The Cascade-Correlation Learning Architecture

Scott E. Fahlman and **Christian Lebiere**

August 29, 1991
CMU-CS-90-100

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Cascade-Correlation is a new architecture and supervised learning algorithm for artificial neural networks. Instead of just adjusting the weights in a network of fixed topology, Cascade-Correlation begins with a minimal network, then automatically trains and adds new hidden units one by one, creating a multi-layer structure. Once a new hidden unit has been added to the network, its input-side weights are frozen. This unit then becomes a permanent feature-detector in the network, available for producing outputs or for creating other, more complex feature detectors. The Cascade-Correlation architecture has several advantages over existing algorithms: it learns very quickly, the network determines its own size and topology, it retains the structures it has built even if the training set changes, and it requires no back-propagation of error signals through the connections of the network.

This research was sponsored in part by the National Science Foundation under Contract Number EET-8716324 and by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under Contract F33615-87-C-1499 and monitored by: Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, OH 45433-6543.

1. Why is Back-Propagation Learning So Slow?

The Cascade-Correlation learning algorithm was developed in an attempt to overcome certain problems and limitations of the popular back-propagation (or "backprop") learning algorithm [Rumelhart, 1986]. The most important of these limitations is the slow pace at which backprop learns from examples. Even on simple benchmark problems, a back-propagation network may require many thousands of epochs to learn the desired behavior from examples. (An *epoch* is defined as one pass through the entire set of training examples.) We have attempted to analyze the reasons why backprop learning is so slow, and we have identified two major problems that contribute to the slowness. We call these the *step-size problem* and the *moving target problem*. There may, of course, be other contributing factors that we have not yet identified.

1.1. The Step-Size Problem

The step-size problem occurs because the standard back-propagation method computes only $\partial E/\partial w$, the partial first derivative of the overall error function with respect to each weight in the network. Given these derivatives, we can perform a gradient descent in weight space, reducing the error with each step. It is straightforward to show that if we take infinitesimal steps down the gradient vector, running a new training epoch to recompute the gradient after each step, we will eventually reach a local minimum of the error function. Experience has shown that in most situations this local minimum will be a global minimum as well, or at least a "good enough" solution to the problem at hand.

In a practical learning system, however, we do not want to take infinitesimal steps; for fast learning, we want to take the largest steps that we can. Unfortunately, if we choose a step size that is too large, the network will not reliably converge to a good solution. In order to choose a reasonable step size, we need to know not just the slope of the error function, but something about its higher-order derivatives—its curvature—in the vicinity of the current point in weight space. This information is not available in the standard back-propagation algorithm.

A number of schemes have been proposed for dealing with the step-size problem. Some form of "momentum" [Rumelhart, 1986] is often used as a crude way of summarizing the slope of the error surface at earlier points in the computation. Conjugate gradient methods have been explored in the context of artificial neural networks by a number of researchers [Watrous, 1988, Lapedes, 1987, Kramer, 1989] with generally good results. Several schemes, for example [Franzini, 1987] and jacobs:dynamic, have been proposed that adjust the step-size dynamically, based on the change in gradient from one step to another. Becker and LeCun [Becker, 1988] explicitly compute an approximation to the second derivative of the error function at each step and use that information to guide the speed of descent.

Fahlman's quickprop algorithm [Fahlman, 1988] is one of the more successful algorithms for handling the step-size problem in back-propagation systems. Quickprop computes the $\partial E/\partial w$ values just as in standard backprop, but instead of simple gradient descent, Quickprop uses a second-order method, related to Newton's method, to update the weights. On the learning benchmarks we have collected, quickprop consistently out-performs other backprop-like algorithms, sometimes by a large factor.

Quickprop's weight-update procedure depends on two approximations: first, that small changes in one weight have relatively little effect on the error gradient observed at other weights; second, that the error function with respect to each weight is locally quadratic. For each weight, quickprop keeps a copy of

$\partial E/\partial w(t-1)$, the slope computed during the previous training cycle, as well as $\partial E/\partial w(t)$, the current slope. It also retains $\Delta w(t-1)$, the change it made in this weight on the last update cycle. For each weight, independently, the two slopes and the step between them are used to define a parabola; we then jump to the minimum point of this curve. Because of the approximations noted above, this new point will probably not be precisely the minimum we are seeking. As a single step in an iterative process, however, this algorithm seems to work very well. (In practice, some complications must be added to the simplified algorithm presented here; see [Fahlman, 1988] for details.)

1.2. The Moving Target Problem

A second source of inefficiency in back-propagation learning is what we call the *moving target problem*. Briefly stated, the problem is that each unit in the interior of the network is trying to evolve into a feature detector that will play some useful role in the network's overall computation, but its task is greatly complicated by the fact that all the other units are changing at the same time. The hidden units in a given layer of the net cannot communicate with one another directly; each unit sees only its inputs and the error signal propagated back to it from the network's outputs. The error signal defines the problem that the unit is trying to solve, but this problem changes constantly. Instead of a situation in which each unit moves quickly and directly to assume some useful role, we see a complex dance among all the units that takes a long time to settle down.

