GENETIC GENERATION OF BOTH THE WEIGHTS AND ARCHITECTURE FOR A NEURAL NETWORK

John R. Koza

Computer Science Department
Stanford University
Stanford, CA 94305 USA
Koza@Sunburn.Stanford.Edu 415-941-0336

James P. Rice

Stanford University Knowledge Systems Laboratory
701 Welch Road
Palo Alto, CA 94304 USA
Rice@Sumex-Aim.Stanford.Edu 415-723-8405

ABSTRACT: *This paper shows how to find both the weights and architecture for a neural network (including the number of layers, the number of processing elements per layer, and the connectivity between processing elements). This is accomplished using a recently developed extension to the genetic algorithm which genetically breeds a population of LISP symbolic expressions (S-expressions) of varying size and shape until the desired performance by the network is successfully evolved. The new "genetic programming" paradigm is applied to the problem of generating a neural network for the one-bit adder.*

1. INTRODUCTION AND OVERVIEW

Most of the various neural network architectures and training paradigms described in the literature (Hinton 1989) presuppose that the architecture of the neural network has been determined before the training process begins. That is, they presuppose that a selection has been made for the number of layers of linear threshold processing elements, the number of processing elements in each layer, and the nature of the allowable connectivity between the processing elements. In this paper, we now show how to design a neural network for both its weights and its architecture simultaneously. In particular, this paper describes the recently developed "genetic programming" paradigm which genetically breeds populations of computer programs to solve problems. In genetic programming, the individuals in the population are hierarchical compositions of functions and arguments of various sizes and shapes (i.e. LISP symbolic expressions). Each individual S-expression in the population is evaluated for its fitness in handling the problem environment and a simulated evolutionary process is driven by this measure of fitness.

2. BACKGROUND ON GENETIC ALGORITHMS

Genetic algorithms are highly parallel mathematical algorithms that transform populations of individual mathematical objects (typically fixed-length binary character strings) into new populations using operations patterned after (1) natural genetic operations such as sexual recombination (crossover) and (2) fitness proportionate reproduction (Darwinian survival of the fittest). Genetic algorithms begin with an initial population of individuals (typically randomly generated) and then iteratively (i) evaluate the individuals in the population for fitness with respect to the problem environment and (ii) perform genetic operations on various individuals in the population to produce a new population. John Holland of the University of Michigan presented the pioneering formulation of genetic algorithms for fixed-length character strings in *Adaptation in Natural and Artificial Systems* (Holland 1975). Holland established, among other things, that the genetic algorithm is a mathematically near optimal approach to adaptation in that it maximizes expected overall average payoff when the adaptive process is viewed as a multi-armed slot machine problem requiring an optimal allocation of future trials given currently available information. Recent work in genetic algorithms and genetic classifier systems can be surveyed in Goldberg (1989).

The conventional genetic algorithm operating on fixed length character strings (typically bit strings) has been successfully used to discover and optimize neural nets (Belew, McInerney and Schraudolph 1991, Miller and Todd 1989, Whitley, Starkweather and Bogart 1990, and Chalmers 1991).

3. BACKGROUND ON THE GENETIC PROGRAMMING PARADIGM

Representation is a key issue in genetic algorithm work because genetic algorithms directly manipulate the coded representation of the problem and because the representation scheme can severely limit the window by which the system observes its world. Fixed length character strings present difficulties for some problems — particularly problems where the desired solution is hierarchical and where the size and shape of the solution is unknown in advance. The need for more powerful representations has been recognized for some time.

We have recently shown that entire computer programs can be genetically bred to solve problems in a variety of different areas of artificial intelligence, machine learning, and symbolic processing. Specifically, this recently developed genetic programming paradigm has been successfully applied (Koza 1989, 1990, 1991) to example problems in several different areas, including

