Report Number: WUCS-91-35

1991-05-01

Genetic Algorithms: Usefulness and Effectiveness for Pattern Recognition

Mohit Verma

Genetic Algorithms have been gaining much interest since the early 1970's and have intrigued people from the fields of machine learning, artificial intelligence, neural networks and operations research. This paper describes the approach of genetic algorithms applied to neural networks. The experiments were conducted using various functions such as XOR,AND,SINE and different network sizes. Based on the experimental data, we concluded that for small network architectures represented by the functions (SINE,ENCODE,etc), genetic algorithms were not effective and the desired results were not achieved within a reasonable period of time.
... Read complete abstract on page 2.

Genetic Algorithms: Usefulness and Effectiveness for Pattern Recognition

Mohit Verma

Complete Abstract:

Genetic Algorithms have been gaining much interest since the early 1970's and have intrigued people from the fields of machine learning, artificial intelligence, neural networks and operations research. This paper describes the approach of genetic algorithms applied to neural networks. The experiments were conducted using various functions such as XOR,AND,SINE and different network sizes. Based on the experimental data, we concluded that for small network architectures represented by the functions (SINE,ENCODE,etc), genetic algorithms were not effective and the desired results were not achieved within a reasonable period of time.

Genetic Algorithms: Usefulness and Effectiveness for Pattern Recognition.

Mohit Verma

WUCS-91-35

May 1991

Department Of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

This work was supported by the KUMON MACHINE project.

Abstract

Genetic Algorithms have been gaining much interest since the early 1970's and have intrigued people from the fields of machine learning, artificial intelligence, neural networks and operations research. This paper describes the approach of genetic algorithms applied to neural networks. The experiments were conducted using various functions such as XOR,AND,SINE and different network sizes. Based on the experimental data, we concluded that for small network architectures represented by the functions (XOR,AND,etc), genetic algorithms were very effective, but for larger network architectures represented by the functions (SINE,ENCODE,etc.), genetic algorithms were not effective and the desired results were not achieved within a reasonable period of time.

Table Of Contents

List Of Figures

Genetic Algorithms: Usefulness and Effectiveness for Pattern Recognition

Mohit Verma

1. History

Genetic algorithms were invented by John Holland in the early 1970's. Holland was intrigued by Darwin's theory of evolution i.e. how the human species which has very complex biological functions had rapidly evolved over the centuries from a species which had only a few simple biological functions available to them.

1.1. Biological Background

The evolutionary theory proposed by Charles Darwin received the acclaim and support of biologists who were long baffled by "evolution" and its causes. The evolutionary theory has some important features which nurtured the growth of genetic algorithms:

- Evolution operates on chromosomes and not physically on a living organism. Each chromosome is made of genes which partly determine the living organism.
- Natural selection is a process that allows the *fittest* organism to replicate. The *fittest* organism is one which is able to adjust to its environment quickly and efficiently.
- Evolution occurs at the time of reproduction. *Mutations* or "sudden changes" cause a biological offspring completely different from its biological parents.
- Biological evolution has no memory: the only knowledge of producing new individuals is contained in the *gene pool*- which is the set of all the chromosomes of the current individuals.

Holland was intrigued by these features of evolution, and believed that if these features could be correctly incorporated into a computer algorithm, they might yield a technique to solve difficult problems the way nature has solved through the process of evolution. These algorithms were named *genetic algorithms* [1,Pg 2].

In nature, mutations have played a key role for the process of evolution. Mutations occur when genes of one chromosome are randomly swapped with other genes of another chromosome or from within, since the position of these genes on a chromosome partly determine the living organism. So, the key question for biologists is which arrangement of genes is the best ; that is, which arrangement of these genes will lead to the fittest living organism for a certain given environment.

Similarly, this applies to neural networks where we have to search for a weight vector which gives the optimal result for a given fuction, serving as the environment in this case.

2. Principles

According to David Goldberg, *genetic algorithms* are defined to be "search algorithms based on the mechanics of natural selection and natural genetics" [2,Pg 1].

Based on this definition, the idea of genetic algorithms has been applied to neural networks. Mutations play a key role in natural selection and natural genetics, and in neural networks we have to try and simulate the role of mutations or "sudden changes".

