Complex Systems **7** (1993) 249–268

A Constructive Algorithm for the Training of a Multilayer Perceptron Based on the Genetic Algorithm

Hans Christian Andersen[*]
Ah Chung Tsoi[†]
Department of Electrical Engineering,
University of Queensland,
St. Lucia, Queensland 4072, Australia

Abstract. A genetic algorithm is proposed for the training and construction of a multilayer perceptron. The genetic algorithm works on a layer-by-layer basis. For each layer, it automatically chooses the number of neurons required, computes the synaptic weights between the present layer of neurons and the next layer, and gives a set of training patterns for the succeeding layer.

The algorithm presented here constructs networks with neurons that implement a threshold activation function. This architecture is suitable for classification problems with a single binary output.

The method is applied to the XOR problem, the n-bit parity problems, and the MONK's problems, and its performance is found to be comparable to that of other techniques.

1. Introduction

Recently there has been much study into the training of multilayer perceptrons using constructive algorithms [1, 2, 3, 4]. These algorithms are useful because there are still no practical methods for obtaining a good estimate of the number of hidden layer neurons, or the number of hidden layers for a particular classification problem. A good introductory account of the types of algorithms available is given in [1]. Among the algorithms a noteworthy one is the Cascade Correlation algorithm [2]. This algorithm progressively works out the number of neurons needed, as well as the number of hidden layers that are required.

In this paper we will introduce a new constructive algorithm that is based on the genetic algorithm [5, 6, 7]. It differs from the other algorithms in that

[*]Electronic mail address: `andersen@s1.elec.uq.oz.au`
[†]Electronic mail address: `act@s1.elec.uq.oz.au`

it trains the multilayer perceptron layer by layer, that is, a layer at a time. For each layer, it automatically chooses the number of hidden layer neurons, the synaptic weights of connections with the preceding layer, and a set of training patterns for the succeeding layer. The resulting network architecture is the same as the classic multilayer perceptron (MLP) structure [1]. This contrasts with the Cascade Correlation algorithm [2], which produces a structure that is different from the classic MLP.

We have applied the method to the XOR problem, n-bit parity problems (up to $n = 8$), and to the MONK problems, which have been used as benchmarks for various classification algorithms [8]. It is found that the proposed technique works well. It yields MLP architectures for each of the three MONK problems that are comparable to the ones obtained by other researchers [8].

The structure of the paper is as follows. First a brief introduction to genetic algorithms is presented, followed by a description of the proposed algorithm. Then results of the application of the algorithm on standard test problems—namely the XOR problem, n-bit parity problems, and the MONK's problems—are presented and discussed. Also in the discussion is a description of a simple pruning algorithm that can dramatically reduce the number of neurons needed.

2. A brief introduction to genetic algorithms

The basic objective of genetic algorithms (GAs) is to harness the power of natural evolution to optimize problems. Natural evolution itself is, in essence, an optimization algorithm. From an extensive gene pool it finds combinations of genes that produce near-optimal organisms that are able to survive and prosper in their environment.

GAs are based on this observation. They hope to emulate what nature does, and in so doing obtain a robust optimization algorithm for the computation of the "global" optimum of a given function. Conceptually the gene-pool is the *solution space*, the environment is the function to be optimized (the *objective* or *cost function*), and the organisms are the trial functions (*individuals*) used to work out a solution. Having developed this analogy, one has to identify the mechanics of natural evolution and then translate them into something mathematically concrete. This results in an effective optimization method.

The first issue to resolve is how each individual will be encoded. We know that each individual must contain all information necessary to produce an offspring similar to itself. One way to do this—the method used by GAs—is to represent all of the parameters describing one individual as a bit-string. The exact manner in which this is done is up to the GA designer. For example, if the function $f(x)$ is to be minimized, an individual must possess information about the x value that defines it. One might encode this x as a bit-string of 16 bits, representing a fixed-point real number within a certain numerical range, such as -2 to 2. Once this has been done, the designer can choose the operators (for the manipulation of the genes) that will be used. The fundamental operators are:

Initialization sets the initial values of the individuals. This is usually done by filling the chromosome (bit-string) of each individual with a random distribution of 0s and 1s.

Evaluation calculates a *fitness value* for each individual on the basis of how well it solves the problem at hand.

Selection chooses individuals to pass information on to the next iteration (*generation*) on the basis of their fitness values. This operator should realize the principle that individuals with higher fitness values have higher probabilities of "surviving."

Reproduction uses selected "fit" individuals from the current generation to produce the next generation. GAs use sexual reproduction, which means that pairs of individuals are used to form offspring. This is usually done by somehow mixing the bit-streams, that is, taking some bits from one individual and the rest from the other (called *crossover*). Often some randomness is added by inverting some bits (called *mutation*). *Cloning* is an operation that copies an individual from the present generation to the next.

