Neural Networks, Part 2:

What are they and why is everybody so interested in them now?

Philip D. Wasserman, Anza Research, Inc.
Tom Schwartz, Tom Schwartz Associates

An application example

Figure 5 shows a typical neural network application involving recognition of handwritten input characters. Here, we construct the 63-component **X** input vector by observing the 7 × 9 pixel area on which the character is written. If a line passes through the pixel, we give the **X** vector's associated component value 1—otherwise, it has value 0. The neural network is trained through an algorithm adjusting its weights until application of the **X** vector (the handwritten character) produces the output **Y** vector (which can be the ideal character).

Broadly stated, we can consider neural networks as systems that map vectors in one space into another space. Vectors, considered in their generalized mathematical sense, can be composed of elements having a wide range of characteristics.

Learning

Rather than programming them, we train neural networks by example. As children need to know nothing about comparative physiology to recognize their mothers, programmers need not provide neural networks with quantitative descriptions of objects being recognized, nor sets of logical criteria to distinguish such objects from similar objects. Instead, we show a neural network examples of objects (faces, parts, or scenes) along with their identifications (mother, carburetor, or mountains). It memorizes these by altering values in its weight matrix, and will produce the proper response when an object is seen again.

This learning ability has profound implications for pattern recognition. In the past, conventional computers have required one or more algorithms (step-by-step procedures) to solve problems. To implement pattern recognition on a conventional computer, for example, programmers select from standard algorithms that they combine with custom-made programs—a combination that is more often art than science. Ideal algorithms for one problem can be totally inappropriate for another superficially similar instance. As a result, pattern recognition requires highly trained and experienced computer scientists. While resultant accomplishments have been impressive in some cases, this is a slow and costly way to solve problems—typically requiring expensive high-speed computers to run programs in real time.

Will neural networks replace programmers? Probably not, but in certain applications—including pattern recognition—they provide a far simpler and more powerful paradigm.

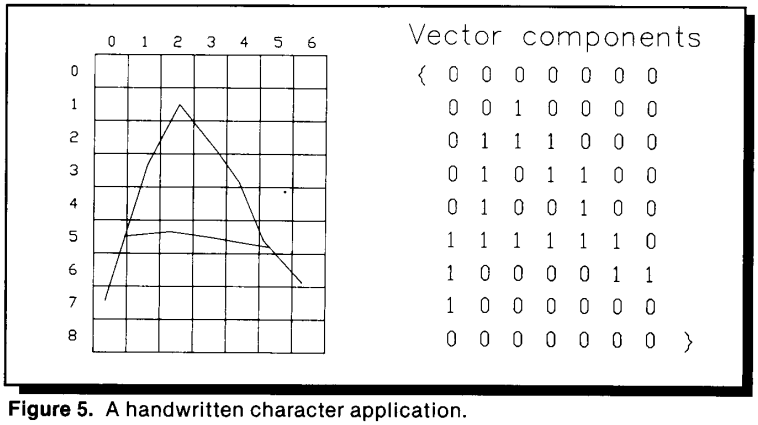

Figure 5. A handwritten character application.

Generalization

A frustrating characteristic of conventional computers is the literal, precise inputs required to produce the desired output. Neural networks, on the other hand, can accommodate variations in their input and still produce the correct output. For example, a system trained to recognize printed letters did so even when noise corrupted 40 percent of the input characters.

That is, the system recognized letters despite never having seen anything exactly like them before—much as humans understand incomplete and partially incorrect input. Studies show that most people can read text in which more than half the letters are obliterated.

The real world rarely presents information with the precision required by a computer program. Of course, conventional computers have been programmed to tolerate "noisy" inputs, but the computational load often precludes using these algorithms in practical applications. Neural networks accomplish the needed generalization by virtue of their structure rather than through elaborate programming (which tends to be application dependent). As such, neural networks provide a far more natural interface to the real world—a world including human users.

0885-9000/88/0200-0010 $1.00©1988 IEEE

Abstraction

Neural networks can abstract the "ideal" from a non-ideal training set. A network trained to recognize the alphabet, for example, was shown only images heavily corrupted by noise. Nevertheless, after training, the system input corrupted letters but output perfectly formed characters. In effect, the system remembered something it had never been taught. It had, in some imperfectly understood fashion, abstracted and stored the training set's essence and could respond appropriately to imperfect inputs.

