Training Product Unit Neural Networks with Genetic Algorithms

David J. Janson and James F. Frenzel, University of Idaho

PRODUCT UNIT NEURAL NETWORKS ARE USEFUL BECAUSE THEY CAN HANDLE HIGHER ORDER COMBINATIONS OF INPUTS. TRAINED USING TRADITIONAL BACKPROPAGATION, HOWEVER, THEY ARE OFTEN SUSCEPTIBLE TO LOCAL MINIMA. USING GENETIC ALGORITHMS CAN HELP.

TRADITIONAL NEURAL NETworks contain multiple layers of simple summation units, where each input is multiplied by a weight and then summed. Typically, this summation is then squashed by a nonlinear equation such as a logistic function. Given enough summation units, such networks can approximate any function to any arbitrary degree of accuracy.[1] However, many functions are complicated enough that the number of summation units needed to duplicate them is prohibitive. One such common task is the formation of higher order combinations of inputs, such as X^2 or $X*Y$, which facilitates the construction of polynomial expressions.[2-4]

One proposed solution is the *sigma-pi unit*,[5] in which a weight is applied not only to each input but also to the second and possibly higher order products of the inputs. While this is much more powerful than the traditional summation unit, the number of weights increases rapidly with the number of inputs, and soon becomes unmanageable when applied to large problems. Also, since many problems require only one or at most a few of these terms, the sigma-pi unit can be "overkill."

An alternative to the sigma-pi unit--called the *product unit*--computes the product of its inputs, each raised to a variable power:[6]

$$y = \prod_{i=1}^{N} X(i)^{p(i)}$$

The $p(i)$ term is treated the same as weights for summation units.

Product units are much more general than sigma-pi units. While a sigma-pi unit is constrained to using just polynomial terms, product units can use fractional and even negative terms; they can also form simple product expressions by constraining weights to integer values.

Product units can be used in a network in many ways, but the overhead required to raise an arbitrary base to an arbitrary power makes it more likely that they will supplement rather than replace summation units.[6] In this article, we use the term *product neural networks* (or product networks) to refer to networks containing both product and summation units (see Figure 1).

While product units increase a neural network's capability, they also add complications. Traditional neural networks are typically trained using backpropagation, a form of gradient descent optimization. Starting from a random point in the solution space, training data is applied to the network, and the sum of the squared error is calculated. The weights are then adjusted such that the network's operating point moves in the direction of the most negative slope.

Backpropagation works best when the solution space is relatively smooth, with few local minima or plateaus. Unfortunately, the solution space for product networks can be extremely convoluted, with numerous local minima that trap backpropagation. This is because small changes in the $p(i)$ exponent can cause large changes in the total error. (As an example, Figure 5 shows the error surface between two points in the solution space corresponding to the network in Figure 2; both figures will be discussed in more detail later.) The complexity of the solution space motivated us to investigate genetic algorithms, an alternative class of optimization methods. Because genetic algorithms do not rely on the slope for optimization, we expected that they would be more successful than traditional backpropagation at training the network.

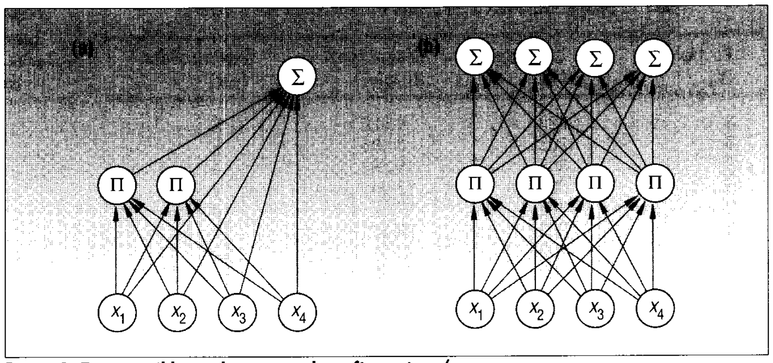
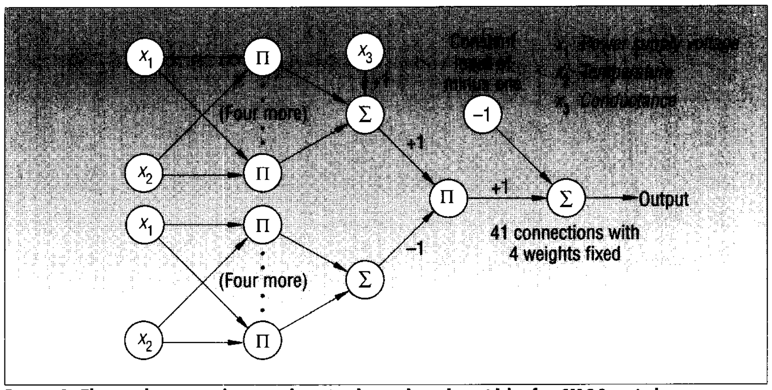


Figure 1. Two possible product network configurations.[6]



x_1
x_2
x_3
Π
Σ
(Four more)
Constant input of minus one
x_1 Power supply voltage
x_2 Temperature
x_3 Conductance
−1
+1
Output
41 connections with 4 weights fixed

Figure 2. The product neural network trained to select the width of a CMOS switch.