- machine learning of functions (e.g. learning the Boolean 11-multiplexer function),
- planning (e.g. navigating an artificial ant along an irregular trail. developing a robotic action sequence that can stack an arbitrary initial configuration of blocks into a specified order),
- automatic programming (e.g. discovering a computational procedure for solving pairs of linear equations, solving quadratic equations for complex roots, and discovering trigonometric identities),
- optimal control (e.g. centering a cart and balancing a broom on a moving cart in minimal time by applying a "bang bang" force to the cart) (Koza and Keane 1990),
- pattern recognition (e.g. translation-invariant recognition of a simple one-dimensional shape in a linear retina),
- sequence induction (e.g. inducing a recursive procedure for the Fibonacci and the Hofstadter sequences),
- symbolic "data to function" regression, integration, and differentiation, and symbolic solution to general functional equations (including differential equations with initial conditions and integral equations),
- empirical discovery (e.g. rediscovering Kepler's Third Law, rediscovering the well-known econometric "exchange equation" MV = PQ from actual, noisy time series data for the money supply, the velocity of money, the price level, and the gross national product of an economy), and
- emergent behavior (e.g. discovering a computer program which, when executed by all the ants in an ant colony, produces interesting overall "emergent" behavior).

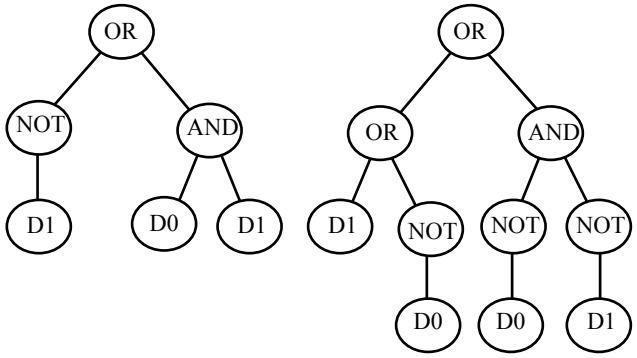
In the genetic programming paradigm, the individuals in the population are compositions of functions and terminals appropriate to the particular problem domain. The set of functions used typically includes arithmetic operations, mathematical functions, conditional logical operations, and domain-specific functions. Each function in the function set must be well defined for any combination of elements from the range of every function that it may encounter and every terminal that it may encounter. The set of terminals used typically includes inputs (sensors) appropriate to the problem domain and various constants. The search space is the hyperspace of all possible compositions of functions that can be recursively composed of the available functions and terminals. The symbolic expressions (S-expressions) of the LISP programming language are an especially convenient way to create and manipulate the compositions of functions and terminals described above. These S-expressions in LISP correspond directly to the "parse tree" that is internally created by most compilers.

The basic genetic operations for the genetic programming paradigm are fitness proportionate reproduction and crossover (recombination). Fitness proportionate reproduction is the basic engine of Darwinian reproduction and survival of the fittest and operates for genetic programming paradigms in the same way as it does for conventional genetic algorithms. The crossover operation for genetic programming paradigms is a sexual operation that operates on two parental LISP S-expressions and produces two offspring S-expressions using parts of each parent. In particular, the crossover operation creates new offspring S-expressions by exchanging sub-trees (i.e. sub-lists) between the two parents. Because entire sub-trees are swapped, this genetic crossover (recombination) operation produces syntactically and semantically valid LISP S-expressions as offspring regardless of which point is selected in either parent.

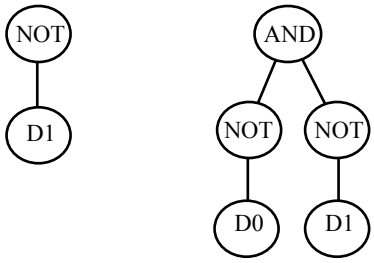
For example, consider the following two parental LISP S-expressions:

```
(OR (NOT D1) (AND D0 D1))
(OR (OR D1 (NOT D0)) (AND (NOT D0) (NOT D1)))
```

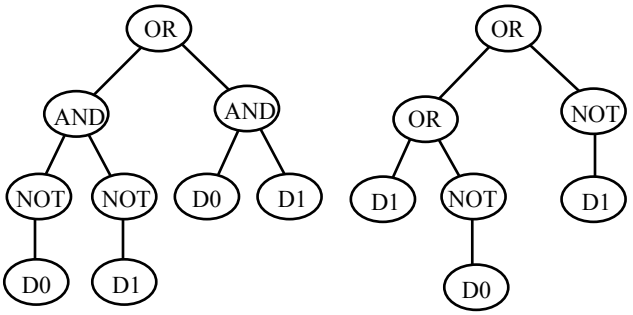
These two LISP S-expressions can be depicted graphically as rooted, point-labeled trees with ordered branches. Assume that the points of both trees are numbered in a depth-first way starting at the left. Suppose that the second point (out of 6 points of the first parent) is randomly selected as the crossover point for the first parent and that the sixth point (out of 10 points of the second parent) is randomly selected as the crossover point of the second parent. The crossover points are therefore the NOT in the first parent and the AND in the second parent, as shown below:



The two crossover fragments are two sub-trees shown below:



These two crossover fragments correspond to the bold, underlined sub-expressions (sub-lists) in the two parental LISP S-expressions shown above. The two offspring resulting from crossover are shown below.