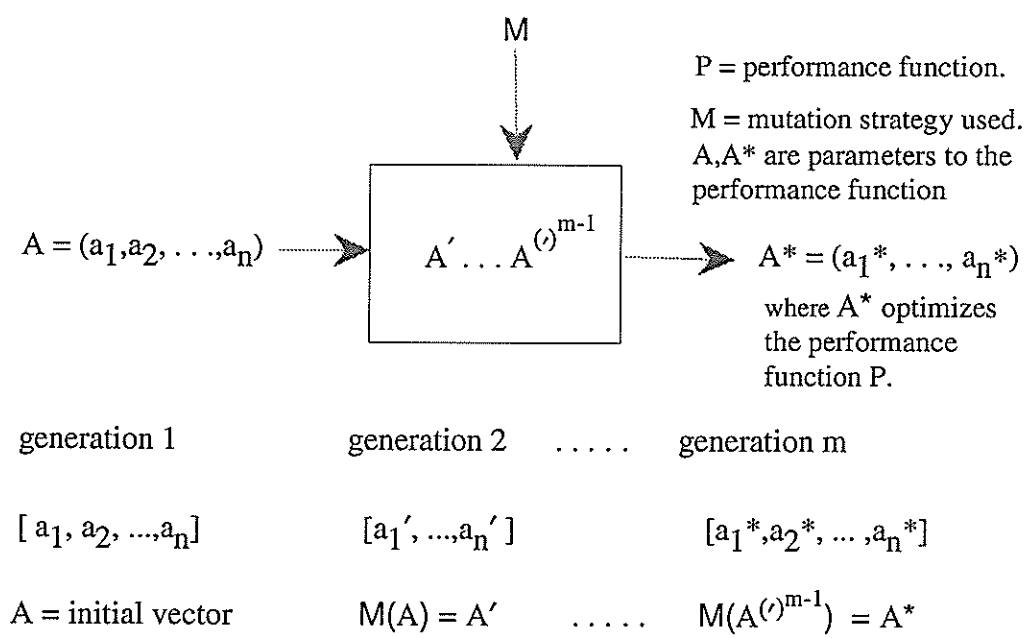


Figure 1: symbolic interpretation

Figure 1 illustrates how the genetic algorithm approach is applied to engineering problems. In the above figure, A is the initial vector whose components are parameters to the performance function. Our goal is to optimize the performance function P, so we have to search for an A=A* which optimizes the performance function. M denotes the mutation strategy which is applied to the parameters(A,etc), here A' is generated from A using the mutation strategy M. So at the end of each generation we have a new parameter set which is used for the next iteration. This procedure is repeated until the optimal A=A* is found.

3. Experiments

Several experiments were conducted using the genetic algorithm approach to determine their effectiveness for simple to complex functions for neural networks.

3.1. Mutation Strategies used

Two different types of mutation strategies (M's in figure 1) were used. In the the next two sections these strategies have been described.

3.1.1. Random Swapping

The example below illustrates the random swapping strategy. In the example, S_1 and S_2 represent the parent chromosomes where the 0's and 1's indicate the genes present on the chromosome. According to this strategy, initially two random numbers (R_1 and R_2) are generated in the range of L (length of string S_1) and L_2 (length of string S_2) respectively, which indicate the positions of the genes to be swapped.

Example:

Before Swapping:

$$S_1 = 1\ 0\ 1\ 0\ 1\ 0$$
$$S_2 = 0\ 1\ 0\ 0\ 1\ 1$$

where, S_1 and S_2 represent the parent chromosomes.

$$R_1 = 3, R_2 = 4$$

where, R_1 represents the position of the gene in S_1 to be swapped and R_2 represents the position of the gene in S_2 to be swapped.

$$L_1 = 6,\ L_2 = 6$$

where, L_1 represents the number of genes present in S_1 and L_2 represents the number of genes present in S_2.

After Swapping:

$$S_1' = 1\,0\,0\,0\,1\,0$$
$$S_2' = 0\,1\,0\,1\,1\,1$$

where, S_1' and S_2' represent the chromosomes of the new offsprings.

In the example above, the random numbers generated are $R_1=3$, $R_2=4$. After the swapping is completed, the resultant strings S1 ' and S2 ' are displayed. Since the positions are chosen randomly, a large number of new strings can be generated.

3.1.2. One-Point Crossover

Biologists have used the concept of *crossover* to explain the "sudden changes" occuring in the offsprings. Crossover is the interchange of genes between parents. It is believed that during reproduction parents exchange their genetic material (genes) which can result in radically different offsprings. John Holland used this process to formulate the *one-point crossover* strategy [1,Pg 16] [3,Pg 16].

Example:

Before Crossover:

$$S_1 = 1\,0\,1\,0\,1\,0$$
$$S_2 = 0\,1\,0\,0\,1\,1$$

where, S_1 and S_2 represent the parent chromosomes.

$$P = 4$$

where, P represents the position in S_1 and S_2 where the crossover of genes begins.

$$L = 6$$

where, L represents the number of genes present in S_1 and S_2.

After Crossover:

$$S_1' = 1\,0\,1\,0\,1\,1$$
$$S_2' = 0\,1\,0\,0\,1\,0$$

where, S_1' and S_2' represent the chromosomes of the new offsprings.

The above example illustrates the one-point crossover strategy. In the above example, S_1 and S_2 represent the parent chromosomes where the 0's and 1's indicate the genes present on the chromosome. According to this strategy, initially a random number P is generated in the range L (length of the strings S_1 is S_2 are restricted to be equal to L) , which indicates the position where the crossover is to begin.

In the above example P = 4. After the crossover is completed , the resultant strings S_1 ' and S_2 ' are displayed. Since the positions are chosen randomly, a large number of new strings can be generated.

3.2. Applications to Neural networks

The mutation strategies described in section 3.1. have been used to simulate the genetic algorithm approach for neural networks and the experiments were conducted using these strategies. The next two subsections describe how these strategies have been applied to the case of neural networks.