Through training, neural networks form internal representations of the training set's salient features. This feature extraction can produce surprising results. One researcher trained a network, coding short sentences as binary vectors. Upon examining the pattern of trained weights, he discovered that the network had organized itself to parse the sentence and correctly categorize words as verbs, nouns, adjectives, and so forth.

Such abstracting ability harks back to Plato's *Republic* and the platonic concept of ideals. How do we determine that a given animal is a dog when every dog we've seen is different? Do we have an internal model—an "ideal dog"—to which we compare all instances? Neural nets can generate similar ideals from a set of imperfect examples.

Speed

One can view neural networks as associative memory, associating input patterns with desired output patterns. When new patterns are presented to the input, associated patterns are produced at the output. In neural networks, significantly, the time required to produce outputs is independent of the number of associations stored. Thus, nothing corresponds to a "search"; the only time required is that associated with network stabilization—a constant in most architectures. A given network storing ten million associations is just as fast as one storing ten thousand.

Multiprocessing

Many computer architects feel that today's fastest computers operate within a factor of 10 of a single processor's theoretical speed limits. For this reason, a major effort toward multiprocessing exists—an effort dividing problems into subproblems, each of which can be assigned to separate processors.

This approach presents two major difficulties. First, many problems cannot be fully solved in parallel. Logical constraints force some or all of the program to be computed serially, in which case systems with 1000 processors may have 999 idle most of the time. Second, computer scientists have proven that efficient scheduling is in the NP-complete problem class. This means that no known algorithm for finding an optimal schedule for multiple processors is better in the worst case than trying all possibilities (usually, a prohibitively time-consuming approach). Still, it has not been proven that no efficient algorithm exists. Discovery of one example would solve hundreds of intractable problems—problems that have defied computer scientists for decades.

With processed, packaged, and tested silicon costing around $1,000,000 per square meter, idle processors make no sense. Developers have tried various architectures to eliminate this problem. For example, Dataflow has succeeded in some specialized areas but failed abjectly in others.

Wasserman and Schwartz's second installment exemplifies and examines the characteristics, capabilities, and history of neural networks. In addition, it addresses neural network architectures, training, and performance. Part one, their introductory description, appeared in our last issue (winter 1987). We welcome reader commentary on both substance and manner of presentation.

Henry Ayling
Managing Editor

Inherently, neural networks schedule themselves—that is, each node can be viewed as a processor operating on its inputs independently of all other processors in the system. Thus, while the network converges to a solution, all processors are busy. Hence, no expensive silicon remains idle. Furthermore, we can add processors in a modular fashion to suit problem size without restructuring the system.

History

At the least, research on neural networks must date back to Marvin Minsky's 1954 doctoral dissertation. Interest surged during the 1960s, when some highly optimistic claims were made regarding neural network capabilities. By this time, Minsky had become disenchanted with the approach as a result of his ongoing research. He found that many patterns could not be learned by simple one-layer networks. Multilayer networks were poorly understood. No theory existed regarding multilayer training, and no proof existed that multilayer networks were inherently superior to single-layer networks.

Consequently, Minsky wrote a book with Seymour Papert titled *Perceptrons*, presenting detailed theoretical treatment of single-layer networks and proving that these networks could not solve such simple problems as implementing an exclusive-or. They wrote that

> The perceptron has shown itself worthy of study despite (and even because of!) its severe limitations. It has many features that attract attention: its linearity; its intriguing learning theorem; its clear paradigmatic simplicity as a kind of parallel computation. There is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgment that the extension is sterile. Perhaps some powerful convergence theorem will be discovered, or some profound reason for the failure to produce an interesting 'learning theorem' for the multilayered machine will be found.

Minsky's prestige, combined with his book's convincing theoretical treatments, effectively killed federal funding for neural network research for 15 years. Moreover, the Soviet Union canceled several promising projects, and researchers throughout the world sought more promising fields.

Despite this loss of support, a few researchers continued their work. Steven Grossberg, Tuevo Kohonen (at the University of Helsinki), K. Fukushima, Jim Anderson, S.A. Amari, C. von der Malsberg, and others gradually developed theoretical foundations for multilayer networks and produced practical algorithms for their implementation. Unfortunately, they experienced severe difficulties in getting their papers published. As a result, their work is scattered among a large number of obscure journals—a fact that further hampers research in this area.