Genetic algorithms

A genetic algorithm is an exploratory procedure that can often locate near-optimal solutions to complex problems. To do this, the GA maintains a set of trial solutions (called *chromosomes*) and forces them to *evolve* toward an acceptable solution. A representation for possible solutions must first be developed. Then, with an initial random population, the algorithm uses "survival of the fittest" as well as old knowledge in the gene pool to improve each generation's ability to solve the problem. This improvement is achieved through a four-step process of evaluation, reproduction, breeding, and mutation.

Representation. Before applying a GA to a task, a representation for possible solutions must be found. The most common way to represent possible solutions is with a bitstring. Higher order strings (such as character strings) or trees (such as binary trees) have also been used.[7,8] Since the designer knows the architecture of the product networks to be trained, a binary string representation containing a fixed number of bits for each weight can be constructed. This representation permits each chromosome to be decoded easily, while still allowing each weight a large degree of freedom. The typical population in our experiments contained 30 to 100 members, with 32 bits representing a weight. In the network in Figure 2, there are 37 adjustable weights, resulting in a chromosome length of 1,184 bits.

Evaluation. The first step in each generation is to evaluate the current chromosomes. This is the only step where we use the interpretation of the chromosome; in all other steps, the chromosome is just treated as a bitstring. Each chromosome in the population is decoded, and the resulting network is tested with the training data. To evaluate product networks, we calculate the sum of the squared error (SSE) for the training set, with the fitness of the chromosome equal to $1/(1+\text{SSE})$. This means that the better a network performs, the higher its fitness, with a perfect network having a fitness of 1.

Reproduction. The next step in each generation is to create a new population based on an evaluation of the current one. Every chromosome generates a number of copies of itself based on its performance, with the best chromosomes producing several copies of themselves, and the worst not producing any. This is the step that allows GAs to take advantage of a survival-of-the-fittest strategy.

There are several ways to calculate the number of offspring generated by each chromosome. The most common technique is *ratioing*: Each chromosome produces a number of offspring proportional to its fitness, with the restriction that the total number of chromosomes per generation remains constant. Thus, if one chromosome's fitness is twice that of another, the superior chromosome would produce twice as many offspring. However, there are two major problems with this method. First, if all the chromosomes have a similar fitness, each member in the population will produce one offspring. This results in little pressure to improve the fitness of the population. Second, if one chromosome has a fitness much larger than any other, that chromosome will create most, if not all, of the new offspring. The chromosome will dominate the population, resulting in a loss of genetic diversity. This problem has been labeled *premature convergence*.

The method we used to train the product networks is *ranking*,[9] in which the whole population is sorted by fitness. The number

Table 1. Sample from the data points used to train the network.

Voltage (V)	Temperature (°K)	Conductance (μmho)	Desired width (μm)
3	303	1.026	2
3	303	3.806	3
3	303	6.593	4
3	303	12.04	6
3	303	17.52	8
3	303	28.51	12
3	303	39.51	16
3	303	61.52	24

Table 2. Representative output from the resulting product network, bred without a penalty function.

Expected output	Calculated output
2	9.179998
3	9.183343
4	9.186628
6	9.193188
8	9.199744
12	9.212848
16	9.225952
24	9.252160

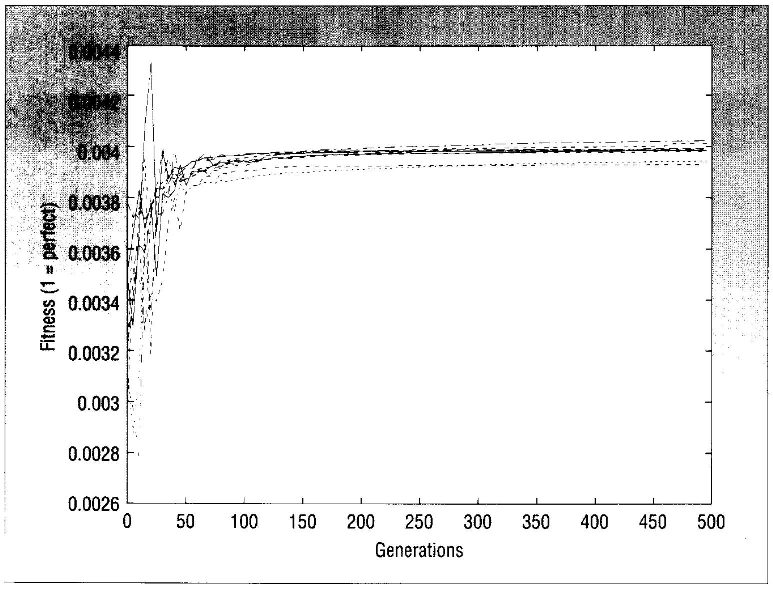


Figure 3. Ten runs with a population of 100, a mutation rate of 0.1 percent, and no penalty function.