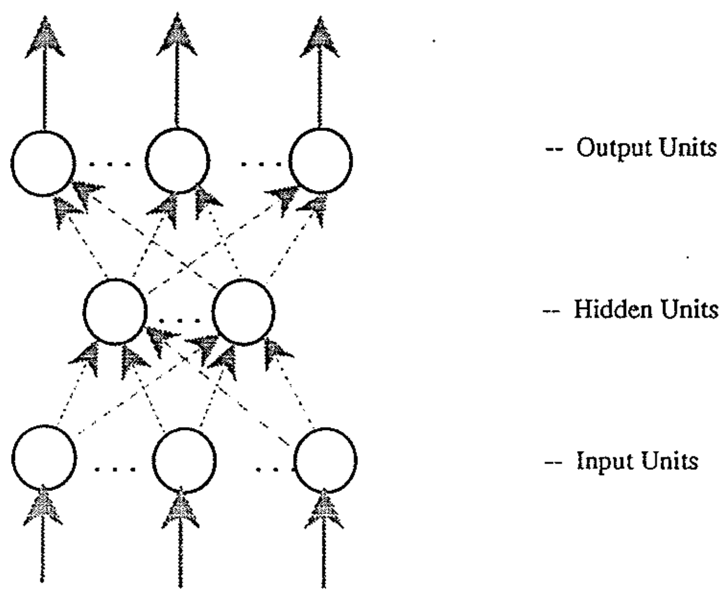
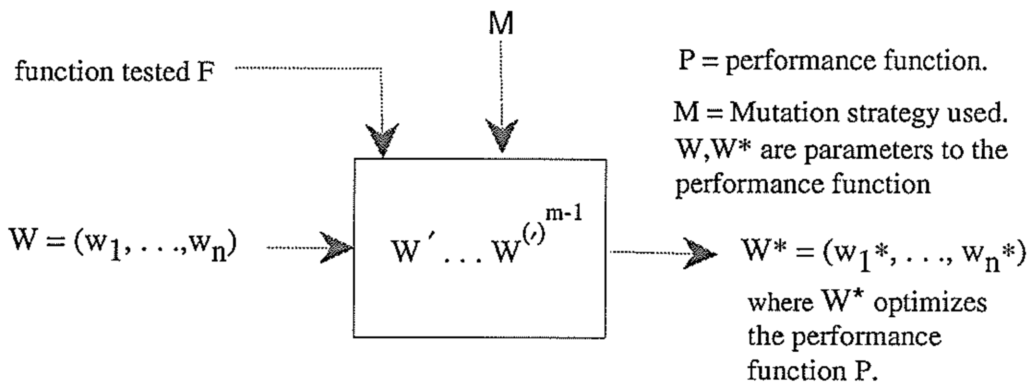


Figure 2: Network Structure

Figure 2 shows the neural network used for the experiments conducted. The network architecture used was hierarchical; where the number of output, input, and hidden units were arbitrary and could be chosen at the discretion of the experimenter. The arrows indicate the inputs and outputs passed along the network (*chopped* arrows indicate data passed within the network, *solid* arrows indicate the inputs given to the network and the outputs generated by the network). Also, each of the arrows (solid and chopped) drawn in figure 2 holds a weight value. For simplicity, the network was restricted to three layers.

In figure 3 a dataflow chart is shown, which illustrates how the genetic algorithm approach is applied to the neural network (pattern recognition) problem. In figure 3, W is the initial vector whose components are parameters to the performance function. Our goal is to minimize the performance function P, so we have to search for an W=W* which minimizes the performance function P. M denotes the mutation strategy which is applied to the parameters (W,etc) and F denotes the function which is to be tested. W′ is generated from W using the mutation strategy M. So, at the end of each generation we have a new parameter set which is used for the next iteration. This procedure is repeated until the optimal W=W* is found.



For neural nets $P(w_1, w_2, \ldots, w_n) = \Sigma\ 0.5 * (y_i - F(x_i))^2$

where, F is the function to be tested(XOR,SINE,etc)

y_i's are the teaching values

x_i's are the function inputs

generation 1	generation 2	.....	generation m
$[w_1, \ldots, w_n]$	$[w_1', \ldots, w_n']$		$[w_1^*, \ldots, a_n^*]$
W = initial vector	$M(W) = W'$	.....	$M(W^{(\prime)^{m-1}}) = W^*$

Figure 3: neural nets representation

3.2.1. Random Swapping(M1) for Neural Nets

The idea behind Random Swapping was described in section 3.1.1. This idea has been applied to neural networks where we search for an optimal weight vector each of which represents a chromosome, and whose components (weights) represent genes .

Example: To illustrate this idea of random swapping, let us take a formal example. Initially, a set of M weight vectors W (below), is generated randomly within a given range. The input function is evaluated for each of these initial weight vectors, then the best half (M/2) weight vectors and worst half (M/2) weight vectors are separated into two lists. The weight vectors are ordered according to their performance, and from them these two lists are determined. The random swapping strategy is applied to two weight vectors at a time, so random swapping will be applied initially to the first weight vector in the best list and the first weight vector in the worst list, respectively. This procedure is repeated until the last set of (best & worst) weight vectors has been used.

$$W = [w_1 w_2 \ldots w_M]$$

Before Swapping:

$$W_1 = [\ w_1\ w_2\, w_3 \ldots w_n]$$
$$X_1 = [\ x_1\ x_2\ x_3 \ldots x_n\ ]$$

where, W_1 and $X_1$are weight vectors, W_1 is the first vector in the best list and X_1 is the first weight vector in the worst list, and their n components are the weights needed for the network.

$$R_1 = 2, R_2 = 3$$

where, R_1 represents the position of the weight in W_1 to be swapped and R_2 represents the position of the weight in X_1 to be swapped with each other, respectively.

$$L_1 = n, L_2 = n.$$

where, L_1 represents the number of weights of W_1 and L_2 represents the number of weights of X_1. For our purposes, the number of weights of W_1 and X_1were restricted to be equal.

