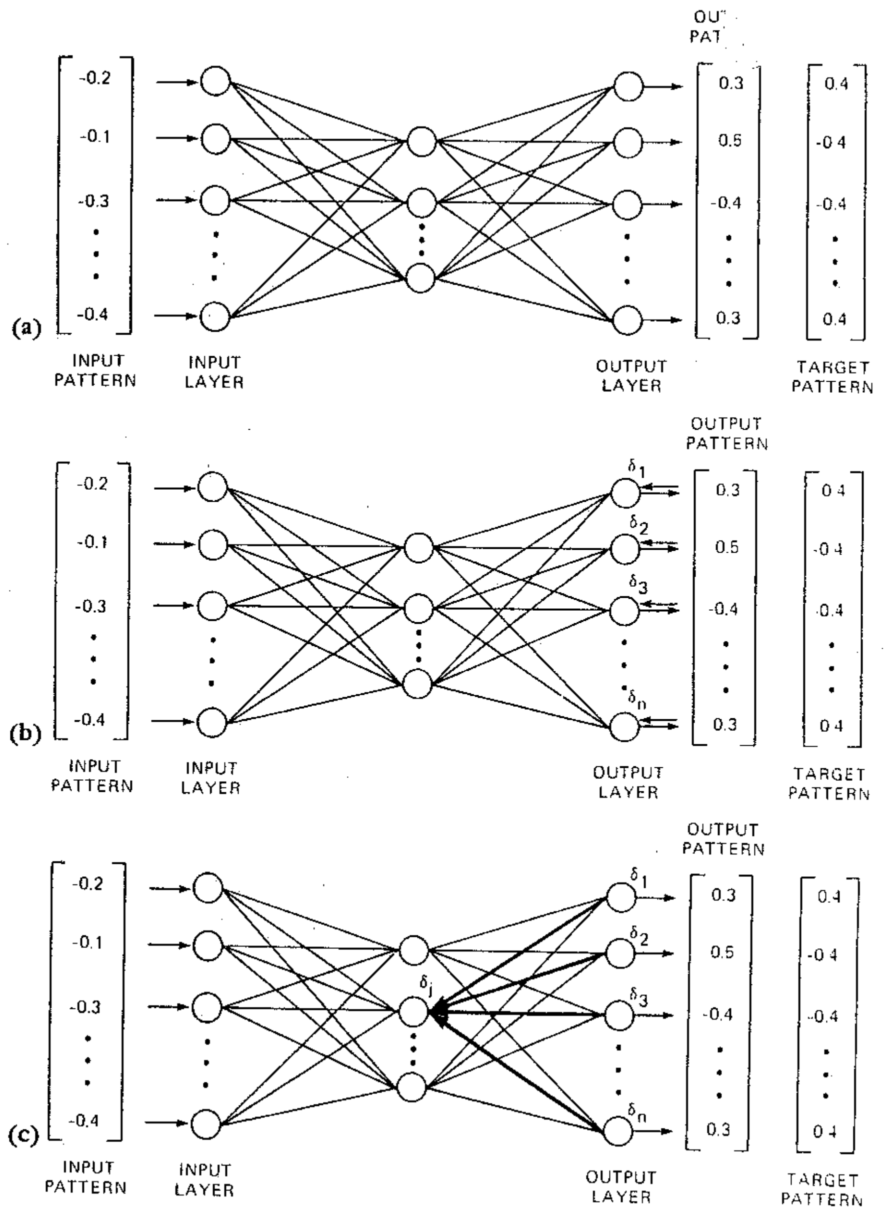


Figure 4-8. Basic back-propagation dynamics. (a) After forward propagation, the target pattern is compared to the output pattern. (b) δ values are calculated for the output layer. Arrows represent flow of information. After δ values are calculated for the output layer, its incoming weights are adjusted. (c) δ values are calculated for the hidden layer. Heavy lines indicate that δ values are communicated from the output layer to the hidden layer. After δ values are calculated for the hidden layer, its incoming weights are adjusted.

go into the output layer. Next an error value is calculated for all of the units in the hidden layer that is just below the output layer (Fig. 4-8c). Then the weights are adjusted for all interconnections that go into the hidden layer. The process is continued until the last layer of weights has been adjusted.

The error value, denoted by the variable δ, is simple to compute for the output layer and somewhat more complicated for the hidden layers. If unit j is in the output layer, then its error value is:

$$\delta_j = (t_j - a_j) f'(S_j) \qquad (4\text{-}3)$$

where

t_j = the target value for unit j
a_j = the output value for unit j
$f'(x)$ = the derivative of the sigmoid function f
S_j = weighted sum of inputs to j

The quantity $(t_j - a_j)$ reflects the amount of error. The f' part of the term "scales" the error to force a stronger correction when the sum S_j is near the

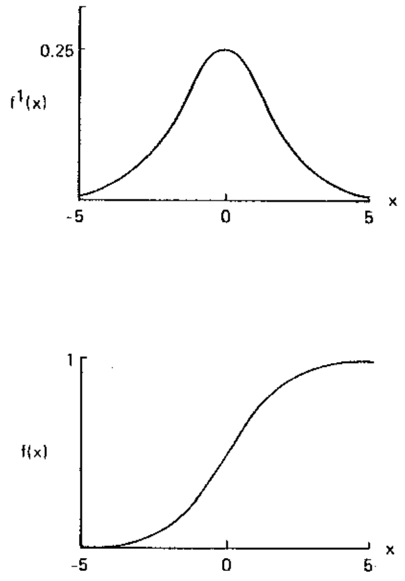


Figure 4-9. The sigmoid function (below) compared to its derivative (above).

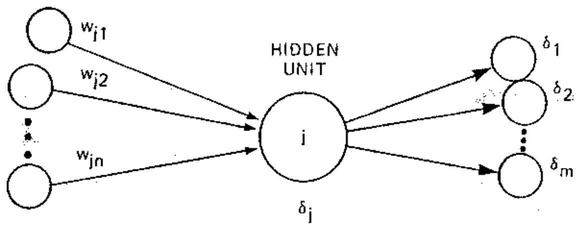


Figure 4-10. A processing unit in a hidden layer.

rapid rise in the sigmoid curve. Figure 4-9 illustrates the form of the f' function, with a peak in the same position as the rise in the sigmoid curve.

Figure 4-10 illustrates j as a unit in a hidden layer. In such a situation, the error value of j is computed as:

$$\delta_j = \left[\sum_k \delta_k w_{kj} \right] f'(S_j)$$

In this case, a weighted sum is taken of the δ values of all units that receive output from unit j. The f' again serves to "scale" this output by emphasizing the region of rapid rise of the sigmoid function.

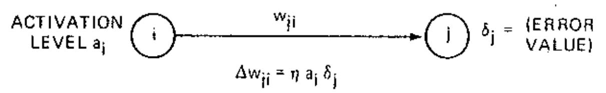
The adjustment of the connection weights is done using the δ values of the processing unit. Each interconnection weight is adjusted by taking into account the δ value of the unit that receives input from that interconnection. The connection weight adjustment is done as follows:

$$\Delta w_{ji} = \eta \delta_j a_i \qquad (4\text{-}4)$$

Figure 4-11 diagrams the adjustment of weight w_{ji}, which goes to unit j from unit i. The amount adjusted depends on three factors: δ_j, a_i, and η. This weight adjustment equation is known as the generalized δ rule (Rumelhart & McClelland 1986).

The size of the weight adjustment is proportional to δ_j, the error value of the target unit. Thus a larger error value for unit j results in larger adjustments to its incoming weights.

The weight adjustment is also proportional to a_i, the output value for the originating unit. If this output value is small, then the weight adjustment is small. If this output value is large, then the weight adjustment is large. Thus a



ACTIVATION LEVEL a_i
i
w_{ji}
$\Delta w_{ji} = \eta\, a_i\, \delta_j$
j
δ_j = (ERROR VALUE)

Figure 4-11. Updating a weight.

higher activation value for incoming unit *i* results in a larger adjustment to its outgoing weight.

The variable η in the weight-adjustment equation is the *learning rate.* Its value—commonly between 0.25 and 0.75—is chosen by the neural network user, and usually reflects the rate of learning of the network. Values that are very large can lead to instability in the network, and unsatisfactory learning. Values that are too small can lead to excessively slow learning. Sometimes the learning rate is varied in an attempt to produce more efficient learning of the network; for example, allowing the value of η to begin at a high value and to decrease during the learning session can sometimes produce better learning performance.

NETWORK TRAINING

Back-propagation networks are trained by a technique called *supervised learning,* whereby the network is presented with a series of pattern pairs—each pair consisting of an input pattern and a target output pattern. Each pattern is a vector of real numbers. The target output pattern is the desired response to the input pattern and is used to determine the error values in the network when the weights are adjusted.

The target output pattern is sometimes designed to represent a classification for the input pattern. In this way, the network may be presented with a series of input patterns together with the classification for each input pattern. In other applications, the target output is simply a desired pattern response to the input pattern, and the network is trained to be a pattern-mapping system.

Figure 4-12 shows an example training set that may be used for a back-propagation network. The input patterns are shown along with their corresponding target output patterns. Each pattern is a vector of three numbers. The neural network to be trained, shown at the bottom, has three processing units in the input layer and three processing units in the output layer.

The training task in this case is to learn to classify the three different types of graphs shown at the left. The target outputs require that a different output unit become active with each different type of graph. Thus, the target outputs reflect the correct classification for each input pattern.

The patterns in the training set are presented to the network repeatedly. Each training iteration consists of presenting each input/output pattern pair once. When all patterns in the training set have been presented, the training iteration is completed, and the next training iteration is begun. A typical back-propagation example might entail hundreds or thousands of training iterations.

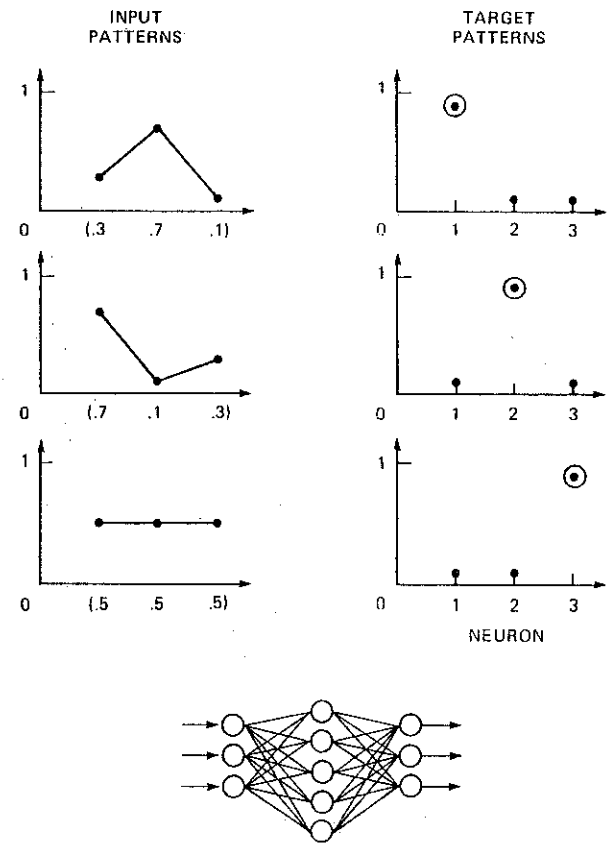


Figure 4-12. An example training set.

CONVERGENCE

When a network is trained successfully, it produces correct answers more and more often as the training session progresses. It is important, then, to have a quantitative measure of learning. The root-mean-squared (RMS) error is usually calculated to reflect the degree to which learning has taken place in the network [Eq. (2-5)]. This measure reflects how close the network is to getting the correct answers. As the network learns, its RMS error decreases (as in Figure 2-5). Generally, an RMS value below 0.1 indicates that a network has learned its training set.

Note that whether an answer is correct or not is a binary yes/no decision.

The target value to a network is a real number, and so is the output value. Thus the network does not provide a yes/no response that is either "correct" or "incorrect." The network gets closer and closer to the target value incrementally with each step. It is possible then to define a cutoff point when the network's output is said to match the target values, and allow this to define a "correct" answer.

Convergence is a process whereby the RMS value for the network gets closer and closer to 0. Convergence is not always easy to achieve because the process may take an exceedingly long time and sometimes the network gets stuck in a local minimum and stops learning altogether.

It is possible to represent convergence intuitively in terms of walking about on mountainous terrain. The terrain is the graph of the RMS value as a function of all of the weights in the network. Using this analogy, the back-propagation algorithm is seeking a minimum height in this mountainous terrain.

Ideally, we seek a global minimum — the bottom of the valley that is the lowest in the entire terrain. This corresponds to the lowest RMS value possible. Unfortunately, it is possible to encounter a local minimum — a valley that is not the lowest possible in the entire terrain. Nevertheless, a local minimum is surrounded by higher ground, and the network usually does not leave a local minimum by the standard back-propagation algorithm described. Special techniques should be used to get out of a local minimum.

The appearance of a local minimum is not always a significant problem. Back-propagation networks typically converge to a good RMS value when the training examples are clearly distinguishable. When a local minimum is encountered, the network may be able to avoid entering that local minimum by a number of techniques, for example, changing the learning parameter or the number of hidden units. These techniques tend to change the scenario involved with moving about on the "mountainous terrain" and may cause the network to avoid the local minimum.

Adding small random values to the weights allows the network to escape from a local minimum once it is encountered by moving the position of the network from a local minimum to a random point some distance away. If the new position is sufficiently removed from the valley of the local minimum, then convergence may proceed in a new direction without getting stuck in the same local minimum again. The amount of noise required depends on the local landscape, which is typically unknown to the investigator. Thus, there is some degree of luck involved in getting a network out of a local minimum.