Initialization is done only at the beginning of a run, but evaluation, selection, and reproduction are repeated for a number of generations, after which the optimal (maximum or minimum) value of the objective function would hopefully have been located.

Many other operators and modifications to the existing ones have been proposed. In the algorithm presented in this paper the evaluation operator will be modified so that the GA performs *niche formation* [9, 10]. For a more in-depth description of GAs see references [5, 6, 7].

3. A description of the proposed algorithm

In this section the data structure on which the algorithm operates is introduced. Secondly, a list of each of the steps of the algorithm is provided, and lastly each of these steps is explained.

3.1 Data encoding

The algorithm involves operations on a set of individuals, which we will describe as *weight vectors* (WVs). A WV contains all of the information needed to define a single neuron, that is, the weight of each of the inputs to the neuron and its threshold. A neuron with I regular inputs and 1 threshold input will thus be defined by $I+1$ weights. Each of the weights is stored as a fixed-point real number, or an integer, of B bits. Hence each WV is defined by $(I+1)B$ bits. The weights of the WV are arranged such that the weight of the connection to the first input appears first, the weight of the connection to the last input appears next to last, and the weight of the threshold is at the end.

Note that the data representation is intrinsically an integer in the present situation. However, the GA designer can impose a real-number interpretation by assuming that the W_b bits represent a real number within a particular range. For purely boolean problems this can offer no advantage because one would merely be scaling the weights.

Our WV can be described mathematically as

$$y = f\left(\sum_{i=0}^{I} w_i x_i - t\right) \tag{1}$$

where y is the output of the neuron, and w_i, x_i, and t are the input synaptic weights, the inputs, and the threshold, respectively, with $i = 1, 2, \ldots, I$. I is the number of regular inputs.

The nonlinear function $f(\alpha)$ is chosen to be

$$f(\alpha) = \begin{cases} 0 & \text{if } \alpha < 0 \\ 1 & \text{if } \alpha \geq 0 \end{cases}$$

Note that this is the usual definition of a neuron [1]. The only difference is that we have assembled the input weights w_i, $i = 0, 1, \ldots, I$, and the threshold t_i into an aggregate weight vector so that we can perform the genetic algorithm operations more easily. Hence, we choose to call it a weight vector to emphasize this point.

The output of the WV is assumed to be binary. Therefore, the nonlinearity of the neuron is assumed to be a Heaviside function, rather than the usual sigmoidal function.

3.2 Outline of the algorithm

The WVs form the population pool. The algorithm involves initializing the WVs once, doing a search for good WVs by continually altering and evaluating them, and finally eliminating ineffective ones. When a solution has been settled upon—when the set of WVs has partitioned itself into classes with members of identical classes performing the same function—one representative from each class is chosen and hence a group of WVs, which represents a layer of neurons, is obtained. To allow training of the next layer the training set used for this layer is propagated through the neurons, producing a training set for the next layer (if one is needed).

The steps of the algorithm are listed below and will be explained subsequently.

1. Initialize N weight vectors (WVs).
2. Evaluate WVs with respect to the training set.
3. Search, repeating G_s times:

   (a) Select WVs to survive.
   (b) Reproduce new WVs by genetic operators (crossover and/or mutation), or cloning those selected in (a).
   (c) Evaluate WVs with respect to the training set.

4. Clean-up, repeating G_c times:
   (a) Select WVs to survive.
   (b) Reproduce WVs for use in the next generation by cloning only.
   (c) Evaluate WVs with respect to the training set.

   This will result in an arbitrary number N_l of classes of WVs, which will be the number of neurons for the current layer.

5. Choose one representative from each class of WVs.
6. Produce a training set for the next layer.
7. Repeat the entire algorithm using the training set produced in step 6, but only if more than one class was found in step 5. This implies that the training set cannot be classified by one neuron, and that another layer is needed.

G_s and G_c are the numbers of generations needed for the search and clean-up phases, respectively.

3.3 Initialization: randomization of weight vectors

The $(I + 1)W_b$ bits of the WV are set to random values. This means that when decoded to fixed-point reals or integers, the weights will have random values that are evenly distributed within the range specified by the user.

3.4 Evaluation: calculation of a fitness value

Evaluating fitness values is the most critical operation for the proposed algorithm. As in other niche formation schemes [9, 10] the fitness value of an individual (a number indicating how good it is) is made to depend on the characteristics of not only the individual itself but also on the characteristics of other individuals. In other words, the overall fitness of an individual is determined in part by the objective function, but also by the relative fitnesses of other individuals. The significance of the particular neuron is relative to the fitness of all of the other neurons.

Smith, Forest, and Perelson [10] introduced a new name for niche formation sometimes called "speciation." The name *cooperative populations* is perhaps a more descriptive name for the concept. Intuitively a WV will receive a relatively high fitness value if it is able to classify parts of the training set that not many others can. Conversely, a WV is judged to be less valuable if it only classifies training vectors that many other WVs also classify. Thus, the performance of each WV is evaluated relative to the performance of other WVs in the population.

