Biol. Cybern. 63, 487–493 (1990)

Biological Cybernetics

Evolving Neural Networks

D. B. Fogel, L. J. Fogel, and V. W. Porto

ORINCON Corporation, 9363 Towne Centre Dr., San Diego, CA 92121, USA

Received October 31, 1989/Accepted in revised form May 3, 1990

Abstract. Neural networks are parallel processing structures that provide the capability to perform various pattern recognition tasks. A network is typically trained over a set of exemplars by adjusting the weights of the interconnections using a back propagation algorithm. This gradient search converges to locally optimal solutions which may be far removed from the global optimum. In this paper, evolutionary programming is analyzed as a technique for training a general neural network. This approach can yield faster, more efficient yet robust training procedures that accommodate arbitrary interconnections and neurons possessing additional processing capabilities.

1 Introduction

Neural networks are parallel processing structures consisting of nonlinear processing elements interconnected by fixed or variable weights. These topologies can be constructed to generate arbitrarily complex decision regions for stimulus-response pairs, hence they are well suited for use as detectors and classifiers. Classic pattern recognition algorithms (e.g. detection, classification, target recognition) require assumptions concerning the underlying statistics of the environment. Neural networks are nonparametric and can effectively address a broad class of problems (Lippman 1987). Further, neural networks have an intrinsic fault tolerance. Some "neurons" may fail and yet the overall network can still perform well because information is distributed across all of the elements of the networks (Rumelhart and McClelland 1986). This is not possible in strictly Von Neumann architectures.

Neural network pradigms can be divided into two categories: supervised learning and unsupervised learning. In supervised learning, input data is associated with some output crtierion in a one-to-one mapping, with this mapping known a priori. This mapping is then learned by the network in a training phase. Future inputs which are similar to those in the training sample will be classified appropriately. Unsupervised learning uses the network as a self-organizing classifier. Decision regions are formed with respect to the similarity of input exemplars. No a priori clustering is given. The network adapts its outputs to minimize a function of the spacing between elements in each developed cluster. In this paper, supervised learning will be considered.

Multiple layer perceptrons, also known as feedforward networks, are typically used in supervised learning applications. Each computation node sums N weighted inputs, subtracts a threshold value and passes the result through a logistic function. An appropriate choice of logistic function provides a basis for global stability of these architectures. Single layer perceptrons (i.e., feedforward networks consisting of a single input layer) form decision regions separated by a hyperplane. If the input from the given different data classes are linearly separable, a hyperplane can be positioned between the classes by adjusting the weights and bias terms. If the inputs are not linearly separable, containing overlapping distributions, a least mean square (LMS) solution is typically generated to minimize the mean squared error between the calculated output of the network and the actual desired output. Two layer perceptrons (i.e., networks with a hidden layer of processing elements) can form unbounded arbitrary convex polytopes in the hyperspace spanned by the inputs. These regions are generated by the intersections of multiple hyperplanes and have at most as many sides as there are nodes in the hidden layer. Three layer perceptrons can form arbitrarily complex decision regions. No more than three layers of elements in perceptron networks are necessary to solve arbitrary classification mapping problems (Kolmogorov 1957).

Both continuous valued inputs and continuous valued outputs may be implemented, allowing for a wide range of input types and output categories. The input layer consists of a vector containing the input feature values to be studied. These may consist of state-space components, frequency components, pixel values, transform coefficients, or any other features considered important and representative of input exemplar data contents to be learned.

Given a network architecture, a training set of input patterns, and the associated target outputs, every set of weights and bias terms defines the output of the network to each presented pattern. The error between the actual output of the network and the target value defines a response surface over an n-dimensional hyperspace, where there are n weights and bias terms to be adapted. Training of a multi-layered network can be achieved through a back propagation algorithm (Werbos 1974; Parker 1985) which implements a gradient search over the error response surface for the set of weights which minimizes the sum of the squared error between the actual and target outputs.