of offspring each chromosome generates is determined by how it ranks in the population. With the ranking algorithm we used, the top 20 percent of the population generates two offspring each, the bottom 20 percent generates no offspring, and the rest generate one offspring each. No one chromosome can overpower the population in a single generation, and no matter how close the actual fitness values are, there is always pressure to improve. The primary disadvantage of ranking is speed, because better chromosomes cannot easily guide the population, forcing good answers to develop more slowly.

Breeding. The previous step creates a population whose members are currently the best at solving the problem; however, many of the chromosomes are identical, and none differ from those in the previous generation. Breeding combines chromosomes from the population and produces new chromosomes that, while they did not exist in the previous generation, maintain the same gene pool. In natural evolution, breeding and reproduction are the same step, but in GAs they have been separated to allow different methods for each to be experimented with and independently evaluated. It is during breeding that GAs can exploit knowledge of the gene pool by allowing good chromosomes to combine with chromosomes that aren't as good. This is based on the assumption that each individual, no matter how good it is, doesn't contain the answer to the problem. The correct answer is contained in the population as a whole, and can only be found by combining chromosomes.

There are several methods for breeding, the most common being *crossover*. Crossover typically swaps parts of two chromosomes to create two new ones. Many variations on crossover have been used, but there is no consensus as to which is best. We used a simple two-point crossover, in which two random points are chosen in the chromosome, and the bitstring between the two points is swapped between the two chromosomes.

Mutation. The last step in creating a new generation is based on the assumption that while each generation is better than the previous, the individuals that produce no offspring might have some information that is essential to the solution. It is also possible that the initial population didn't have all the necessary information. Thus the process of mutation reinjects information into the population. There are many ways to implement mutation, but essentially all choose and change members of the population randomly.

The method we used was to randomly distribute a constant number of mutations every generation (approximately 0.1 percent of the total number of bits in the entire population). This means that any specific chromosome might or might not mutate, with a small chance that it could mutate severely.

An application

The last decade has seen a tremendous increase in the availability of computer-aided design tools. For example, synthesis tools for digital logic can transform schematic or language circuit descriptions into VLSI circuit layouts. The number of analog computer-aided design tools has also increased; however, while many excellent analysis tools (such as Spice) are available, very few software packages can transform performance specifications into a complete circuit schematic.

CMOS circuit designers might be able to use the product network proposed by Thelen.[10] Given temperature, supply voltage, and minimum conductance as inputs, the network could calculate the optimal transistor widths for a CMOS switch. Such a tool would eliminate the typical iterative process of estimating proper component values, simulating, and redesigning. This network configuration (see Figure 2) uses existing information about the equations for modeling a CMOS switch (other applications should consider the more generic configurations suggested by Durbin and Rumelhart[6]). Four of the 41 weights are fixed, allowing the network to approximate an equation used to calculate the switch conductance. For example, the −1 weight forces the output of the lower half of the network to be treated as the denominator, whereas the node with a constant −1 input provides an offset. However, Thelen was unable to train the network using traditional backpropagation, so we selected it as a vehicle for evaluating GAs for training product networks.

We extracted our training data from several Spice simulations with differing transistor dimensions, temperatures, and power supply voltages. In the training set created from this data, the voltages ranged from 3 to 12 volts, the temperature from 303 to 403° K, and the transistor width from 2 to 20 micrometers. Using these inputs, the conductance could range from approximately 1 to 500 µmhos. Table 1 shows a sample from the 200 data points collected.

Results. The first attempts at training the product network produced consistently incorrect results. Through many runs of the GA, every solution represented a network that gave outputs of approximately 10 µm for the transistor width, with no regard for the input. Figure 3 shows, for several runs, the fitness of the best chromosome as the population evolved over 500 generations. Each run used a population of 100 chromosomes and a mutation rate of 0.1 percent. Table 2 shows the output from a product network found by one of these 10 runs.

These initial results surprised us. The GA's inability to find an appropriate solution meant that either the network could not solve the problem, or that the real solution to the problem was extremely difficult to find. Previous work by Thelen showed that a solution to this problem did indeed exist.[10] This meant that the real solution must be difficult for the GA to find.

The first success came when we seeded the population with an approximation to the solution, which we derived with a curve-fitting program using the training data. When seeded, the GA quickly improved the approximation and found a network that gave the desired output. While seeding verified that there was a correct answer and that the GA could find it, we wanted the GA to be able to find the answer using an initial random population.