The evaluation function is as follows:

$$\text{Fitness} = \sum_{i=1}^{T} \alpha_i \tag{2}$$

$$\alpha_i = \begin{cases} 0 & \text{if WV classifies training vector } i \text{ incorrectly} \\ \text{Bias}(n_i) & \text{if WV classifies training vector } i \text{ correctly} \end{cases}$$

where

T = the total number of vectors in the training set

n_i = the total number of WVs that classify training vector i correctly

$\text{Bias}(x)$ = a monotonically decreasing function for $x \geq 0$.

In general, the function can be chosen as $\text{Bias}(x) = 1/x^\beta$, where β is a non-negative integer; for example, $\text{Bias}(x) = 1/x^2$, or $\text{Bias}(x) = 1/x^3$. Note that a neuron classifies a training vector correctly if the neuron's output matches the desired output.

3.5 Selection: choosing weight vectors for survival

The selection operator is identical for both the search and the clean-up stages. A breeding population the size of the current population is constructed. This population is used in the reproduction stage to produce individuals for the next generation. Individuals for the breeding population are selected from the current population with a probability defined by

$$\text{probability of selecting } WV_i = \frac{r_i}{\sum_{j=1}^{N} j} = \frac{2r_i}{N(N+1)} \tag{3}$$

where

WV_i = the ith individual of the current population

N = total number of WVs

r_i = the rank of the fitness of individual i, where individuals with high fitnesses receive high ranks. No two individuals can have the same rank and clashes are resolved randomly.

This type of selection is commonly known as *rank selection* and was chosen because of its suitability for parallel implementation.

3.6 Reproduction: making new weight vectors from old ones

There are three operations that can be used to produce new individuals from the old ones, namely cloning, crossover, and mutation.

Individuals of the breeding population are grouped randomly into pairs. For each pair of WVs, there is a probability P_c that it will be crossed to produce new pairs of WVs. If a pair is not to be crossed, it will be cloned (left as is) for reuse in the next generation. In the *clean-up* stage P_c is set to 0 (i.e., cloning operation only) to make sure that no new WVs are created.

To explain the mixing/crossover of individuals, WVs must be thought of as simple strings of bits, and to fix the terminology, the natural analogy will be used. The old WVs will be called the mother and the father, and the new WVs will be the daughter and the son.

For each couple, a location L in the string is chosen randomly. The daughter inherits the bits to the left of L from the corresponding bits of her mother and the bits to the right of L from the corresponding bits of her father. Conversely, the son will inherit the bits to the left of L from his father and the bits to the right from his mother.

The mutation operation does not require two parents. There is a probability P_{m1} that a WV will be mutated. If it is mutated, each bit of the WV has a probability P_{m2} of inverting.

Both crossover and mutation can be applied separately; that is, the individuals of a pair that have been crossed over can be mutated independently.

The crossover operator is called *single-point crossover* and the mutation operator is standard. For the clean-up stage P_c, P_{m1}, and P_{m2} are set to zero. This has the effect of stopping the search for better individuals, but it also means that the bad individuals currently in the population are removed. After a few iterations with $P_c = P_{m1} = P_{m2} = 0$, only good WVs remain and hence the population has been cleaned up.

The clean-up stage could be replaced by setting a threshold for the fitnesses of all individuals, hence facilitating the consequent selection of the best ones. However, since this would add another user-definable parameter, and hence an additional level of uncertainty to the algorithm, we feel that the clean-up stage is preferable. Experiments have shown that clean-up requires very few generations compared to the rest of the algorithm.

3.7 Choosing class representatives

Weight vectors and neurons are considered to be in the same class if they perform exactly the same function. That is, they respond in the same way to each training vector in the training set. This does not necessarily mean that all of the WVs are numerically identical.

One representative from each class is taken to make up a layer of the multi-layer perceptron. In choosing representatives, one individual is simply chosen at random from each class.

3.8 Producing a training set for the next layer

The set of representatives chosen in the previous operation is a solution for a single layer of a multi-layer perceptron for classifying the training set used. To generate a training set for the next layer, the inputs of the training vectors in the current training set are propagated through this layer and hence become the inputs for the neurons in the next layer. The desired output for each training vector is made equal to the desired output for the corresponding training vector in the current training set.

After being passed through a layer, it is possible that two different input vectors are mapped to identical output vectors. This means that subsequent layers have no way of differentiating them. In this case, if they have different desired outputs, the network cannot possibly classify both correctly, and in the training sets for the subsequent layers the two training samples are said to be "conflicting." If the algorithm is allowed to continue without intervention, it will get confused and attempt to classify both of the conflicting vectors correctly. This can never be done by a single neuron, so no layer will ever be produced with just one neuron; because this is our stopping criterion, the algorithm will never terminate.