Nevertheless, training algorithms with proven convergence characteristics were developed for multilayer networks despite Minsky's pessimism. These net-

works proved capable of solving the exclusive-or problem and, with suitable complexity, exhibited highly generalized learning abilities.

Architectures

Researchers have produced many different arrangements of artificial neurons and have devised many algorithms to train them. Most are presented as "biologically inspired," or at least "biologically plausible." In fact, knowledge about biological neuronal systems reveals levels of complexity and diversity far beyond the highly simplified models of these artificial architectures. Furthermore, the operational functions of many biological features are poorly understood. Hence, artificial neural network architectures represent grossly simplified approximations to the biological systems after which they are modeled. Since the functional architecture of biological systems remains incompletely understood, many artificial neural network architectures represent various speculations regarding actual structure.

Learning. Most learning algorithms employ some version of Hebb's Law, first enunciated by D.O. Hebb in 1949. Hebb hypothesized that the synaptic connections between neurons were reinforced when both the source and destination neurons were active. Specifically, the change in a weight is proportional to the product of source and destination neuron output levels. In equation form, this would be written

$$\text{DELTA } W_{ij} = k * (O_i * O_j)$$

Grossberg has used a learning scheme in which changes in weight are proportional to "activation function" A_i, evaluated at the difference between higher layer output minus the weight between layers, or

$$\text{DELTA } W_{ij} = k * A_i * (O_j - W_{ij})$$

Although researchers use many other learning formulas in varying architectures, simple Hebbian learning is still widely employed.

Neural networks are trained in two ways—supervised and unsupervised. In supervised learning, a training pair consists of an input vector and a desired target vector. The input vector is applied to the network, which then produces an output vector that is subtracted from the target. The difference constitutes an error that is used to modify network weights in a manner that reduces the error on subsequent training cycles.

In unsupervised learning, no target vector exists. The input vector is applied to the network and the system "self organizes" so that a consistent output (possibly unpredictable before training) is produced whenever that input vector is applied. Such systems are used for classification problems and have been highly developed by Kohonen and Grossberg.

Some have argued that supervised learning is not biologically plausible—that no "teacher" resides in a biological system to direct training or to compare responses against desired outcome. For this reason, Grossberg and others have emphasized unsupervised learning. Nevertheless, many supervised learning algorithms produce excellent results and seem promising solutions to practical problems.

We sometimes divide neural networks into auto-associative and hetero-associative types. In auto-associative types, desired output is the same as input; such systems are useful for pattern completion, where a pattern segment (or a noisy version) is provided and the full, perfect pattern is produced. We will see in our counter-propagation examples that we can use an unsupervised stage with a hetero-associative stage to produce a hetero-associative system. Hetero-associative types are useful in general vector mapping, where an input vector is mapped to the target output vector. These types are also capable of pattern completion, but only in terms of the desired target vector.

Architectural examples

The following sections present several prominent architectures and learning techniques. Through this nonexhaustive treatment of neural net architectures, we hope the reader develops an understanding that will permit further study.

Counter propagation. Based upon Kohonen and Grossberg's work, Robert Hecht-Nielsen devised this architecture, which is capable of mapping an arbitrary input vector to an output vector and has been applied to such tasks as handwritten character recognition. For purposes of explanation, we will first consider counter propagation's "forward" path. As Figure 6 shows, it consists of three neuron layers—an input layer that simply receives the input vector, an intermediate "hidden" layer connected to the input layer outputs through an array of weighted connections W, and an output layer connected to the hidden layer by an array of weighted connections K.

Training the counter-propagation network. We accomplish training in two phases and by two different algorithms. During phase 1, we apply a sequence of training vectors **X** and an algorithm known as "Kohonen learning" to adjust weights W. Kohonen learning is an unsupervised learning algorithm; that is, only input vectors (without desired output vectors) are applied during training and the weights W are adjusted so that a given neuron or set of neurons in layer 2 is activated. This is accomplished in the following procedure:

(1) Apply an input vector **X**.

(2) Find the dot product of **X** with each set of weights connected to each neuron. If W_i is the set of weights associated with neuron i, then—for all i—find dot product $\mathbf{X} \bullet W_i$.

(3) Find the neuron with the largest dot product and call it number c.

(4) Adjust the weight vector associated with neuron c according to the following formula:

$$\mathbf{W}_c^{new} = \mathbf{W}_c^{old} - \text{alpha}(\mathbf{X} - \mathbf{W}_c^{old})$$

where alpha is a constant less than 1 representing the training rate. This constant is generally reduced during training.