The convergence process of back-error propagation is basically the same as the gradient-descent method, which derives from traditional statistical methodology. Intuitively, gradient descent can be visualized as a skier who is dropped at a randomly selected point on a mountainous terrain. The skier looks about and finds the direction of steepest descent, and then takes an incremental step in that direction. He repeats this process until he reaches a minimum in the terrain. A complete proof of the analogy between back-propagation and gradient descent is given in *Parallel Distributed Processing* (Rumelhart & McClelland 1986, Vol. 1, 323–4).

It should be noted that back-propagation provides more than just the gradient-descent optimization method. The back-propagation network, when successfully trained, finds a way of mapping an arbitrary set of input patterns to an arbitrary set of output patterns. This mapping is found without any knowledge of a mathematical function that may relate the output patterns to the input patterns. Traditional curve-fitting analysis techniques rely on *a priori* knowledge of the form of this mathematical function.

NETWORK TESTING AND PERFORMANCE

Typically an application of back-propagation requires both a training set and a test set. Both the training set and the test set contain input/output pattern pairs. While the training set is used to train the network, the test set is used to assess the performance of the network after training is complete. In a typical application both sets are taken from real data, although sometimes simulated data is used as well. If available data is scarce, then small amounts of noise may be added to the data to simulate additional patterns for the training or test sets. In any case, the training and test sets should use patterns typical of the type of data that the network is to encounter later. To provide the best test of network performance the test set should be different from the training set.

As an example, the network in Figure 4-12 classifies patterns with the same general shapes as those shown in the training set. Patterns with a peak in the middle are of type 1, patterns with a valley in the middle are of type 2, and flat patterns are of type 3. Both the training and test sets include patterns of these three general types, but the same pattern does not appear in both sets.

Figure 4-13 shows an example test set for the network given in Figure 4-12. The test set in this case had 28 input patterns, and included six noisy versions of each of the training patterns. Noise values of ± 0.1 were added to one of the values in each of the input patterns (shown as error bars). The network performed 100% successfully on these test patterns. Additional test patterns were made from horizontal lines of various heights. The network was able to classify the higher lines as type 3, but failed to classify the lower lines because the network failed to produce much activation in any output unit, and thus responded incorrectly.

To put this example in context, suppose that each pattern in the training set from Figure 4-12 is the sales record of a company for a three-month period. A neural network is built to classify three different types of sales patterns — a

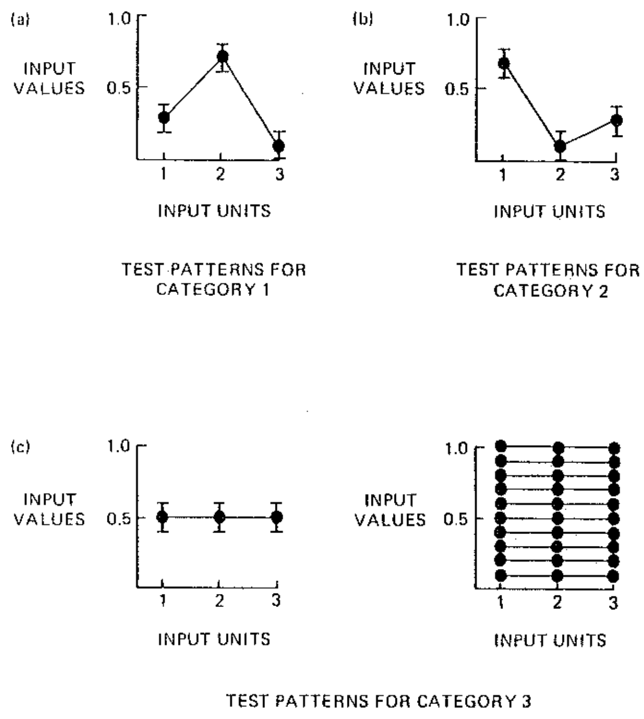


Figure 4-13. A test set for the training set in Figure 4-12.

peak, a valley, and a flat sales record. The neural network is to be used to input sales records and output the type of sales record. If performance is not satisfactory, another experiment may be done using a larger training set to train a second network. A training set that incorporates more of the variation present in the test examples might lead to better performance of the network. A series of similar experiments may be done to improve the network further. A realistic application of sales or financial assessments would probably require more input data and more inputs to the network, but the same general techniques can be employed to train the networks, evaluate their performance, and improve the training results.

FEATURE DETECTORS—AN EXAMPLE

To illustrate how feature detecting units can be developed by back-error propagation, we have constructed an example that is simple enough that the features organized by the network are obviously distinguishing characteristics for the pattern classes. The network to be trained (shown in Figure 4-14)

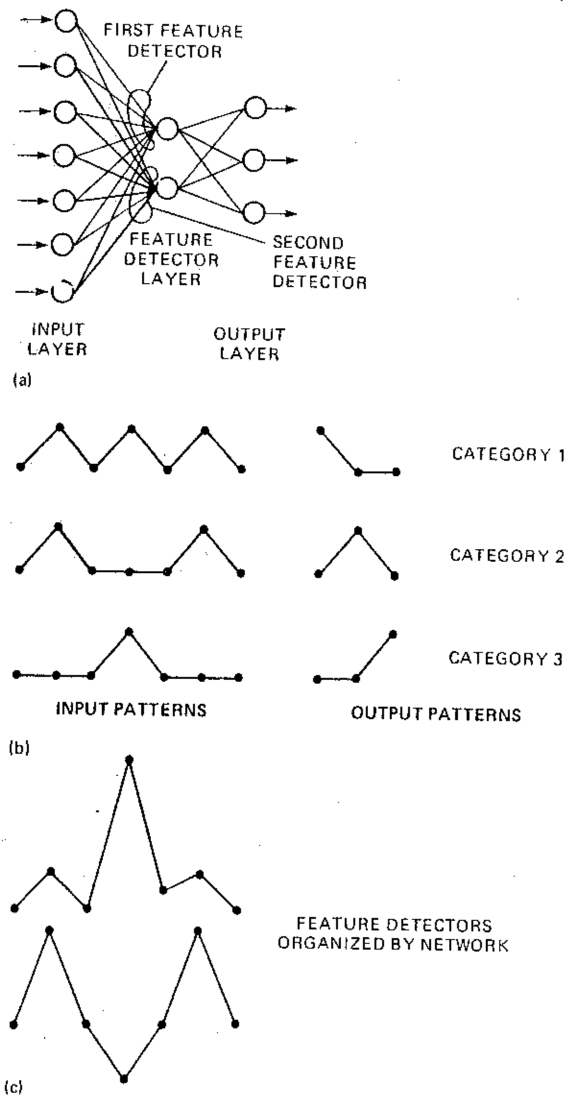


Figure 4-14. Example network training set, and feature detectors.

consists of three layers, with seven processing units in the input layer, two processing units in the hidden layer, and three processing units in the output layer. There are three patterns in the training set (also shown in Figure 4-14), which are to be classified into three categories.

From the combinations of peaks in the training set it is apparent that only two feature detectors are needed. The second and third patterns are combined in the first pattern. Thus, if feature detectors are organized to respond to the second and third patterns, then the first pattern can be identified when both feature detectors are activated.

The network in Figure 4-14 was trained with 460 presentations of the training set. The learning rate was set at 0.6, and the network converged to an RMS value of 0.06. The trained network was able to classify all training patterns correctly.

Features were found by reading weight values from the trained network, as shown at the bottom of Figure 4-14. The weights read were from the first layer of weights, for interconnections that originated at the input layer and terminated at the two hidden "feature-detector" units. Graphs of these weights reflect the features to which each of the hidden units responds. These features roughly match the contours of the second and third training patterns, providing distinguishing characteristics for all three patterns in the training set.

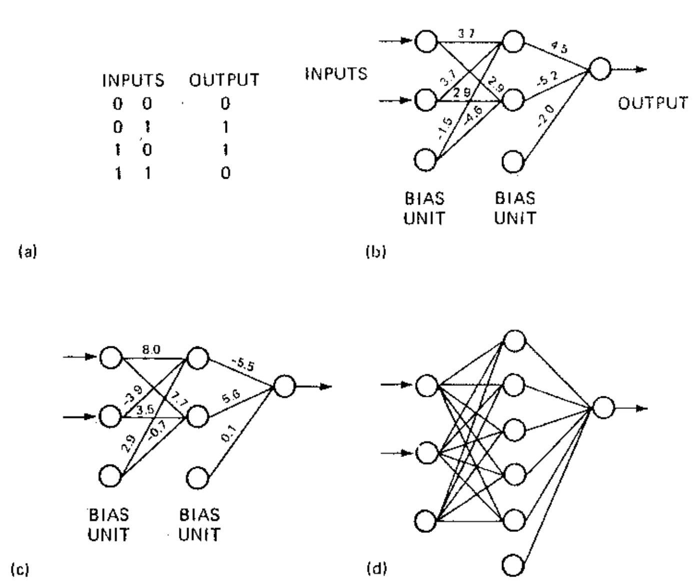
The top layer of the back-propagating network uses the response of the feature detectors in the hidden layer. Each unit in the top layer emphasizes each feature detector according to the value of the interconnecting weight. Thus, the connection from the first feature detector to the third output unit is strong, and the connection from the second feature detector to the second output unit is strong. Both feature detectors influence the first output unit, which stands for a pattern class that has both units' features.

This simple example shows feature detectors that were organized automatically by back-propagation and reflect the salient distinguishing features of the training-set patterns. Although the initial weight values were set to small random numbers, the final weight values "self-organized" to become features that could be used to perform the classification task. The network's only input for this self-organization task was the set of training patterns used during learning.

Although the features organized in this example appeared to the eye to be effective for the classification task, feature detectors do not always organize in a way that is obviously correct to a human observer. For example, in the previous section, the network was trained to classify patterns that looked like a peak, a valley, or a flat graph. The trained network organized feature detectors that appeared like a peak and a valley, but no feature detectors responded explicitly to the third training pattern, a flat graph. The response of the net to the flat graph came about by weighing in many features, none of which appeared flat.

THE EXCLUSIVE-OR: A CLASSIC PROBLEM

The exclusive-or (Xor) function is a classic example for the back-error paradigm. In this problem, a back-error propagating network is trained to perform



INPUTS		OUTPUT
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4-15. The exclusive-or problem: (a) Training set. (b) A two-layer exclusive-or network, trained. (c) An apparent local minimum for the exclusive-or problem. (d) An alternative exclusive-or problem network that is larger and gets caught in local minima less often.

the exclusive-or logic operation shown in Figure 4-15. The perceptron network was unable to solve this problem with a single output unit because the problem is not linearly separable, and its solution requires two layers of adjustable weights.

An Xor network has two inputs and one output. The training set for this problem is simply the Xor function shown in Figure 4-15a. The working network shown in Figure 4-15b is the smallest layered network that accomplishes the Xor function. It is possible to evaluate by hand the summation equations for each of the units to verify that the output is correct.

One of the common problems in doing the Xor problem with a standard back-error paradigm is the presence of local minima. Sometimes when the network in Figure 4-15b is trained, starting with small random weights, the network gets caught in a local minimum. In the case of an apparent local minimum (Figure 4-15c), the network evaluated the first and third entries in the training set correctly, but failed on the second and fourth. Application of more training iterations failed to get better convergence. Although the network adjusted its weights, it failed to converge on the correct answers for all of the patterns in the training set.

Figure 4-15d shows an alternative network topology that trains readily on the exclusive-or problem. This network topology has five units in its hidden layer as opposed to two, and tends to get into local minima less often.

STRENGTHS AND LIMITATIONS OF BACK-ERROR PROPAGATION

The principal strength of back-error propagation is its relatively general pattern-mapping capability; it can learn a tremendous variety of pattern-mapping relationships. It does not require any a priori knowledge of a mathematical function that maps the input patterns to the output patterns; back-propagation merely needs examples of the mapping to be learned. The flexibility of the paradigm is enhanced by the large number of design choices available—choices for the number of layers, interconnections, processing units, the learning constant, and data representations. As a result, back-error propagation might be able to address a broad spectrum of applications.

The largest drawback with back-error propagation appears to be its convergence time. Training sessions can require hundreds or thousands of iterations for relatively simple problems. Realistic applications may have thousands of examples in a training set, and it may take days of computing time (or more) to complete training. Usually this lengthy training needs to be done only during the development of the network, because most applications require a trained network and do not need on-line retraining of the net.

Back-error propagation is susceptible to training failures—the network never converges to a point where it has learned the training set. We have illustrated one such example with the exclusive-or problem, where the network becomes stuck in a local minimum. Additional training does not appear to improve this network.

A variety of special techniques have been developed in an attempt to decrease convergence time and to avoid local minima. A "momentum" term is sometimes used to speed convergence procedures (see Lippman 1987). Parker (1987) summarized a variety of additional techniques that can help the network adapt more efficiently. Improvements in convergence have also been found by varying the learning parameter η by starting with a larger value for η and progressing to smaller values. Techniques for avoiding local minima include changing the network or the training set, and adding noise to the weights. In spite of these improvements, applications developers utilize a variety of specialized accelerator boards, parallel processing machines, and other fast computers in training back-error propagation nets.

It should be noted that back-error propagation was not designed to model biological systems, and does not include many of the biological structures found at synapses and in nerve interconnections. Although biological systems have neurons that perform a type of summation of inputs, and have varying interconnection strengths, a scheme that does error-differencing and back-error propagation has not yet been identified in biological systems.