This surface may contain many local minima that may be far removed from the global optimum solution. A gradient technique may lead to entrapment in these suboptimal solutions so that the network inaccurately classifies input patterns. One strategy to avoid this problem is simply to restart the optimization with a new random set of weights, in the hope that a different optimum will be found. Of course, there is no guarantee that such a minimal energy well will not also be a local solution. Another technique is to perturb the weights whenever the algorithm seems to be in a local minimum point and then continue training, but this does not guarantee that the same local solution will not be rediscovered (Rumelhart and McClelland 1986). Further, should the response surface be pocked with many local optima, the constant modification of the weights may make the gradient search technique ineffective at finding even "good" locally optimal solutions. Simulated annealing (Szu 1986) has been used with some success at overcoming local optima, but the required execution time makes this an unsatisfactory approach to many problems.

The search for an appropriate set of weights and bias terms is a complex, combinatorial optimization problem. No single parameter can be optimized without regard to all other parameters. Evolutionary programming has been used to address difficult combinatorial optimization problems such as the traveling salesman problem (Fogel 1988). In this paper, a review of evolutionary programming is offered. The efficiency of the technique for training feedforward networks is examined in two experiments. A comparison is made to back propagation. Finally, some theoretical and computational issues are addressed.

2 Evolutionary Programming

The original evolutionary programming concept (Fogel 1962, 1964; Fogel et al 1966) focused on the problem of predicting any stationary or nonstationary time series with respect to an arbitrary payoff function, modeling an unknown transducer on the basis of input-output data, and optimally controlling an unknown plant with respect to an arbitrary payoff function. Natural evolution optimizes behavior through iterative mutation and selection within a class of organisms. Behavior can be described in terms of the stimulus-response pairs that depend on the state of the organism. Each organism can be portrayed as a finite state machine (i.e., a Mealy machine), a mathematical function that does not constrain the represented transduction to be linear, passive, or without hysteresis.

The evolutionary process is simulated in the following manner: an original population of "machines" (arbitrarily chosen or given as "hints") are measured in their individual ability to predict each next event in their "experience" with respect to whatever payoff function has been prescribed (e.g. squared error, linear error, all-none, or another reasonable choice). Progeny are now created through random mutation of these "parent" machines. These offspring are scored on their predictive ability in a similar manner to their parents. Those organisms which are most suitable for achieving the task at hand are probabilistically selected to become the new parents. An actual prediction is made when the predictive-fit score demonstrates that a sufficient level of credibility has been achieved. The surviving machines generate a prediction, indicate the logic of this prediction, and become the progenitors for the next sequence of progeny, this is in preparation for the next prediction. Thus, aspects of randomness are selectively incorporated into the surviving logics. The sequence of predictor machines demonstrate phyletic learning.

Wright (1932) introduced the concept of an "adaptive landscape" or "adaptive topography" which describes the fitness of organisms. This concept has come to be a central facet of evolutionary biology. Individual genotypes are mapped to respective phenotypes which are in turn mapped onto the adaptive topography. Each peak in the topography corresponds to an optimized phenotype, and thus an optimized genetic structure. The adaptive topography is not stationary and undergoes time varying transformations corresponding to environmental variation and changes. Evolution proceeds up the slopes of the topography towards the peaks as natural selection acts in concert with reproduction, mutation, and competition.

When applying evolutionary programming to the training of neural networks, the corresponding adaptive topography is inverted, with evolution proceeding toward valleys as error is minimized. Rather than perform mutation and selection on finite state machines, the actual weight and bias terms of a network can be varied directly. A population of vectors whose components are the weight and bias terms of the network is maintained at each generation. Each vector has a corresponding error score. Those vectors with the smallest error are probabilistically selected to become parents of the next generation. Mutation is accomplished by adding a Gaussian random variable with zero mean to each component of a parent vector. The variance is made proportional to the error of the parent to simulate the effect of genetic buffering that occurs in natural evolution. As this selective random walk iterates, appropriate sets of weights and bias terms are evolved.