Many experimenters have reported that backprop learning slows down dramatically (perhaps exponentially) as we increase the number of hidden layers in the network. In part, this slowdown is due to an attenuation and dilution of the error signal as it propagates backward through the layers of the network. We believe that another part of this slowdown is due to the moving-target effect. Units in the interior layers of the net see a constantly shifting picture as both the upstream and downstream units evolve, and this makes it impossible for such units to move decisively toward a good solution.

One common manifestion of the moving-target problem is what we call the *herd effect*. Suppose we have two separate computational sub-tasks, A and B, that must be performed by the hidden units in a network. Suppose that we have a number of hidden units, any one of which could handle either of the two tasks. Since the units cannot communicate with one another, each unit must decide independently which of the two problems it will tackle. If task A generates a larger or more coherent error signal than task B, there is a tendency for all the units to concentrate on A and ignore B. Once problem A is solved, redundantly, the units can then see task B as the only remaining source of error. However, if they all begin to move toward B at once, problem A reappears. In most cases, the "herd" of units will eventually split up and deal with both of the sub-tasks at once, but there may be a long period of indecision before this occurs. The weights in a backprop network are given random initial values to prevent all of the units from behaving identically, but this initial variability tends to dissipate as the network is trained.

One way to combat the moving-target problem is to allow only a few of the weights or units in the network to change at once, holding the rest constant. The cascade-correlation algorithm uses an extreme version of this technique, allowing only one hidden unit to evolve at any given time. It might seem that by holding most of the network constant most of the time we would slow down the learning, but in the cases we have tested this strategy actually allows the network to learn faster. Once the moving-target effect is eliminated, any unit that is not frozen can quickly choose some useful role in the overall solution and then move decisively to fill that role.

2. Description of Cascade-Correlation

Cascade-Correlation combines two key ideas: The first is the *cascade architecture*, in which hidden units are added to the network one at a time and do not change after they have been added. The second is the learning algorithm, which creates and installs the new hidden units. For each new hidden unit, we attempt to maximize the magnitude of the *correlation* between the new unit's output and the residual error signal we are trying to eliminate.

The cascade architecture is illustrated in Figure 1. It begins with some inputs and one or more output units, but with no hidden units. The number of inputs and outputs is dictated by the problem and by the I/O representation the experimenter has chosen. Every input is connected to every output unit by a connection with an adjustable weight. There is also a *bias* input, permanently set to +1.

The output units may just produce a linear sum of their weighted inputs, or they may employ some non-linear activation function. In the experiments we have run so far, we use a symmetric sigmoidal activation function (hyperbolic tangent) whose output range is -1.0 to +1.0. For problems in which a precise analog output is desired, instead of a binary classification, linear output units might be the best choice, but we have not yet studied any problems of this kind.

We add hidden units to the network one by one. Each new hidden unit receives a connection from each of the network's original inputs and also from every pre-existing hidden unit. The hidden unit's input weights are frozen at the time the unit is added to the net; only the output connections are trained repeatedly. Each new unit therefore adds a new one-unit "layer" to the network, unless some of its incoming weights happen to be zero. This leads to the creation of very powerful high-order feature detectors; it also may lead to very deep networks and high fan-in to the hidden units. There are a number of possible strategies for minimizing the network depth and fan-in as new units are added. We are currently exploring some of these strategies.

The learning algorithm begins with no hidden units. The direct input-output connections are trained as well as possible over the entire training set. With no need to back-propagate through hidden units, we can use the Widrow-Hoff or "delta" rule, the Perceptron learning algorithm, or any of the other well-known learning algorithms for single-layer networks. In our simulations, we use the quickprop algorithm, described earlier, to train the output weights. With no hidden units, quickprop acts essentially like the delta rule, except that it converges much faster.

At some point, this training will approach an asymptote. When no significant error reduction has occurred after a certain number of training cycles (controlled by a "patience" parameter set by the user), we run the network one last time over the entire training set to measure the error. If we are satisfied with the network's performance, we stop; if not, there must be some residual error that we want to reduce further. We attempt to achieve this by adding a new hidden unit to the network, using the unit-creation algorithm described below. The new unit is added to the net, its input weights are frozen, and all the output weights are once again trained using quickprop. This cycle repeats until the error is acceptably small (or until we give up).