References

Lippmann, R. P. April 1987. An introduction to computing with neural nets. *IEEE ASSP Magazine,* pp. 4–22.

Parker, D. B. 1982. "Learning Logic," Invention Report S81-64, File 1, Office of Technology Licensing, Stanford University.

Parker, D. B. 1987 "Optimal Algorithms for Adaptive Networks: Second Order Back Propagation, Second Order Direct Propagation, and Second Order Hebbian Learning." Proc. of IEEE the First Intl Conference on Neural Networks. San Diego, CA.

Parker, D. B. 1987. Second order back propagation: Implementing an optimal $O(n)$ approximation to Newton's method as an artificial neural network.

Rumelhart, D. E. and J. L. McClelland. 1986. *Parallel Distributed Processing,* Vols. 1 & 2. Cambridge, Mass.: MIT Press.

Werbos, P. J. 1974. Beyond regression: New tools for prediction and analysis in the behavioral sciences. Thesis, Harvard University.

5 Back-Propagation Applications and Examples

Back-error propagation networks demonstrate a surprising capability for a very broad spectrum of applications, including image classification, speech synthesis, sonar return classifications, knowledge base systems, information encoding, and many other pattern classification and perceptual problems. In this chapter we illustrate the basic techniques used in applying back-error propagation by describing a series of applications studies and examples.

We begin with a description of NETTalk, the neural network designed by Sejnowski and Rosenberg to learn to read English text aloud. The NETTalk study addressed a number of important points that usually arise in applications of back-error propagation: choosing the number of layers, the number of hidden units, and the length of training. Also studied were issues of damage to the trained network and retraining after damage. This application also demonstrates progressive stages in training the network, and illustrates a distributed internal representation in the trained network.

The second example of back-propagation is taken from two-dimensional shape recognition. The actual problem described is a highly simplified classification task motivated by medical image processing. In this task, the network is trained to distinguish four different images by their shape and general visual pattern. A hidden layer of units organized a striking set of feature detectors that can be displayed as two-dimensional patterns. This example illustrates two important aspects of back-propagation: the presence of distinguishing characteristics in the feature detectors and redundancy in the trained network.

The third study described addresses the use of back-propagation for classifying patterns that are presented over time (i.e., not all the pattern is presented at the same time). The time-delay neural network (Waibel 1988;

Waibel et al 1989) employs time-delay components at each layer of the network to utilize a time history of each incoming data signal. In this way the network can be trained to identify patterns that arrive over a period of time. Here we describe how such a network is trained to recognize spoken syllables.

NETTALK

Perhaps the most striking neural network application is the network that learns to read aloud. NETTalk, developed by Sejnowski and Rosenberg (1987) at Johns Hopkins University, learns to translate segments of English text into phonetic notation for pronouncing the text. The phonetic notation that is output can be given to a speech generator and pronounced out loud automatically.

NETTalk is trained using actual examples of English text combined with the phonetic notation. Training NETTalk networks took 1 – 2 weeks of CPU time on a VAX; after training was complete, performance levels were over 90% correct. This performance level is very high considering that NETTalk was the first applications study of its kind. A realistic applications use, however, would require better performance. Future network designs may possibly enable us to attain these higher performance levels.

Figure 5-1 shows the network topology for NETTalk. The network consists of three fully interconnected layers. The input layer has 203 input units, the hidden layer 80, and the output layer 26. The bottom layer inputs English

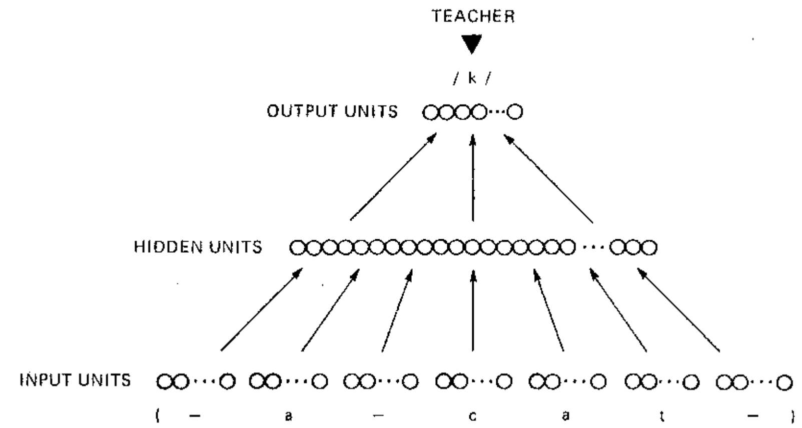


Figure 5-1. The NETTalk back-propagating neural network (from Sejnowski and Rosenberg, 1987. Parallel Networks That Learn to Pronounce English Text, *Complex Systems*).

text, and the top layer is presented with pronunciation notation for sounds to be pronounced from the input text. The NETTalk study illustrates the extent to which pronunciation rules can be mastered by a three-layer back-propagation net with only a single feature detection layer.

Because the English language requires context for recognition of letter pronunciation, an input window that centered on the letter to be pronounced but also included three letters to the right and left of the center letter was used. This input window of seven letters was stepped character-by-character through the English text. Each position was treated as an input pattern in the training set.

The 203 input units were divided into seven groups of 29 units each. Each of the seven groups represented a single character in the text stream and the seven groups together represented the seven-letter window. Each of the 29 units in a group was assigned to represent a single character: the 26 letters of the alphabet plus two punctuation indicators and a word boundary notation. A single unit was activated in each group, corresponding to the letter or character in the text stream. For example, if the first letter in the seven letter window was an A then the first unit in the first group was activated and the other 28 units in the first group were deactivated.

The 26 output units represent different phonetic notation symbols, for individual sounds or stresses: The 21 articulatory features were each represented by a single unit, and five additional units represented specific stresses and syllable boundaries. The output unit with the highest activation level indicated the phonetic symbol the network produced. This symbol was the network's response to the character at the center of the input window, and was presented to a sound generator for pronunciation.

The size of the hidden layer was chosen after experimentation with this back-propagation system. There is an inherent trade-off to be made — more hidden units result in more time required for each iteration of training; fewer hidden units result in a faster update rate but provide fewer feature detectors. NETTalk's hidden layer of 80 units was chosen as a result of learning experiments that indicated that 80 units were sufficient for good performance but not prohibitive in terms of training time.

The experiments done to arrive at the hidden layer size used a range of sizes. Some training experiments used up to 120 hidden units, some none at all. Performance was surprisingly high with no hidden units — up to about 80% correct pronunciation was found. When there were 120 hidden units, the network gave a greater than 90% correct performance. However, there was not much difference between 60 and 120 hidden units — apparently the first 60 feature detectors were the most important to maintain high performance levels.

The network was trained from a database of 1,000 words of transcribed

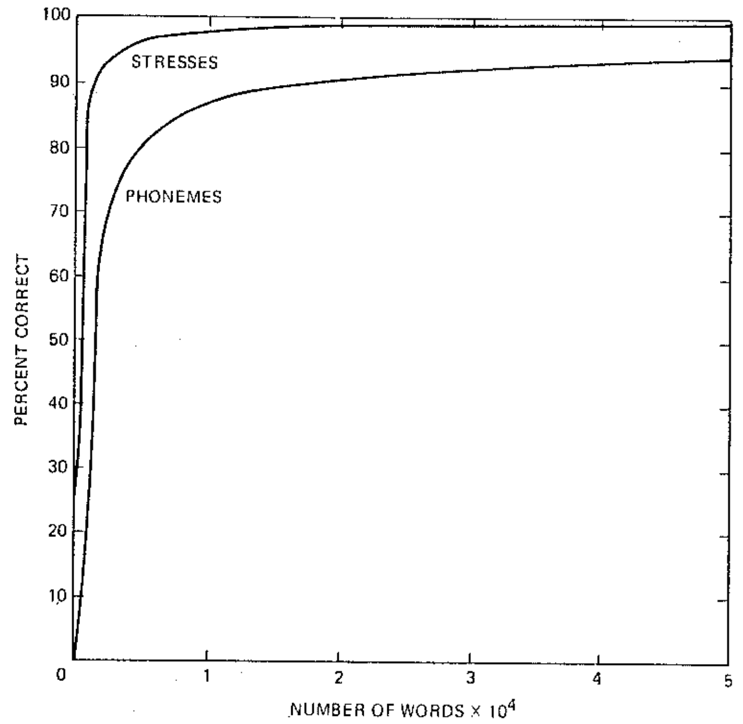


Figure 5-2. Performance of a NETTalk network as a function of the amount of training (from Sejnowski and Rosenberg, 1987. Parallel Networks That Learn to Pronounce English Text, *Complex Systems*).

speech (for details, see Sejnowski & Rosenberg 1987). Figure 5-2 shows the learning curve. The training set was presented up to 50 times to obtain these results. Output units that represent stresses are graphed separately from output units that represent sounds. The stresses were learned very quickly — the network had near perfect performance for stresses after five passes through the training set. Phonemes were learned more slowly — a 95% correct performance for phonemes required 50 presentations of the training set.

The network progressed through different stages during training. First it distinguished between vowels and consonants. At this stage the network predicted the same vowel for all vowels and the same consonant for all consonants. Then, at the next stage, the network recognized boundaries of words, and its pronunciation sounded like pseudowords. By 10 passes through the training set, most of the text was intelligible. These different stages were reminiscent of the way that children learn to speak. A further similarity between the network and people is seen in Figure 5-2, which shows that the

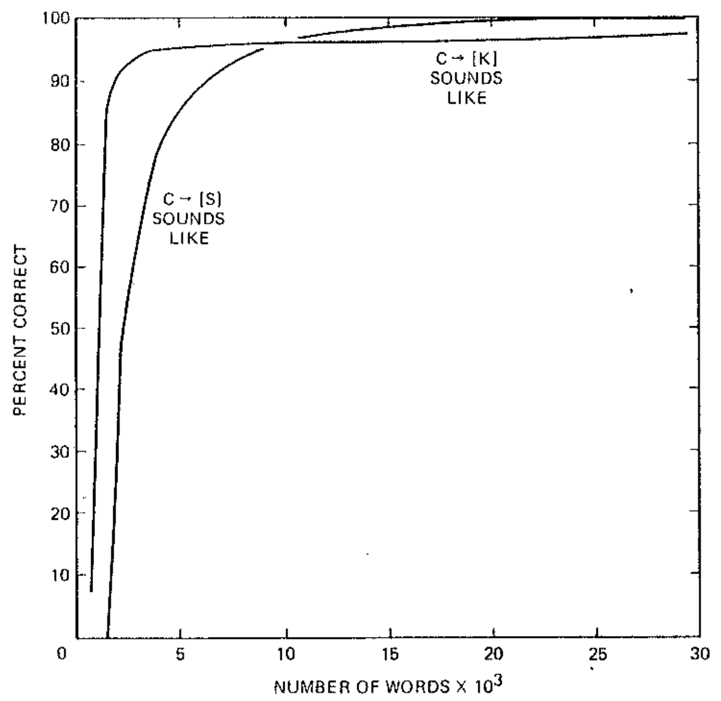


Figure 5-3. Performance of NETTalk on distinguishing between the *k* and *s* sounds for the letter *c*. Percent correct pronunciation is graphed as a function of the amount of training (from Sejnowski and Rosenberg, 1987. Parallel Networks That Learn to Pronounce English Text, *Complex Systems*).

network's learning follows a power law, which is characteristic of human skill learning.

Experiments with NETTalk indicated that there were some phonetic rules that the network never learned. Figure 5-3 shows such an example: Here, the network was required to distinguish between the hard and soft pronunciation of the letter *c*. Figure 5-3 shows the performance level (percent correct) as a function of how many words were presented during training. One curve shows pronunciation performance for words that have a *c* that is pronounced hard (as a *k* sound). The other curve shows pronunciation performance for words that have a *c* that is pronounced soft (as an *s* sound). If the network in Figure 5-1 is trained on more and more words, correct pronunciation of each type of *c* increases. However, a plateau is reached. More than 3,000 words in the training set does not increase the network's performance for pronouncing the letter *c* as *k*. The performance level here stays at about 95%.

Certain rules apparently are not learned by a three-layered back-propagation network. Increasing the number of examples in the training set does not cause the network to learn the distinction between soft and hard pronunciation of the letter *c* any better.

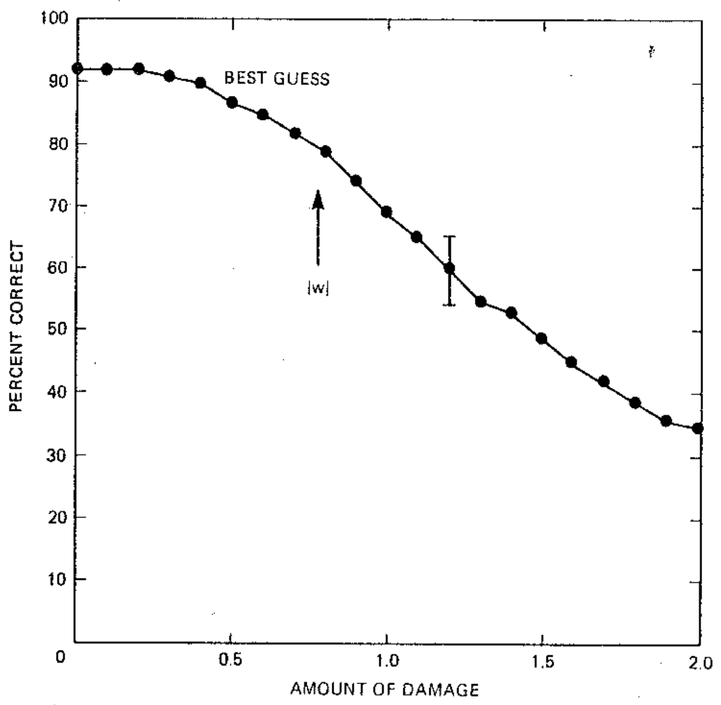


Figure 5-4. Performance of a trained NETTalk network as a function of the amount of damage done to the weights (from Sejnowski and Rosenberg, 1987. Parallel Networks That Learn to Pronounce English Text, *Complex Systems*).