To give an example of how evolutionary programming moves the parent vectors over a response surface,

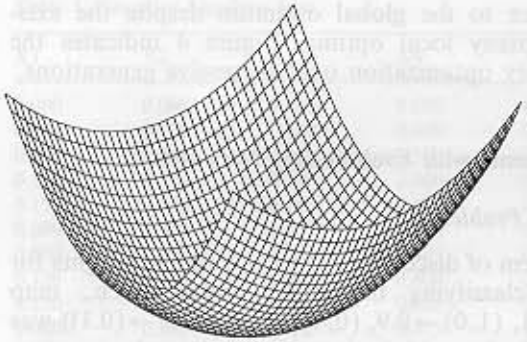

Fig. 1. $F(x, y) = x^2 + 2y^2 - 0.3\cos(3\pi x) - 0.4\cos(4\pi y) + 0.7$. $x, y \in [-50, 50]$

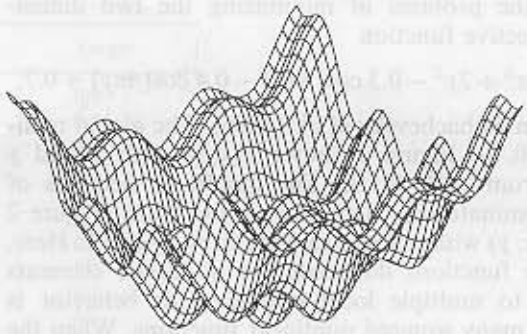

Fig. 2. $F(x, y) = x^2 + 2y^2 - 0.3\cos(3\pi x) - 0.4\cos(4\pi y) + 0.7$. $x, y \in [-1, 1]$

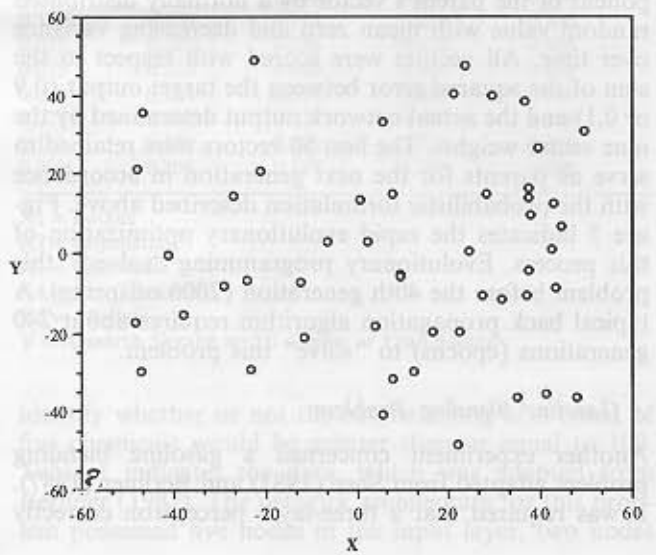

Fig. 3. **a** Distribution of parent vectors at generation #1. **b** Distribution of parent vectors at generation #5. **c** Distribution of parent vectors at generation #10. **d** Distribution of parent vectors at generation #20.

consider the problem of minimizing the two dimensional objective function

$$F(x, y) = x^2 + 2y^2 - 0.3\cos(3\pi x) - 0.4\cos(4\pi y) + 0.7,$$

taken from Bohachevsky et al. (1986). The global minimum is $(0, 0)$. Figure 1 shows $F(x, y)$ with x and y ranging from -50 to 50. The quadratic elements of $F(x, y)$ dominate the trigonometric functions. Figure 2 shows $F(x, y)$ with x and y ranging from -1 to 1. Here, the cosine functions dominate the quadratic elements and lead to multiple local optima. This behavior is typical of many squared nonlinear functions. When the estimated parameters are far from optimum, the squared error dominates; when the estimates are closer to the optimum, the nonlinearities generate a pocked surface.