After Swapping:

$$W_1' = [\ w_1\, x_3\, w_3 \ldots w_n]$$
$$X_1' = [\, x_1\, x_2\, w_2 \ldots x_n]$$

Now, $W' \quad = \quad [\ W_1'\ X_1'\ W_2'\ X_2' \ldots W_n'\ X_n'\ ]$

Please note:

W' is the newly generated set of weight vectors.

W_1' and $X1'$ are the new weight vectors generated after the swapping.

n = (number of hidden units) *(number of input units + number of output units) of the network.

M = n if n is even, otherwise M = n + 1.

For this example, W_1 is the first weight vector in the best list and X_1 is the first weight vector in the worst list. The two random numbers generated are $R_1 = 2$, $R_2 = 3$, the corresponding weights w_2 and x_3 are swapped, and the resultant weight vectors W_1' and X_1' are shown after the swapping has occurred. This procedure is repeated M/2 times, so at the end of one iteration we have a new set of M weight vectors which are used in the next iteration. The upper bound on the number of iterations acceptable is 20000 iterations. This procedure is repeated until a weight vector is generated which optimizes the given function or the 20000 iterations are completed. This strategy was found to be $O(n^2)$.

To illustrate this strategy, let us give an example.

$$W = \Big[\ (a\,b\,c\,d)\ \ (g\,h\,i\,j)\ \ (m\,n\,o\,p)\ \ (s\,t\,u\,v)\ \Big]$$

$$BH = \Big[\ (a\,b\,c\,d)\ \ (g\,h\,i\,j)\ \Big]\ , WH = \Big[\ (m\,n\,o\,p)\ \ (s\,t\,u\,v)\ \Big]$$

where,

W is the set of weight vectors in order of performance,
$M = n = 4$ (number of weight vectors generated),
BH is the best half list of vectors,
WH is the worst half list of vectors.

After Random Swapping,

$$W' = \Big[\; (a\,m\,c\,p) \;\; (b\,n\,o\,d) \;\; (s\,h\,t\,j) \;\; (g\,i\,u\,v) \; \Big]$$

where,

W' is the set of new weight vectors generated after swapping.
There are 4 div 2 = 2 swaps ((b with m , p with d), (s with g, t with i)).
div, represents the integer division function.

3.2.2. One-Point Crossover(M2) for Neural Nets

The idea behind one-point crossover was described in section 3.1.2. This idea has been applied to neural networks where we search for an optimal weight vector each of which represents a chromosome, and whose components (weights) represent genes.

Example: To illustrate this strategy, let us take a formal example. Initially, a set of M weight vectors W(below), is generated randomly within a given range. The input function is evaluated for each of these initial weight vectors, then the best half (M/2 weight vectors which give the optimal results) of the M weight vectors are separated into a list, the other half is discarded Then a new set of M/2 weight vectors are generated randomly into another list. The one-point crossover strategy is applied to two weight vectors at a time and the crossover point is chosen initially, so one-point crossover will be applied to the first weight vector in the best half list and the first weight vector in the newly generated weight vectors list, respectively. This procedure is repeated until the last set of weight vectors has undergone this strategy.

$$W = [\, W_1\, W_2 \ldots W_3 \ldots W_M]$$

Before Crossover:

$$W_1 = [\, w_1\; w_2\, w_3 \ldots w_n]$$
$$X_1 = [\, x_1\; x_2\; x_3 \ldots x_n \,]$$

where, W_1 and $X_1$are weight vectors. W_1 is the first weight vector in the best half list and X_1 is the first weight vector in the newly generated list. Their n components are individual weights needed for the network.

For this example, we choose $P = 3$ as n is arbitrary. For other cases, $P = n \text{ div } 2$ if n is even, otherwise $P = (n+1) \text{ div } 2$.

where, P represents the position the crossover is to begin and div is the integer division function.

$$L_1 = n,\ L_2 = n.$$

where, L_1 represents the number of weights of W_1 and L_2 represents the number of weights of X_1. For our purposes, the number of weights of W_1 and X_1 were restricted to be equal.

After Crossover:

$$W_1' = [\ w_1\, w_2\, x_3 \ldots x_n]$$
$$X_1' = [\, x_1\, x_2\, w_2 \ldots w_n]$$

Now, $W' \quad = \quad [\, W_1'\ X_1'\ W_2'\ X_2' \ldots W_n'\ X_n' \,]$

Also, please note here:

W' is the newly generated set of weight vectors.

W_1' and X_1' are the new weight vectors generated after the interchange.

n = (number of hidden units) *(number of input units + number of output units) of the network.

M = n if n is even, otherwise M = n + 1.