There are three ways to make a problem difficult for a GA to solve:

- the solution space misleads the GA,
- the solution space is extremely convoluted, or
- the best solution occupies a very small portion of the solution space.

Since it is hard to prove whether a GA is being misled, we considered the other two possibilities. Comparing the solutions found in different runs of the GA showed that they converged to the same answer each time. If the solution space were extremely convoluted, we would have found many different solutions; thus, we rejected this possibility.

The third possible problem for GAs occurs when *local* minima occupy so much of the search space that the best solution is almost impossible to find. We can correct for this by adding a penalty function, which decreases a chromosome's fitness by adding constraints to the solution. The penalty we used to train the product network added a value to a chromosome's error based on how close the output of two consecutive data points were. The closer the two outputs for the two points, the larger the penalty. Figure 4 shows how the GA trained with the addition of this penalty, keeping all the other GA parameters the same as before. Table 3 shows the output from the product network found by the best run.

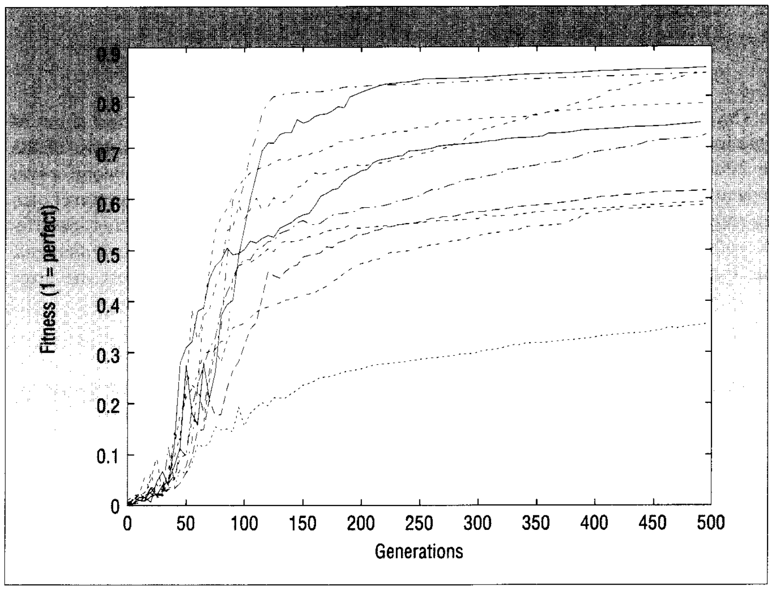


Figure 4. Ten runs with a population of 100, a mutation rate of 0.1 percent, and a penalty function.

Table 3. Representative output from the resulting product network, bred with a penalty function.

Expected output	Calculated output
2	1.959561
3	2.804309
4	4.101839
6	5.501904
8	7.928023
12	12.300184
16	14.906841
24	23.877340

The fact that the GA could find a correct solution using a penalty function leads us to believe that an incorrect answer did indeed dominate the solution space. Unfortunately, because there are 37 free weights, the solution space is 37-dimensional, making visualization of the error surface very difficult. However, we can view slices of the error surface by holding most of the weights fixed and varying only one or two weights. Alternatively, we can travel on a straight line through the solution space by incrementing the weights by fixed amounts. For example, Figure 5 shows the error surface along the line between the local minimum and the global minimum. The position labeled "local minimum" is the solution found by the GA without penalty, and the position labeled "global minimum" is the solution found by the GA with the aid of the penalty function. Several important regions on the error surface help explain the GA's behavior. Near the center of the figure is a large flat region, bounded by high peaks. The error of this region is large enough that the GA tends to discount it early in the evolution and avoids becoming trapped. The areas to the left and right of this region are much more interesting in determining why the GA cannot find the correct solution.

On the left is an *area of attraction* for the local minimum, and on the right is one for the global minimum. The error for the area of attraction around the global minimum is about 2.5 times greater than that for the local minimum. In fact, there is only a very small region about the global minimum where the error is lower than that around the local minimum. This explains why it was difficult for the GA to locate the global minimum unless the population was seeded with a solution close to that region. However, this is a one-dimensional view of a 37-dimensional space; additional features might exist off of the line between the two minima.

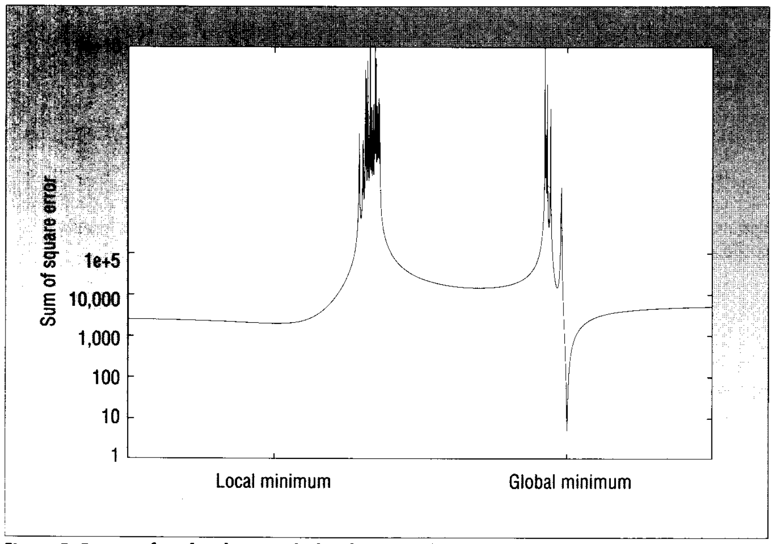


Figure 5. Error surface for the straight line between the two minima, with no penalty.

