FACTA UNIVERSITATIS (NIŠ)
Ser. Math. Inform. 15 (2000), 133–146

CONSTRUCTIVE NEURAL NETWORKS DESIGN USING GENETIC OPTIMIZATION

Zoran Obradović and Rangarajan Srikumar

Dedicated to Prof. Radosav Ž. Đorđević for his 65th birthday

Abstract. A new approach for designing feedforward neural networks using genetic algorithms has been proposed. In previous attempts to design neural networks using genetic algorithms, the objective was to optimize the complete architecture at once which is a large optimization problem. In contrast, in the proposed algorithm the objective is reduced is to design a neural network through a sequence of small optimization problems by growing one hidden unit at a time, each constructed using genetic search. An additional nice feature of the proposed algorithm is that no need for additional adaptation of connections from input to the hidden layer after construction. The analysis shows that the algorithm always has a convergence property, while experiments indicate that the number of hidden units constructed by the algorithm is close to optimum which results in good generalization abilities.

1. Introduction

Research on using genetic algorithms for neural networks learning is increasing. Typically, genetic search is used for the weights optimization on a pre-specified neural network topology (for survey, see [15]). However, determining the appropriate size of a neural network is one of the most difficult tasks in its construction. An attempt to overcome the fixed architecture problem are *constructive learning algorithms* that grow or shrink the network in an application specific manner [3, 4, 5, 10]. An interesting system can

Received November 26, 1998
2000 *Mathematics Subject Classification.* 68Q70, 08A60, 08A70, 20M35

be obtained by combining an existing constructive learning algorithm and the weights optimization of neural networks using genetic algorithms [8, 13]. However, the space defined by a combination of two techniques might be difficult for the genetic search. Another previously suggested approach to neural networks design using genetic algorithms is architecture optimization only [9]. In those applications once the networks are constructed, slow gradient descent optimization using the backpropogation algorithm [18] is used for learning appropriate interconnection weights.

Genetic algorithms [7] are optimization techniques inspired by natural selection and natural genetics. They consist of a sequence of steps (called *generations*), where in every generation a better set of strings, called *species*, is created using bits and pieces of the fittest species of the old generation. In addition, an occasional random change of strings is tried. While randomized, genetic algorithms are no simple random walk. They efficiently exploit historical information to perform search on new solution candidates with expected improved performance. In this paper we use genetic search, based on the generation-by-generation optimization of a set of candidates for a next hidden layer unit, to design an appropriate neural network architecture for a given classification problem. Typically, there are four stages in the genetic search process: creation, selection, crossover, and mutation. In the *creation stage*, a set (*population*) of possible solution strings (*species*) is randomly generated. After the creation stage, each of the species is evaluated using a *fitness function* and assigned a fitness value. The fitness function must be tightly linked to the eventual goal. The usual criterion for success in pattern classification tasks is the percentage of test examples classified correctly. *Selection* eliminates bad candidates based on fitness and allows better to replicate. During *crossover*, portions of the parent species are exchanged with the hope of generating new fitter species consisting of useful parts of both parents. When all species are similar, the crossover operation loses its ability to generate new species since exchanging portions of identical strings generates the same strings. *Mutation*, a random alteration of bits of the string (with small probability), is performed on each of the new species to prevent the whole population from becoming similar.

In the evolutionary algorithm that we propose here there is no need for additional adaptation of connections from input to the hidden layer after an architecture is designed. The algorithm designs a neural network growing one hidden unit at a time, each constructed using genetic search on a relatively simple space. Each hidden unit is immediately assigned appropriate connection weights from the input layer. The pocket algorithm [5] is then

used to learn the connection weights between the hidden and the output layer. The analysis shows that the algorithm always converges, while experiments indicate that the algorithm generates small networks with good generalization ability. The drawback of this algorithm is that it is highly computation intensive when implemented on a sequential machine, which makes it inappropriate for large scale problems. In a companion paper [12], the proposed algorithm is speeded-up through parallelization of the genetic search and the pocket algorithm to the level that it is applicable to large real life problems. A preliminary version of the results from both journal manuscripts appear in [11].