To cure this problem the conflict must be resolved. Conflict resolution is done by choosing one of the desired outputs and setting it equal to the output for both input vectors. If there are more than two vectors involved, the desired output chosen is the more common one.

Conflict resolution results in a loss of accuracy, but if it is not used a final solution will not be obtained.

3.9 Parameters

The parameters that the algorithm needs are the following:

- Number of bits per weight, W_b
- Range of magnitude of weights, W_r
- Population size N, the number of individuals
- Crossover probability P_c, the probability that crossover will occur
- Individual mutation probability P_{m1}, the probability that an individual will be mutated
- Per bit mutation probability P_{m2}, the probability that a bit in a mutated individual will be inverted
- Number of generations for search G_s
- Number of generations for clean-up G_c
- The parameter controlling bias β

W_b and W_r are dependent on the training set. W_b controls the precision of the weights, and W_r determines the range within which the hyperplane of any neuron can exist. N controls the probability of finding a good solution to a problem. The larger N, the higher the probability of finding a good solution; however, run time increases with population size.

The algorithm, like most GAs [5], is quite robust with respect to the value of P_c, P_{m1}, and P_{m2}, which is fortunate since little concrete theory exists on how to determine them. Values of about 0.5 for P_c and about 0.01 for both P_{m1} and P_{m2} have been found to work.

The G values control the amount of run time used by the algorithm. First, the algorithm searches for solutions for G_s generations, then sorts out which solutions are good for G_c generations. G_s varies with the difficulty of the problems presented. G_c should only need a few generations, although this may also vary from one training set to another.

As in all algorithms of this kind, the setting of these parameters depends on the user's experience and on the nature of the problem. These parameters are often chosen by trial and error, as little theory exists that helps in accomplishing this task. The examples shown in section 4 give some indications of how these parameters can be chosen for typical problems.

3.10 Computational time issues

The computational time for the proposed method is almost instantaneous for a small number of inputs, a small population size, and a small data sample set. However, the GA can be quite time consuming when run on a sequential machine. This is because the operations crossover and mutation are essentially parallel operations. Hence, to emulate these parallel operations on a sequential machine takes longer.

We have implemented two versions of the algorithm: one on a sequential machine, and the other on a massively parallel machine. The massively parallel machine we used was a MasPar MP-1 SIMD (Single Instruction Multiple Data) computer with 4096 processor elements, each element having a 4-bit processor. We have resorted to using the MasPar computer because it is ideally suited for studying problems of this nature. Each processor element can be used to emulate an individual in the population. Mutation and crossover can be performed by simple operations, either on the processor itself, or among a small number of processors. The speed of the algorithm when run on this machine is insensitive to population size as long as it is less than the total number of processors. Hence, population size was always set to the maximum 4096 individuals. It is possible to implement even larger population sizes on the MasPar computer as well, except that the speedup is expected to be a linear function of the number of individuals in a population.

4. Results and discussions

As mentioned previously, the algorithm has so far been tested on the XOR, the n-bit parity, and the MONK's problems. In this section results for all of these are presented. We focus our analysis of the algorithm on the XOR problem.

4.1 The exclusive-or (XOR) problem

With population sizes (N) upwards of about 150, the algorithm almost always arrives at the same general solution to the XOR problem (see Table 1 for the

Training	Inputs		
Vector	A	B	Output
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	0

Table 1: Training set for the XOR problem.

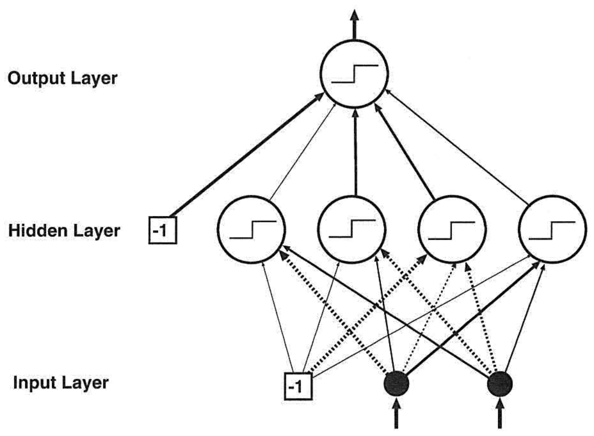

Figure 1: Sample solution to the XOR problem. The magnitude of the weights is proportional to the thickness of the connecting lines, and broken lines have negative weights.

training set). A sample of this solution is shown in Figure 1, which shows a network of 4 nodes in a single hidden layer and 1 node in the output layer.