(5) Repeat Steps 1 through 4 as required to train the system.

During training phase 2, we train the weights connecting layers 1 and 2 using the Grossberg Outstar method. Here, both an input vector **X** and a desired output vector **Y** are supplied as pairs in the training set; therefore, this method is supervised training.

The Grossberg Outstar training process consists of the following steps:

(1) Input a vector **X** and a corresponding desired output vector **Y** as shown in Figure 6.

(2) Take the input vector's dot product with each of the weight vectors $\mathbf{W}_i$. Find the largest of these dot products, call the neuron (that they feed) in layer 2 number c, and call its output Z_c. Set Z_c to 1.0.

(3) Adjust each of the weights K_{cj} between neuron c in layer 1 and the neurons in layer 2 according to the following rule:

$$K_{cj}^{new} = K_{cj}^{old} + (d * Y_j - c * K_{cj}^{old})$$

Where c and d are constants less than 1 that are reduced during training, repeat Steps 1 through 3 as needed to train the system.

Computation with the counter-propagation network. We perform computation, like training, in two stages. First, we apply the input vector

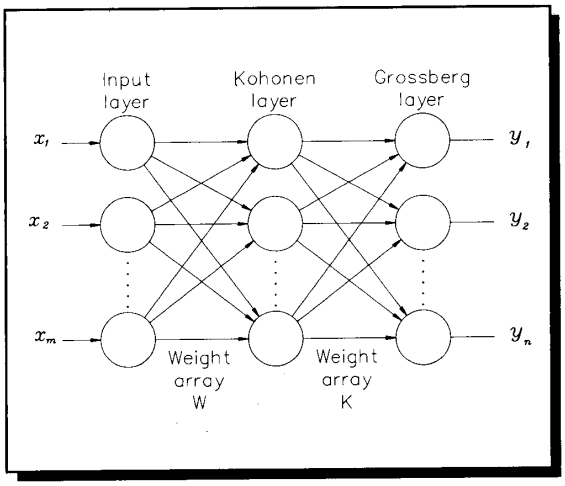

Figure 6. A counter-propagation network.

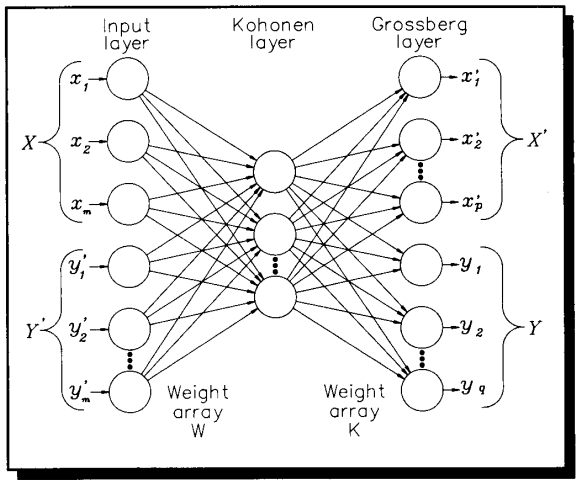

Figure 7. A full counter-propagation network.

X and find its dot product with each weight vector $\mathbf{W}_i$ in layer 1. Let's call the neuron with the largest dot product in layer 1 number *c*.

Next, we find each neuron's output Y_j in layer 2 as simply equal to K_{cj} (the weight connecting neuron *c* in layer 1 to output Y_j).

To optimize counter-propagation network performance, some preprocessing is necessary. First, each input vector should be normalized to unit length. To accomplish this, compute the square root of the sum of the squares of the components, and divide each component by this sum.

Also, all network weights should be set to small random numbers. Each set of weights providing input to a Kohonen neuron should be treated as a vector and normalized to unit length just as was done for input vectors.

These two normalizations allow each input and Kohonen weight vector to be considered as a point on a unit hypersphere. Applying Kohonen training then becomes a matter of selecting an input vector point, finding the weight vector point closest to it on the hypersphere's surface, and adjusting the position of the weight vector to be closer to the input vector.

Figure 7 shows a full counter-propagation network. Two input vectors (**X** and **Y**′) and two output vectors (**X**′ and **Y**) now exist. When training the Kohonen layer, the two input vectors are simply considered as one large vector and the weight array W is adjusted as previously described. Similarly, training the Grossberg layer requires that the two output vectors (**X**′ and **Y**) be treated as one large output vector and, again, the training equations are the same.