In summary, the objective of this paper is to design a neural network through a sequence of small optimization problems by growing one hidden unit at a time, each constructed using genetic search. A description of the proposed algorithm is presented in Section 2, followed by the theoretical analysis in Section 3, experimental results in Section 3, and conclusions in Section 4.

2. The Evolutionary Algorithm

By Occam's razor principle, "the simplest explanation of the observed phenomena is most likely to be a correct one". So, the objective of this study is to design a small number of hidden layer units in a neural network with a single hidden layer that classifies well training set examples for a given classification problem. Then, assuming that the training set is large enough, we could expect a good generalization with high confidence.

For a given domain $D \subset \mathbb{R}^m$ we define a *region* by its bounding set of hyperplanes in $\mathbb{R}^m$. A region is said to be *resolved* if almost all training examples (high percentage) belonging to that region are of one class, otherwise the region is *unresolved*.

Let us assume that the given problem can be represented by a feedforward neural network of a single hidden layer with units computing threshold functions of their weighted input sum (usually called hard limiter units). Each hidden unit in such neural network can be interpreted as a hyperplane through the problem domain $\mathbb{R}^m$. Hyperplanes corresponding to the network's hidden units partition the domain into resolved regions. Consequently, learning goal can be interpreted as a construction of a small set of hyperplanes (corresponding to hidden layer units) which partition the training set into resolved regions.

The outline of the proposed constructive learning algorithm follows:

(1) Start from a single unresolved region of all training examples, and with a neural network without hidden units.

(2) Generate a new hidden unit (inter-connection weights and threshold) using genetic search and add it to the current network.

(3) Partition the current unresolved region(s) further with the generated unit. Discard resolved regions.

(4) Go back to step 2 until all regions are resolved.

(5) Learn hidden to output layer inter-connection weights.

Let us form pairs of training examples, each pair consisting of two examples belonging to different classes. Let A and B be such a pair of training examples. For perfect classification of training examples by a single hidden layer neural network, there must be a hyperplane corresponding to a hidden unit separating A from B. Assume that the line connecting A and B is completely covered by k disjoint *building blocks* all of the same length. For a perfect classification there must be a building block between A and B such that the hyperplane separating A from B passes through it. For large k the block is small and consequently the hyperplane can be assumed to pass through the center of it. A hyperplane in $\mathbb{R}^m$ is uniquely determined by m points. So, the equation of the hyperplane separating A from B is defined by m building blocks appropriately positioned between m pairs of training examples similar to A and B. A genetic search can be used to determine m such appropriately positioned building blocks that define a desired separating hyperplane in $\mathbb{R}^m$ (for $m = 2$, see an example on Figure 1).

In the genetic algorithm, population represents a set of hyperplanes each being a candidate for the next hidden unit that we want to add to the neural network. Each individual hyperplane is represented by a fixed length binary string. For m dimensional input space, each string consists of m concatenated substrings of equal length. Each of those substrings encodes a 4-tuple: a region, pair of a positive and a negative training examples both belonging to that region, and the index (one of k possibilities) of a building block between those two examples that defines a point on the hyperplane. This has been illustrated for a two dimensional problem in the Figure 2.

Here, in every generation of genetic search the objective is to create a better population of candidates for the next hidden unit to be added to the neural network. For that matter, although there are many variations of genetic algorithms [6], an *incremental static population model* [17] where