Evolutionary programming was implemented to minimize $F(x, y)$. A set of 50 parent vectors were maintained at each generation. Offspring were generated by adding a Gaussian random variable with mean zero and variance equal to the objective value of the parent. Probabilistic survival was handled by forcing each vector to face 10 other vectors in competition for a "win." The probability of obtaining a win was set equal to the opponent's error score divided by the sum of both error scores. For example, if a vector with error = 1.0 faced another vector with error = 2.0, the probability of the first vector obtaining a win would be $2/(1+2) = 2/3$. This weighting is justified by viewing each vector's fitness as a mean time to obtain a given resource, with the distribution of the time being exponential. The probability of a win then becomes the probability that the vector with the lower error will obtain the resource before its competitor. The 50 vectors with the most wins are selected to be parents for the next generations. Figures 3a through 3d indicates the distribution of the parents over successive generations. There is a rapid convergence to the global optimum despite the existence of many local optima. Figure 4 indicates the evolutionary optimization over successive generations.

3 Experiments with Evolutionary Networks

The XOR Problem

The problem of discovering a set of suitable weights for correctly classifying the XOR problem (i.e., map $(0,0) \rightarrow 0.1$, $(1,0) \rightarrow 0.9$, $(0,1) \rightarrow 0.9$, $(1,1) \rightarrow (0.1)$) was considered. A multiple layer perceptron having two input nodes, a single hidden layer of two nodes, and a final output node was used. There were nine weighted connections, including the bias terms. A population of 50 parent vectors were initialized randomly, having components valued between -0.5 and $+0.5$. Offspring vectors were generated by randomly altering each component of the parent's vector by a normally distributed random value with mean zero and decreasing variance over time. All vectors were scored with respect to the sum of the squared error between the target output (0.9 or 0.1) and the actual network output determined by the nine vector weights. The best 50 vectors were retained to serve as parents for the next generation in accordance with the probabilistic formulation described above. Figure 5 indicates the rapid evolutionary optimization of this process. Evolutionary programming "solved" this problem before the 40th generation (2000 offspring). A typical back propagation algorithm requires about 240 generations (epochs) to "solve" this problem.

A Gasoline Blending Problem

Another experiment concerned a gasoline blending problem adapted from Snee (1981) and Berliner (1987). It was required that a three-layer perceptron correctly

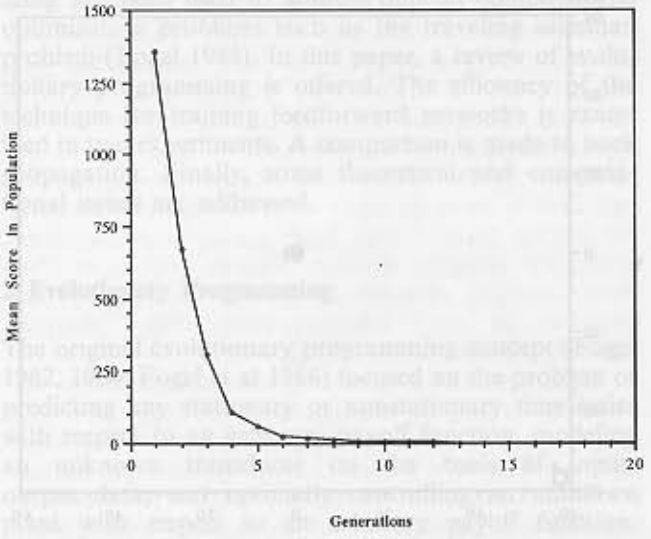

Fig. 4. Optimization of mean score in population. $F(x, y) = x^2 + 2y^2 - 0.3\cos(3\pi x) - 0.4\cos(4\pi y) + 0.7$