For the above example, W_1 is the first weight vector in the best list and X_1 is the first weight vector in the list of newly generated weight vectors. The point of crossover is chosen to be P = n div 2, in this case we choose P = 3, as n is arbitrary. The resultant set of weight vectors W_1' and $X_1'$are shown in the example. This procedure is repeated M/2 times, so at the end of this iteration a new set of M weight vectors is generated which are used in the next iteration. The upper bound on the number of iterations acceptable is 20000 iterations. This procedure is repeated until a weight vector is generated that optimizes the given function or the 20000 iterations are completed. This strategy was found to be $O(n^2)$.

An example is given to illustrate this strategy.

$$W = \Big[\ (a\,b\,c\,d)\ \ (g\,h\,i\,j)\ \ (e\,f\,k\,l)\ \ (q\,r\,w\,x)\ \Big]$$

$$BH = \big[\ (a\,b\,c\,d)\ \ (g\,h\,i\,j)\ \big]\ ,\ NH = \big[\ (m\,n\,o\,p)\ \ (s\,t\,u\,v)\ \big]$$

where,

W is the set of weight vectors in order of performance,

M = n = 4 (number of weight vectors generated),

BH is the best half list of vectors,

NH is the newly generated half list of vectors.

After Crossover,

$$W' = \left[\;(a\,b\,o\,p)\;\; (m\,n\,c\,d)\;\; (g\,h\,u\,v)\;\; (s\,t\,i\,j)\; \right]$$

where,

W' is the set of new weight vectors generated after swapping.

The crossover begins at position 2, since 4 div 2 = 2.

3.3. Experimental Results

Several experiments were conducted using the strategies mentioned earlier, in order to compare the performance of genetic algorithms to traditional neural network approaches. A genetic algorithm simulator was designed, and later implemented in the C programming language. For simplicity, the hierarchical neural network was chosen. The program was executed on the Next Cube (68040) machine running Unix, which was the available hardware at the time. The source code of the simulator implemented in the C programming language is given in the Appendix (Section 7).

To compare the performance of genetic algorithms, a set of functions were chosen (XOR, EQUIVALENCE (EQUIV), AND, OR, SINE, ENCODE) to be the test functions. XOR, EQUIV, AND, OR, each represented functions which had relatively small architectures, while SINE and ENCODE[4] represented functions which had larger architectures and greater complexity. A total of 12 experiments were conducted for each mutation strategy. Also, two different network sizes were used for each function, so two experiments were conducted for each of the above functions for each mutation strategy respectively. Two different network sizes were used, in order to check if the performance of genetic algorithms differed for different network sizes. The results obtained confirmed our suspicions that genetic algorithms seemed to work well with small architectures, but performed rather poorly with larger architectures. An upper bound of 20000 generations was chosen which gave the machine sufficient CPU time, this bound was used to represent nonconvergence. The XOR, EQUIV, OR, AND functions each had 2 binary inputs and 1 binary output. The SINE and ENCODE functions had 8 binary inputs and 8 binary outputs, respectively.

The next two sections describe the experimental results obtained using the two different mutation strategies. The entries in the tables are to be interpreted in the following way:

- MSE represents that the mean square error was used to determine the accuracy of the data.
- Avg. # of iterations gives the avg. # of the generations needed for convergence over T trials.
- # of Trials (T) indicates that the data was gathered over T executions of the program.
- Avg. Time indicates the time taken in seconds for the program to achieve results over T trials.
- Total Units indicates the network size used for the given function.
- Total Wts. indicates the number of weights needed for the network.

- Converge indicates how many Trials (T) were able to give the desired results in 20000 iterations.
- A dash (-) indicates the data was not available, since the program did not achieve convergence.
- An infinity (∞) indicates that the program did not converge in 20000 iterations.
- The range for the weights generated was chosen to be $-200 \le 0 \le 200$.
- For the SINE function, 20 function patterns were tried, which represented *a half period*(0 to π /2).

3.3.1. Results obtained for the random swapping strategy(M1)

Figure 4 displays the data collected for the random swapping strategy. Please note that for large network (SINE,ENCODE) problems the results are incomplete.

Function	MSE	Avg. # of Iterations	# Of Trials(T)	Avg. time	Total Units	Total Wts.	Converge
XOR	5e-6	208	20	3.38	6	9	20
EQUIV	0	294	20	4.65	6	9	20
OR	2.6e-3	6563	20	140	7	12	19
AND	1.9e-3	3657	20	93.7	7	12	20
SINE	-	∞	20	∞	24	128	0
ENCODE	-	∞	20	∞	22	96	0
XOR	0	90	20	2.23	7	12	20
EQUIV	2e-6	77	20	3.89	7	12	20
OR	2.5e-3	3366	20	147.8	9	18	20
AND	1.8e-3	979	20	43.8	9	18	20
SINE	-	∞	20	∞	32	256	0
ENCODE	-	∞	20	∞	24	128	0

Figure 4: Table for random swapping data

3.3.2. Results obtained for the one-point crossover strategy(M2)

Figure 5 displays the data collected for the one-point crossover strategy. Please note that for large network (SINE,ENCODE) problems the results were incomplete.