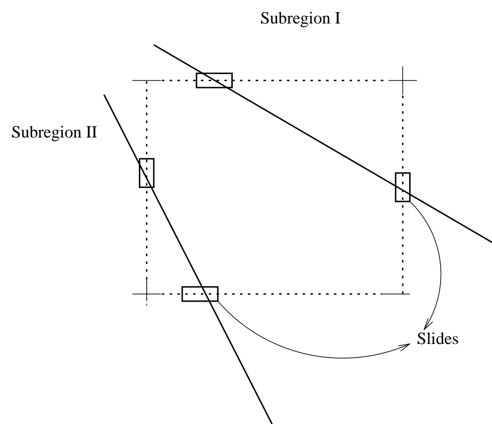


FIG. 1: The XOR function realized by the net with two hidden units. The figure shows building blocks on the dotted lines between the examples belonging to the opposite classes (between $(0,0)$ and $(0,1)$; $(0,0)$ and$(1,0)$; $(0,1)$ and $(1,1)$; $(1,0)$ and $(1,1)$). Each block can be in one of the k different positions on the dotted line.

the population is ranked according to fitness was used in this paper. The two best ranked strings from previous generation are copied into the next generation. The rest $n-2$ strings of the new generation are the result of crossover and mutation on the n strings of the previous generation. This model ensures that the best strings are not destroyed.

In each generation the hyperplanes in the population are evaluated for their fitness. If τ_i is the percentage of all training examples from class i correctly classified by the hyperplane, then its *fitness* is defined as the sum of τ_i over all classes. The crossover occurs with equal probability between any two adjacent bits.

Initially, the problem domain is a single unresolved region. Genetic search for a pre-specified number of generations is performed and the hidden unit corresponding to the generated hyperplane is added to the existing network hidden layer. This hidden unit is used to partition the unresolved regions further. All resolved regions can be ignored in future constructive steps of adding new hidden units because a set of hidden units that can classify those regions correctly is already designed. The unresolved regions are maintained in a linked list. The process continue construction of new hidden units, each using new genetic search, until all the regions are resolved.

The hidden layer units and the connections from the input to the hidden layer can be easily generated from the constructed hyperplanes. The final

00010111	00010011	000011101	00101	11111011	000000010	011011110	11010
Point 1	Slide 1	Point 2	Region 1	Point 3	Slide 2	Point 4	Region 2

FIG. 2: Representation of a hyperplane in a two dimensional problem space. Points 1 and 2, as well as points 3 and 4 are pairs of a positive and a negative example belonging to Region 1 and 2 respectively. The slide positions between the Points 1, 2 and 3, 4 are Slide1 and Slide2 respectively. In this example Point 1 and 2 are 23'rd positive and 29'th negative example belonging to 5'th region. If the value encoded in a substring representing the point (or slide, or region) is larger than the number of training examples, then modulo arithmetic is performed.

step in the algorithm is to learn the connection weights between the hidden and the output layer. This task is performed using the pocket algorithm [5], a modification of the perceptron algorithm [14] able to produce the optimal separation between non-linearly separable classes with high probability.

3. The Analysis

In this section we address questions of convergence of the proposed algorithm and quality of generated architecture. It is easy to show the following :

The Convergence Theorem: The proposed evolutionary algorithm for constructive neural network design will always converge.

Proof: With the addition of each hidden unit (hyperplane) one of the following will occur: (1) number of training examples belonging to unresolved regions decreases; (2) number of existing unresolved regions increases. Consequently, all regions will eventually be resolved since both the number of training examples belonging to unresolved regions, and the number of unresolved regions are non-negative integers upper bounded by the number of training examples. □

By the Convergence Theorem we have a procedure which constructs hyperplanes partitioning the problem domain to resolved regions only. Unfortunately, this does not guarantee that corresponding hidden layer representation is linearly separable. For example consider the XOR problem of Figure 3, where database consists of four examples: a and c are positive and b and d are negative. It is easy to see that if an algorithm first discovers separating planes 1 and 2, it should continue a search in order to