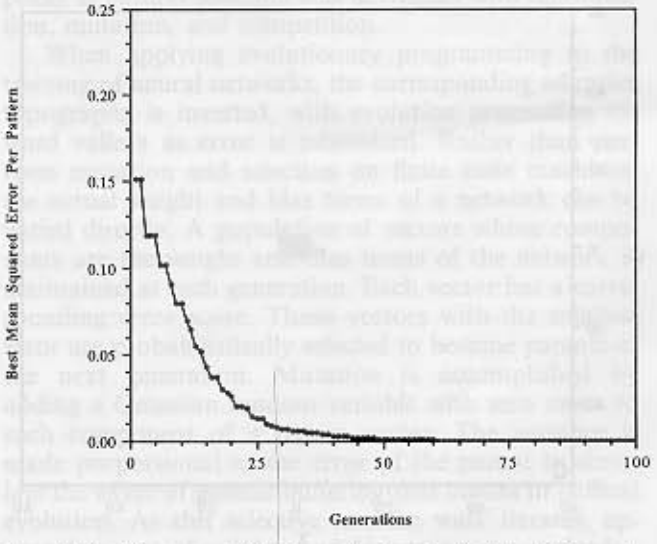

Fig. 5. Evolutionary optimization of network for the XOR problem

Table 1. Gasoline blending data

X1	X2	X3	X4	X5	Y	Target
0.000	0.000	0.350	0.600	0.060	100.0	0.9
0.000	0.300	0.100	0.000	0.600	101.0	0.9
0.000	0.300	0.000	0.100	0.600	100.0	0.9
0.150	0.150	0.100	0.600	0.000	97.3	0.1
0.150	0.000	0.150	0.600	0.100	97.8	0.1
0.000	0.300	0.049	0.600	0.051	96.7	0.1
0.000	0.300	0.000	0.489	0.211	97.0	0.1
0.150	0.127	0.023	0.600	0.100	97.3	0.1
0.150	0.000	0.311	0.539	0.000	99.7	0.1
0.000	0.300	0.285	0.415	0.000	99.8	0.1
0.000	0.080	0.350	0.570	0.000	100.0	0.9
0.150	0.150	0.266	0.434	0.000	99.5	0.1
0.150	0.150	0.082	0.018	0.600	101.9	0.9
0.000	0.158	0.142	0.100	0.600	100.7	0.9
0.000	0.000	0.300	0.461	0.239	100.9	0.9
0.150	0.034	0.116	0.100	0.600	101.2	0.9
0.068	0.121	0.175	0.444	0.192	98.7	0.1
0.067	0.098	0.234	0.332	0.270	100.5	0.9
0.000	0.300	0.192	0.208	0.300	100.6	0.9
0.150	0.150	0.174	0.226	0.300	100.6	0.9
0.075	0.225	0.276	0.424	0.000	99.1	0.1
0.075	0.225	0.000	0.100	0.600	100.4	0.9
0.000	0.126	0.174	0.600	0.100	98.4	0.1
0.075	0.000	0.225	0.600	0.100	98.2	0.1
0.150	0.150	0.000	0.324	0.376	99.4	0.1
0.000	0.300	0.192	0.508	0.000	98.6	0.1

X1 = Butane
X2 = Isopenetane
X3 = Reformate
X4 = Cat Cracked
X5 = Alkylate
Y = Research Octane at 2.0 Grams of Lead/Gallon

identify whether or not the octane rating of a blend of five chemicals would be greater than or equal to 100. Table 1 indicates the data, which was adapted from Berliner (1987). The network architecture for this problem possessed five nodes in the input layer, two nodes in the first hidden layer, five nodes in the second hidden layer, and a final output node. The target output for the network was 0.9 if the octane was greater than or equal to 100, and 0.1 otherwise. Figure 6 indicates the evolutionary optimization of the 33-component weight vectors. The fully trained network made no errors in classification. An appropriate set of weights was discovered in fewer than 100 generations. Using standard training rates and momentum terms, a typical back propagation algorithm requires between 400 to 500 generations (epochs) to "solve" this problem.