To create a new hidden unit, we begin with a *candidate unit* that receives trainable input connections from all of the network's external inputs and from all pre-existing hidden units. The output of this candidate unit is not yet connected to the active network. We run a number of passes over the examples of the training set, adjusting the candidate unit's input weights after each pass. The goal of this adjustment is to maximize S,

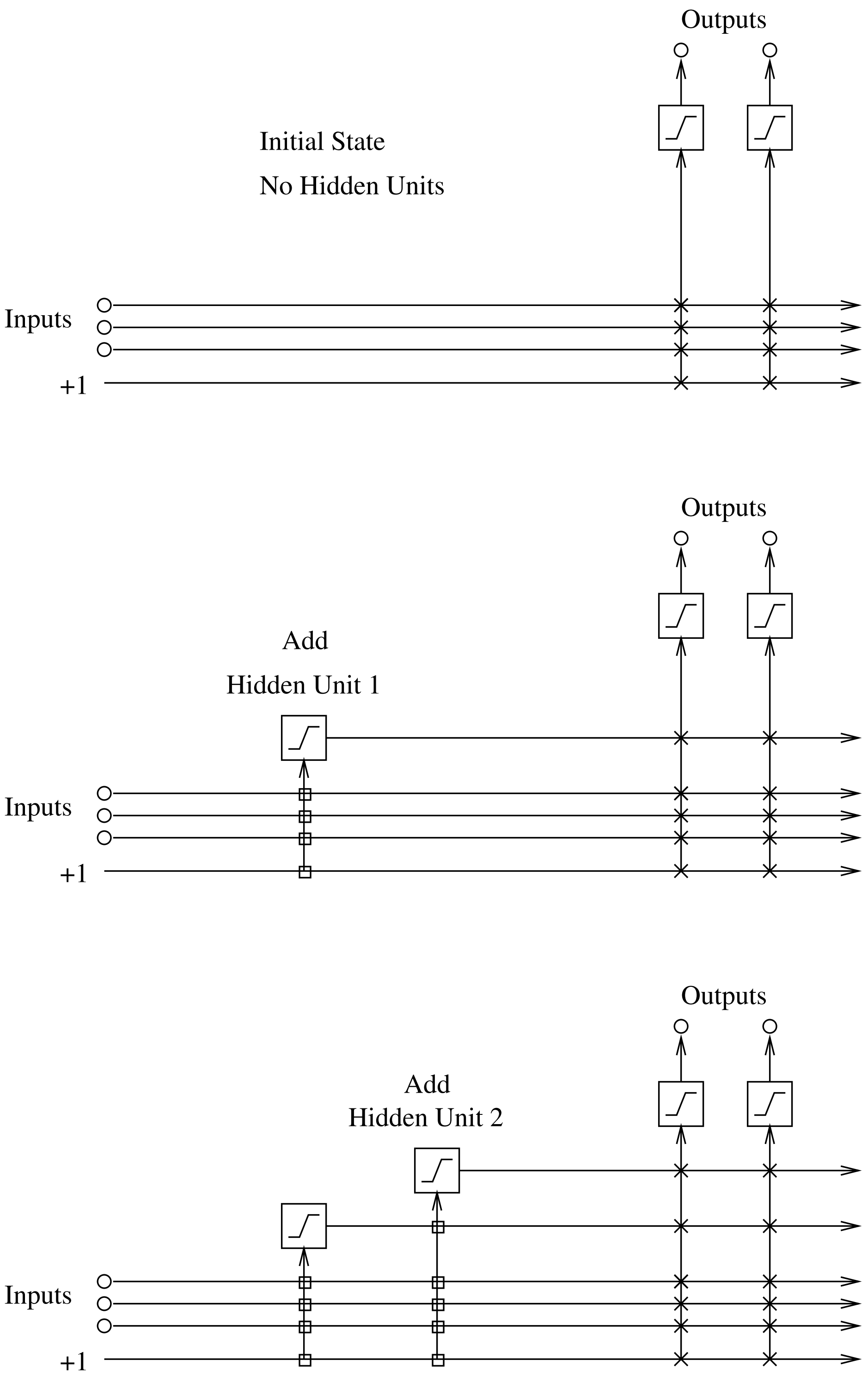

Figure 1: The Cascade architecture, initial state and after adding two hidden units. The vertical lines sum all incoming activation. Boxed connections are frozen, X connections are trained repeatedly.

the sum over all output units o of the magnitude of the correlation[1] between V, the candidate unit's value, and E_o, the residual output error observed at unit o. We define S as

$$S = \sum_o \left| \sum_p (V_p - \overline{V})\,(E_{p,o} - \overline{E_o}) \right|$$

where o is the network output at which the error is measured and p is the training pattern. The quantities $\overline{V}$ and $\overline{E_o}$ are the values of V and E_o averaged over all patterns.

In order to maximize S, we must compute $\partial S/\partial w_i$, the partial derivative of S with respect to each of the candidate unit's incoming weights, w_i. In a manner very similar to the derivation of the back-propagation rule in [Rumelhart, 1986], we can expand and differentiate the formula for S to get

$$\partial S/\partial w_i = \sum_{p,o} \sigma_o (E_{p,o} - \overline{E_o}) f'_p I_{i,p}$$

where σ_o is the sign of the correlation between the candidate's value and output o, f'_p is the derivative for pattern p of the candidate unit's activation function with respect to the sum of its inputs, and $I_{i,p}$ is the input the candidate unit receives from unit i for pattern p.