Figure 5 gives us valuable insight into the error surface for our network, but how does adding the penalty function change this surface? Figure 6 shows the error surface between the same two points with the penalty function added. As we expected, the error surface around our global minimum has changed very little; the penalty was specifically designed not to affect the desired solution. However, the surface around the local minimum is completely changed. The GA can ignore this local minimum and continue searching for the best solution.

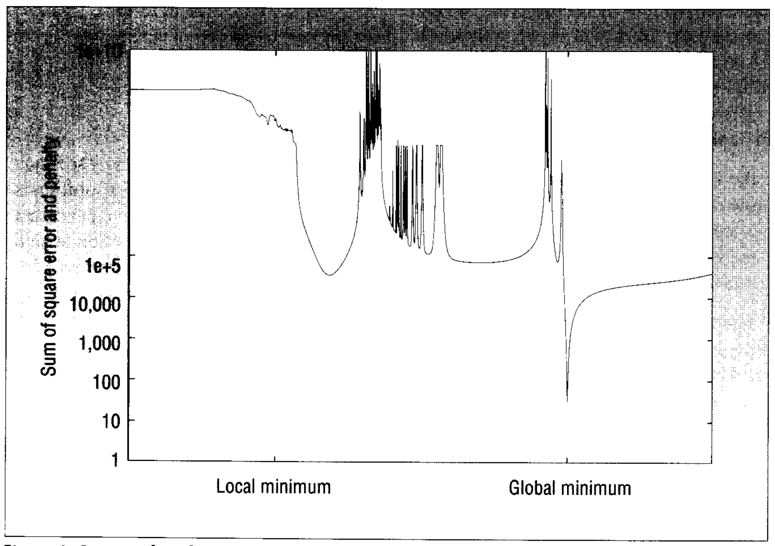


Figure 6. Error surface for the straight line between the two minima, with penalty added.

This illustrates one possible problem with GAs: They are often used when the solution space is not well known, and suboptimal answers can dominate the solution space. In this example, the penalty function distorts the solution space by placing a pole in the middle of the unwanted solution, thus allowing the GA to continue searching without the distraction of this particular local minimum. However, a penalty function is added only after the GA has tried to find the optimal answer, so it is designed to discourage unwanted answers. Consequently, penalty functions are problem specific, and their usefulness depends on the features of the particular solution space.

Parameter sensitivity. Next we explored the sensitivity of the GA's performance to several of its parameters. Figure 7 shows 30 runs of the GA with a population of 100 and a mutation rate of 0.1 percent. Figure 8 shows the same parameters except with a population of 30. Both settings were run for 500 generations. However, because the population of 100 has 10/3 more individuals than the population of 30, it does 10/3 more computations to run the same number of generations. To make a fair comparison based on computational complexity, the performance of the larger population should be compared to the smaller one after 150 generations.

From these figures, we see that each run has two stages of evolution: a period of rapidly increasing fitness, followed by a period of lesser improvement. These periods correspond directly to the population's diversity. At first there are significant differences between individuals within the population; then, as the GA converges to an answer, this diversity is lost and the population becomes homogeneous. Also, the larger population maintains its diversity longer and finds better solutions. The smaller population seems to become homogeneous quickly, and from that point on improvement is slow, driven primarily by mutation. However, the smaller population arrives at better answers much faster than the larger population. This indicates that large populations are less influenced by good potential solutions early on, analogous to having a greater degree of inertia. Consequently, there is a tradeoff between the convergence rate and the fitness of the final answer.

Because smaller populations must rely on mutation to compensate for a smaller gene pool, it is natural to ask, What happens if we increase the mutation rate? Figure 9 shows 30 runs of the GA with a population of 100 and the mutation rate increased to 1 percent. Figure 10 also shows 30 runs of the GA and a 1-percent mutation rate, but with a population of 30. By comparing these plots with the previous two, we see that a higher mutation rate is disruptive to larger populations, but beneficial to smaller populations that lack genetic diversity.