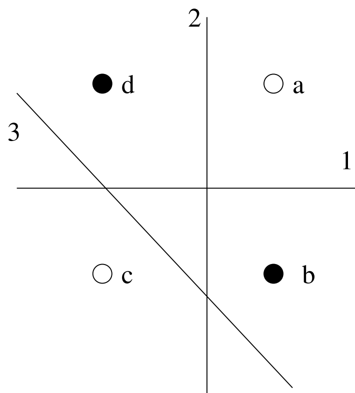


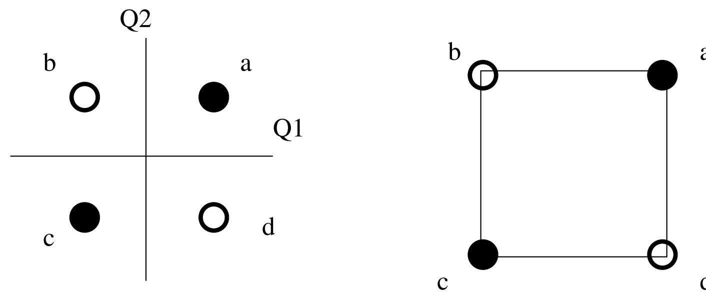
FIG. 3: If the exclusive OR problem data is first partitioned by planes 1 and 2, then for the hidden representation to be linearly separable an additional plane 3 is needed. This results in a non-minimal neural network architecture.

discover plane 3 as otherwise the constructed hidden representation will not be linearly separable. However, if the first two hyperplanes discovered by the proposed algorithm are planes Q1 and Q2 shown in Figure 4(a) the algorithm would stop too early as such a partitioning gives resolved regions only although the corresponding hidden layer representation is not linearly separable as shown in Figure 4(b) (point a will map to (1,1), point b to (0,1), point c to (0,0) and point d to (1,0)). It is important to observe that such an example is not possible in practice because the proposed algorithm will select first planes P1 and P2 shown in Figure 5(a) which results in linearly separable hidden layer representation shown in Figure 5(b). This is so since planes P1 and P2 have greater fitness values than plane P3 (50% for plane P3 versus 75% for planes P1 and P2).

So, the analysis shows that for the exclusive OR problem, the proposed algorithm discovers a minimum configuration of 2 hidden units as desired. In general, the algorithm uses genetic search with an aim to discover the best hidden unit that is added to the existing network. This results in a near-minimal number of generated hidden units even for much more complex domains as demonstrated by the experimental results of the next section.

4. Experimental Results

In the literature, one can see three types of tests for neural networks algorithms. One choice is testing on highly structured, human constructed



(a) (b)

FIG. 4: (a) The input representation for the exclusive-OR problem partitioned by planes Q1 and Q2. (b) The corresponding hidden layer representation is not linearly separable (it is identical to the input representation).

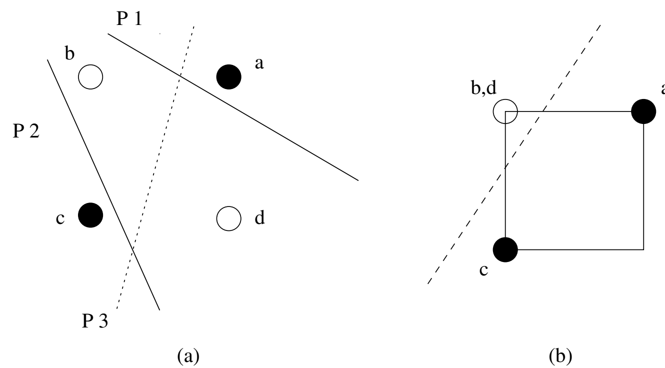


FIG. 5: (a) The separating planes for the XOR problem generated by the proposed algorithm. (b) The corresponding hidden layer representation is linearly separable and minimal.

problems such as Two spirals problem [3]. These tests are important since they can be arbitrary hard to learn, but as artificial domains they might not to have much relation to real-life pattern recognition problems of interest. The second choice is to test on natural data which is important but potentially biased as such a test usually provides insufficient test-bad for a new algorithm. The problems here include strong dependence of the learning problem on the database and non-existence of well-accepted benchmark data bases. Also, real data usually involves a number of crucial design issues (e.g. preprocessing) which makes testing effectiveness of an algorithm more dependent on the implementation. A third type of test addresses the above concerns by generating random functions computable by feed-forward

networks and testing the ability of the algorithm to learn these [1]. Note that the target function so generated is not an arbitrary random function, but it is a function that can be represented by a feedforward network of known complexity. These tests, similarly to human crafted functions, have a nice property that a training and testing set of arbitrary size, dimension and complexity is easy to construct, while also providing flexibility in testing learning for larger classes of functions.