Additional experiments were done on damage to the network. Damage was done to the network by adding a small random number to each of the weights. The random number was taken from a uniform distribution over intervals from 0.1 to 2.0. Performance of the resulting network was affected by the damage, and deteriorated as the amount of damage grew (see Figure 5-4). The performance was affected only slightly for small amounts of damage; performance dropped off more rapidly with larger amounts of damage. This relationship shows that network performance is resilient to small amounts of damage.

A further indication of the robustness of the network in the face of damage is shown in Figure 5-5, which shows that if the damage is not too severe, the network can recover readily with more training. The bottom trace shows the curve for original learning, giving performance (percent correct) as a function of the number of words presented during training. The top trace shows relearning after damage; damage in this case was done by adding random values between −1.2 and 1.2 to all weights. Relearning after damage occurred faster than the original learning.

Networks with four layers were also trained in the NETTalk study. One four-layered network used the input and output layers described previously in addition to two hidden layers with 80 units each. Performance was then compared to the three-layered network with 120 hidden units; the total number of weights in these two networks differed by only 10%. Both networks did about the same in absolute performance. The four-layer network, however, was able to generalize better — it pronounced new words better than the three-layer network with 120 hidden units.

The NETTalk network organized feature detectors that tended to respond to particular sounds or groups of sounds. The patterns that individual feature detectors responded to were not obvious in many cases; however, the hidden layer taken as a whole had a distributed representation of the key features of the data. An interesting point about this organization arose when the same network was trained twice on the same data, each time starting at different randomized weight values. The resulting networks could have the same performance and learning characteristics on a particular task, but differ completely at the levels of synaptic strengths and single-unit responses. Hence, the details of the feature detectors differed even though the performance of the networks were the same.

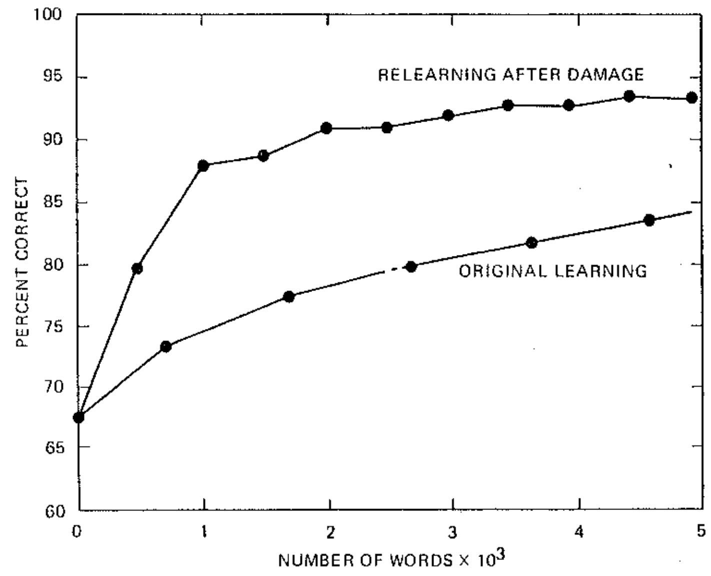


Figure 5-5. A comparison of original learning with relearning after damage. Performance is shown as a function of the amount of training (from Sejnowski and Rosenberg, 1987. Parallel Networks That Learn to Pronounce English Text, *Complex Systems*).

NETTalk demonstrates that most of the expertise needed to read English text aloud can be learned by a three-layered back-propagation system. It is humbling to observe that such a simple network can actually learn a task that is so fundamental to human existence. The NETTalk system also illustrates a number of important considerations in back-propagation studies. Experiments dealt with hidden layer sizes, training sets, learning curves, network damage, relearning after damage, and the particular organization of the feature detectors. These issues arise in other back-propagation applications as well.

TWO-DIMENSIONAL FEATURE ORGANIZATION

Systems that can identify and classify two-dimensional images based on their shapes and other visual characteristics are very much needed in applications such as the identification of handwritten numerals (e.g., on checks), reading of handwritten characters, sorting of parts in industrial production, automated defect inspection, and medical image processing. Although these applications are not expected to be solved entirely by back-propagating systems, neural networks are likely to be a key component in their solution.

Two-dimensional patterns provide us with a visually intuitive way to show how back-propagating systems work internally. The feature organization can readily be seen by eye in a two-dimensional image classification task. Here we show a simple example in which a back-propagating neural network (Wasserman 1988)[1] organizes a set of two-dimensional features when it learns to classify a collection of different shapes (Dayhoff & Dayhoff 1988). This shape classification problem was motivated by medical image classification tasks that have the potential to benefit from the development of automated classification systems.

Figure 5-6 diagrams the three-layered network used in this study. The layers are fully interconnected. Both the input and output layers are arranged as two-dimensional grids (7 × 9), and the hidden layer in this case has 16 processing units.

Figure 5-7 shows the training set adapted from a medical image processing task. A coarse-grid representation of four images was made to simplify the

[1] Thanks to Phil Wasserman for making his software available.

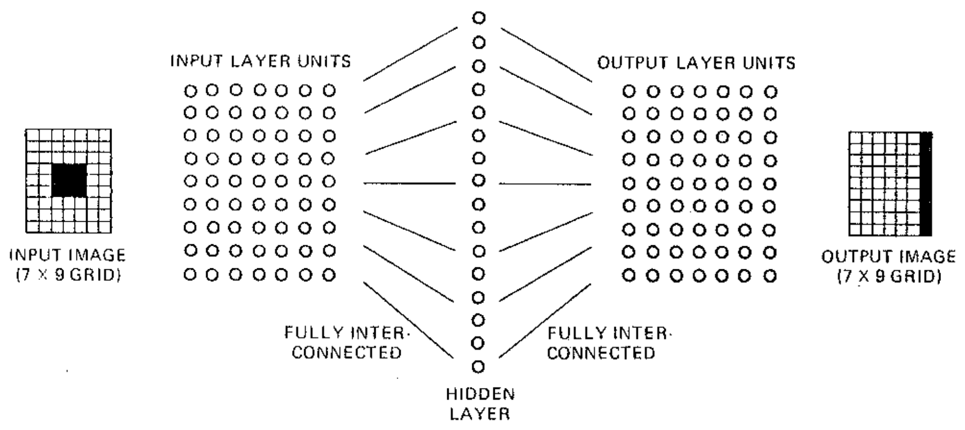


Figure 5-6. A three-layered back-propagation network used to perform a simplified medical image classification task (from Dayhoff and Dayhoff, 1988, © IEEE).

pattern recognition task. The four images were based on the shapes of different one-celled organisms. The organisms were amoeboid cells, distinguishable by the thickness of their membrane and the size and organization of their nucleus. The coarse-grid representation is shown in Figure 5-7 along with the original images of the amoebae. (Note that the training set is a considerable simplification of real-life images.)

The network in this study was trained to output a grid with a vertical slab of activity. The slab is to appear at a different place based on the network's classification of cell type. Four different positions are shown, corresponding to the four different cell types. These are the target outputs of the neural network.

The training set was presented to the network 3,000 times. The resulting network was able to classify all pictures in the training set, and in addition showed good performance on a test set with noisy pictures.

The feature detectors that were organized by the network during training are shown in Figure 5-8a. Figure 5-8b illustrates how these feature detectors were drawn based on the trained network. Each feature detector corresponds to a single unit in the hidden layer of the network. The incoming connection weights for each hidden unit make up a 7 X 9 grid. The values of these weights were quantized and drawn in a grid. Black boxes represent inhibitory connections, striped boxes represent positive connections, and cross-hatched boxes were strongly positive. White boxes represent weights with very low absolute values. Feature detectors developed emphasized the following areas of the images: outer edge, inner edge, central square, four corners of central square, "+" inside central square, and background.

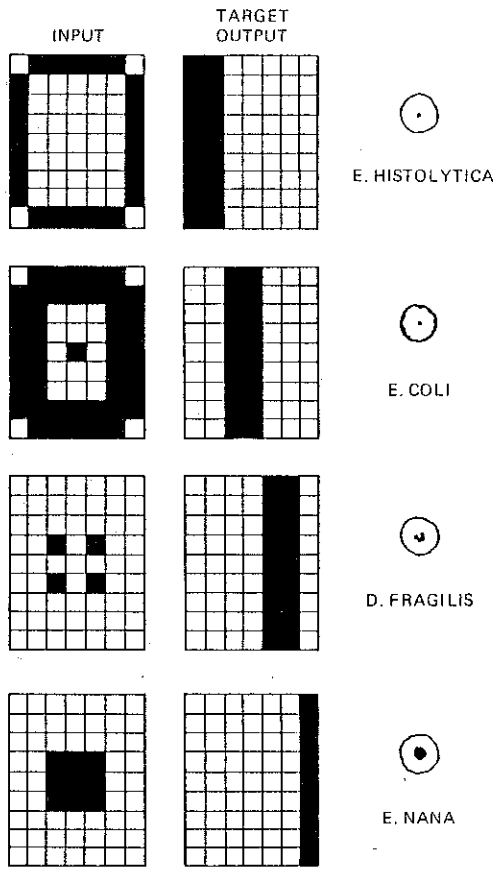


Figure 5-7. Four different types of ameboid cells with their coarse-grid representations. The target output patterns for the network are also shown (from Dayhoff and Dayhoff, 1988, © IEEE).

Each feature detector illustrates the incoming pattern that the hidden unit responds to best. Thus, an input pattern that is exactly the same as a given feature detector will activate the corresponding feature-detecting unit best. If the input pattern matches a subset of the feature, then the corresponding unit will be activated but not at maximum activation. An input pattern that does not match the feature will not activate the corresponding hidden unit significantly.

The feature detectors that were organized (Figure 5-8a) are similar to fragments of the images in the training set. It is possible to pick out by eye some of the features that are distinguishing characteristics of two or more images. For example, the presence of a thin cell boundary appears in features

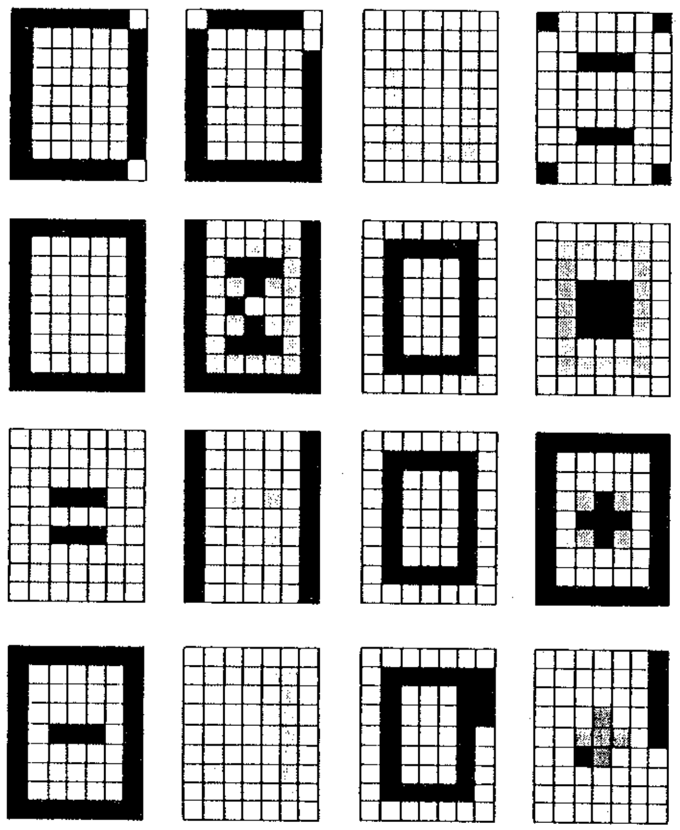


Figure 5-8. (a) Sixteen feature detectors organized by the network shown in Figure 5-6 to solve the classification task shown in Figure 5-7 (from Dayhoff and Dayhoff, 1988, © IEEE).

1-2, 5-6, and 12-13. An inner cell boundary elicits a response from features 7, 11, and 15. The nuclei in the last two training patterns are distinguished by the + pattern in feature 16, and can be partially distinguished by the pattern of four dots in features 1, 10, and 13.

The features organized in this example are highly redundant—some are almost exactly the same (e.g., 7 and 11). Many pairs of feature detectors contain similar components. This redundancy is a natural part of back-error propagation training because redundancy is organized automatically when an excess of hidden units is supplied. Furthermore, the redundancy is usually distributed throughout the hidden unit layer, as in Figure 5-8a. Redundancy in feature detectors contributes to the capabilities of the network by helping to make the network performance resilient to errors or failures in individual units or weights.