To illustrate how the algorithm converges, a graph has been included (see Figure 2) that shows the relative sizes of selected output classes at each generation of a single experiment. The XOR problem has 4 training vectors, so there are 16 ($= 2^4$) different ways in which a neuron can classify these. Hence, there are at most 16 possible output classes to which individuals can belong. We have numbered these 0 to 15, where 0 corresponds to the type of neuron that classifies all input vectors as 0s, and 15 corresponds to the one that classifies them all as 1s. The numbering scheme is described in Table 2. The classes plotted are the four "good" ones (2, 4, 7, and 14) and two of the "bad" ones (0 and 15). We could have plotted all 16 classes, but the resulting graph would be too cluttered to illustrate the points we wish to make. The significance of this graph will be discussed later.

The parameters used for this run were:

- $W_b = 16$ bits per weight

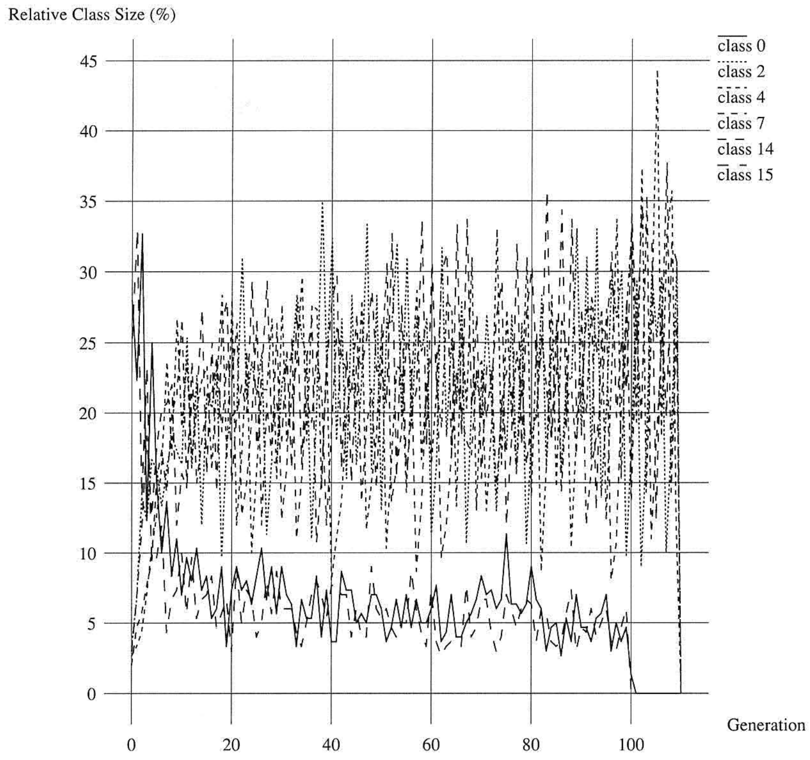

Figure 2: Sizes of output classes 0, 2, 4, 7, 14, and 15 during a run of the algorithm on the XOR problem.

- W_r = same as that of a signed 16-bit integer (2's complement)
- $N = 150$ individuals
- $P_c = 50\%$ probability of crossover
- $P_{m1} = 5\%$ probability of individual mutation
- $P_{m2} = 5\%$ probability of bit inversion in a mutated individual
- $G_s = 100$ generations for search
- $G_c = 10$ generations for clean-up

As can be seen in Figure 1, the algorithm solves the XOR problem with 4 neurons in the hidden layer. This was disconcerting at first because we know that a network for solving this problem requires only 2 nodes in the hidden layer. An explanation for this behavior is that the algorithm is objective in its choice of neurons; that is, it evaluates *individual* neurons in terms of their contribution toward the complete solution. Once it has located each of the

Output	Output for Training Vector			
Class	0	1	2	3
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

Table 2: Definition of output classes of the XOR problem.

4 classes of neurons in question, it will not eradicate any of them because they all contribute equally toward the complete solution. One could describe the solutions generated by this algorithm as being *individualistic*. Although these individualistic networks will sometimes contain more neurons than are necessary, it is thought that this will occur primarily in symmetric problems like the XOR problem, which have several equally good sets of solutions. In the last section of the paper a pruning algorithm is introduced that, to a large extent solves this problem of duplication of effort.

The graph in Figure 2 illustrates some characteristics of the algorithm's convergence. The first observation is the instability of the class sizes with respect to time. This is probably due to the discrete nature of the fitness function (i.e., there are only 14 ($= 2^4 - 2$) possible fitness values), which has the effect of not allowing a class's ideal size to be matched by its ideal fitness. It is possible that a slightly modified fitness function will alleviate this problem.

