An Evolutionary Approach to Training Feed-Forward and Recurrent Neural Networks.

Jeff Riley
Hewlett Packard Australia
31-41 Joseph St.
Blackburn 3130
Australia
jeff@aus.hp.com

Victor B. Ciesielski
Department of Computer Science
Royal Melbourne Institute of Technology
GPO Box 2476V
Melbourne 3001
Australia
vc@goanna.cs.rmit.edu.au

Abstract

This paper describes a method of utilising genetic algorithms to train fixed architecture feed-forward and recurrent neural networks. The technique described uses the genetic algorithm to evolve changes to the weights and biases of the network rather than the weights and biases themselves. Results achieved by this technique indicate that for many problems it compares very favourably with the more common gradient descent techniques for training neural networks, and in some cases is superior. The technique is useful for those problem which are known to be difficult for the gradient descent techniques.

1 Introduction

The most common techniques for training both feed-forward and recurrent neural networks are variations of the gradient descent technique. Gradient descent techniques suffer from well-known problems, so more efficient and effective methods to determine network weights are desirable. One such method combines another biologically inspired technique, that of genetic algorithms, with neural networks.

Developed by John Holland [5], a genetic algorithm is a biologically inspired search technique. In simple terms, the technique involves generating a random initial population of individuals, each of which represents a potential solution to a problem. Each member of that population's fitness as a solution to the problem is evaluated against some known criteria. Members of the population are then selected for reproduction based upon that fitness, and a new generation of potential solutions is generated from the offspring of the most fit individuals. The process of evaluation, selection, and recombination is iterated until the population converges to an acceptable solution.

The evaluation of an individual's worth as a solution is achieved by the use of a fitness function. The goal of the fitness function is to numerically encode the performance of the individual with reference to the problem for which it is a potential solution. This is an extremely important process, for without a fitness function which accurately evaluates the performance of potential solutions, the search will fail.

There are several advantages to using genetic algorithms over the gradient descent techniques. Two of the most important are that they require no knowledge about the response surface, including gradient information, and that they are far less likely to become trapped in a local minimum.

Several variations of the genetic algorithm-neural network (GA-NN) hybrid exist; the most common of which are the determination of network weights by the use of genetic algorithms [2], [7], [8], [12], [18], and the evolutionary design of the network architecture [1], [4], [11], [13].

This paper focuses on the determination of network weights and biases for fixed architecture feed-forward and recurrent networks.

A major obstacle to using genetic algorithms to evolve the weights of a fixed network is the encoding of the weights onto the chromosome. The weights of a neural network are generally real-valued and unbounded, whereas a chromosome in a genetic algorithm is generally a string of bits of some arbitrary length. Encoding real values onto

such a chromosome presents problems both in the precision of the representation and the resultant length of the chromosome. The length of the chromosome impacts upon the size of the search space of the genetic algorithm, and the efficiency of the search. Obviously for very large neural networks with many connections, a chromosome encoded with full precision real numbers would be extremely long and difficult to deal with computationally.

Schraudolph and Belew [15] describe a mechanism for avoiding the sacrifice of either precision or search efficiency when encoding real-valued weights onto a fixed length chromosome. This technique, called *Dynamic Parameter Encoding* by Schraudolph and Belew, adapts the encoding scheme used for the chromosome such that the genetic algorithm concentrates its efforts on the most significant part of the weights early in the search, and on refining the search in the later stages. The assumption of this technique is that by using a coarse representation of the weights initially, the genetic algorithm will converge to a solution which, while not being the best solution, is a good solution. The search for the best solution may then continue, but since only the region already identified needs to be searched, the weights may now be re-encoded onto the same chromosome with greater precision.

Montana and Davis [12] describe a method of using real number coding instead of binary value coding for the chromosome. This avoids the problems associated with encoding real numbers onto the genetic algorithm's chromosome. In the method described by Montana and Davis the genes in the chromosome are not bits, but real numbers. The chromosome is just an ordered list of real numbers representing the weights of the network. The crossover operator is implemented such that two parents create a single offspring by, for each of the offspring's non-input units, randomly selecting one parent and copying the incoming weights from that parent's corresponding unit to the offspring. The mutation operator adds a random number between *−1.0* and *+1.0* to the incoming weights of a selected number of non-input units. This technique avoids the problems associated with encoding real numbers onto fixed length bit strings, but since the only way in which the weights can be changed is by mutation, it relies to some degree on the best weights being part of the initial population. What is really evolved by this technique is the best combination of weights rather than the weights themselves.

The work presented in this paper investigates a method of encoding network weights and biases onto a chromosome similar to [12], and utilising a genetic algorithm to determine the weights and biases of fixed architecture artificial neural networks. The goal of this work is to show that this new method of using a genetic algorithm to train neural networks is capable of training both feed-forward and recurrent networks.

2 Method Description

The method described here is based on the 2DELTA-GANN technique [7, 8] which uses a genetic algorithm to train fixed architecture feed-forward neural networks.

2.1 2DELTA-GANN

The approach used in 2DELTA-GANN is to have the genetic algorithm evolve a change, or delta value, to the weights and biases of the neural network being trained. This is done by modifying the weights and biases of the network by some value according to a combination of fixed rules. Rather than have the genetic algorithm evolve the actual weights and biases of the network, only the way in which the rules are to be applied and the delta values are evolved. As a result, the chromosome being modified by the genetic algorithm does not need to represent each weight and bias of the network; only the method by which the rules are to be applied and the delta values need be represented.