In this paper, the algorithm is first tested on various random feedforward target networks. Second, it is tested on a real-life votes database, which has been previously used as a real-life bench-mark in machine learning literature (17 attributes, 2 classes, 435 examples - 300 training and 135 testing). Finally, the algorithm has been tested on the Monks problems which are generic problems constructed originally to test the performance of different machine learning systems including neural networks [16]. All results reported in this section are obtained by averaging over ten experiments and rounding to the nearest integer. The experiments are identical in all parameters except for the seed of the random number generator. In all experiments a region is considered to be resolved only if it is 100% pure. The intention was to test if this oversimplification will result in an overfit. However, in our experiments the obtained networks generalized quite well as demonstrated by the following results.

4.1. Random Feedforward Networks

The proposed constructive learning algorithm is first tested on random target feedforward networks with m input units, one output unit and k hidden units in a single hidden layer generated as follows. The k hidden units with threshold and connections from input layer are obtained from k hyperplanes in $\mathbb{R}^m$. Each hyperplane is generated randomly by choosing m non-collinear points and finding the equation of the plane passing through them. The output unit was also randomly initialized. The examples were drawn randomly and classified according to the target function. With this kind of test we know that the test function is actually computable on a neural network with k hidden units. Consequently if the algorithm constructs a networks with more than k hidden units we immediately know that it is of a non-optimal size.

Tests were conducted on two nets with m=2, k=3 and m=2, k=5 learning from training set of about 50 examples. In our tests the algorithm almost always learned the minimum configuration of hyperplanes required to com-

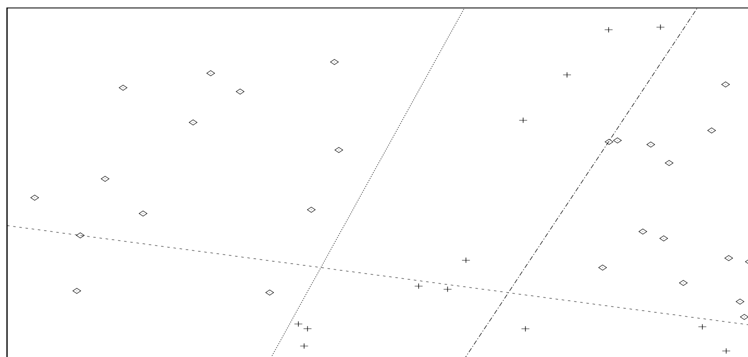


FIG. 6: Test on a randomly generated binary function computable on a neural network with three hidden units. A minimal neural network with three hidden units was discovered for this problem.

pute the target function. The output layer was then trained using the pocket algorithm. In both types of experiments average accuracy of 99% on the test set was achieved. The results are depicted in Figures 6 and 7. Experiments indicate the number of constructed hidden units is optimal or close to it.

4.2. Votes Database

This data set includes votes for each of the U.S House of Representatives congressmen on 16 key votes identified by the Congressional Quarterly Almanac (CQA) [2]. The CQA contains nine different types of votes: voted for, paired against, voted against, and announced against. This two class database consists of 435 examples each of 17 attributes, and has previously published classification results of about 90 - 95%. The proposed constructive algorithm learning from 300 training examples of this database designed a network with two hidden units. The obtained classifiers generalization performance on the remaining portion of the database was 97% correct.