This arrangement is interesting because it can learn two functions simultaneously. For example, if one desired function is $\mathbf{Y} = f(\mathbf{X})$ and another is $\mathbf{X}' = g(\mathbf{Y}')$, both can be learned at the same time. Just apply **X** and **Y**′ as inputs, **X**′ and **Y** as desired outputs, and train in the usual fashion. Once trained, application of **Y**′ with all components of **X** set to zero will produce **X**′. Conversely, **X** will produce **Y** if all components of **Y**′ are set to zero.

In the special case where g is the inverse function of f, the network can simultaneously learn a function and its inverse (if it exists). This characteristic has found application in image data compression, where f is the compression function and g is the inverse recovery function.

The functionality of counter propagation. The counter-propagation network has the virtue of rapid learning. However, since its internal layer produces only a single neuron having an output of 1 (all others are set to 0), it cannot develop the complex internal representations needed to perform some mappings. Furthermore, it is often difficult to separate input vectors that are close in the L2 norm sense (the conventional euclidean distance) as these will tend to activate the same Kohonen neuron. Despite this limitation, the architecture is interesting and potentially useful because (once classified) an input vector can be mapped to an arbitrary output vector.

Back propagation. The back-propagation algorithm provides a theoretically sound method for training multilayer networks. This answered Minsky's challenge (presented in *Perceptrons*) by converging to a solution of the exclusive-or and many other difficult problems. In addition, David Rumelhart has developed and presented an elegant convergence proof.

Controversy abounds regarding the actual inventor of back propagation. David Parker claimed authorship in 1982 but subsequently announced that, after independently inventing the algorithm, he discovered that Paul Werbos (now at the Energy Information Agency) had published a dissertation in 1974 in which the entire algorithm was disclosed. While Rumelhart referenced Parker's work, he referenced none published before 1985. Such confusion highlights the problems associated with limited communication in this field. Researchers have often been unaware of other work (often published in obscure journals). Consequently, much effort has been wasted or duplicated.

Computing with back propagation. During computation, since the back propagation network is a simple multilayer feed-forward configuration, each neuron produces an output called Net representing weighted sums of the previous layer's outputs. No intralayer connections exist, nor any feedback paths from an output layer to an earlier layer.

In addition to Net output, each neuron produces an Out signal by simply passing the Net through a "squashing"

function. Many functions have been proposed, however the most common usage is

$$\text{Out} = 1/(1 + \exp(-\text{Net})).$$

This satisfies the convergence requirement —namely, that each neuron's output be a nondecreasing, differentiable function of its input's weighted sum (bounded in this case by 0 and 1).

Each operation applies an input vector to the first layer, producing a Net signal. This passes through the squashing function to yield an Out signal that propagates through the weights to the next layer. There, each neuron receives the Out of every neuron in the previous layer (multiplied by a weighting factor). The operation repeats layer by layer until it produces a set of Outs at the output layer—a set comprising the desired output vector.

Training the back-propagation network. An example of supervised learning, the back-propagation algorithm applies an input vector to the input, thereby producing an output vector. It then compares the output vector to a desired "target" vector and adjusts network weights to minimize the difference between these vectors.

Training is a two-stage process that applies an input vector, and computes and stores Net and Out for each neuron in the network. Next, the process calculates output error and propagates it backward through the network, training weights as it's passed from layer to layer. This two-step process is repeated over the training vector set's elements until the network converges and produces the desired responses.

Output layer training resembles single-layer training algorithms. The difficulty arises when training internal layer weights where no target is available. The solution lies in propagating the error back, layer by layer, from the output to successively deeper layers. Thus, neurons in an interior or "hidden" layer receive an error signal that is the weighted sum of the following layer's error signals. The neuron uses this error to train its associated weights, then passes its error back to all neurons to which it connects in the next lower layer.

Training the output layer is relatively simple. For each neuron, we subtract the output from the target vector's associated component and produce a difference. We then multiply this difference by the squashing function's derivative, evaluated at the net output's current value for that neuron, producing a value of DEL for this neuron.

This is expressed in equation form as

$$\text{DEL}_j = (T_j - O_j) * F'(\text{Net}_j)$$

where

DEL_j = the error to be propagated back for the jth neuron in a layer.

T_j = the target (or desired output) for the jth neuron in a layer.

O_j = the previously calculated output of the jth neuron in a layer.