After computing $\partial S/\partial w_i$ for each incoming connection, we can perform a gradient ascent to maximize S. Once again we are training only a single layer of weights. Once again we use the quickprop update rule for faster convergence. When S stops improving, we install the new candidate as a unit in the active network, freeze its input weights, and continue the cycle as described above.

Because of the absolute value in the formula for S, a candidate unit cares only about the *magnitude* of its correlation with the error at a given output, and not about the sign of the correlation. As a rule, if a hidden unit correlates positively with the error at a given unit, it will develop a negative connection weight to that unit, attempting to cancel some of the error; if the correlation is negative, the output weight will be positive. Since a unit's weights to different outputs may be of mixed sign, a unit can sometimes serve two purposes by developing a positive correlation with the error at one output and a negative correlation with the error at another.

Instead of a single candidate unit, it is possible to use a *pool* of candidate units, each with a different set of random initial weights. All receive the same input signals and see the same residual error for each training pattern. Because they do not interact with one another or affect the active network during training, all of these candidate units can be trained in parallel; whenever we decide that no further progress is being made, we install the candidate whose correlation score is the best.

The use of this pool of candidates is beneficial in two ways: it greatly reduces the chance that a useless unit will be permanently installed because an individual candidate got stuck during training, and (on a parallel machine) it can speed up the training because many parts of weight-space can be explored simultaneously. In the simulations we have run, we have typically used rather small pools with four to eight candidate units; this was enough to ensure that we had several good candidates (almost equally good) in each pool.

[1] Strictly speaking, S is a covariance, not a true correlation, because the formula leaves out some normalization terms. Early versions of our system used a true correlation, but the version of S given here worked better in most situations.

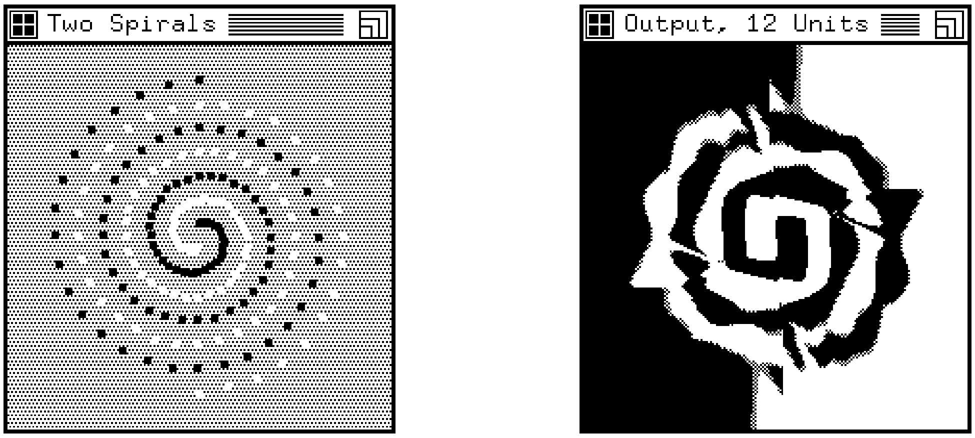

Figure 2: Training points for the two-spirals problem, and output pattern for one network trained with Cascade-Correlation.

The hidden and candidate units may all be of the same type, for example with a sigmoid activation function. Alternatively, we might create a pool of candidate units with a mixture of nonlinear activation functions—some sigmoid, some Gaussian, some with radial activation functions, and so on—and let them compete to be chosen for addition to the active network. The resulting networks, with a mixture of unit-types adapted specifically to the problem at hand, may lead to more compact and elegant solutions than are possible in homogeneous networks. To date, we have explored the all-sigmoid and all-Gaussian cases, but we do not yet have extensive simulation data on networks with mixed unit-types.

One final note on the implementation of this algorithm: While the weights in the output layer are being trained, the other weights in the active network are frozen. While the candidate weights are being trained, none of the weights in the active network are changed. In a machine with plenty of memory, it is possible to record the unit-values and the output errors for an entire epoch, and then to use these cached values repeatedly during training, rather than recomputing them for each training case. This can result in a tremendous speedup, especially for large networks.

3. Benchmark Results

3.1. The Two-Spirals Problem

The "two-spirals" benchmark was chosen as the primary benchmark for this study because it is an extremely hard problem for algorithms of the back-propagation family to solve. It was first proposed by Alexis Wieland of MITRE Corp. The net has two continuous-valued inputs and a single output. The training set consists of 194 X-Y values, half of which are to produce a +1 output and half a -1 output. These training points are arranged in two interlocking spirals that go around the origin three times, as shown in Figure 2a. The goal is to develop a feed-forward network with sigmoid units that properly classifies all 194 training cases. Some hidden units are obviously needed, since a single linear separator cannot divide two sets twisted together in this way.