4.3. Monks Problems

The Monks problems rely on an artificial domain in which robots are described by six different attributes [16]. It comprises of three different tasks. Each learning task is a binary classification problem given by a logical description of a class. Robots belong either to a particular class or not, but

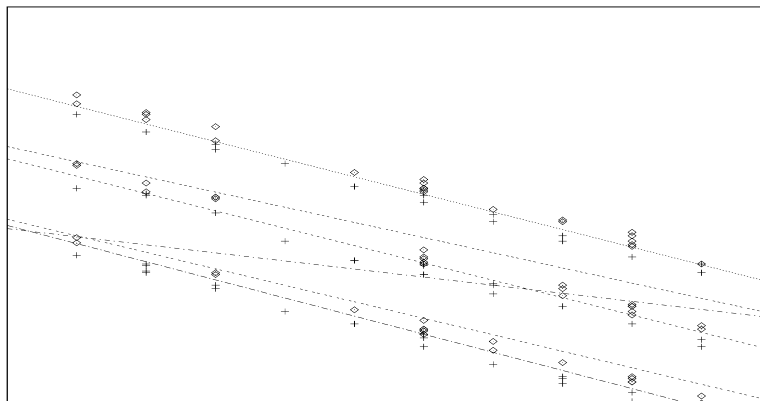


FIG. 7: A 2% noise is applied to data resulting from a randomly generated binary function computable on a neural network with five hidden units. A neural network with six hidden units was discovered for this problem.

instead of providing a complete class description to the learning problem, only a subset of 432 possible robots with their classification is given. The learning task is then to generalize using these examples. The results of the evolutionary algorithm on these problems are shown in Table 1, where the training set had 125, 170, 350 examples respectively. The proposed evolutionary design algorithm is compared to another popular constructive technique called the Tower algorithm [5], and its performance on all three problems was superior as clearly evident from the table.

Table 1: The Tower and the evolutionary algorithm comparison on the Monks problems. Here, U_{gen} is the number of units generated and P_{test} is the performance on the testing set.

Problem	**Tower**		**Evolutionary**	
	U_{gen}	P_{test}	U_{gen}	P_{test}
Monk1	6	81%	3	95%
Monk2	8	82%	8	87%
Monk3	5	83%	6	99%

Table 2: Effects of multipoint crossover performed on data generated using a random feedforward network.

Population Size	No. of X-over	Iter. to Conv.
50	1	5
50	3	1
100	1	4
100	3	3
100	4	1
1000	1	1
1000	4	1

4.4. Multipoint Crossover Effects

A test of the effects of multipoint crossover to performance of the proposed algorithm was conducted on the various sizes of population (see Table 2). These experiments used randomly generated feedforward network of Figure 6, and all converged to the same network having 100% accuracy.

The results can be explained by observing that the genetic operators that are more disruptive are more likely to create new individuals from parents with nearly identical genetic material. Smaller population sizes tend to converge faster to a homogeneity level which reduces crossover productivity. With larger population sizes the effects of multipoint crossover appear to be less important. In a small population, more disruptive multiple crossover operators may yield better results because they help to overcome the limited information capacity of the population. However, in a larger population less disruptive single point crossover operator is more likely to work better, as suggested by Holland's original analysis [7].

5. Conclusion

A new algorithm has been proposed which learns binary functions computable on single hidden layer feedforward networks. Using genetic algorithm, though tried before, has not been used for directly constructing the input to hidden layer weights as shown. In addition, the concept of finding a point on or near a classification boundary by genetic algorithms is new. On a variety of tests the proposed algorithm is shown to perform better than some well known algorithms for neural network construction.

In a genetic search species correspond to hyperplanes in the problem domain and their fitness is computed as the percentages of examples of various classes classified correctly by the hyperplane. Genetic optimization used in construction of separating hyperplanes creates hyperplanes with better generalization capability from one generation to another. Thus if a population consists of n species, in each generation the fitness will have to be evaluated for all n species. This is the most expensive step in the algorithm since in practice n can be quite large. The experiments show that in the sequential implementation of the algorithm and with n ranging between 50 to 200 more than 80% of the time is spent computing the fitness. Fortunately, the estimation of the fitness values of the species are independent of one another, and this makes it an appropriate candidate for distributed computing. In a companion paper [12], this algorithm is efficiently parallelized and thereby made applicable to large real life problems.