The event that caused the change near generation 100 is the start of the algorithm's clean-up stage (step 4). The two lower lines almost immediately go to 0. In fact, all class sizes go to 0 except the four good ones; these are the four that will make up the solution. The purpose of the clean-up stage is to do this filtering, "to clean out the barn." The question of why the bad ones disappeared is perhaps best answered by asking why they were there at all. They were not present because they received good fitness values

n	Average Accuracy (%)	Example of MLP Generated
3	100	3-4-1
4	100	4-15-1
5	100	5-15-1
6	100	6-41-1
7	100	7-39-1
8	100	8-54-1

Table 3: Accuracies and examples of networks used to solve the n-bit parity problems. Each accuracy figure is an average of 5 solutions generated from consecutive runs. The notation used to describe the architectures gives the number of neurons in each layer with the number of neurons in the input layer being the first number.

and reproduced of their own accord. In fact, their fitness values would most likely have been quite low throughout. The reason they were there was that they were the result of the crossover of members of separate good classes. For example, mating a member of class 2 with a member of class 14 might result in an individual of class 6. Hence there is a constant production of inter-class offspring due to crossover between different good classes. When the probability of crossover and mutation are reduced to zero at generation 100, this production of mutant inter-class offspring is eliminated. Therefore, since the bad classes cannot survive by themselves and are not being produced through inter-class mating, they disappear. In the XOR problem clean-up occurs almost instantaneously, but experiments with the MONK's problems indicate that it sometimes takes longer (see section 4.3).

4.2 The n-bit parity problems

The algorithm has been tested on n-bit parity problems for n up to and including eight. The accuracy of some networks constructed using the algorithm are shown in Table 3. This table also gives examples of typical networks produced for each of the problems.

The number of hidden layer neurons appears to be greater than those found by other algorithms. However, as explained in subsequent sections, this number can be reduced by using a pruning algorithm.

4.3 The MONK's problems

The MONK's problems is a set of three problems used at the Second European Summer School on Machine Learning to benchmark a number of expert systems and artificial neural network techniques [8]. The problems are set in an artificial domain in which robots are described by six different attributes:

x_1	:	head_shape	$\in$	round, square, octagon.
x_2	:	body_shape	$\in$	round, square, octagon
x_3	:	is_smiling	$\in$	yes, no
x_4	:	holding	$\in$	sword, balloon, flag
x_5	:	jacket_color	$\in$	red, yellow, green, blue
x_6	:	has_tie	$\in$	yes, no

There are a total of 17 inputs. The learning task is a binary classification task, each problem is given by a logical description of a class. Robots belong either to this class or not. There are three problems as follows:

1. Problem 1 (MONK-1):
   (head_shape = body_shape) or (jacket_color = red)
   From 432 possible examples, 124 were selected randomly for the training set. There are no misclassifications, so there is no noise in the training set.

2. Problem 2 (MONK-2):
   exactly two of the six attributes have their first value
   (For example, body_shape = head_shape = round implies that robot is not smiling, holding no sword, jacket_color is not red and has no tie, since then exactly two (body_shape and head_shape) attributes have their first value.)
   From 432 possible examples, 169 were selected randomly to be in the training set. Again, there is no noise introduced.

3. Problem 3 (MONK-3):
   (jacket_color is green and holding a sword) or (jacket_color is not blue and body_shape is not octagon)
   From 432 examples, 122 were selected randomly, and among them there were 5% misclassifications; that is, there is noise in the training set.

The multi-layer perceptrons constructed and trained by the proposed algorithm are described in Table 4.

It should be noted that the training set for MONK-3 has noise in the form of 5% incorrectly classified training vectors. In the original MONK report [8], it was reported that the classification accuracy for both the MONK-1 and MONK-2 problems is 100%. However, this is true only if the initial conditions are chosen judicially. In general, if the initial conditions are chosen randomly, it is possible that the classification accuracy may not be 100% using the back-propagation algorithm. In our case, the average classification accuracy is shown after 5 runs with random initial conditions.

These problems were all run on a MasPar MP-1 computer with 4096 processor elements. The parameters that were changed from the runs on the XOR problem were $N = 4096$, $G_s = 400$, and $G_c = 100$. The run-time for each layer of each problem was about 1 minute.

The MONK's problems are less "clinical" than the XOR and n-bit parity problems and did not seem to suffer from the problem of an excessive

Problem	Average Accuracy (%)	Example of MLPs Generated
MONK 1	99.64	17-5-1
MONK 2	97.54	17-6-1
MONK 3	96.42	17-4-1

Table 4: Testing accuracies and examples of networks used to solve the MONK's problems using the genetic algorithm. Each accuracy figure is an average of 5 solutions generated from consecutive runs.

Problem	Accuracy (CASCOR) (%)	Accuracy (BP) (%)
MONK 1	100.0	100.0
MONK 2	100.0	100.0
MONK 3	97.2	97.2

Table 5: Testing accuracies of the MONK's problems using cascade correlation (CASCOR) and back-propagation (BP) as reported in [8]. The figures are apparently those of a single run of each algorithm.

number of neurons (see Table 4). Table 5 shows the performance of the back-propagation and cascade correlation algorithms on the MONK's problems.