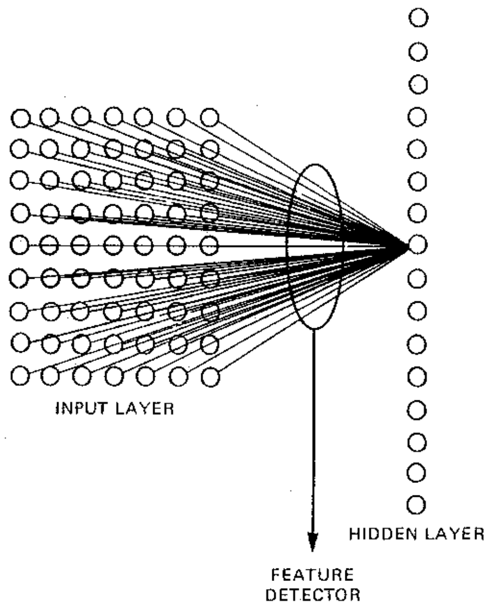


Figure 5-8. (b) Reading a feature detector from the trained network.

If the applications goal is to include redundancy in the system, the redundancy makes the system more tolerant to damage of individual units or weights. Computing systems that are tolerant to damage of components are needed, especially in applications of military or medical systems. If the applications goal is to construct the smallest system possible, then the number of hidden units can be decreased and performance would continue to be good. The example shown here can be decreased to 12 hidden units without significant degradation of performance.

The way that these feature detectors are used by the output layer is specified by the second layer of weights. Weights from the hidden layer to the output layer indicate how strongly each feature is weighed by the network. Feature 16, for example, is strongly weighted in its connection to the third and fourth slab of output units, and is used to distinguish these images.

THE TIME-DELAY NEURAL NETWORK

One of the chief limitations of traditional neural network architectures is in dealing with patterns that may vary over time and that require a period of time to be presented. Standard neural network architectures such as back-error propagation require an input pattern to be presented all at the same time and have no explicit architecture built into the network to deal with patterns such as speech, which are presented in parts over a period of time, with a lag between when the pattern begins and when it ends. A spoken word is presented to the listener over a period of time, and a spoken sentence is presented over a longer period of time. The human nervous system can recognized these patterns almost immediately even though they are not presented all at once.

The basic architecture of the time-delay neural network (TDNN), as described by Waibel and colleagues (Waibel 1988; Waibel et al 1989) has a variety of time-delay components built into its structure so that it can deal with temporally presented patterns. Back-error propagation is used as the learning algorithm. Considerable success has been obtained with this network in the area of speech-recognition studies.

Figure 5-9 shows the basic unit of the TDNN. A series of input channels

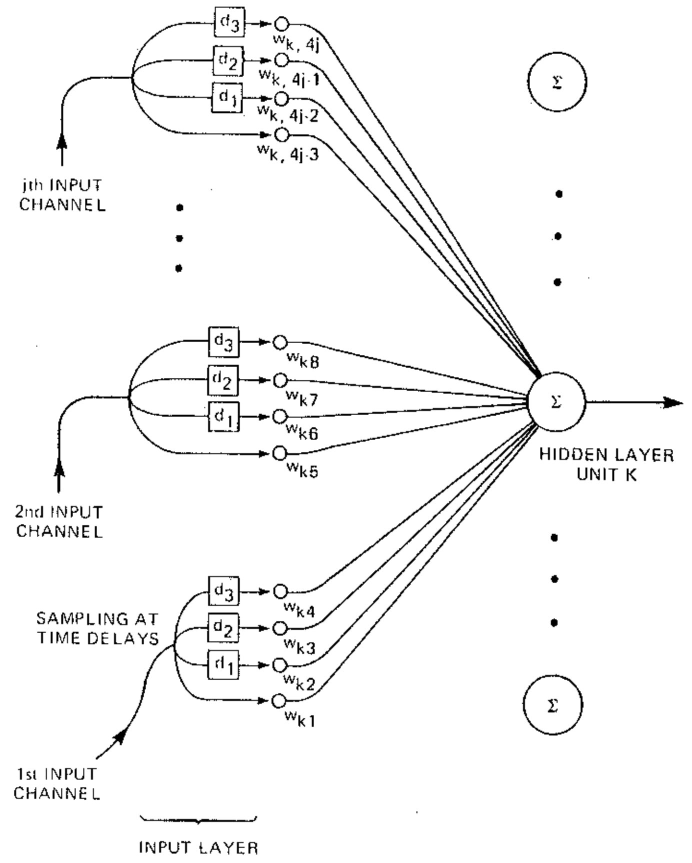


Figure 5-9. A processing unit from the time-delay neural network.

provides signals for the network. Each channel is sampled over a period of time, and past values are input to the network after delays. The bottom input for each channel is the present value of the channel signal; the other inputs are previous values, each delayed by a fixed amount. In principle, one can include any number of channels, any number of time delays, and any collection of time delay values in such a unit.

The input units and delays are considered the first layer of the network. The second layer is a hidden layer, and the unit shown at right of Figure 5-9 is a hidden layer unit. This unit produces a sequence of outputs, which are stored over a period of time. Then the present output together with delayed outputs from the past are fed into the next layer. The TDNN can employ two or more layers of time-delay units like the one in Figure 5-9.

Each interconnection of the network has an associated weight. The weight is adjusted according to the standard back-error propagation paradigm. Temporal relationships, then, become encoded in the network by means of the values of weights associated with the different time delays. For example, suppose that the weight for a particular interconnection has an unusually high positive value. Observe the time delay associated with the unit where the interconnection originates—it is emphasized by the high weight value and has more influence on the output of the network than other time delays.

In a series of initial experiments done with a time-delay neural network, a four-layer network was trained to recognize and classify a set of spoken syllables. The incoming speech was split into 16 separate frequency bands and each band was assigned to a separate input channel. The input signal for each channel was the amplitude for a narrow band of frequency as a function of time. Three input units were assigned to each channel. The first unit received the input signal itself and the other two units received the input signal delayed in time, with a different time delay for each unit.

The trained network was able to classify three different consonant sounds: *D, B,* and *G.* A single unit in the output layer was taught to respond to each consonant. The neural network was trained to recognize the consonant sounds regardless of the vowel sound that followed, and a variety of different vowel sounds were used.

Figure 5-10a shows the response of the trained network to the activation from the syllable *DA*. For comparison, Figure 5-10b shows the activation pattern resulting from input of the syllable *DO*. Although the activation patterns in the first and second layers differ considerably, the third layer has a similar activation pattern for both. These cause the output layer of the network to respond with the classification *D*.

The TDNN performed successfully, and its performance was better than a leading nonneural network approach, which utilizes hidden Markov models. Furthermore, the researchers have continued their work to obtain even better

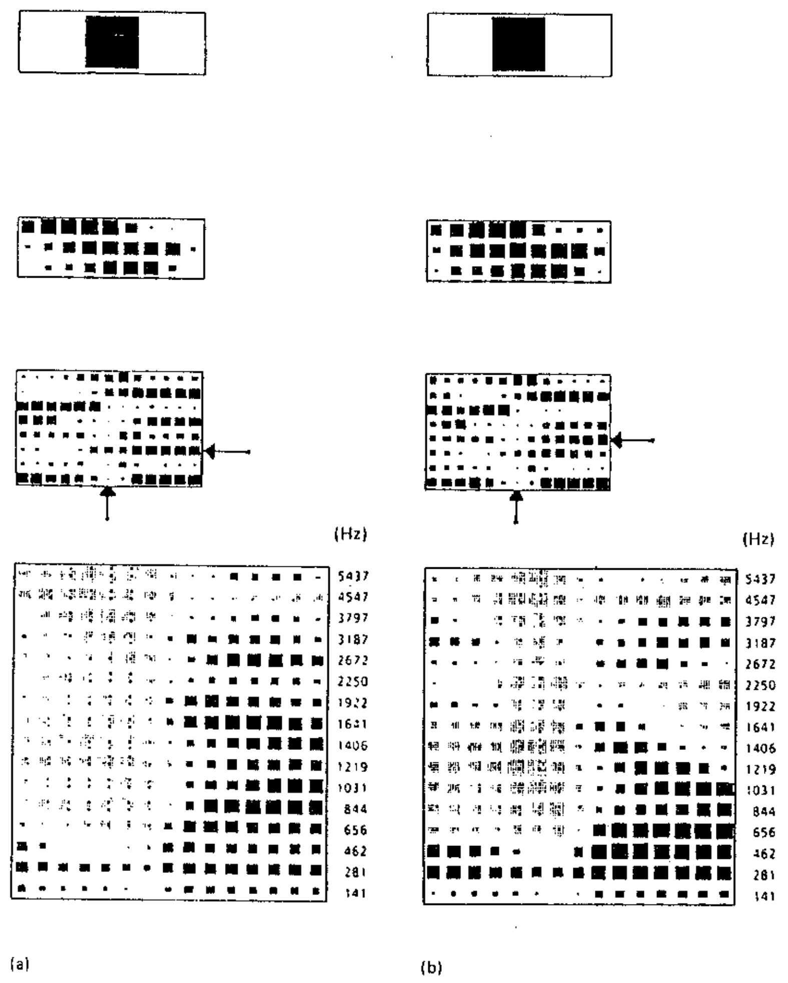


Figure 5-10. (a) TDNN activation patterns in response to *DA* (b) TDNN activation patterns in response to *DO* (from Waibel et al. Phoneme recognition using time delay neural networks, © 1989, IEEE *Trans. on Acoustics, Speech, and Signal Processing*).

performance with a series of similar subnets that divide the classification task (Waibel 1988). These research studies have made striking progress toward designing a system that can dynamically recognize speech, a very difficult task.

References

Sejnowski, T. and C. Rosenberg. 1987. Parallel networks that learn to pronounce English text. *Complex Systems* 1(1):145–68.

Dayhoff, R. E. and J. E. Dayhoff. 1988. Neural networks for medical image processing. IEEE SCAMC Proceedings, pp. 271–75.

Waibel, A.; T. Hanazawa, G. Hinton, K. Schikano, and K. Lang. 1989. Phoneme recognition using time delay neural networks. *IEEE Trans. on Acoustics, Speech and Signal Processing* 37, March 1989.

Waibel, A. 1988. Consonant recognition by modular construction of large phonemic time-delay neural networks. Neural Information Processing Systems, Morgan Kaufman Publishers, San Mateo, Calif., pp. 215–23.

3

Hopfield Networks

In a breakthrough paper published in 1982, John Hopfield introduced the network architecture that has come to be known as the Hopfield Net. In clear and simple terms he described how computational capabilities can be built from networks of neuronlike components. He illustrated an associative memory that can be implemented with his network, and later demonstrated optimization problems that could be solved. The appearance of the Hopfield network renewed interest in previous research results and precipitated a rebirth of enthusiasm for neural networks. (Fukushima, Miyake, & Ito 1983; Grossberg 1987, 1988; Reilly 1982, Anderson, 1972; Anderson et al 1977).

THE BINARY HOPFIELD NETWORK

Basic Structure

The binary Hopfield Net has a single layer of processing units. Each processing unit has an activity value, or "state" that is binary — one of two possible values. Here we use the binary states 0 and 1 (the network works the same way if values of $+1$ and -1 are used, but slight changes in the equations are required).

The entire network is considered to have a "state" at each moment. The state is a vector of 0s and 1s. Each entry in the vector corresponds to an individual processing unit in the network. Thus, at any given moment, the state of the network is represented by a state vector such as:

$$U = (u_1, u_2, \ldots, u_n) = (++\cdots\cdot+\cdots+)$$

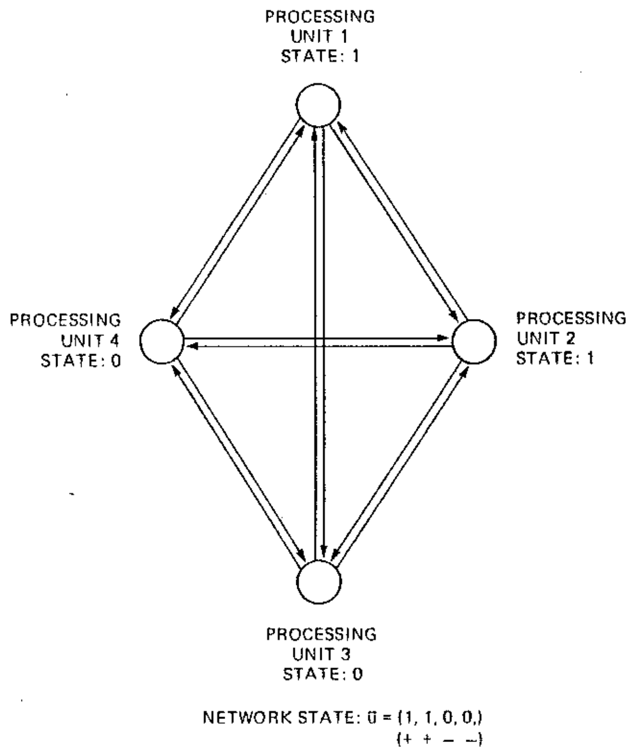

PROCESSING UNIT 1 STATE: 1
PROCESSING UNIT 2 STATE: 1
PROCESSING UNIT 3 STATE: 0
PROCESSING UNIT 4 STATE: 0
NETWORK STATE: $\vec{u}$ = (1, 1, 0, 0,)
(+ + – –)

Figure 3-1. A binary Hopfield network.

This vector reflects a network of *n* processing units, where element *i* has state u_i. In this notation, a + represents a processing unit with the binary value 1, and a – represents a processing unit with the value 0. Figure 3-1 shows a diagram of the processing units in a Hopfield network, together with an example state. The state of the network can change over time as the values of individual units change.

The processing units in the Hopfield network are *fully interconnected*—each unit is connected to every other unit. In fact, the connections are "directed," and every pair of processing units has a connection in each direction (see Figure 3-2). This interconnection topology makes the network "recursive" because the outputs of each unit feed into inputs of other units in the same layer. As we shall see, this recursive organization will allow the network to relax into a stable state in the absence of external input.