Note that the first offspring above is a perfect solution for the odd-parity (exclusive-or) function, namely

```
(OR (AND (NOT D0) (NOT D1)) (AND D0 D1)).
```

Details on the genetic programming paradigm can be found in Koza (1990).

4. NEURAL NETWORK DESIGN

We begin by considering the set of function and terminals appropriate for constructing a neural network. One such function set suitable for constructing neural networks contains six functions. In particular, the function set is F = {P, W, +, -, *, %}. The function P is the linear threshold processing function. The function W is the weighting function

used to give a weight to a signal going into a linear threshold processing function. This weighting function W is, in fact, merely multiplication; however, we give it this special name to distinguish it from the ordinary arithmetic operation of multiplication (which is used for the entirely different purpose of creating and modifying the numerical constants in the neural network in this problem). Both the linear threshold processing function P and the weighting function W appear in the function set with a varying number of arguments (i.e. typically 2, 3, 4). The function set also contains the four arithmetic operations addition (+), subtraction (-), multiplication (*), and the protected division function (%) which returns zero in event of a division by zero. The four arithmetic operations are used to create and modify the numerical constants (weights) of the neural network.

The set of terminals (arguments) contains the input signals of the problem and the ephemeral random floating point constant atom ("R"). In this section, we focus on neural networks with two input signals. If there were two input signals (D0 and D1) for a particular problem, then the terminal set would be T = {D0, D1, R}.

The problem of designing neural networks illustrates the need for creating the initial random population so that all individuals comply with a particular set of syntactic restrictions. Not all possible compositions of the functions from the function set above and terminals from the terminal set above correspond to what we would choose to call a "neural network." Thus, the problem of designing neural networks requires rules of construction that specify what structures are allowable for this particular problem. S-expressions must be initially created in conformity to these rules of construction. Moreover, all operations that modify the individuals in the population (in particular, crossover) must be performed in a structure-preserving way.

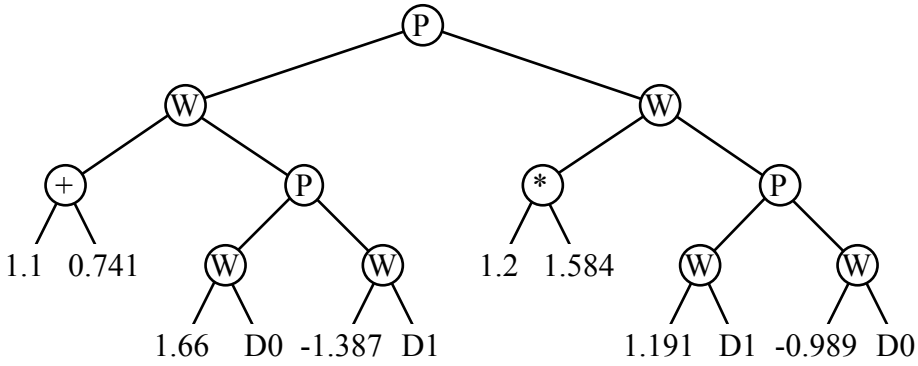
In particular, the rules of construction for neural networks with one output signal require that the root of the tree must be a linear threshold processing function P. Then, the functions at the level immediately below any linear threshold processing function P must always be weighting functions (W). At the level below a weighting function W, there is greater flexibility. In particular, there can be an input data signal (such as D0 and D1), any arithmetic function (addition, subtraction, division, or multiplication), a floating point random constant, or another P function. At the level immediately below an arithmetic operation, there can only be an arithmetic operation or a floating point random constant. Once a P function again appears in the tree, the rules of construction again require that the functions immediately below the P function again be only weighting functions W. These rules are applied recursively until a full tree is constructed. Note that the external points of the tree are either input signals (i.e. D0 or D1) or floating point random constants.