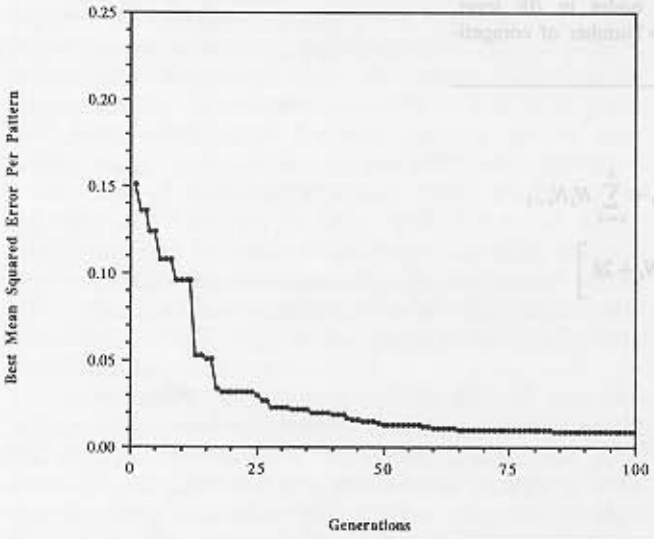

Fig. 6. Evolutionary optimization of network for the gasoline blending problem

4 Conclusions

The typical back propagation algorithm is limited to multiple layer perceptrons. Other search techniques can be constructed for various network topologies, but specific considerations must be addressed in each case. In contrast, evolutionary programming can be implemented to optimize the weighted interconnections of any generalized network. Feedback loops and feedforward loops can extend over more than one layer without computational difficulty. Each node can be given additional processing capability. There is no need to give concern to the physical realization of the network. Only the desired and actual outputs are compared with the weights being mutated randomly in accordance

with a Gaussian distribution with zero mean and variance proportional to the network's error. The class of networks that can be examined using evolutionary programming becomes much more general.

The response (energy) surfaces generated in real world neural network pattern classification problems are typically pocked with multiple optima. While a gradient technique such as back propagation is guaranteed to find locally optimal solutions if the step size tends to zero, these local solutions often fail to provide satisfactory performance on the given training set. In these cases, the entire search is restarted from a different random point or additional nodes are added to the network until the training algorithm discovers a suitable solution. But the resulting network may be severely overdefined. Any training data can be correctly classified if the network is given sufficient degrees of freedom. Such a network is unlikely to perform well on new data taken independently from the training data.

Evolutionary programming offers a parallel search which can overcome local optima. The Gaussian relationship between each parent and offspring guarantees that every combination of weights and biases can be generated. Each contending solution is probabilistically selected to become a parent in the next generation. Simulated evolution can therefore discover globally optimal sets of weights. Whereas back propagation can lead to overdetermined networks, evolutionary programming can effectively train smaller networks which may be more robust.

This versatility has some computational cost. Table 2 indicates the number of multiplies and additions required in one training iteration of a multiple layer perceptron by back propagation and evolutionary programming. On a serial computer, the evolutionary technique requires significantly more computation and execution time per iteration. Obversely, the evolutionary technique generally discovers a suitable solution in fewer iterations than back propagation. If a machine is available which can fully parallelize both algorithms then the difference in execution times becomes negligible. It should be noted that the evolutionary algorithm requires sorting out the "best" half of the population at each iteration. While this does not entail any multiplies or additions, it can take significant execution time if the population is very large (say > 1000). For typical population sizes, such as 50 parents, very little time is spent in sorting on a serial machine, and even less would be required on a parallel machine.