Each interconnection has an associated weight. This weight is a scalar value, considered intuitively to be the *connection strength*. We let T_{ji} denote the weight to unit *j* from unit *i*. In the Hopfield network, the weights T_{ji} and T_{ij} have the same value, therefore

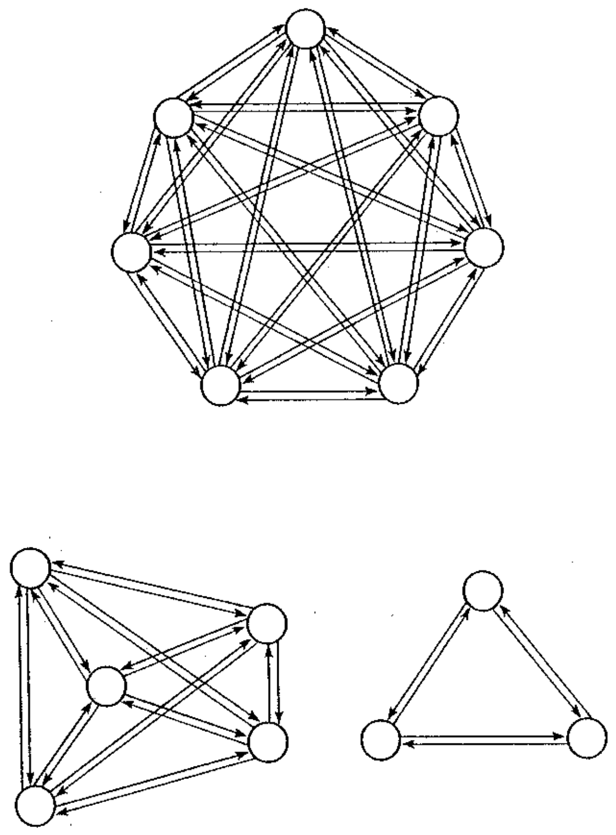


Figure 3-2. Fully interconnected one-layer networks, with connections in both directions between each pair of processing units.

$$T_{ji} = T_{ij}$$

Mathematical analysis has shown that when this equality is true, the network is able to converge—that is, it eventually attains a stable state. Convergence of the network is necessary in order for it to perform useful computational tasks such as optimization and associative memory. Many networks with unequal weights ($T_{ji} \neq T_{ij}$) also converge successfully.

Figure 3-3 shows an alternative method of diagramming the interconnections: The processing units appear in a row and the interconnections form a

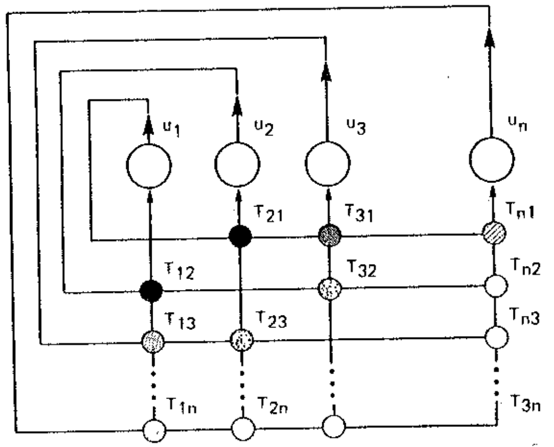


Figure 3-3. A Hopfield network, with weight values shown as circles. Shading of the circles represents the values of the weights.

gridwork. Note that with n units, there are $n(n\text{-}1)$ interconnections. Each connection weight is shown at an intersection point in the grid. The size of the weight is depicted by the shade of the circle drawn at that intersection. The weight from node i to j is shown below node j, where the line below node j intersects the line that originates at node i.

Connection weights are set at the beginning of an application. The method of setting the weights depends on the application. Two different applications will be described: the associative memory and the traveling salesman problem.

The Updating Procedure

Initially, the network is assigned a state for each processing unit. An updating procedure is applied to the units in the Hopfield network. One unit at a time is updated. The updating procedure affects the state of each unit, sometimes changing it and sometimes leaving it the same. The updating of processing units continues until no more changes can be made.

Figure 3-4 illustrates the basic processing done by a binary Hopfield Net unit during the updating procedure. Each neuron takes a weighted sum of its inputs, according to the following equation:

$$S_j = \sum_{\substack{i=1 \\ i \neq j}}^{n} u_i T_{ji} \qquad (3\text{-}1)$$

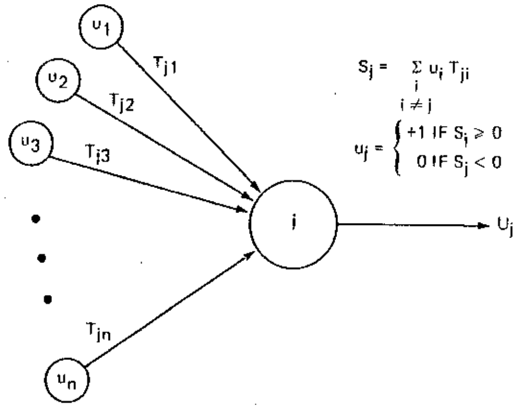

u_1
u_2
u_3
u_n
T_{j1}
T_{j2}
T_{j3}
T_{jn}
i
U_j
$S_j = \sum_{i, i \neq j} u_i T_{ji}$
$u_j = \begin{cases} +1 \text{ IF } S_j \geq 0 \\ 0 \text{ IF } S_j < 0 \end{cases}$

Figure 3-4. Binary Hopfield processing unit.

When this sum is calculated, the processing unit then evaluates whether the sum is greater or smaller than 0. If the sum is at least 0, then the output of the unit is set at +1. If the sum is less than 0, the output of the unit is set at 0. In mathematical terms,

$$\begin{array}{ll} \text{if } S_j \geq 0 & \text{then } u_j = 1 \\ \text{if } S_j < 0 & \text{then } u_j = 0 \end{array} \qquad (3\text{-}2)$$

Suppose that processing unit j is to be updated, as shown in Figure 3-5. The weighted sum of inputs is calculated for element j, according to Eq. (3-1). Then the rule in (3-2) is applied. The result is that unit j has the value of 0 (top of Figure 3-5). In the middle frame of Figure 3-5, another processing unit (top left) is updated, and changes its state from 1 to 0. Next, unit j is updated (bottom of Figure 3-5). The weighted sum of its inputs is now 0.5, which is positive and hence changes the state of unit j to 1.

The previous value of a processing unit is not taken into account when it is updated. The unit may or may not change its value due to the updating procedure. The example in Figure 3-5 shows a processing element that changed values through the updating procedure. Its change was caused by a prior change in the state of the unit at the top left.

One system for the updating process is to update the units in sequence, then repeat the sequence until a stable state is attained. The update mechanism posed by Hopfield is not as simple — it chooses the next unit to be updated at random, which allows all units to have the same average update rate. Eventually the network reaches a stable state: All units retain the same value upon updating. Random updating has advantages both in implementation (each unit can generate its next update time) and in function (sequential updating

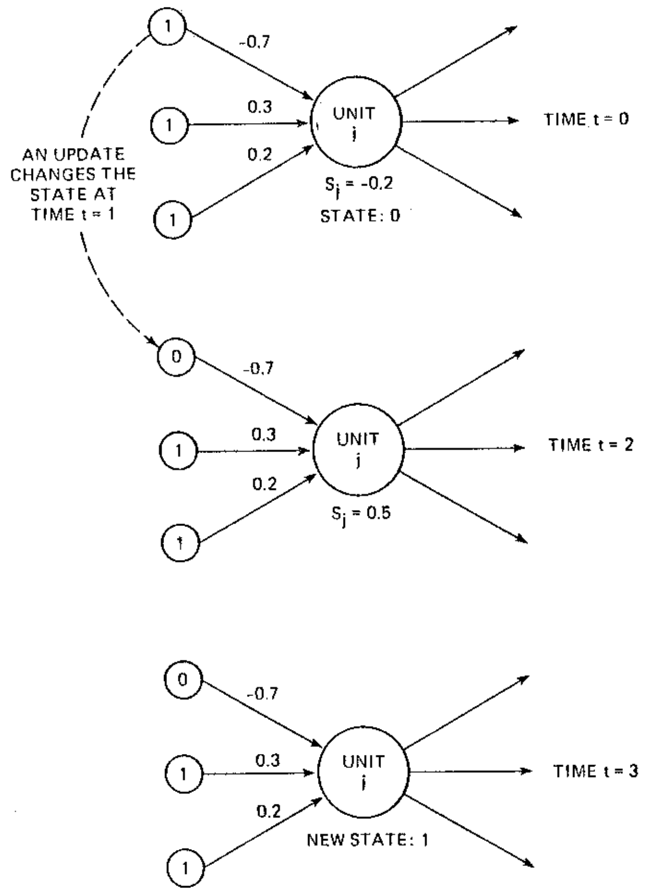


Figure 3-5. State of processing unit j at three successive times ($t = 0,1,2,3$). A change in state of the unit at top left results in a change in state for unit j.

can restrict the output states of the network in cases in which different stable states are equiprobable).

Usually, many updates must be done to all of the processing units before the network reaches a stable state. Hopfield (1982) showed that his network does eventually reach such a stable state.

The randomized updates used in the Hopfield network provide an important difference between it and other paradigms. Most other neural network paradigms have a layer of processing units updated at the same time (or nearly the same time). In contrast, the asynchronous updating of the Hopfield Net is a closer match to biological reality—biological neurons update their own states due to events that impinge upon the neuron. These impinging events are not synchr[illegible]ized from neuron to neuron.

Convergence

Each state of the Hopfield network has an associated "energy" value. This value is defined by:

$$E = -\frac{1}{2} \sum_{j} \sum_{\substack{i \\ j \neq i}} T_{ji}\, u_j\, u_i \tag{3-3}$$

The equation is referred to as "energy," although it does not represent the real energy of any physical system. The energy function in Eq. (3-3) is an objective function that is minimized by the network.

The successive updating of the Hopfield network provides a "convergence" procedure whereby the energy of the overall network gets smaller and smaller. Eventually the network goes into a stable state; at this stable state, the energy is at a minimum. This minimum may be local or global.

It is possible to prove that each time a processing unit is updated, the energy of the network either stays the same or decreases. As a result, this updating procedure will always allow the energy of the network to converge to a minimum.

There is also an argument that the updating procedure either decreases the energy or leaves it the same. Suppose that unit j is the next processing unit to be updated. Then, the portion of E affected by processing unit j is given by:

$$E_j = -\frac{1}{2} \sum_{\substack{i \\ i \neq j}} T_{ji}\, u_j\, u_i \tag{3-4}$$

which rearranges to

$$E_j = -\frac{1}{2}\, u_j \sum_{\substack{i \\ i \neq j}} T_{ji}\, u_i \tag{3-5}$$

When unit j is updated, if there is no change in its state, then the energy E_j remains the same. If there is a change in its state, then the difference in E_j is:

$$\Delta E_j = E_{j_{new}} - E_{j_{old}} = -\frac{1}{2}\, \Delta u_j \sum T_{ji}\, u_i \tag{3-6}$$

where

$$\Delta u_j = u_{j_{new}} - u_{j_{old}}$$

If u_j changes from 0 to 1, then

$$\Delta u_j = 1$$

and

$$\sum_i T_{ji} u_i \geq 0$$

after updating, according to Eq. (3-2).

Plugging these nonnegative values into Eq. (3-6), we get

$$\Delta E_j \leq 0$$

If u_j changes from 1 to 0, then

$$\Delta u_j = -1$$

and

$$\sum_i T_{ji} u_i < 0$$

after updating, by (3-2). Plugging these two negative values into (3-6), we get

$$\Delta E_j < 0$$

Since ΔE_j is the product of three negative numbers. Thus, the change in E is always negative or 0 no matter what change there is in the state of unit j upon updating. The network is guaranteed to converge, with E taking on lower and lower values until the network reaches a steady state.

Seeking a minimum of the energy function is analogous to seeking a minimum in a mountainous terrain. Figure 3-6 depicts a two-dimensional version of such terrain. The energy function value is reflected in the height of the graph. Each position on the terrain corresponds to a possible state of the network, and the network moves toward a minimum position in this graph. Two local minima and a global minimum are depicted. If the state of the network is changed, then a corresponding change is made in the x coordinate position on the graph. This change in turn results in a movement downhill, toward one of the minima.

The initial state of the network may be thought of as the position of a skier who has been dropped randomly onto mountainous terrain. The updating procedure moves the skier downhill until he gets to the bottom of the most accessible valley. (This valley may be at a local or a global minimum.)

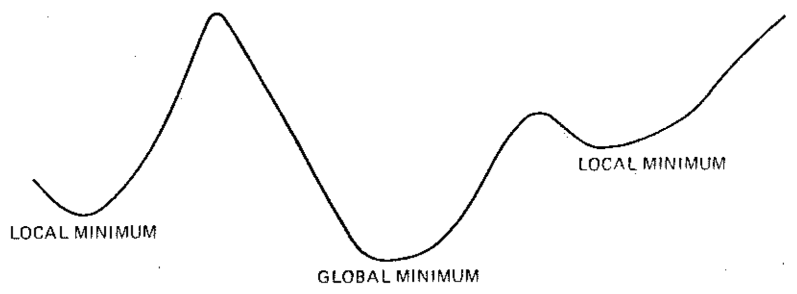


Figure 3-6. An energy terrain plotted in two dimensions.

Although we may visualize this search in two or three dimensions, there are actually as many dimensions as there are processing units. The Hopfield convergence procedure seeks a minimum in such a multidimensional mountainous terrain.