Wieland (unpublished) reported that a modified version of backprop in use at MITRE required 150,000 to 200,000 epochs to solve this problem, and that they had never obtained a solution using standard backprop. Lang and Witbrock [Lang, 1988] tried the problem using a 2-5-5-5-1 network (three hidden layers of five units each). Their network was unusual in that it provided "shortcut" connections: each unit received incoming connections from every unit in *every* earlier layer, not just from the immediately preceding layer. With this architecture, standard backprop was able to solve the problem in 20,000 epochs, backprop with a modified error function required 12,000 epochs, and quickprop required 8000 epochs. This was the best two-spirals performance reported to date. Lang and Witbrock also report obtaining a solution with a 2-5-5-1 net (only ten hidden units in all), it required 60,000 quickprop epochs to train this network.

We ran the problem 100 times with the Cascade-Correlation algorithm using a sigmoidal activation function for both the output and hidden units and a pool of 8 candidate units. All trials were successful, requiring 1700 epochs on the average. (This number counts both the epochs used to train output weights and the epochs used to train candidate units.) The number of hidden units built into the net varied from 12 to 19, with an average of 15.2 and a median of 15. Here is a histogram of the number of hidden units created:

Hidden Units	Number of Trials	
12	4	####
13	9	#########
14	24	########################
15	19	###################
16	24	########################
17	13	#############
18	5	#####
19	2	##

In terms of training epochs, Cascade-Correlation beats quickprop by a factor of 5 and standard backprop by a factor of 10, while building a network of about the same complexity (15 hidden units). In terms of actual computation on a serial machine, however, the speedup is much greater than these numbers suggest, for three reasons:

- In backprop and quickprop, each training case requires a forward and a backward pass through all the connections in the network; Cascade-Correlation requires only a forward pass.
- In Cascade-Correlation, many of the training epochs are run while the network is much smaller than its final size.
- The cacheing strategy described above makes it possible to avoid repeatedly re-computing the unit values for parts of the network that are not changing.

Suppose that instead of epochs, we measure learning time in *connection crossings*, defined as the number of multiply-accumulate steps necessary to propagate activation values forward through the network and error values backward. This measure leaves out some computational steps, such as the sigmoid computations, but it is a reasonably accurate measure of relative computational cost—much more accurate than comparingd comparing epochs of different sizes or comparing runtimes on different machines.