These rules of construction produce a tree (LISP S-expression) that we would call a "neural network." The structure consists of linear threshold processing elements (the "P" functions) that process weighted inputs to produce a discrete signal (i.e. 0 or 1) as its output. The number of inputs to a processing element (P function) can vary; however, the inputs are always weighted signal lines. The threshold is fixed at 1.0 for this problem. The weights may be a single floating point constant or can be the result of a composition of arithmetic operations and floating point random constants. The signal lines can be the input to the network from the outside, the output of other processing elements, or compositions of arithmetic operations and random constants. If a signal line consists only of a composition of arithmetic operations and random constants, it is called a bias.

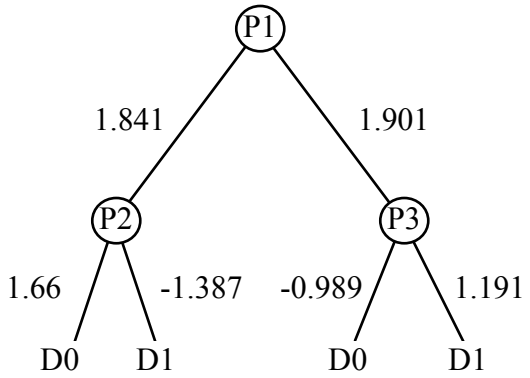
Consider the following LISP S-expression representing a neural network which l00% correctly performs the odd-parity (exclusive-or) task on the inputs D0 and D1:

```
(P (W (+ 1.1 0.741) (P (W 1.66 D0) (W -1.387 D1)))
   (W (* 1.2 1.584) (P (W 1.191 D1) (W -0.989 D0)))).
```

This S-expression can be graphically depicted as the a rooted tree below:



The root of this tree contains the linear threshold processing function P with two arguments. The functions at the level immediately below the uppermost P function are both weighting functions (W). The left argument of the left weighting function is 1.841 (created by adding the random constant 1.1 to the random constant 0.741). The right argument of the left weighting function is the output of another linear threshold processing function P with two arguments. The functions at the level immediately below this second linear threshold processing function P are both weighting functions (W). The arguments to the left weighting function of this second linear threshold processing function P are the random constant 1.66 and the input data signal D0. The arguments to the right multiplication function of this second P function are the random constant -1.387 and the input data signal D1. This S-expression can be converted to the form that one would typically see in the neural network literature, as follows:



Here the input signal D0 is weighted by 1.66 and the input signal D1 is weighted by -1.387 so that these two weighted input signals become the input to the linear threshold processing function P2 with two input lines. Since the input D0 is either 0 or 1, the first input line to P2 is either 0 or 1.66. Similarly, the second input line to P2 is either 0 or -1.387. The linear threshold processing function P2 adds up its two weighted input lines and emits a 1 if the sum exceeds the threshold of 1.0 and emits a 0 otherwise. If D0 and D1 are both 0, the sum of the inputs will be 0 (which is less than the threshold of 1) and therefore, P2 will emit 0. If D0 is 1 and D1 is 0, the sum will be 1.66 and P2 will emit a 1. If D0 is 0 and D1 is 1, the sum will be -1.387 and P2 will emit a 0. If both D0 and D1 are 1, the sum will be 0.273 . This is less than the threshold of 1.0. Thus, P2 will emit a 0. In other words, P2 emits a 1 if and only if the input lines D0 and D1 are 1 and 0, respectively. Similarly, the input signal D0 is weighted by -0.989 and the input signal D1 is weighted by 1.191 so that these two weighted input signals become the input to the linear threshold processing function P3 with two input lines. As will be seen, P3 will emit a 1 if and only if the input signals D0 and D1 are 0 and 1, respectively.

Then, the output of P2 is weighted by 1.841 and the output of P3 is weighted by 1.901 so that these two weighted input signals become the input to the linear threshold processing function P1 with two input lines. As will be seen, the effect of these weights is that the weighted sum of the inputs to P1 exceeds the threshold of 1.0 if and only if either signals coming out of P2 and P3, or both, are non-zero.In other words, the output of P1 is 1 if either (but not both) D0 or D1 are 1, and the output of P is 0 otherwise. That is, the output P1 is the odd-parity (exclusive-or) function on the inputs D0 and D1.