F' = The derivative of the squashing function. If we use the sigmoid function $F(x) = 1/(1+\exp(x))$, it may be shown that $F'(x) = F(x)(1-F(x))$.

$$\text{Net}_j = \sum_i O_i * W_{ij}$$

where W_{ij} represents the weight from the ith neuron of the next lower layer to neuron j in the current (output) layer.

Next, we adjust the current (output) layer weights according to the following formula:

$$W_{ij}(n+1) = W_{ij}(n) + \text{DEL}_j * O_i * \eta$$

where $W_{ij}(n)$ equals the weight from the ith neuron in the previous layer to the jth neuron in the current (output) layer at the nth step in the calculation, O_i is the neuron's output in the lower layer connected to the weight in question, and η is the learning rate typically less than 1.0.

We must now adjust the weights of the penultimate layer. First, we compute each neuron's DEL_j in this layer using the following formula:

$$\text{DEL}_j = F'(\text{Net}_j) * \sum_k (\text{DEL}_k * W_k)$$

Thus, to find the DEL at an interior layer's neuron, take the summation of each neuron's DELs in the next layer—each multiplied by the weight connecting it to that neuron. Then, multiply this summation by the squashing function derivative evaluated at the current neuron's previously calculated Net output.

Having found the DEL for an interior layer's neuron j, the weights can then be adjusted by applying the formula used for the output layer. We repeat this process for every neuron in each layer from output to input. By convention, the lowest layer has no variable weights; each neuron receives one input vector component directly (through a constant weight of 1).

Various elaborations of this algorithm have proven beneficial. First, it can give each neuron a weight to a constant "one" level—a weight trained in the same way as all of the others, with its contribution included in all computations. In this way, we add a trainable bias that helps convergence with some training sets.

Perhaps even more important, we can add a "momentum" to the calculation. In this case, the weight modification equation is modified as follows:

$$\text{DELTA}(W_{ij}(n+1)) = \eta * \text{DEL}_j * O_i + \text{ALPHA} * \text{DELTA}(W_{ij}(n))$$

where

$\text{DELTA}(W_{ij}(n+1))$ = a change in weight i,j at Step $n+1$

ALPHA = the momentum coefficient (approximately .9).

Using the momentum term, change in a weight depends upon the previous change's value. This stabilizes the convergence with error surfaces containing long ravines that display sharp curvature across the ravine and a gently sloping floor. This low-pass filter allows larger values of η, thereby speeding convergence.

Back-propagation network performance. The back-propagation algorithm has solved the exclusive-or problem, as well as numerous other problems that are intractable using a single-layer network. However, it has four major problems—slow training, local minima, temporal instability, and biological implausibility.

Slow training. Training is slow. With n neurons in each of two adjacent layers, we must train $(n)^2$ weights. Furthermore, the error must be back propagated through all of these weights. Complete training may take 10,000 passes through a training set, so it is not unusual for a three-layer network with 100 neurons per layer to take weeks to train on a fairly capable general-purpose computer. Once trained, however, the network can be used indefinitely. If viewed as a one-time product development process, training time is usually not a serious problem.

Local minima. The back-propagation network computes an error for each output neuron proportional to the difference between its actual output and the desired output. The algorithm operates by adjusting network weights so as to minimize the sum of the squares of these errors as training progresses. It's quite possible for the training algorithm to become trapped in a local error minimum, thereby converging to an unacceptable solution. Since the solution depends upon the starting point, nothing remains but to train again with different initial conditions for the weights. Rumelhart indicates that this occurs infrequently. Other researchers

have shown that many large networks can be successfully trained. Still, cases have existed where training has used a large amount of computer time but failed to produce a usable set of weights.

Temporal instability. Grossberg and Gail Carpenter (also of Boston University) have investigated back propagation and rejected it due to its tendency to "forget" previously learned patterns when taught new ones. It is certainly true that a back-propagation network, trained using a given training set, will have its training upset by introducing a new training vector. Such a characteristic would be ruinous if expected to adapt to a continuously changing environment. However, if the training set is fully known at the time of training and continuous adaptation is unnecessary, this aspect is insignificant.

Biological implausibility. Grossberg and others have sought to restrict their research to architectures not violating known properties of biological systems as we understand them today. Two factors motivate this attitude. First, evolution has spent millions of years testing every possibility and has probably found optimal solutions to the problems we want neural networks to solve; therefore, networks unlike those found in nature are probably nonoptimal. Second, many researchers attempting to understand the brain's physiological and psychological functions have little interest in producing algorithms unrelated to their quest.