Comparison to backpropagation. For comparison purposes, we also used backpropagation to train the network. As mentioned earlier, backpropagation is the traditional method for training neural networks, relying on gradient descent to minimize the network error. Figure 11 is a histogram of the results of 50 runs using backpropagation with random initial starting points. The sum of the squared error for even the

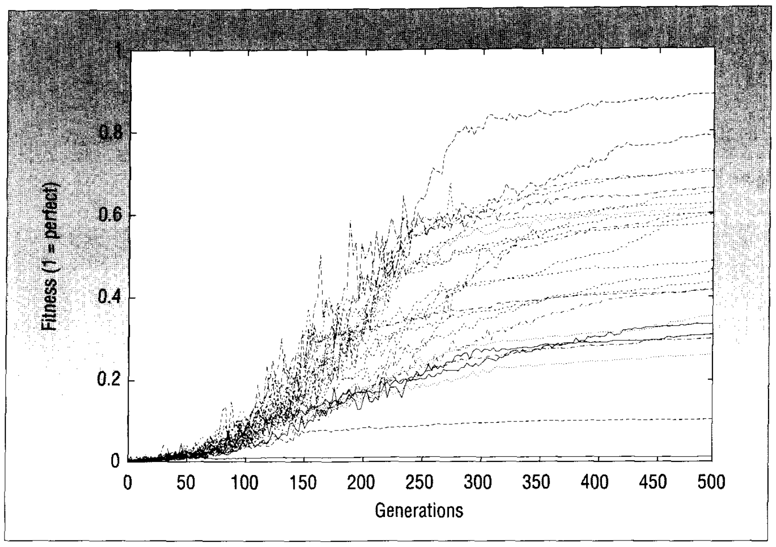


Figure 7. Thirty runs with a population of 100 and a 0.1-percent mutation rate.

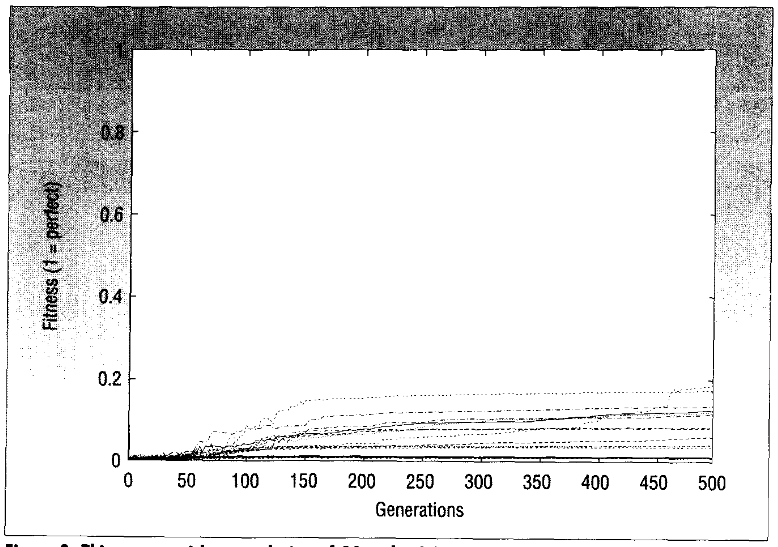


Figure 8. Thirty runs with a population of 30 and a 0.1-percent mutation rate.

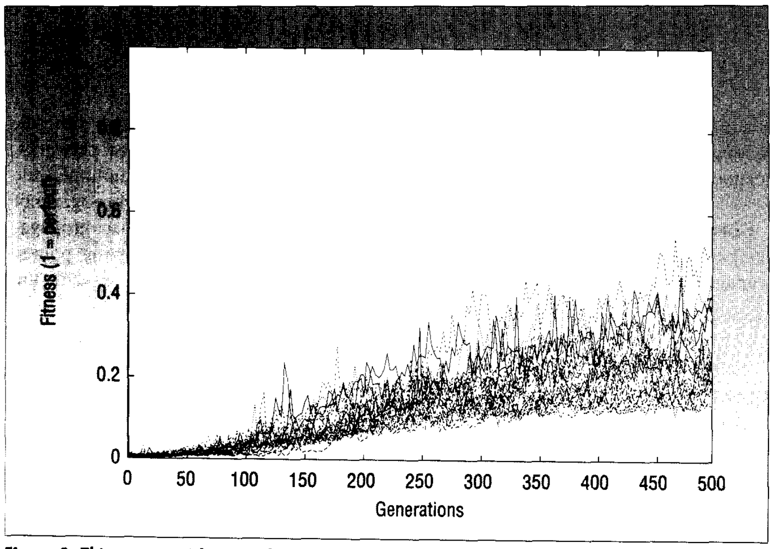


Figure 9. Thirty runs with a population of 100 and a 1-percent mutation rate.