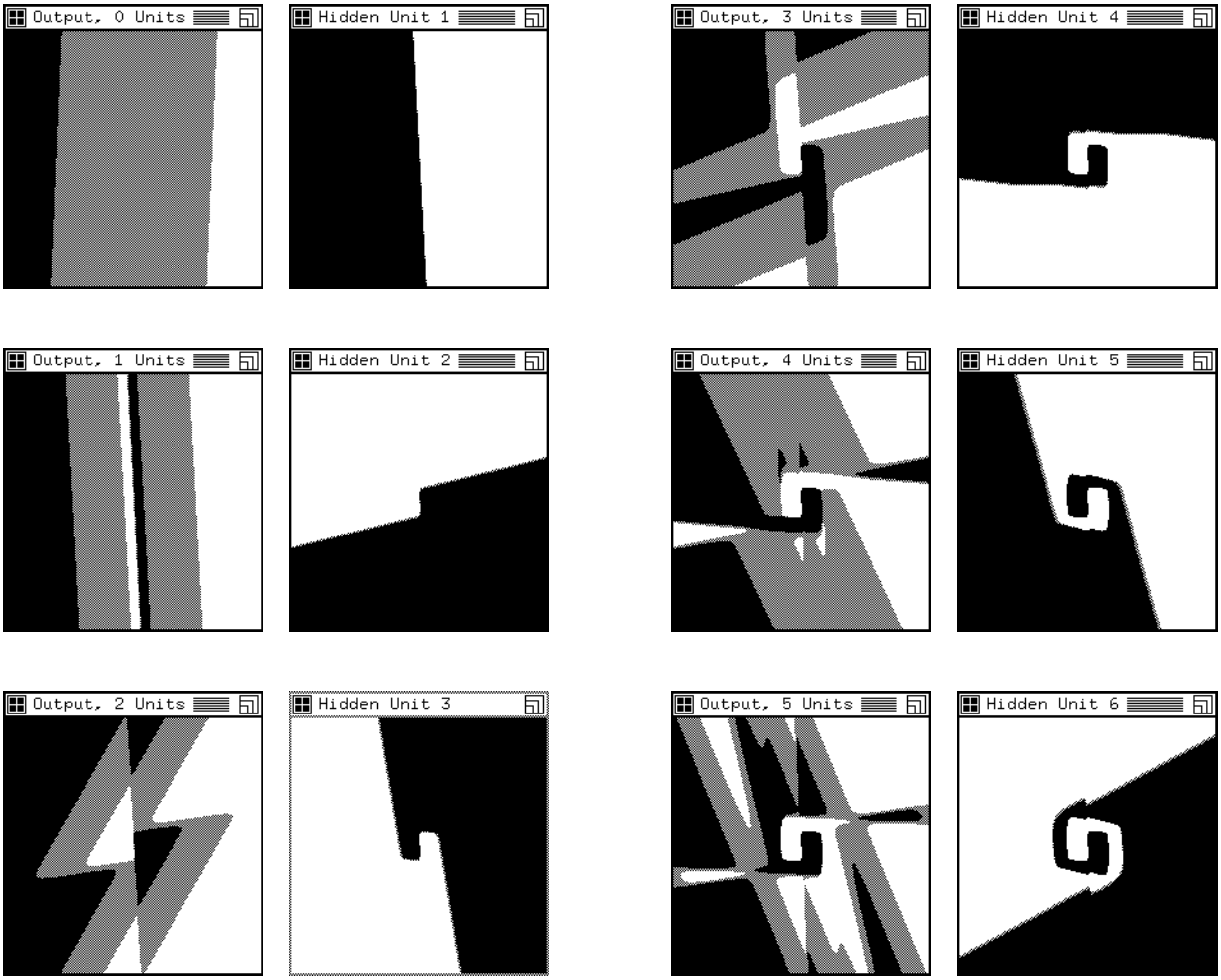

Figure 3: Evolution of a 12-hidden-unit solution to the two-spirals problem.

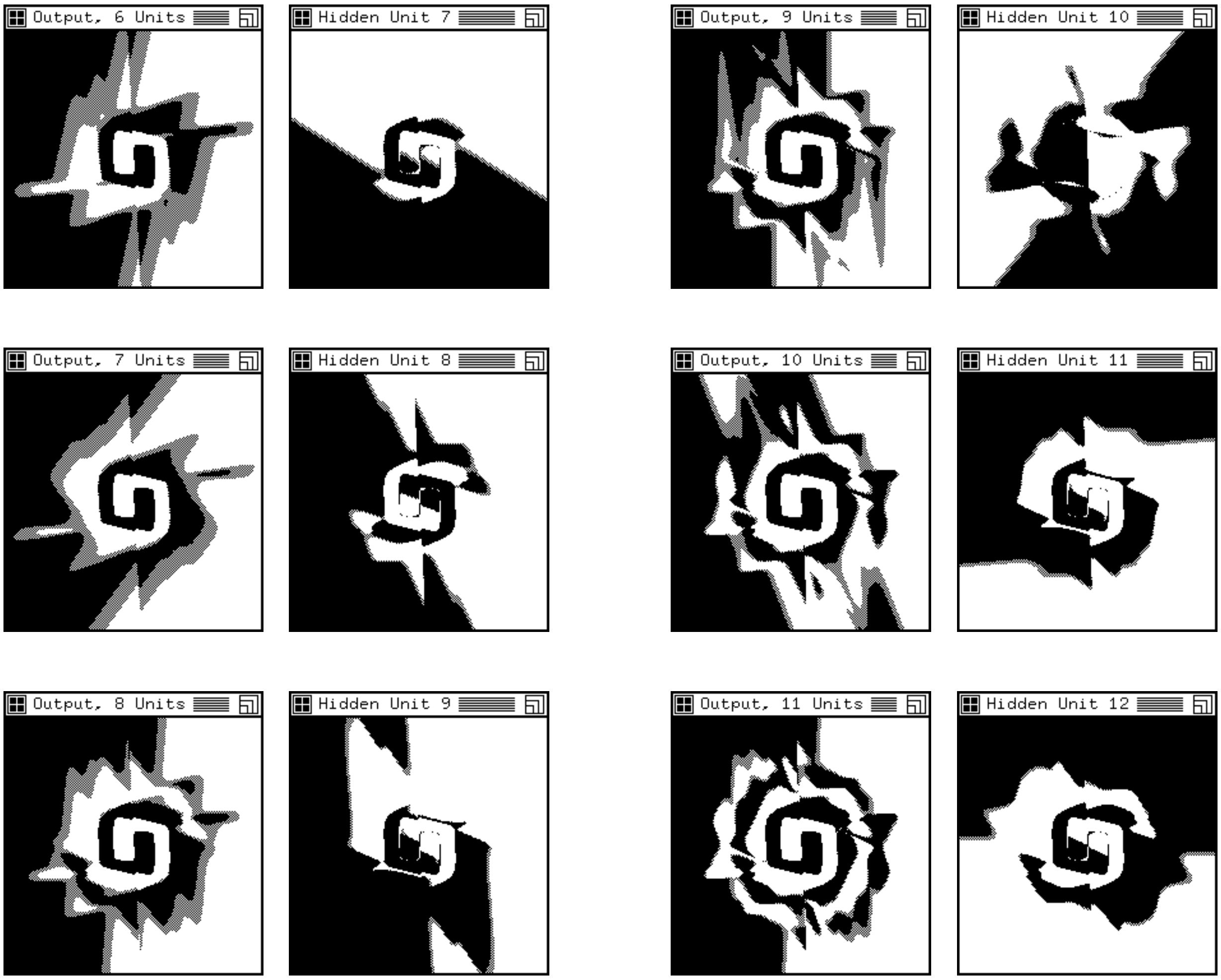

Figure 4: Evolution of a 12-hidden-unit solution to the two-spirals problem (continued).