If there is more than one output signal from the neural network, the output signals from the neural network are returned as a LIST. That is, the function set is enlarged to F = {LIST, P, W, +, -, *, %}. The rules of construction for neural networks with more than one output signal require that the root of the tree must be a LIST function. The number of arguments to the function LIST equals the number of output signals. This is the only time the LIST function is used in the S-expression. Then, the rules of construction require that the function at the level of the tree immediately below the LIST function must be linear threshold processing functions P. Thereafter, the previously described rules of construction apply.

For example, the following LISP S-expression represents a neural network with two output signals:

```
(LIST (P (W D1 -1.423) (W D0 (+ 1.2 0.4))
      (P (W D0 (* -1.7 -0.9)) (W D0 (- 1.1 0.5)))).
```

The first argument to the LIST function is shown in underlined bold type. This first argument represents the first of the two output signals. This first output signal is the output from one linear threshold processing element P. This linear processing element has two input lines. The first input line is connected to data signal line D1 and is weighted (multiplied) by -1.423 as it comes into the linear threshold processing element. The second input line is connected to input data signal line D0 and is weighted by the sum (+ 1.2 0.4) as it comes into the linear threshold processing element. The second argument to the LIST function is the second output signal and has a similar interpretation.

Note that a given input data signal (such as D0 or D1) can appear in more than one place in the tree. Thus, it is possible to create connectivity between any input data signal and any number of processing elements. Similarly, the "define building block" operation (see Koza 1990 for details) makes it possible to have connectivity between the output from any linear processing element function P in the network and the inputs of other linear processing element functions. "Defined functions" are produced dynamically by the "define building block" operation in the genetic programming paradigm so as to encapsulate potentially useful sub-expressions so that they can be referred to later . The connectivity created using the "define building block" operation can even create feedback (i.e. a recurrent neural network); however, if only a feed-forward neural networks is desired, an additional rule of construction could implement that restriction.

THE ONE BIT ADDER

We now show how to simultaneously do both the "architectural design" and the "training" of the neural network to perform the task of adding two one-bit inputs to produce two output bits.

The environment for the task of adding two one-bit inputs to produce two output bits consists of the four cases representing the four combinations of binary input signals that could appear on D0 and D1 (i.e. 00, 01, 10, and 11). The correct two output bits (00, 01, 01, and 10, respectively) are then associated with each of these four environmental cases (i.e. 00, 01, 10, and 11). The raw fitness function would add up the differences between the output signals from the neural network and the correct value of the one-bit adder function.

The crossover operation in the genetic programming paradigm manipulates the structure of individuals in the population. This operation must be performed so as to preserve the allowable structure of individuals for the problem. That is, the output of this operation must be allowable neural networks in every case. Structure-preserving crossover is performed as follows: Any point may be selected as the crossover point for the first parent without restriction. However, the selection of the crossover point in the second parent is then restricted to a point of the same "type" as the point just chosen from the first parent. For this problem, there are four "types" of points, namely

- A point containing a processing element function P,
- A point containing a weighting function W,
- A point containing an input data signal (such as D0 or D1), and
- A point containing an arithmetic operation or a random constant.

When multiple values are returned by an S-expression, the fitness function is the absolute value of the difference between the value of the first dependent variable returned by the S-expression and the target value of the first dependent variable plus the absolute value of the difference between the value of the second dependent variable returned by the S-expression and the target value of the second dependent variable.

In one particular run of the genetic programming paradigm (using a population of 500 individuals), an individual emerged on generation 31 which l00% correctly performs the one-bit adder task. This individual was simplified by

consolidating each sub-expression consisting of only numerical constants and arithmetic operations into a single numerical constant (weight). The simplified form of this 100% correct individual is shown below:

```
(LIST (P (W 1.53 (DF103))
         (W 1.30 (P (W 5.25 (P (W -1.62 (P (W 0.99 D1) (W 0.043 D1)))
         (W 1.21 D0))) (W 0.50 (P (W -1.01 D1) (W 0.29 D0))))))
      (P (W -0.22 (DF111)) (W 1.21 D0))).
```