Function	MSE	Avg. # of Iterations	# Of Trials(T)	Avg. time	Total Units	Total Wts.	Converge
XOR	0	185	20	3.33	6	9	20
EQUIV	5e-6	226	20	3.92	6	9	20
OR	2.6e-3	4607	20	131.2	7	12	19
AND	1.9e-3	2233	20	55.1	7	12	20
SINE	-	∞	20	∞	24	128	0
ENCODE	-	∞	20	∞	22	96	0
XOR	1e-6	94	20	2.66	7	12	20
EQUIV	2e-6	91	20	2.69	7	12	20
OR	2.7e-3	3497	20	175.2	9	18	20
AND	2.8e-3	1036	20	53.9	9	18	20
SINE	-	∞	20	∞	32	256	0
ENCODE	-	∞	20	∞	24	128	0

Figure 5: Table for one-point crossover data

4. Observations

During the experimentations using genetic algorithms, we made some interesting observations:

- XOR, EQUIV, OR, AND converged rapidly if the network size was increased, but the time complexity also increased.
- For SINE and ENCODE both strategies converged for 1 or 2 patterns of the function, but convergencewas not achieved for greater number of patterns.
- For both the mutation strategies, the results were consistent with each other.
- If the range of the weights was decreased significantly, the convergence rates (# of iterations) decreased for XOR, OR, AND, EQUIV. Convergence for the SINE and ENCODE functions was not achieved if the range of weights was changed.

5. Conclusion

Based on the experiments conducted and the data gathered, we concluded that genetic algorithms seem to be very effective for simple functions (XOR, AND, etc) and small architectures, but as the complexity of the function increases (SINE, etc) and the network architectures become larger, genetic algorithms become ineffective and are less useful.

6. References

[1] Handbook of Genetic Algorithms: Lawrence Davis. Van Nostrand Reinhold. 1991.
[2] Genetic Algorithms in Search, Optimization & Machine Learning: David E. Goldberg. The Addison-Wesley Publishing Company,Inc. 1989.
[3] Genetic Algorithms and Simulated Annealing, Research Notes in Artificial Intelligence: Lawrence Davis. Morgan Kaufman Publishers,Inc. 1987.
[4] Parallel Distributed Processing, Explorations in the Microstructure of Cognition,Volume 1: David E. Rumelhart, James L. McClelland and the Research Group. The MIT Press. 1988.