The proposed algorithm assumes that functions to be learned can be represented by a reasonably small single layer feedforward neural networks. The constructed hidden layer representation, although sparse, is not always linearly separable. It is an interesting open problem to further modify the proposed algorithm such as to insure linear separability of the constructed hidden layer representation.

REFERENCES

1. E. B. Baum: *What can back-propagation and k-nearest neighbor learn with feasible sized sets of examples?* Neural Networks EURASIP Workshop, 1990, pp. 2–25.
2. Congressional Quarterly Almanac, 98th Congress, 2nd session 1984, Volume XL: Congressional Quarterly Inc., Washington, D.C., 1985.
3. S.Fahlman and C. Labiere: *The cascade correlation learning architecture.* Advances in Neural Information Processing Systems, Vol. 2, Morgan Kaufmann, pp. 524–532, 1990.
4. J. Fletcher and Z. Obradovic: *A discrete approach to constructive neural network learning.* Neural, Parallel and Scientific Computations **3** (1995), 307–320.
5. S. I. Gallant: *Perceptron-based learning algorithms.* IEEE Transaction on Neural Networks **1** (1990), 179–191.
6. D. E. Goldberg: *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.
7. J. H. Holland: *Robust algorithms for adaptation set in a general formal framework.* In: Proceedings of the IEEE Symposium on Adaptive Processes, in Decision and Control, XVII, Sec 5.1–5.5, 1970.

8. N. Karunanithi *et al*: *Genetic cascade learning for neural networks.* In: Proc. Int. Workshop on Combinations of Genetic Algorithms and Neural Networks, pp. 134–145.
9. R. Keesing and D. G. Stork: *Evolution and learning in neural networks: The number and distribution of learning trials after the rate of evolution.* In: Advances in Neural Info. Processing Systems 3 (Eds. R. P. Lippman, J. E. Moody, D. S. Touretzky), San Mateo, CA, Morgan-Kaufmann, 1991, pp. 804–810.
10. S. Milenkovic, Z. Obradovic, and V. Litovski: *Annealing based dynamic learning in second-order neural networks.* In: Proc. IEEE Int. Conf. on Neural Networks, Washington D.C., 1996, pp. 458-463.
11. Z. Obradovic and R. Srikumar: *Evolutionary design of application tailored neural networks.* In: Proc. IEEE Int. Symp. on Evolutionary Computation, Orlando, FL, 1994, pp. 284–289.
12. Z. Obradovic, R. Srikumar: (in review) *Parallelizing design of application tailored neural networks.*
13. M. A. Potter: *A genetic cascade – correlation learning algorithm.* In: Proc. Int. Workshop on Combinations of Genetic Algorithms and Neural Networks, pp. 123–133.
14. F. Rosenblatt: *Principles of Neurodynamics: Perceptrons and the theory of brain mechanisms.* Washington, D.C: Spartan Press, 1961.
15. J. D. Schaffer *et al*: *Combinations of genetic algorithms and neural networks: A survey of the state of the art.* In: Proc. Int. Workshop on Combinations of Genetic Algorithms and Neural Networks, 1992, pp. 1–37.
16. S. B. Thrun *et al*: *The Monk's Problems: A performance comparison of different learning algorithms.* Carnegie Mellon University, Tech. report CMU-CS-91-197, Dec. 1991.
17. D. Whitley: *The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials in best.* In: Proc. Third Int. Conf. of Genetic Algorithms, Washington, DC, June 1989.
18. P. Werbos: *Beyond Regression: New Tools for Predicting and Analysis in the Behavioral Sciences.* Harvard University, Ph.D. Thesis, 1974; Reprinted by Willey & Sons, 1995.

School of Electrical Engineering
and Computer Science
Washington State University
Pullman, WA 99164-2752, USA

Microsoft Corporation
3219 Building 16, One Microsoft Way
Redmond, WA 98052-6399, USA