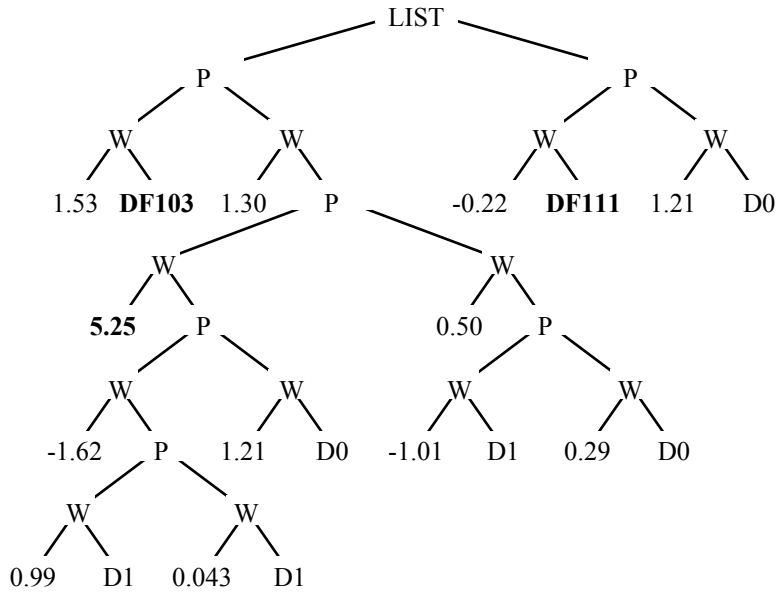
Note that this S-expression contains two defined functions (DF103 and DF111). Defined function 103 was defined in the particular run cited above so as to return the S-expression shown below:

```
DF103 = (P (W 1.052 D1) (W -1.496 D0))
```

Defined function DF111 was defined so as to return the S-expression below:

```
DF111 = (P (W 5.25 (P (W -1.62 (P (W 0.99 D1) (W 0.043 D1)))
             (W 1.51 (P (W -0.42 D1) (W 1.57 D0)))))
             (W -0.48 (P (W -0.017 D1) (W 0.077 D0))))
```

The interpretation of the 100% correct individual shown above is as follows: The first element of the LIST is the low order bit of the result. Upon examination, this first element is equivalent to (OR (AND D1 (NOT D0)) (AND D0 (NOT D1))), which is the odd-parity (exclusive-or) function of the two input bits D0 and D1. This is the correct expression for the low order bit of the result. The second element of the LIST is the high order bit of the result. Upon examination, this second element is equivalent to (AND D0 (NOT (OR (AND D0 (NOT D1)) (NOT D1))), which is equivalent to (AND D0 D1). This is the correct expression for the high order bit of the result. This individual is graphically depicted below.



It is important to note that each of the numeric constants in the S-expressions and graphical representation above is a consolidation of a generally large sub-expression involving arithmetic operations (+, -, *, %) and floating point random constants (see Koza 1990). The constant 5.25 (in bold type above) arose from the following S-expression:

```
(% (+ (* (% 0.865 -1.058) (* -0.354 0.843))
      (+ (% 1.082 -0.728) (- 0.543 0.265)))
   (* (+ -0.549 (* 1.71 0.49)) -0.636)).
```