Evolutionary programming can be viewed as a parallel implementation of simulated annealing, although it is less greedy. In generalized simulated annealing (Bohachevsky et al. 1986; Szu 1986) steps which lead to improved solutions are always accepted. Steps which lead to poorer performance are probabilistically accepted with respect to an "annealing schedule." Simulated annealing can overcome local optima but is generally slower than evolutionary programming because it utilizes only a single point on the response surface at each iteration. By maintaining a population of candidate solutions, the response surface can be searched more efficiently and effectively.

It is important to note that payoff functions other than the typical squared error can easily be incorporated into the evolutionary algorithm. In practice, equally correct classifications are rarely of equal worth. Similarly, errors are not equally costly. Simulating natural evolution provides a paradigm for discovering an optimal set of interconnection weights which determine the appropriate network behavior in the context of a given criterion.

Table 2. Computational complexity of back propagation (BP) and evolutionary programming (EP). L = Number of layers excluding input layer, N_i = Number of nodes in ith layer, N_0 = Number of input nodes, P = Number of parents in population, k = Number of competitions per Parent

	Required execution time for one iteration on serial machine	
	Multiplies	Additions
BP	$3\sum_{i=1}^{L} N_i N_{i-1} + 3\sum_{i=1}^{L} N_i + \sum_{i=2}^{L} N_i N_{i-1}$	$3\sum_{i=1}^{L} N_i N_{i-1} + 3\sum_{i=1}^{L} N_i + \sum_{i=2}^{L} N_i N_{i-1}$
EP	$P\left[2\sum_{i=1}^{L} N_i N_{i-1} + \sum_{i=1}^{L} N_i + 3k\right]$	$P\left[2\sum_{i=1}^{L} N_i N_{i-1} + \sum_{i=1}^{L} N_i + 2k\right]$
	Required execution time for one iteration on parallel machine	
	Multiplies	Additions
BP	$\sum_{i=0}^{L-1} N_i + 2L + 1$	$\sum_{i=0}^{L-1} N_i + L + 1$
EP	$\sum_{i=0}^{L-1} N_i + 4$	$\sum_{i=0}^{L-1} N_i + 3$

References

Berliner LM (1987) Bayesian control in mixture models. Technometrics 29:455–460

Bohachevsky IO, Johnson ME, Stein ML (1986) Generalized simulated annealing for function optimization. Technometrics 28:209–218

Fogel DB (1988) An evolutionary approach to the traveling salesman problem. Biol Cybern 60:139–144

Fogel LJ (1962) Autonomous automata. Industr Res 4:14–19

Fogel LJ (1964) On the organization of intellect, Ph.D. Dissertation, UCLA

Fogel LJ, Owens AJ, Walsh MJ (1966) Artificial intelligence through simulated evolution. Wiley, New York

Kolmogorov AN (1957) On the representation of continuous functions of many variables by superposition of continuous functions of one function and addition. Dokl Akad Navk VSSR 14:953–956

Lippmann RP (1987) An introduction to computing with neural nets. IEEE ASSP Mag (April):4–22 April

Parker DB (1985) Learning logic. Technical Report TR-47. MIT Center for Computational Economics and Statistics, Cambridge, Mass

Rumelhart DE, McClelland JL (1986) Parallel distributed processing, vol I. MIT Press, Cambridge, Mass pp 423–443, 472–486

Snee RD (1981) Developing blending models for gasoline and other blends. Technometrics 23:119–130

Szu H (1986) Non convex optimization. Proceedings of the Society of Photooptical Instrumentation Engineers, 698, Real Time Signal Processing IX, 59–65

Werbos P (1974) Beyond regression: new tools for prediction and analysis in the behavioral sciences, Ph.D. Dissertation, Harvard

Wright S (1932) The roles of mutation, inbreeding, crossbreeding, and selection in evolution, Proc. 6th Int. Cong. Genetics, Ithaca, NY 1:356–366

David B. Fogel
ORINCON Corporation
9363 Towne Centre Dr.
San Diego, CA 092121
USA