In the Hopfield Net, there is no way to reach the global minimum from a local minimum. A different network paradigm, such as the Boltzmann Machine, must be used. The Boltzmann Machine uses noise to "shake" the network state out of a local minimum (Hinton & Sejnowski 1986). The Hopfield Net, however, can be restarted at a different initial position, which may allow the network to then find a global minimum.

Associative Memory

Hopfield originally proposed the application of associative memory for his binary network. This application illustrates the general approach of using a neural network architecture for developing an associative memory system. Although the binary Hopfield Net implementation is not a very efficient or reliable way to build an associative memory, the methodology is highly informative. The associative memory application illustrates the basic application technique for Hopfield networks—the energy equation is devised to model the applications problem and then used to assign the weight values. The network then converges to a solution.

The basic scheme for associative memory is shown in Figure 3-7, where each "memory" is represented by a vector of +/−'s. In this example, each memory is a person's name, together with a color associated with the name. Each memory vector is a state of the network that corresponds to a minimum in the terrain defined by the network energy. When the network starts at an initial state, the updating procedure moves the state of the network until it is at a minimum. That minimum, then, is expected to correspond to one of the "memories" of the network. Thus the network may "converge" to the stored memory that is most similar or most accessible to the initial state.

Figure 3-7a shows three example memories that may be stored in a Hopfield network with 24 processing units. In this scheme, (--++++----+-) stands for Karen. If the network is presented with this code for Karen, followed by a random string of -'s instead of a color association, then the network is expected to converge to the representation for Karen-Green. In this sense, the network has stored the association of Karen with Green (Figure 3-7b).

A similar use for the associative memory, that of recall from a noisy pattern pair — is shown in Figure 3-7c. A noisy version of the pattern pair Paul-Blue is input to the network. The network is then expected to converge to the correct +/− code for Paul-Blue.

Figure 3-8 illustrates another candidate application of the associative memory. An array of black/white pixels (Figure 3-8a) is used to represent individual characters. (Figure 3-8). Each of these characters can be represented as a +/− vector, and thus may be a memory in a Hopfield network. Associative recall allows the possibility that some fraction of the array entries be correct initially, with the rest incorrect or random (noisy). The network then begins

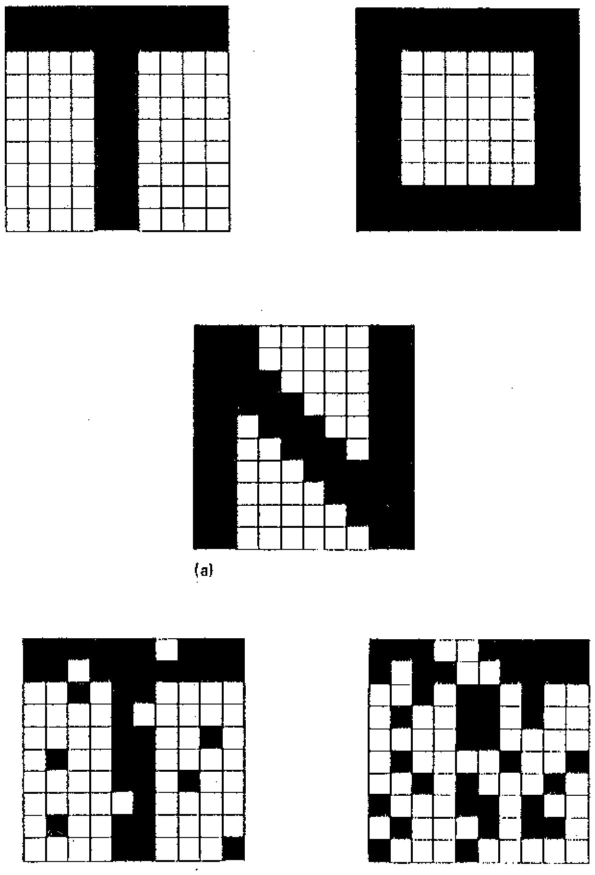
```
(a)  ASSOCIATED PATTERN PAIRS:

        KAREN   − − + + + + − − − − + − | − − + + − + + − − + − − GREEN
        GEORGE  + − − − + − − + + − + + | − + − − − + + − + − + + RED
        PAUL    − + + − − + + + − − − + | + − + + + + − − − − − + BLUE


(b)  PARTIAL DATA PRESENTED:

        KAREN   − − + + + + − − − − + − | − − − − − − − − − − − − (BLANK)

     RECALL AFTER CONVERGENCE:

        KAREN   − − + + + + − − − − + − | − − + + − + + − − + − − GREEN


(c)  NOISY DATA PRESENTED:

     (NOISY) PAUL   − + + + − + + − − − − + | + − − − − + − − − − − + (NOISY) BLUE

     RECALL AFTER CONVERGENCE:

        PAUL    − + + − − + + + − − − + | + − + + + + − − − − − + BLUE
```

Figure 3-7. Associative memory examples. (a) A 24-unit network stores three associations of names with colors. (b) The name is presented to the network, which recalls the associated color. (c) A noisy pattern pair is presented to the network, which recall the exact pattern pair.

with an initial state that is a noisy version of a memory pattern. The network is expected to converge to the memory that is most accessible to its initial state; if the noisy initial pattern is close enough to the memory pattern, then the network will converge to the correct character. The correct character may then, in principle, be recovered in spite of noise in some of the data.

The pattern in Figure 8b is a *T* with 10% of the pixels incorrect (noisy). When this pattern is presented as an initial state, the network converges to the *T* shown at top left. Figure 8c shows a pattern that started as a *T* but had 20% of its pixels changed. Here the network failed to converge to a *T*, selecting an

Hop61R



Hop67C

Figure 3-8. Pattern completion by a 100-unit Hopfield network that has stored three characters as memories. (a) Three memory patterns. (b) Character T with 10% noise. (c) Character T with 20% noise.

O instead because of the large amount of noise in the initial pattern state. These examples illustrate the pattern completion capacity and limitations of the Hopfield Net.

Setting the Weights

In the associative memory application, the patterns to be stored as memories are chosen a priori. The number of processing units in the network is equal to the number of entries in the patterns to be stored. The weights are fixed based on the choice of patterns.

First we must define notation for the patterns to be stored. Pattern **p**, a vector of +/−'s, such as (++--+-+ ⋯ +), is denoted by

$$A_p = (a_{p1}, a_{p2}, \ldots, a_{pn})$$

where a_{pi} = ith entry in pattern vector **p**. There are m patterns total: $A_1, A_2, \ldots, A_m$.
Then

$$T_{ji} = \sum_{p=1}^{m} \underbrace{(2a_{pi} - 1)}_{1 \text{ or } -1}\underbrace{(2a_{pj} - 1)}_{1 \text{ or } -1} \tag{3-7}$$

where the sum is taken over all patterns to be stored. Note that the term $2a$-1 simply leaves $a = 1$ the same, and changes $a = 0$ to -1. Equation (3-7) accomplishes the following: T_{ji} is incremented by 1 when $a_{pj} = a_{pi}$ (two entries of pattern **p** are the same); T_{ji} is decremented by 1 when $a_{pj} \neq a_{pi}$ (two entries of pattern **p** are different). This increment/decrement process is done for all pairs ij $(i \neq j)$ in all patterns A_p. One can add a pattern by doing the appropriate increment/decrement process for the new pattern.

Adding pattern memories by this process is analogous to "learning." This is a striking result because Hebb proposed a similar type of learning law with "neural assemblies" in 1949,[8] when it was virtually impossible to justify how such a system might work.

Recall Limitations

Although the Hopfield Net was a significant achievement, there are still many limitations to its actual use. First, the evoked memory is not necessarily the memory pattern that is most similar to the input pattern. A second disadvantage is that all memories are not remembered with equal emphasis; some are

evoked inappropriately often. A third disadvantage is that sometimes the network evokes *spurious states*—patterns that were not on the original list of patterns to be stored as memories.

In practical use, the evoked memory may not be the memory that is the closest match to the initial state of the network. (This is true, for example, when the number of matching entries is used as a measure of closeness.) An intuitive description of why is as follows: The initial state of the network defines a point on the "mountainous terrain." This point is on the side of a particular "valley." The minimization procedure finds the bottom of that valley. The bottom point corresponds to the evoked memory. Thus, a memory that is at the bottom of a larger valley will be evoked from a larger number of initial states.

A further practical limitation is the appearance of spurious states. An example of such a state is shown in Figure 3-9. The pairs Walter-White and Walter-Black are both stored in memory in the Hopfield Net. It is possible then to recall Walter-Gray. The reason that Gray is substituted for White is the correlation between the vector for White and the vector for Gray; likewise for the substitution of Black with Gray. Although the network is usually more likely to recall the pairs that were originally encoded (Walter-White and Walter-Black), it is still possible to get the spurious state Walter-Gray. Spurious states have been problematic in other applications of the Hopfield Net as well.

An unlearning process proposed by Hopfield (Hopfield, Feinstein, & Palmer 1983) is intended to address two limitations of the Hopfield Net—the existence of spurious states and the fact that some memories are evoked disproportionately often. In this process, changes in the weights are done according to the equation:

$$\Delta T_{ji} = -\epsilon\,(2a_{pj} - 1)\,(2a_{pi} - 1)$$

where ϵ is small, $0 < \epsilon \ll 1.0$.

As a result, this "unlearning" process equalizes the accessibility of memories; it also causes some suppression of spurious states. Although the suppression is not complete, it is significant, as documented by Hopfield and his colleagues (1983).

MEMORY 1:	WALTER	+ + + + − − − −	+ + − + − + − −	WHITE
MEMORY 2:	WALTER	+ + + + − − − −	− − + − + − + +	BLACK
MEMORY 3:	HAROLD	+ + − − + + − −	+ − − + + − − +	GRAY
SPURIOUS STATE:	WALTER	+ + + + − − − −	+ − − + + − − +	GRAY

Figure 3-9. A spurious state in a Hopfield Net associative memory.

There is a maximum limit to the number of memories that a Hopfield Net can store, and this limit increases when the number of units increases. The total number of memories that can be stored is about 0.15N memory states, where N is the number of processing units in the network. This limit was determined in experiments with computer simulations (Hopfield 1982) in which memory patterns were generated at random and stored in Hopfield networks. If more such memories are stored, then a significant number of errors occur during recall. Performance can become so poor that memory retrieval is almost impossible. A few errors do occur just below the 0.15N limit, and the number of errors goes down if the number of memories is decreased.

The capacity limit of the network depends in part on the choice of memory patterns. If the memory patterns are orthogonal vectors instead of randomly generated vectors, then more memories can be stored. A number of theoretical studies analyze these and other relationships in the capacity of the Hopfield network (Abu-Mostafa & St. Jacques 1985; McEliece et al. 1987).

CONTINUOUS-VALUED HOPFIELD NETWORK

In 1984 Hopfield extended the binary network design to allow for processing units that could attain a continuous range of values. This extension to the binary Hopfield Net is the continuous-valued Hopfield Net. Its activity corresponds directly to the activities in the binary net, but it is capable of doing much more because its units are not limited to binary (0/1) values and its architecture is more complex.

The continuous-valued Hopfield Net retains the same network topology as the binary Hopfield Net. The symmetric weight equation

$$T_{ji} = T_{ij}$$

is also retained. One difference is the use of the sigmoid function instead of the hard threshold given in Eq. (3-2). The sigmoid function provides a continuous-valued, nonlinear—"soft"—threshold.

Figure 3-10 plots the sigmoid function whose equation is

$$f(x) = \frac{1}{(1 + e^{-x})}$$

The soft threshold operates as follows: For values of x sufficiently below a threshold (0), the value of $f(x)$ is almost 0. For values of x sufficiently above a threshold (0), the function is close to 1. For a small number of values of x close to 0, the function increases rapidly. The sigmoid can be translated right or left to perform a thresholding at a value other than 0. The sigmoid can be translated or scaled vertically to approach limits other than 0 and 1. (Sigmoid functions also appear in the back-propagation paradigm, see Chapter 4.)

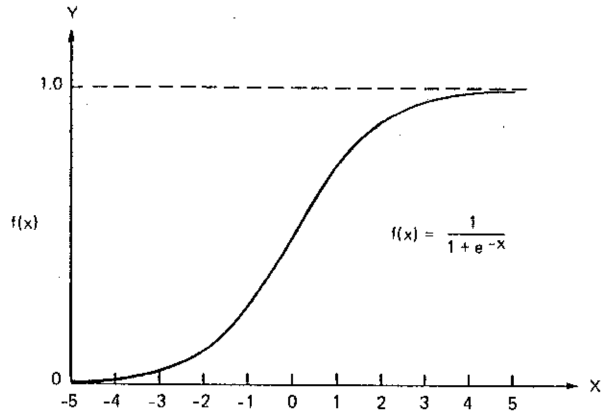


Figure 3-10. The sigmoid function.

In the continuous-valued Hopfield Net, the architecture of the network is specified so that changes in time are described continuously, rather than as discrete update times for units. The processing units are governed by the equation:

$$C_j \frac{du_j}{dt} = \sum_i T_{ji} V_i - \frac{u_j}{R_j} + I_j \qquad (3\text{-}8)$$

where C_j = a constant > 0, R_j = controls unit j's decay resistance ($R_j > 0$), I_j = external input to unit j, and V_i is the output of unit i after the sigmoid function is applied.

The energy equation is

$$E = -\frac{1}{2} \sum_j \sum_i T_{ji} V_j V_i - \sum_j V_j I_j \qquad (3\text{-}9)$$

where I_j is an input bias term. For applications, T_{ji} and I_j are chosen appropriately so that E represents the function that is minimized to solve the applications problem.