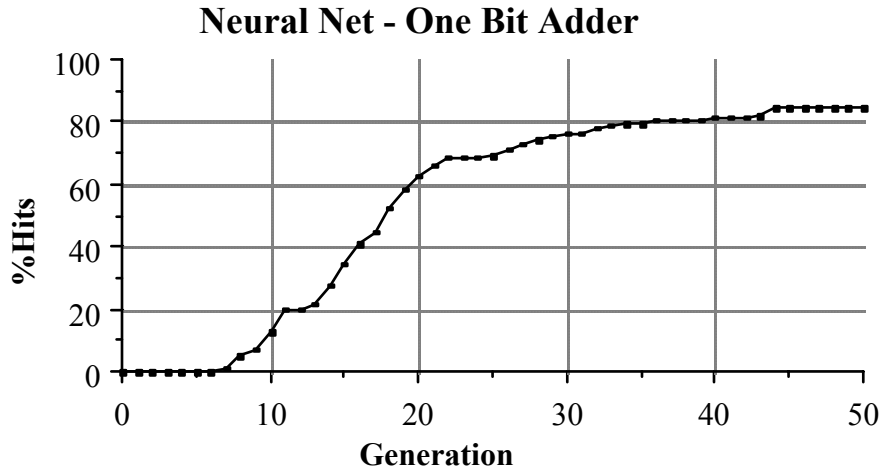
LEARNING CURVE FOR THE ONE BIT ADDER

Genetic algorithms contain probabilistic steps at several different points, namely, the random creation of the initial population of individuals, the probabilistic selection (proportionate to fitness) of individual(s) on which to perform the operations of fitness proportionate reproduction or genetic recombination (crossover), and the crossover points in the two parents. As a result, we rarely obtain a solution to a problem in the precise way we anticipate and we rarely obtain a solution twice in the precise same way.

We can measure the number of individuals that need to be processed by a genetic algorithm to produce a desired result with a certain probability, say 99%. Suppose, for example, a particular run of a genetic algorithm produces the desired result with only a probability of success p_S after a specified choice (perhaps arbitrary and non-optimal) of number of generations N_{gen} and population of size N. Suppose also that we are seeking to achieve the desired result with a probability of, say, $z = 1 - \varepsilon = 99\%$. Then, the number K of independent runs required is

$$K = \frac{\log (1-z)}{\log (1-p_S)} = \frac{\log \varepsilon}{\log (1-p_S)}, \text{ where } \varepsilon = 1-z.$$

For example, we ran 102 runs of the one-bit adder problem with a population size of 500 and 51 generations. The graph below shows that the probability of success p_S of a run is 86% for a population size of 500 with 51 generations. With this probability of success p_S on a particular single run, we need K = 3 independent runs to assure a 99% probability of solving this problem on at least one run. That is, processing 153,000 individuals is sufficient.



5. CONCLUSIONS

We have shown how to find both the weights and architecture for a neural network (including the number of layers, the number of processing elements per layer, and the connectivity between processing elements) for the one-bit adder.

6. REFERENCES

Belew, R., McInerney, J. and Schraudolph, N. N. Evolving networks: Using the genetic algorithm with connectionist learning. CSE Technical Report CS 90-174, University of California, San Diego.

Chalmers, David J. The evolution of Learning: An Experiment in Genetic Connectionism. In Touretzky, David S., Elman, Jeffrey L., Sejnowski, Terrence J., and Hinton, Geoffrey E. *Connectionist Models: Proceedings of the 1990 Summer School*. San Mateo, CA: Morgan Kaufmann 1991. Pages 81-90.

Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley l989.

Hinton, Geoffrey. Connectionist Learning Procedures. *Artificial Intelligence*. 40 (1989) 185-234.

Holland, John H. *Adaptation in Natural and Artificial Systems*, Ann Arbor, MI: University of Michigan Press 1975.

Koza, John R. Hierarchical Genetic Algorithms Operating on Populations of Computer Programs. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence.* San Mateo: Morgan Kaufman 1989.

Koza, John R. *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems.* Stanford University Computer Science Department Technical Report STAN-CS-90-1314. June 1990.

Koza, John R. Genetic Evolution and Co-Evolution of Computer Programs. In Farmer, Doyne., Langton, Christopher, Rasmussen, S., and Taylor, C. (editors) *Artificial Life II, SFI Studies in the Sciences of Complexity*. Volume XI. Redwood City, CA: Addison-Wesley. 1991.

Koza, John R. and Keane, Martin A. Genetic Breeding of Non-Linear Optimal Control Strategies for Broom Balancing. In *Proceedings of the Ninth International Conference on Analysis and Optimization of Systems*. *Antibes, June, 1990.* Pages 47-56. Berlin: Springer-Verlag, 1990.

Miller, Geoffrey F. Todd, Peter M. and Hegde, S. U. Designing Neural Networks using Genetic Algorithms. In Schaffer, J. D. (editor) *Proceedings of the Third International Conference on Genetic Algorithms.* San Mateo, CA: Morgan Kaufmann Publishers Inc. 1989. Pages 65-80.

Whitley, D., Starkweather, T., and Bogart, C. Genetic algorithms and neural networks: optimizing connections and connectivity. *Parallel Computing.* 14 (August 1990) 3. Pages 347-361.