7. Appendix

```
/* THIS PROGRAM USES A 3 LAYERED NEURAL NETWORK TO EMULATE THE BEHAVIOUR OF A
FUNCTION.  THE NUMBER OF OUTPUTS, INPUTS AND HIDDEN UNITS ARE ARBITRARY AND
CAN BE CHOSEN BY THE USER. THE NETWORK USED IS HIERARCHICAL.  A RANDOM NUMBER
GENERATOR IS USED TO GENERATE THE INITIAL SET OF WTS, THEN A GENETIC ALGORITHM
IS USED TO RANDOMLY PERMUTE THE WTS TO OBTAIN A MORE QUALIFIED SET OF WEIGHT
VECTORS.  THE PROGRAM DISPLAYS A WT_VECTOR  WHICH SATISFIES THE ERROR
TOLERANCE. THE SYSTEM TIME IS ALSO DISPLAYED. */


#include <stdio.h>
#include <math.h>
#include <sys/file.h>
#include <sys/time.h>
#include "rand.h"
#define MAX        100
#define INPUTFILE   "and.data"   /*INPUT FILE TO BE USED*/
#define NUMLAYERS   3
#define SQR(x)      ((x)*(x))
#define TOLERANCE   0.01
#define RANGE       200.0  /* RANGE OF RANDOM NUMBERS TO BE
                    GENERATED */
#define TOTAL       20000  /* MAXIMUM NUMBER OF GENERATIONS
                    ALLOWED */
#define COUNT       20     /* NUMBER OF TRIALS */

typedef struct cell
      {    float actvalue;
           float inputvalue;
           float wts[MAX];
      } LAYER[NUMLAYERS+1][MAX];  /* UNIT IN NETWORK */


typedef struct in_out
      {    float value[MAX];
      } IN_struct[MAX],OUT_struct[MAX];


typedef float WTS[MAX][MAX];       /*  WEIGHT ARRAY  */

typedef struct node
      {    int index;
           float value;
      } STORE[MAX];


IN_struct in_data;
OUT_struct out_data;


/* THIS ROUTINE PRINTS OUT THE SYSTEM TIME */

void TIME(void)

{  struct timeval tp;              /* SYSTEM STRUCT */
```

```
    struct timezone tzp;

    if (gettimeofday(&tp,&tzp) == 0)
    {  printf("The Time is %s",ctime(&tp.tv_sec));
       printf("Microsecs: %ld\n",tp.tv_usec);
    }
 }

 /* RETURNS A RANDOM NUMBER BETWEEN - BOUND AND BOUND */

 float Randomize(double BOUND)

 {     return(rand01()*2.0*BOUND - BOUND);
 }

 /* THIS ROUTINE GENERATES THE WT-VECTORS REQUIRED.  ALWAYS AN EVEN NUMBER OF
 WEIGHT  VECTORS ARE GENERATED. */

 void INIT(int numout,int numin,int numhid,WTS wt_array)

 {  int i,j;                      /* LOOP INDEXES */
    int numwts = (numin + numout) * (numhid); /* WTS NEEDED */
    int copy = numwts;

    srand(0);     /* INITIALIZE THE RANDOM NUMBER GENERATOR */
    if (ODD(numwts) == 1)
      numwts++;
    for (i=1;i<=numwts;i++)
       for (j=1;j<=copy;j++)
          wt_array[i][j] = Randomize(RANGE);
 }

 /* THIS ROUTINE SETS UP THE NETWORK AND ASSIGNS THE WTS TO THE CONNECTIONS IN
 THE NETWORK */

 void ASSIGN(LAYER net_array,WTS wt_array, int index,
         int numin, int numout,int numhid)

 {  int i,j;
    int count = 1;

    for (i=1;i<=numin;i++)
    {  for (j=1;j<=numhid;j++)
       {  net_array[1][i].wts[j]=wt_array[index][count];
          count++;
       }
    }
    for (i=1;i<=numhid;i++)
    {  for (j=1;j<=numout;j++)
       {  net_array[2][i].wts[j]=wt_array[index][count];
          count++;
       }
    }
 }

 /* ACTIVATION FUNCTION USED */

 float Lambda(float x)
```

```
{ return (1/(1 + exp(-1*x)));}

/* THIS ROUTINE COMPUTES THE INPUT AND ACTIVATION VALUES OF A UNIT IN THE
 NETWORK */

void COMPUTE(int numin,int numout, int numhid,
        LAYER net_array)

{  int i,j;              /* LOOP INDEXES */
   float temp=0;

   for (i=1;i<=numhid;i++)
   {  for (j=1;j<=numin;j++)
         temp += net_array[1][j].wts[i] * net_array[1][j].actvalue;
      net_array[2][i].inputvalue = temp;
      net_array[2][i].actvalue = Lambda(temp);
   }
   temp = 0;
   for (i=1;i<=numout;i++)
   {  for (j=1;j<=numhid;j++)
         temp += net_array[2][j].wts[i] * net_array[2][j].actvalue;

/* THIRD LAYER = OUTPUT UNITS */

      net_array[3][i].inputvalue = temp;
      net_array[3][i].actvalue = Lambda(temp);
   }
}


/* THIS ROUTINE FETCHES THE INPUT FROM A GIVEN INPUT FILE. */

void INPUT(int *numin, int *numout,int *numhid,int *patterns)

{  int i,j;
   FILE *fp;
   int test,fd;

   fd = open(INPUTFILE,O_RDWR,0700);
   fp = fdopen(fd,"r");
   fscanf(fp,"%d",patterns);
   fscanf(fp,"%d",numin);
   fscanf(fp,"%d",numhid);
   fscanf(fp,"%d",numout);
   for (i=1;i<= *patterns;i++)
   {  for (j=1;j<= *numin;j++)
         fscanf(fp,"%f",&in_data[i].value[j]);
      for (j=1;j<= *numout;j++)
         fscanf(fp,"%f",&out_data[i].value[j]);
   }
   fclose(fp);
   close(fd);
}

/* THIS SUBROUTINE COMPUTES THE MEAN SQUARE ERROR FOR EACH PATTERN OF THE
 FUNCTION */
```

```
void COMPUTE_ERR(int numout,LAYER net_array,
              OUT_struct out, float *ans,int index)

{   int i;
    float temp =0;
    float var = *ans;


    for (i=1;i<=numout;i++)
       temp += 0.5 * SQR(out[index].value[i] -
              net_array[3][i].actvalue);
    var += temp;
    *ans = var;
}

/* CHECK IF THE INTEGER IS ODD OR EVEN */

int ODD(int x)

{   if ((x % 2) == 0)
      return 0;
    else
      return 1;
}

/* RETURNS THE ABSOLUTE VALUE OF THE NUMBERS */

float ABS(float x)

{   if (x < 0)
      return (x * -1);
    else
      return x;
}