The Lang and Witbrock result of 20,000 backprop epochs requires about 1.1 billion connection crossings. Their solution using 8000 quickprop epochs on the same network requires about 438 million crossings. An average Cascade-Correlation run with a pool of 8 candidate units requires about 19 million crossings—a 23-fold speedup over quickprop and a 50-fold speedup over standard backprop. With a smaller pool of candidate units the speedup (on a serial machine) would be even greater, but the resulting networks might be somewhat larger.

Figure 2b shows the output of a 12-hidden-unit network built by Cascade-Correlation as the input is scanned over the X-Y field. This network properly classifies all 194 training points. We can see that it interpolates smoothly for about the first 1.5 turns of the spiral, but becomes a bit lumpy farther out, where the training points are farther apart. This "receptive field" diagram is similar to that obtained by Lang and Witbrock using backprop, but is somewhat smoother.

Figures 3 and 4 show the evolution of this 12-hidden-unit network as new units are added one by one. Each pair of pictures shows the output of the network built so far, just prior to the addition of a new unit, and then the receptive field of the unit that is added. The black areas indicate that the network or unit is strongly negative, the white areas are strongly positive, and the gray areas indicate an intermediate output , close to zero.

The first six units follow the first 1.5 turns of the spirals in a very smooth and regular manner. Each new unit builds upon the earlier ones to create a receptive field that wraps farther around the origin. This very regular progression breaks down for the later units, which opportunistically grab chunks of the remaining error, though some wrapping-around of the receptive field is still visible.

3.2. N-Input Parity

Since parity has been a popular benchmark among other researchers, we ran Cascade-Correlation on N-input parity problems with N ranging from 2 to 8. The best results were obtained with a sigmoid output unit and hidden units whose output is a Gaussian function of the sum of weighted inputs. (These should not be confused with the spherical or ellipsoidal Gaussian units used by some other researchers.) Based on five trials for each value of N, our results were as follows:

N	Cases	Hidden Units	Average Epochs
2	4	1	24
3	8	1	32
4	16	2	66
5	32	2–3	142
6	64	3	161
7	128	4–5	292
8	256	4–5	357

For a rough comparison, Tesauro and Janssens [Tesauro, 1988] report that standard backprop takes about 2000 epochs for 8-input parity. Quickprop can do it in about 86 epochs. However, both of these results were obtained in a network with 16 hidden units; cascade-correlation builds a much more compact network by making effective use of shortcut connections.

As a test of generalization, we ran a few trials of Cascade-Correlation on the 10-input parity problem, training on either 50% or 25% of the 1024 patterns and testing on the rest. Note that the nearest-neighbor algorithm will do very poorly on this test, generally getting all the test cases wrong. Once again we used Gaussian hidden units. The results of these individual trials were as follows:

Train Cases	Test Cases	Hidden Units	Train Epochs	Test Errs	% Errs
512	512	4	282	9	1.8%
512	512	7	551	30	5.8%
512	512	7	491	32	6.2%
512	512	5	409	14	2.7%
256	768	4	382	111	14.4%
256	768	4	362	90	11.7%
256	768	4	276	55	7.2%
256	768	4	311	49	6.4%

4. Discussion

We believe that that Cascade-Correlation algorithm offers the following advantages over network learning algorithms currently in use:

- There is no need to guess the size, depth, and connectivity pattern of the network in advance. A reasonably small (though not optimal) net is built automatically. It may be possible to build networks with a mixture of nonlinear unit-types.
- Cascade-Correlation learns fast. In backprop, the hidden units engage in a complex dance before they settle into distinct useful roles; in Cascade-Correlation, each unit sees a fixed problem and can move decisively to solve that problem. For the problems we have investigated to date, the learning time in epochs grows *very roughly* as $N \log N$, where N is the number of hidden units ultimately needed to solve the problem.
- Cascade-Correlation can build deep nets (high-order feature detectors) without the dramatic slowdown we see in back-propagation networks with more than one or two hidden layers.
- Cascade-Correlation is useful for *incremental learning*, in which new information is added to an already-trained net. Once built, a feature detector is never cannibalized. It is available from that time on for producing outputs or more complex features. Training on a new set of examples may alter a network's output weights, but these are quickly restored if we return to the original problem.
- At any given time, we train only one layer of weights in the network. The rest of the network is not changing, so results can be cached.
- There is never any need to propagate error signals backwards through the network connections. A single residual error signal can be broadcast to all candidates. The weighted connections transmit signals in only one direction, eliminating one troublesome difference between backprop connections and biological synapses.

- The candidate units do not interact with one another, except to pick a winner. Each candidate sees the same inputs and error signals at the same time. This limited communication makes the architecture attractive for parallel implementation.

5. Relation To Other Work

The principal differences between Cascade-Correlation and older learning architectures are the dynamic creation of hidden units, the way we stack the new units in multiple layers (with a fixed output layer), the freezing of units as we add them to the net, and the way we train new units by hill-climbing to maximize the candidate unit's correlation with the residual error. The most interesting discovery is that by training one unit at a time instead of training the whole network at once, we can speed up the learning process considerably, while still creating a reasonably small net that generalizes well.