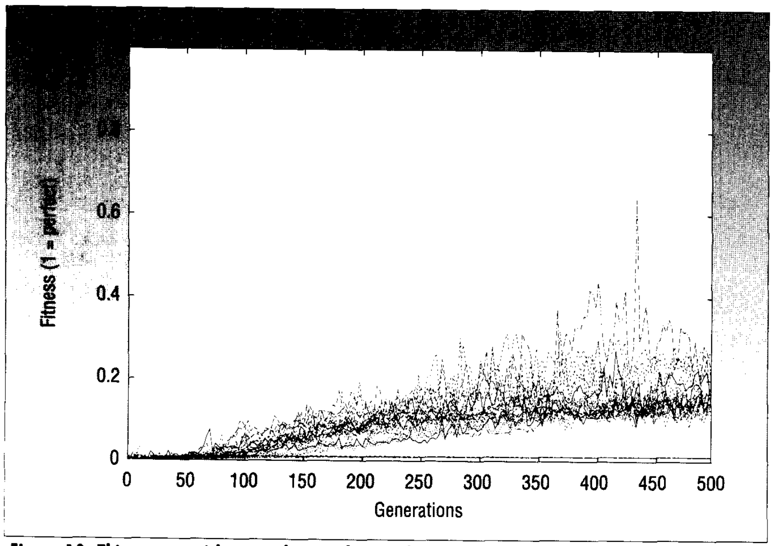


Figure 10. Thirty runs with a population of 30 and a 1-percent mutation rate.

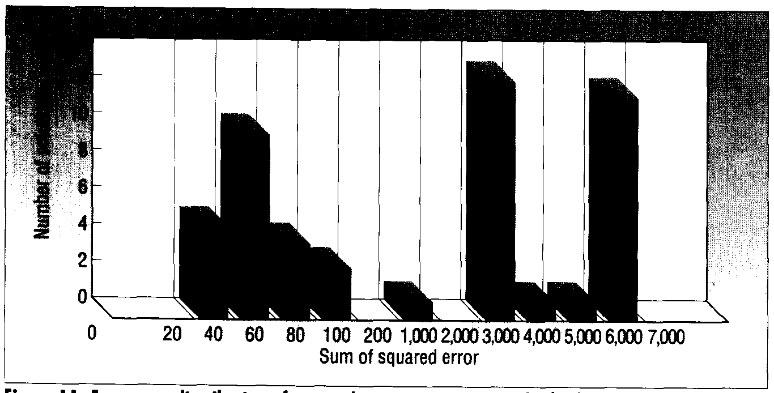


Figure 11. Frequency distribution of network error using training by backpropagation.

best solutions was 5 to 20 times worse than the global answer found using the GA. This indicates that there are several, if not many, local minima that cause backpropagation to get trapped. Further analysis of the solutions revealed at least a dozen different local minima. These results were anticipated based on the features observed in Figure 5. While it is possible for local minima to occur with conventional neural networks, they are particularly prevalent in networks containing product nodes, due to the effect of exponentiation. This characteristic supports the decision to use GAs for training these networks.

RATHER THAN RELYING SOLELY on backpropagation or GAs, a better method might be to combine the two. If we used GAs to find the area of the best solution, we would avoid many (if not all) problems with local minima. We could then use backpropagation to improve the best solutions, avoiding the problems associated with a lack of genetic diversity. This would combine the best of both worlds, and avoid the pitfalls of both. We expect that this technique can be applied to more complicated systems.

Acknowledgment

This research was supported by the National Aeronautics and Space Administration under Space Engineering Research Center Grant NAGW-1406.

References

1. G. Cybenko, "Continuous-Valued Neural Networks with Two Hidden Layers Are Sufficient," tech. report, Dept. of Computer Science, Tufts Univ., Medford, Mass., 1989.
2. K.N. Gurney, "Training Nets of Hardware Realizable Sigma-Pi Units," *Neural Networks*, Vol. 5, No. 2, 1992, pp. 289-303.
3. M. Lee, S.Y. Lee, and C.H. Park, "Neural Controller of Nonlinear Dynamic Systems Using Higher Order Neural Networks," *Electronics Letters*, Vol. 28, No. 3, Jan. 30, 1992, pp. 276-277.
4. P.F. Wyard and C. Nightingale, "A Single-Layer Higher Order Neural Net and Its Application to Context-Free Grammar Recognition," *Connection Science*, Vol. 2, No. 4, 1990, pp. 347 ff.