An Optimization Example: The Traveling Salesman Problem

The traveling salesman problem (TSP) is an extremely difficult classic optimization problem. This problem is in the class of *NP*-complete problems, all of which take a very long time to solve in worst-case analysis. Hopfield and Tank devised a way to attack this problem using the continuous Hopfield network, which found a good solution to the traveling salesman problem in a reasonable amount of time.

Unfortunately, practical limitations exist: Performance is not good with the Hopfield Net, and performance becomes poorer with larger problems (Wilson & Pawley 1988). Nevertheless, the behavior of the network is interesting to observe.

The TSP is stated as follows: A salesman has a number of cities to visit. He starts and ends at a particular city, and travels to each city on his list without visiting any city twice. The goal is to choose the order of the cities visited in such a way that the total path length traveled is minimized.

Let the cities to be visited be *A,B,C,* . . . ,*T,* with the distances between them given as

$$d_{xy} = \text{distance from city } x \text{ to city } y$$

A "tour" is a trip that starts and ends at the same place and visits each city exactly once. The path length of the tour $X_1X_2 \ldots X_t$ is

$$\sum_{i=1}^{n} d_{X_iX_{i+1}} \qquad (3\text{-}10)$$

where the subscripts of X are evaluated modulo n.

The minimum solution to the traveling salesman problem is the tour that has the smallest value for the total distance traveled. A "good" solution is a tour that is relatively close to the minimum solution. Hopfield was able to get a good solution rather than the minimum solution.

The first key to addressing this problem with the Hopfield network is the representation of the tour. A tour can be represented by a matrix of 0s and 1s. The rows of the matrix represent the different cities, and the columns of the matrix are the different positions in a tour (e.g., the first city is in position 1, etc.). A matrix representation for a tour of 10 cities is given in Figure 3-11.

A Hopfield Net can be built with the same number of processing units as there are entries in this matrix. Thus, a 10-city TSP would require a Hopfield Net with 100 (10^2) units. The goal of this application is to get the values of the 100 units in the Hopfield Net to converge to become the values for a matrix that represents a tour that is a good solution of the TSP.

	POSITION IN TOUR									
CITY	1	2	3	4	5	6	7	8	9	10
A	0	0	0	1	0	0	0	0	0	0
B	0	1	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	1	0	0	0
D	0	0	0	0	1	0	0	0	0	0
E	0	0	0	0	0	0	0	0	0	1
F	0	0	0	0	0	0	0	1	0	0
G	1	0	0	0	0	0	0	0	0	0
H	0	0	0	0	0	1	0	0	0	0
I	0	0	0	0	0	0	0	0	1	0
J	0	0	1	0	0	0	0	0	0	0

TOUR: G B J A D H C F I E

Figure 3-11. The traveling salesman matrix, specifying the tour at bottom.

The second key to solving the TSP with the Hopfield network is in the energy equation. The energy equation used is complex, and can be rearranged to match the terms in the original energy equation (3-9) given by Hopfield.

The energy equation for the TSP is:

$$E = (A/2) \sum_{X} \sum_{i} \sum_{\substack{j \\ j \neq i}} V_{Xi} V_{Xj} \qquad \text{(term 1)}$$

$$+ (B/2) \sum_{i} \sum_{X} \sum_{\substack{Y \\ Y \neq X}} V_{Xi} V_{Yi} \qquad \text{(term 2)}$$

$$+ (C/2) (\sum_{X} \sum_{i} V_{Xi} - n^2) \qquad \text{(term 3)}$$

$$+ (D/2) \sum_{X} \sum_{Y} \sum_{i} d_{XY} u_{Xi} (V_{X,i+1} + V_{Y,i-1}) \qquad \text{(term 4)}$$

where *A, B, C,* and *D* are parameter values that may be tuned to aid performance and $V_{X,i}$ is the entry in the table, row *X*, column *i*.

Each term in the energy function may be explained intuitively as imposing a particular restriction on the tour of the salesman, and each restriction is enforced through the minimization of the corresponding term. Ideally, if the constants are sufficiently large, then all of the low-energy states of the network will have the form of a valid tour, and the states with the shortest paths will be the lowest-energy states.

The first term allows only one visit to each city; this term is small when there is only a single 1 in a given row. The second term does not allow the salesman to be in two different cities at the same time; this term is small when there is only a single 1 in a column. The third term allows only *n* cities to appear on the itinerary; this term is small when there are only *n* 1s in the matrix. The fourth term expresses the distance sum to be minimized; this last term is a mathematical expression proportional to the total distance of the tour.

The weights are set initially by Eq. (3-11), which takes into account the distances between the cities

$$T_{Xi,Yj} =$$

$-A\,\delta_{XY}\,(1-\delta_{ij})$	inhibiting connections within each row
$-B\,\delta_{ij}\,(1-\delta_{XY})$	inhibitory connections within each column
$-C$	global inhibition
$-D\,d_{XY}\,(\delta_{j,i+1}+\delta_{j,i-1})$	distance term

The external bias is

$$I_{Xj} = Cn$$

We use the notation:

$$\delta_{ij} = \begin{cases} 1 \text{ if } i = j \\ 0 \text{ otherwise} \end{cases}$$

The weights remain fixed at the values given in Eq. (3-10) throughout the optimization procedure. The initial state of the network is set at random, and updating is performed until the network converges to a steady state. If convergence has been successful, the itinerary of the salesman is then read from the final state of the network.

In Figure 3-12, which showed the convergence of the network for one example run, the value of the matrix entry is proportional to the diameter of each circle. Thus, larger circles correspond to larger entries in the matrix of Figure 3-11. Figure 3-12a–c shows snapshots of intermediate times; Figure 3-12d shows the final state of the network, after it converged to a stable state.

Figure 3-12e also depicts an intermediate state of the network, the route of the salesman's travel plan. Intermediate states of the network can sometimes show the itinerary going to two places at the same time (two entries in the same column), or may omit visiting certain cities at all (no entries in a given row). The intermediate states may appear to be "considering" different possibilities.

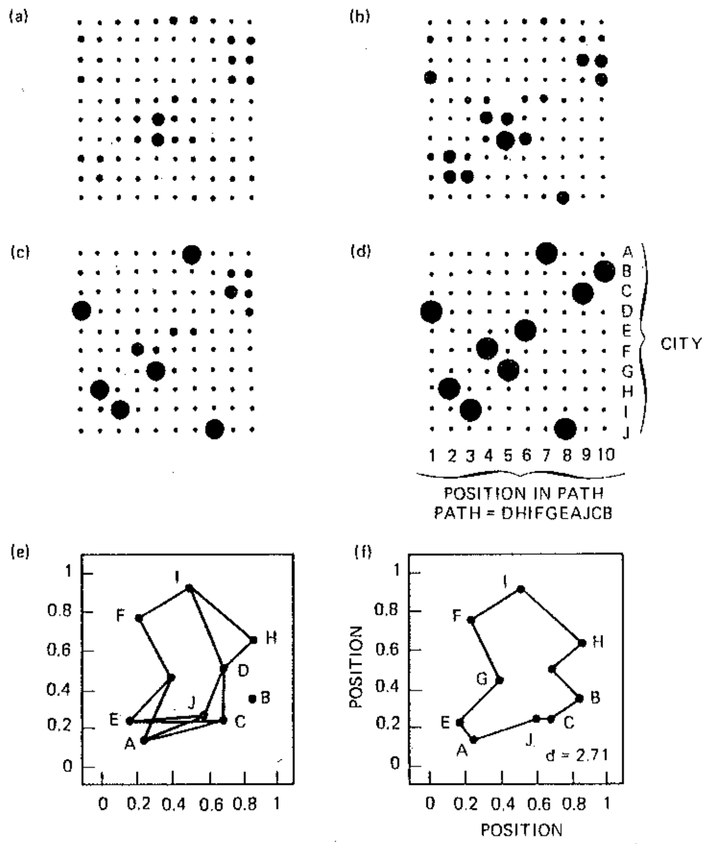


Figure 3-12. A traveling salesman solution. (a,b,c) Progressive intermediate states of the network. (d) Final state of the network. (e) Incomplete tours indicated by an intermediate state. (f) Final tour, indicated from the final state of the network shown in (c) (from Hopfield and Tank. Neural computation of decisions in optimization problems, *Biological Cybernetiks*. Springer-Verlag 1985).

The tour that corresponds to the final state of the network, Figure 3-12f, obeys the basic rules of visiting each city, with each city visited only once. Furthermore, the distance is relatively short compared to other possible tours.

Although Hopfield and Tank reported success with the traveling salesman problem, their work was apparently limited to a subset of problems involving fewer than 30 cities. Other investigators (Wilson & Pawley 1988) have reported that the Hopfield network does not solve larger problems (with more than 10 cities) reliably. However, in the larger problems the network often comes up with a solution that contains segments that are locally optimal.

ADVANTAGES AND LIMITATIONS

In each of these uses of the Hopfield network for associative memory and optimization, the applications problem is solved when the network reaches a stable state at an energy minimum. Such an approach was striking when proposed because of its use of an interconnected net of processing units that "settle" or converge on a final solution.

Applications of the Hopfield Net are unfortunately limited by the network's level of performance: Associative memory has displayed limited capacity, uneven recall ability, and recall of extra (spurious) states. Optimization of the traveling salesman problem provides a limited number of good answers and does not work well for more than 10 cities. The Hopfield network nevertheless gives an excellent demonstration of problems that can be attacked with neural networks. Realistic real-world applications, however, will probably require enhanced network architectures.

Hopfield and Tank have shown additional examples of the Hopfield Net in a variety of optimization problems, including signal decomposition and linear programming. Some job shop scheduling optimization has been addressed with a Hopfield-like network (Foo and Takefuji 1988). Such optimization problems, however, tend to work on examples from a limited domain — the problems solved are not general.

Many possible applications exist for the Hopfield network and similar, more sophisticated architectures, such as the Boltzmann Machine. Candidates include speech processing, database retrieval, image processing, fault tolerant memories, and pattern classification.

The nonsynchronous updating of processing units in the Hopfield network is a unique property, and is especially relevant to the study of biological systems because of their asynchronous updating of nerve cell attributes. In addition, asynchronous updating can be helpful in designing fast hardware implementations.

The most general advantage of the Hopfield network is its inherently parallel architecture. As a result, potential hardware implementations may be very fast. Tradeoffs must be assessed, however, between the size and speed of the network and the size of the applications problems.

References

Abu-Mostafa, Y. S. and J. St. Jacques. 1985. *IEEE Trans. Info. Theory* 31(4): 461–64.

Anderson, James A. 1972 "A Simple Neural Network Generating an Interactive Memory," *Mathematical Biosciences*, Volume 14, pp 197–220, 1972.

Anderson, James A., Silverstein, Jack W., Ritz, Stephen A., Jones, Randall S. 1977 "Distinctive Features, Categorical Perception, and Probability Learning: Some Applications of a Neural Model," *Psychological Review*, Volume 84, Number 5, September 1977.

Foo, Y. S. and Y. Takefuji. 1988. Stochastic neural networks for solving job-shop scheduling: Parts 1 and 2. *IEEE ICNN Proc. 1988* II-275–II-290.

Fukushima, K., S. Miyake, and T. Ito. 1983. Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE Trans. Systems, Man, and Cybernetics* 13(5): 826–34.

Grossberg, S. 1987. *The Adaptive Brain*, Vols. 1 & 2. New York: North-Holland.

Grossberg, S. 1988. *Neural Networks and Natural Intelligence*. Cambridge, Mass.: MIT Press.

Hebb, D. O. 1949. *Organization of Behavior*, New York: Wiley.

Hinton, G. E. and T. J. Sejnowski. 1986. Learning and relearning in Boltzmann Machines. In *Parallel Distributed Processing*. Cambridge, Mass.: 282–317. MIT Press.

Hopfield, J. J. 1982. Neural networks and physical systems with emergent collective computational abilities. *Proc. Natl. Acad. Sci.* 79: 2554–58.

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

Hopfield, J. J., D. I. Feinstein, & R. G. Palmer. 1988. Unlearning has a stabilizing effect in collective memories. *Nature* 304: 158–59.

Hopfield, J. J. and D. W. Tank. 1985. Neural computation of decisions in optimization problems. Biol. Cybernetics. 52: 141–152.

McEliece, R. J., E. C. Posner, E. R. Rodemich, and S. S. Venkatesh. July 1987. The capacity of the Hopfield associative memory. *IEEE Trans. Info. Theory* 33(4): 461–482.

Reilly, D. L., L. N. Cooper, and C. Elbaum, 1982. "A Neural Model for Category Learning." Biological Cybernetics. 45, pp. 35–41.

Wilson, G. V. and G. S. Pawley. 1988. On the stability of the traveling salesman problem algorithm of Hopfield and Tank. *Biol. Cybern.* 58: 63–70.

Suggested Readings

Anderson, James A. 1983 "Cognitive and Psychological Computation with Neural Models", IEEE Transactions on Systems, Man, and Cybernetics, Volume SMC-13, Number 5, September/October 1983.

Hopfield, J. J. and D. W. Tank. 1986. Computing with neural circuits: A model. *Science* 233:625–33.

Tank, D. W. and J. J. Hopfield. 1986. Simple neural optimization networks. *IEEE Trans. CS.* CAS-33(5): 533–41.

Van den Bout, D. E. and T. K. Miller. 1988. A traveling salesman objective function that works. *IEEE ICNN 1988* II-299–II-304.