Certainly, no known biological system has been shown to exhibit the information flow back and forth through the back-propagation network (along with taking derivatives!). Nevertheless, although it may have little to do with the organization of the brain, back propagation has solved many significant problems.

The Cauchy machine

Neural network training can be cast as a nonlinear optimization problem comprising a search of N-dimensional space (where N is the number of weights) that attempts to minimize an objective function. With supervised training, this objective function represents the difference between target and output vectors. Local minima plague a simple gradient search in all but the simplest cases; that is, a local "valley" is found and the algorithm cannot escape even though there may be a much lower valley just over the next "hill." Researchers have used simulated annealing to solve such problems—by analogy, networks are compared to thermodynamic systems having a temperature T.

When annealing a metal, for example, we heat the metal to a temperature high enough so that the atoms have sufficient energy to move about freely; entropy causes the atoms to seek a minimum energy configuration. As we gradually reduce the temperature, atomic mobility is more limited and the atoms can line themselves up in more and more perfect minimum-energy states. In this way, large crystals can be grown covering distances billions of times bigger than a single atom because the crystal represents the system's global minimum energy state.

Nicholas Metropolis first applied these ideas to optimization problems, solving the Traveling Salesman problem in time that was a small power of the number of calculations. Since then, this method has been variously applied to neural network training. One version of the so-called Boltzmann machine uses the following training algorithm:

(1) Set an initially high temperature and randomize all network weights.
(2) Apply an input vector and compute the objective function with current weights.
(3) Change each weight randomly as determined by the Boltzmann distribution $P(x) = \exp(-x^2/T^2)$.
(4) Recompute the objective function; if it is reduced, make the weight change permanent; if it is increased (worse), make the change permanent only infrequently, with a probability determined by the Boltzmann distribution of Step 3.
(5) Compute a new value for temperature as follows: $T(n+1) = T(0) * (1/\log(1+n))$; $n \geq 1$.
(6) Repeat Steps 3 through 6.

Stuart and Donald Geman have proven convergence for the Boltzmann machine. However, it's slow due to Step 5's inverse logarithmic temperature reduction.

To speed convergence, Harold Szu invented the Cauchy machine. Rather than using the Boltzmann distribution in Step 3, it uses the following Cauchy/Lorentzian distribution:

$$P(x) = T/(t^2 + x^2)$$

This distribution, characterized by longer "tails" than the Boltzmann distribution, raises the probability that larger changes will be made. Thus, it preserves local convergence, but the presence of a few huge jumps enables faster escape from local minima. Consequently, a faster temperature reduction is possible and Step 5 can use the following formula:

$$T(n+1) = T(0)/(1 + T)$$

Based upon a stochastic Markovian chain, R.L. Hartley and Szu have developed a rigorous theorem proving convergence. This development has been compared in importance to the Fast Fourier transform, as it moves many more applications into the realm of computational practicality.

Discussion of the Cauchy machine. The Cauchy machine represents a possible solution to the local minima problem encountered with virtually every other neural network training algorithm. However, many unanswered questions exist regarding this technique's use. For example, we know that the algorithm converges faster than the Boltzmann machine. But what is its actual convergence time with typical networks? How often does it find a true global minimum in a reasonable number of iterations? How many weights should be changed at one time? How often should they be changed? What is the best pattern with which to present the training set? While the method shows great promise, further work remains before we can fully evaluate its ability.

Future outlook

Neural networks is a rediscovered field experiencing an explosive growth in research and application interest. Algorithms and architectures proliferate. Claims and counterclaims fill the literature. And the press produces stories at a rapid pace.

Despite its longevity, neural network theory and technology is rudimentary. We find many more questions than answers, and technical knowledge remains narrowly disseminated. Numerous demonstration programs exist, but not one proven commercial application for neural networks (although software and hardware producers vaguely allude to several).

The situation resembles the laser's when it was introduced. The laser had such unique properties that many people felt it must be of immense value. Nevertheless, to develop even a small percentage of its commercial potential required nearly a decade.

If this analogy is valid, some time will pass before neural networks find applications where their unique characteristics make them the clear method of choice. Meanwhile, all parties—researchers, commercial firms, and the press—must understand the risks of promising more than can be delivered. ■