/* CHECKS IF THE ERROR IS WITHIN THE REQUIRED  TOLERANCE OR NOT */

int CHECK(STORE error,int numin,int numout,int numhid,
        int *index)

{
   if (error[1].value < TOLERANCE)
   {   *index = error[1].index;
      return 1;
   }
   else
     return 0;
}


/* SORTS THE ERROR ARRAY IN ASCENDING ORDER */

void SORT(STORE error,int num)

{   int i,j;
   float temp;
```

```
    for (j=2;j<=num;j++)
    {   temp = error[j].value;
        i = j-1;
        while ((i>0) && (error[i].value > temp))
        {  error[i+1].value = error[i].value;
           i--;
        }
        error[i+1].value = temp;
    }
}
```

```
/* THIS ROUTINE RANDOMLY SWAPS THE WEIGHTS OF TWO CHOSEN WEIGHTS VECTORS
 TO GENERATE ANOTHER SET OF WEIGHT VECTORS TO BE USED IN THE NEXT GENERATION */

void MUTATE1(STORE error,WTS wt_array,int numin,int numout, int numhid)

{  int i,j;                          /* LOOP INDEXES */
   int num = (numin+numout)*numhid;             /* NUMBER OF WEIGHTS */
   int copy = num;
   float var;
   int rand1,rand2;


   if (ODD(num) == 1)
     num++;
   num = (int)(num/2);
   for (i=1;i<=num;i++)
      for (j=1;j<=(int)(num/2);j++)
      {  rand1 = (int) ABS(Randomize((double)copy)) + 1;
         rand2 = (int) ABS(Randomize((double)copy)) + 1;
   /* SWAP THE WEIGHTS */
         var = wt_array[i][rand1];
         wt_array[i][rand1] = wt_array[i+num][rand2];
         wt_array[i+num][rand2] = var;
      }
}
```

```
/* THIS ROUTINE USES ONE-POINT CROSSOVER TO GENERATE A NEW SET OF WEIGHT
VECTORS. THE BEST HALF WEIGHT VECTORS ARE RETAINED AND ANOTHER HALF WEIGHT
VECTORS ARE GENERATED. THEN THE ONE-POINT CROSSOVER STRATEGY IS APPLIED TO
THEM TO GENRATE THE NEW SET */

void MUTATE2(STORE error,WTS wt_array,int numin,int numout, int numhid)

{  int i,j;
   int num = (numin+numout)*numhid;    /* NUMBER OF WEIGHTS */
   WTS temp,new_wts,worst;           /* HOLDS THE NEW WEIGHTS GENERATED */
   int copy = num;
   float var;

   if (ODD(num) == 1)
     num++;
   num = (int)(num/2);
```

```
  for (i=1;i<=num;i++)  /* save best weights*/
    for (j=1;j<=copy;j++)
      temp[i][j] = wt_array[error[i].index][j];
  for (i=1;i<=num;i++)
    for (j=1;j<=copy;j++)
  /* GENERATE THE NEW WEIGHTS */
      new_wts[i][j] = Randomize(RANGE);
  for (i=1;i<=num;i++) /* use one-point crossover */
    for (j=1;j<=num;j++)
    {  var = temp[i][j];
      temp[i][j] = new_wts[i][j];
      new_wts[i][j] = var;
    }
  for (i=1;i<=num;i++)
    for (j=1;j<=copy;j++)
      wt_array[i][j] = temp[i][j];
  for (i=num+1;i<= 2*num;i++)
    for (j=1;j <=copy;j++)
      wt_array[i][j] = new_wts[i-num][j];
}


/* THIS SUBPROGRAM INVOKES THE SUBROUTINES TO CREATE THE NEEDED CYCLE IN
ORDER TO DETERMINE THE BEST SUITED WEIGHT VECTOR FOR THE NETWORK */

void START_LOOP(int numin,int numout,int numpatterns,
          int numhid,WTS wt_array,LAYER net_array,
          IN_struct in, OUT_struct out, STORE error,
          int *generations)

{  int i,j,k;                        /* LOOP INDEXES */
  int  temp = (numin+numout)*numhid;          /* NUMBER OF WEIGHTS */
  float result = 0;
  int index =0;
  int sw =-1;                        /* LOOP VARIABLE */
  int gen = 1;                       /* GENERATION COUNT */
  int copy=temp;

  INIT(numout,numin,numhid,wt_array);
  if (ODD(temp) == 1)
    temp++;
  while (sw != 0)
  {  for (i=1;i<=temp;i++)
    {  ASSIGN(net_array,wt_array,i,numin,numout,numhid);
      for (j=1;j<=numpatterns;j++)
      {  for (k=1;k<=numin;k++)
        {  net_array[1][k].inputvalue = in[j].value[k];
          net_array[1][k].actvalue =
          Lambda(in[j].value[k]);
        }
        COMPUTE(numin,numout,numhid,net_array);
        COMPUTE_ERR(numout,net_array,out,&result,j);
      }
      error[i].value = result/(numpatterns);
      error[i].index = i;
      result = 0;
    }
    SORT(error,temp);
```

```
    if (CHECK(error,numin,numout,numhid,&index) == 1)
    { sw == 0;
      break;
    }
    MUTATE2(error,wt_array,numin,numout,numhid);
    gen++;
  }
  if (gen > TOTAL)
  { printf("No convergence in %d generations\n",TOTAL);
    return;
  }
  if (index != 0)
  {  printf("Convergence!! After %d generations\n",gen);
     *generations += gen;
     printf("For weight_vector W:\n");
     for (i=1;i<=copy;i++)
        printf("                %f\n",wt_array[index][i]);
  }
}

/* MAIN DRIVER ROUTINE */

void main(void)

{   LAYER     network;
    WTS       weight_array;
    STORE     error;
    int numin;
    int numout;
    int numhid;
    int numpatterns;
    int i,j;
    float total_error = 0;
    int generations = 0;

    for (i=1;i<=COUNT;i++)
    {  TIME();
       INPUT(&numin,&numout,&numhid,&numpatterns);
       START_LOOP(numin,numout,numpatterns,numhid,
         weight_array,network,in_data,out_data,error,
         &generations);
       if (error[1].value < TOLERANCE)
         total_error += error[1].value;
       TIME();
    }
    printf("Cumulative Error/Nooftrials =  %f\n",total_error/COUNT);
    printf("NoOfGenerations/Nooftrials = %d\n",generations/COUNT);

}
```