Note that our results are not comparable to those presented in the original MONK report [8]. This is because the authors of [8] did not clearly mention the initial conditions they used. Secondly, they did not indicate if their results were a single best-performance figure, or an average-performance figure. In our experience, the classification performance depends on the initial conditions. In our results we have observed 100% correct classification accuracy for both MONK-1 and MONK-2 problems using certain initial conditions. However, in order to be fair to the problem, we have chosen to report the average performance of the proposed algorithm instead. This is closer to the actual performance of the algorithm should initial conditions be chosen at random. However, for the sake of completeness, we have duplicated the results of both the cascade correlation and the backprop as shown in Table 5 for the reader to judge.

4.4 Changing the bias function

As mentioned previously, the bias function x^β is used in fitness evaluation to provide a relationship between the number of WVs that correctly classify a training example and the value attributed to a WV for doing so. It must be a monotonically decreasing function (for $x \geq 0$) so that there is a trend to encourage WVs to classify correctly examples that few others do.

An example is used to clarify this. Three training vectors T_a, T_b, and T_c

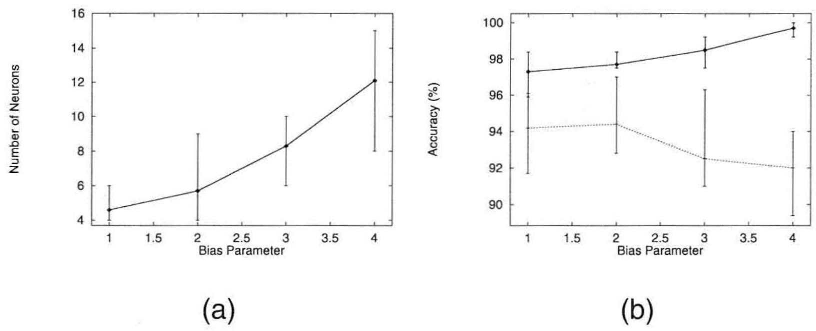

Figure 3: The average number of neurons (a) and the accuracy on the training and testing sets (b) of sets of ten runs on the MONK-3 problem with the bias parameter increasing. In graph (b) the solid line represents the accuracy on the training set, and the dashed line represents the accuracy on the testing set. Error bars are included.

from a large training set have the following numbers of correctly classified individuals: $n_a = 10$, $n_b = 20$, and $n_c = 20$. If $\beta = 1$ then the payoff gained from classifying each of them will be $p_a = 0.1$ and $p_b = p_c = 0.05$. A neuron that correctly classifies only T_a in this case will receive the same fitness, 0.1, as one that classifies both T_b and T_c correctly. However, if $\beta > 1$ then the former neuron will get a total payoff greater than that of the others. It can easily be seen that a greater β will have the effect of promoting the importance of rarely classified training vectors. In the same manner, if β is small then a solution will be searched for where less importance is placed on rare training vectors.

One could envision a case in which a training vector requires a neuron to be devoted to classification alone. If β is low it is unlikely this will happen, but the higher β becomes, the more likely it is that this will occur.

This feature of being able to vary the amount of attention to detail could be thought of as being analogous to controlling *generalization.* The lower the bias parameter, the more the resulting multilayer perceptron will generalize.

The two graphs in Figure 3 show how the accuracies on the training and testing sets, as well as the number of neurons, vary with the bias parameter. Each point on the graphs is the average of ten runs on the MONK-3 problem. This training set has 5% noise in the form of incorrect classifications.

The effect of the bias parameter β being related to the amount of generalization is quite apparent. The higher the bias parameters, the more neurons are utilized, and the higher the accuracy on the training set. However, because of the noise in the training set of MONK-3, a near-perfect accuracy on the training set is not desirable as it results in poor performance on the testing set. Overall performance is best when β is around 2. At this level, the networks produced seem to have the best level of generalization.

Network Produced	Accuracy (%)
2-2-1	100

Table 6: Results on the XOR problem of an algorithm that used pruning on intermediate training sets.

4.5 Pruning the intermediate training sets

It is apparent from the results presented thus far that the networks produced by the algorithm are larger than they need to be. This is particularly obvious in the case of the parity problems whose results are presented in section 3.2. In section 3.1 we described why the algorithm produced a solution with 4 hidden-layer units for the XOR problem. We suspected that something similar was happening in the parity problems, that is, the excessive size of the networks was a result of duplicated effort caused mainly by the symmetry of the problems. In order to test our hypothesis we produced a pruning algorithm that turned out to be a potentially useful addition to the technique. The method is described below.

The pruning algorithm is based on the following idea. If a training set is *faithful* then a neural network can be found to classify correctly every training vector in that set. By faithful we mean that it has no conflicts: in other words, there exists no pair of input-output exemplars in which the inputs are identical but the outputs are different.

The objective is to minimize the number of inputs without introducing conflicts into the training set. Because the inputs come from the neurons in the previous layer, when we know which inputs can be cut out we also know which neurons in the previous layer are redundant. The method itself is merely an implementation of this.

It should be noted that doing this pruning does not necessarily make the training set easier because situations may exist in which several inputs contribute toward a particular classification, although only one of them is vital.