5. D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning Internal Representations by Error Propagation," in *Parallel Distributed Processing 1*, D.E. Rumelhart and J.L. McClelland, eds., MIT Press, Cambridge, Mass., 1986, pp. 318-362.
6. R. Durbin and D. Rummelhart, "Product Units: A Computationally Powerful and Biologically Plausible Extension to Backpropagation Networks," *Neural Computation*, Vol. 1, No. 1, Spring 1989, pp. 133-142.
7. H.J. Antonisse, "A Grammar-Based Genetic Algorithm," *Foundations of Genetic Algorithms*, G. Rawlins, ed., Morgan Kaufmann, San Mateo, Calif., 1991, pp. 193-204.
8. J. Koza, "A Hierarchical Approach to Learning the Boolean Multiplexer Problem," in *Foundations of Genetic Algorithms*, G. Rawlins, ed., Morgan Kaufmann, San Mateo, Calif., 1991, pp. 171-192.
9. D. Whitley, "The Genitor Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best," *Proc. Third Int'l Conf. Genetic Algorithms*, Morgan Kaufmann, San Mateo, Calif., 1989, pp. 116-123.
10. D. Thelen, "A Neural Network for Designing CMOS Switches: An Application for Product Units," *Proc. 1991 Joint WSU/UI Interstate Student Conf. on Neural Computation*, 1991, pp. 100-109. Available from Jack Meador, Electrical Eng. Dept., Washington State Univ., Pullman, WA 99163.

David J. Janson is a masters level graduate student at the University of Idaho. His research involves genetic algorithms, neural networks, and VLSI design. He received his BS in computer engineering from the University of Idaho in 1990 and expects his MS in 1993.

James F. Frenzel is assistant professor of electrical engineering at the University of Idaho and interim assistant director of the Microelectronics Research Center. His research interests include genetic algorithms, VLSI design and testing, computer architecture, and fault-tolerant computing. He received his BS in physics from Bucknell University in 1981, and his MS and PhD in electrical engineering from Duke University in 1983 and 1989, respectively. He is a member of IEEE.

The authors can be reached at the Dept. of Electrical Engineering, University of Idaho, Moscow, ID 83844-1023; e-mail, djanson@kz.ee.uidaho.edu or j.frenzel@ieee.org

A D V A N C E   A N N O U N C E M E N T

5th International Conference on

TOOLS WITH ARTIFICIAL INTELLIGENCE

This conference encompasses the technical aspects of specifying, designing, implementing, and evaluating tools with artificial intelligence and tools for artificial intelligence applications. The topics of interest include the following aspects:

- Machine Learning
- AI and Software Engineering
- Logic and Intelligent Database
- AI Knowledge Base Architectures
- Parallel Processing and Hardware Support
- Artificial Neural Networks
- AI Applications
- Expert Systems and Environments
- Natural Language Processing
- AI Algorithms
- Intelligent Multimedia Systems
- AI and Object-Oriented Systems
- Reasoning Under Uncertainty, Fuzzy Logic

Workshop:

Intelligent Tools & Their Applications
Chair: Nikolaos G. Bourbakis, SUNY - Binghamton
The workshop is included in the conference fee.

Keynote Addresses:

The Architecture of Intelligent Agents
Raj Reddy, Carnegie Mellon University

What is the Trend of Information Technology?
Alan Salisbury, Learning Group International

How Can Knowledge-Based Techniques Help Software Development?
Sam DiNitto, USAF Rome Laboratory

Experience with Knowledge-Based Computer-Aided Design
Steve Szygenda, University of Texas at Austin

Integrating T&E in the Acquisition Process to Reduce Cost
Raymond A. Paul, OEC, U.S. Army

Panel Sessions:

Will Symbolic AI be Replaced by Neutral Networks?
Integration of AI, Database, and Software Engineering: Research Issues, Practical Problems
Quality of Heuristic Programs
Real-Time and AI
Tools for Constraint Satisfaction
The Future Direction of AI Tools

November 8-11, 1993 • Cambridge (Boston), Massachusetts

Sponsored by IEEE Computer Society

IEEE

Register Today!

Please return registration form and fee to:

Prof. Cris Koutsougeras
Dept. of Computer Science
School of Engineering
Tulane University,
301 Stanley Thomas Hall, New Orleans, LA 70118
Phone: (504) 865-5840, E-Mail: ck@rex.cs.tulane.edu

Early registrations for the conference must be received by Monday, October 25, 1993. Payments must be remitted by check or money order in U.S. currency only, payable to TAI '93.

Conference Registration

Advance (Until 10/25/93)		Late/On Site (After 10/25/93)	
☐ Member	$300	☐ Member	$360
☐ Nonmember	$375	☐ Nonmember	$450
☐ Student	$100	☐ Student	$100

Total Enclosed: $ ________

Written requests for refunds must be received by Prof. Koutsougeras no later than Monday, October 25. Refunds are subject to a $50 processing fee. All no-show registrations will be billed in full. Students are required to show current picture ID cards at the time of registration. Registrations after ***October 25*** will be accepted on-site only.

The conference fee includes conference attendance, workshop attendance, refreshments at breaks, banquet, and one copy of the proceedings.

Name ______________________________
LAST/FAMILY FIRST MIDDLE INITIAL

Company ______________________________

Address/Mailstop ______________________________

City/State/Zip/Country ______________________________

Daytime Phone Number ______________________________

Fax Number ______________________________

E-Mail Address ______________________________

IEEE or CS Membership Number ______________________________
REQUIRED FOR MEMBER RATE