A number of researchers [Ash, 1989, Moody, 1989] have investigated networks that add new units or receptive fields within a single layer in the course of learning. While single-layer systems are well-suited for some problems, these systems are incapable of creating higher-order feature detectors that combine the outputs of existing units. For some kinds of problems, the use of higher-order features may be very advantageous.

The idea of building feature detectors and then freezing them was inspired in part by the work of Waibel on modular neural networks for speech [Waibel, 1989], but in Waibel's model the size and structure of each sub-network must be fixed by the experimenter before learning begins.

We know of only a few attempts to build up multi-layer networks in the course of training. Our decision to look at models in which each unit can see all pre-existing units was inspired to some extent by work on progressively deepening threshold-logic models by Merrick Furst and Jeff Jackson at Carnegie Mellon. (They are not actively pursuing this line at present.) Gallant [Gallant, 1986] briefly mentions a progressively deepening perceptron model (his "inverted pyramid" model) in which units are frozen after being installed. However, he has concentrated most of his research effort on models in which new hidden units are generated at random rather than by a deliberate training process. The SONN model of Tenorio and Lee [Tenorio, 1989] builds a multiple-layer topology to suit the problem at hand. Their algorithm places new two-input units at randomly selected locations, using a simulated annealing search to keep only the most useful ones—a very different approach from ours.

Acknowledgments

We would like to thank Merrick Furst, Paul Gleichauf, and David Touretzky for asking good questions that helped to shape this work.

References

[Ash, 1989] Ash, T. (1989) "Dynamic Node Creation in Back-Propagation Networks", Technical Report 8901, Institute for Cognitive Science, University of California, San Diego.

[Becker, 1988] Becker, S. and leCun, Y. (1988) “Improving the Convergence of Back-Propagation Learning with Second-Order Methods” in *Proceedings of the 1988 Connectionist Models Summer School*, Morgan Kaufmann.

[Fahlman, 1988] Fahlman, S. E. (1988) “Faster-Learning Variations on Back-Propagation: An Empirical Study” in *Proceedings of the 1988 Connectionist Models Summer School*, Morgan Kaufmann.

[Franzini, 1987] Franzini, M. A. (1987) “Speech Recognition with Back-Propagation”, Proceedings, 9th Annual Conference of IEEE Engineering in Medicine and Biology Society.

[Gallant, 1986] Gallant, S. I. (1986) "Three Constructive Algorithms for Network Learning” in *Proceedings, 8th Annual Conference of the Cognitive Science Society*.

[Jacobs, 1987] Jacobs, R. A. (1987) “Increased Rates of Convergence Through Learning-Rate Adaptation”, Tech Report COINS TR 87-117, University of Massachusetts at Amherst, Dept. of CIS.

[Kramer, 1989] Kramer, A. H. and Sangiovanni-Vincentelli, A. (1989) “Efficient Parallel Learning Algorithms for Neural Networks” in D. S. Touretzky (ed.), *Advances in Neural Information Processing Systems 1*, Morgan Kaufmann.

[Lang, 1988] Lang, K. J. and Witbrock, M. J. (1988) “Learning to Tell Two Spirals Apart” in *Proceedings of the 1988 Connectionist Models Summer School*, Morgan Kaufmann.

[Lapedes, 1987] Lapedes, A. and Farber, R (1987) “Nonlinear Signal Prediction and System Modelling”, Los Alamos National Laboratory Technical Report LA-UR-87-2662.

[Moody, 1989] Moody, J. (1989) “Fast Learning in Multi-Resolution Hierarchies” in D. S. Touretzky (ed.), *Advances in Neural Information Processing Systems 1*, Morgan Kaufmann.

[Rumelhart, 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986) “Learning Internal Representations by Error Propagation” in Rumelhart, D. E. and McClelland, J. L.,*Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, MIT Press.

[Tenorio, 1989] Tenorio, M. F., and Lee, W. T. (1989) “Self-Organizing Neural Nets for the Identification Problem” in D. S. Touretzky (ed.), *Advances in Neural Information Processing Systems 1*, Morgan Kaufmann.

[Tesauro, 1988] Tesauro, G. and Janssens, B. (1988) “Scaling Relations in Back-Propagation Learning” in *Complex Systems 2* 39-44.

[Waibel, 1989] Waibel, A. (1989) “Consonant Recognition by Modular Construction of Large Phonemic Time-Delay Neural Networks” in D. S. Touretzky (ed.), *Advances in Neural Information Processing Systems 1*, Morgan Kaufmann.

[Watrous, 1988] Watrous, R. L. (1988) “Learning Algorithms for Connectionist Networks: Applied Gradient Methods of Nonlinear Optimization’s, Tech Report MS-CIS-88-62, University of Pennsylvania, Dept. of CIS.