Below is a step-by-step description.

1. Remove the first input from every training vector in the training set (analogous to removing a neuron from the previous layer).

2. Check the training set to determine if there are conflicts. If there are conflicts go to step 3, otherwise go to step 4.

3. Put back the input just removed.

4. If there are more inputs, remove the next input and go to step 2, otherwise finish.

The results obtained when pruning is done on intermediate training sets are shown in Tables 7, 8, and 9. For all of these runs $\beta = 2$. In each case the

n	Average Accuracy (%)	Example of MLP Generated
3	100	3-3-1
4	100	4-4-1
5	100	5-5-1
6	100	6-9-1
7	100	7-9-1
8	100	8-15-1

Table 7: Accuracies and examples of networks used to solve the n-bit parity problems using pruning of intermediate training sets. Each accuracy figure is an average of 5 solutions generated from consecutive runs.

Problem	Average Accuracy (%)	Example of MLPs Generated
MONK 1	99.91	17-3-1
MONK 2	97.78	17-2-1
MONK 3	94.40	17-2-1

Table 8: Accuracies and examples of networks used to solve the MONK's problems. Each accuracy figure is an average of 5 solutions generated from consecutive runs.

pruned architecture is similar to the more established architectures reported in the literature [8].

Without major modifications to the algorithm and using its principles of parallel search, it does not seem probable that pruning-like behavior can be made implicit in the main part of the algorithm. Therefore, because problem heuristics are available, we thought it best to build our pruning algorithm around these.

5. Conclusion

In this paper we have introduced a novel constructive algorithm for the training of a multilayer perceptron based on genetic algorithm concepts. It is shown that such an algorithm can construct the multilayer perceptron layer by layer. In addition, the algorithm yields automatically a training pattern set for the subsequent layer. In contra-distinction to the Cascade-Correlation algorithm introduced by Fahlman, the architecture of the resulting network is similar to the classic multilayer perceptron.

This algorithm has been applied to a number of testing problems, namely the exclusive OR problem, the n-bit parity problem, and the MONK problems. It is found that in all cases the results obtained are comparable to the known results.

It is noted that the proposed algorithm is capable of finding an MLP structure with more than one hidden layer of neurons. However, it so happens that the examples chosen in this paper can all be solved using MLP structures that have only one hidden layer of neurons. It might be interesting to find examples that cannot be solved using a single hidden layer, and see whether the proposed algorithm yields structures that are comparable to those found by more traditional methods.

6. Acknowledgments

The second author, Ah Chung Tsoi, acknowledges partial financial support from the Australian Research Council in performing this research. Both authors also wish to acknowledge fruitful discussions with Steven Young, which led to the development of part of the algorithm. In addition, the authors wish to acknowledge the assistance given by the anonymous referee whose helpful comments led to a more readable version of this paper.

References

[1] J. Hertz, A. Krogh, and R. G. Palmer, *Introduction to the Theory of Neural Computation* (Reading, Mass.: Addison-Wesley, 1991).

[2] S. Fahlman and C. Libiere, "The Cascade-Correlation Learning Architecture," pages 524–532 in *Advances in Neural Information Processing Systems II*, edited by D. Touretzky (San Mateo, Calif.: Morgan Kaufmann, 1990).

[3] M. Frean, "The Upstart Algorithm: A Method for Constructing and Training Feedforward Neural Networks," *Neural Computation*, **2** (1990) 198–209.

[4] J.-P. Mezard and M. Nadal, "Learning in Feedforward Layered Networks: The Tiling Algorithm," *Journal of Physics A*, **22** (1989) 2191–2204.

[5] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning* (Reading, Mass.: Addison-Wesley, 1989).

[6] L. Davis, *Genetic Algorithms and Simulated Annealing* (San Mateo, Calif.: Morgan Kaufmann, 1987).

[7] L. Davis, *Handbook of Genetic Algorithms* (New York: Van Nostrand Reinhold, 1991).

[8] S. B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. Dzeroski, S. E. Fahlman, D. Fisher, R. Hamann, K. Kaufman, S. Keller, I. Kononenko, J. Kreuziger, R. S. Michalski, T. Mitchell, P. Pachowicz, Y. Reich, H. Vafaie, W. Van de Welde, W. Wenzel, J. Wnek, and J. Zhang, "The MONK's Problems: A Performance Comparison of Different Learning Algorithms," Carnegie Mellon University, Report CMU-CS-197 (December 1991).

[9] D. E. Goldberg and J. Richardson, "Genetic Algorithms with Sharing for Multi-modal Function Optimization," *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms* (San Mateo, Calif.: Morgan Kaufman, 1988).

[10] R. E. Smith, S. Forrest, and A. S. Perelson, "Searching for Diverse, Cooperative Populations with Genetic Algorithms," TCGA Report No. 92002 (1992).