The gene structure used by Krishnan is an attempt to overcome the problems associated with encoding real numbers onto a chromosome represented as a bit string. In the 2DELTA-GANN method, each gene on the chromosome is a composite structure representing either a weight or a bias from the neural network. The gene is composed of three rule bits and two floating point values. The floating point values are manipulated in accordance with the rule bits to apply a change to the weights and biases of the network. The three rule bits are denoted $x1$, $x2$ and $x3$; and the two floating point values are denoted **delta1** and **delta2**.

The basic idea of the 2DELTA-GANN method is to use the rule bits to specify a simple heuristic to apply to the **delta2** value, which is then used to modify the value of **delta1**. Finally, **delta1** is used to modify the network weight or bias associated with the gene.

2DELTA-GANN implements a fitness scaling method of selection. Modified methods of crossover and mutation are used to manipulate the chromosome formed from the combination of the gene structures.

2DELTA-GANN uses a modified form of parameterized uniform crossover [17]. Since the gene structure implemented by 2DELTA-GANN involves a composite structure, the chromosome is not a simple bit string, so the crossover operator was modified to accommodate the composite gene structure. As part of the modified uniform crossover operation the 2DELTA-GANN crossover operator will swap the delta values of any gene whose rule bits were affected by crossover.

Similarly, the mutation operator was also modified to accommodate the composite gene structure. Each bit on the chromosome is a candidate for mutation with some arbitrary probability. If a gene has any rule bit mutated, then *delta1* and *delta2* for that gene are assigned randomly generated values.

Because of the modifications to the crossover and mutation operators, the representation of the genetic structure can be somewhat simplified. In the 2DELTA-GANN method each gene is represented as three bits, corresponding directly to the rule bits *x1*, *x2* and *x3*. Each gene then has associated with it two floating point values, corresponding to *delta1* and *delta2*. In reality, only the rule bits are combined to form the chromosome. This allows the chromosome to be kept to a minimum length, and further means that the floating point values do not need to be represented as bit strings on the chromosome. The values of *delta1* and *delta2* become attachments to the chromosome.

The crossover and mutation operators are aware of the chromosome structure and the floating point values associated with each gene. These modified operators are designed such that if any rule bits of a particular gene are affected by the operation, then the *delta1* and *delta2* values associated with that gene are also affected. In the case of crossover, if any bit on a gene is affected by crossover, then the *delta1* and *delta2* values of the offspring will be swapped. For mutation, if any bit on a gene is mutated, then the *delta1* and *delta2* values for that gene will be replaced with randomly generated values. Figure 1 shows the composite gene structure and an example of the modified crossover operator.
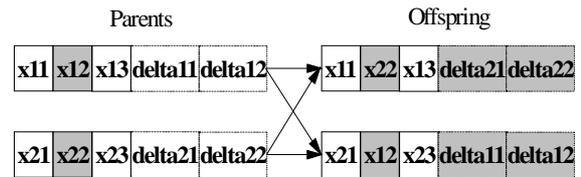

Figure 1 Modified Crossover.

2DELTA-GANN utilises the GAUCSD genetic algorithm package [16] which allows a user supplied fitness function. The standard crossover and mutation functions of GAUCSD were modified for the new gene structure.

Figure 2 shows a simple XOR network, a sample chromosome, and possible corresponding *delta1* and *delta2* values for each gene on the chromosome. The chromosome is applied to the neural network by first applying the heuristic specified by the rule bits *x1*, *x2* and *x3* for each gene to the *delta1* and *delta2* values corresponding the those genes. The weight or bias associated with each gene is then modified by the value of *delta1*.
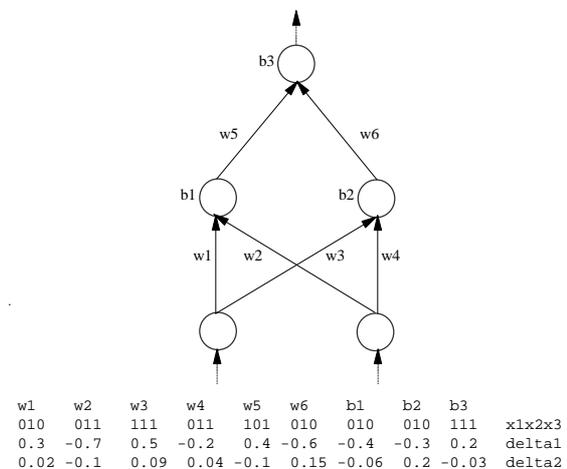


```
w1    w2    w3    w4    w5    w6    b1    b2    b3
010   011   111   011   101   010   010   010   111    x1x2x3
0.3  -0.7   0.5  -0.2   0.4  -0.6  -0.4  -0.3   0.2    delta1
0.02 -0.1   0.09  0.04 -0.1   0.15 -0.06  0.2  -0.03   delta2
```

Figure 2 XOR network with sample chromosome.

The chromosome is applied to the network by interpreting the rule bits in the following way:

```
if x1 = 1 then
     if x2 = 1 and x3 = 1 then
             double delta2
     else
             halve delta2
     endif
     add delta2 to delta1
endif
add delta1 to weight (or bias)
```

3

2.2 The New Method

After analysis of the algorithm and experimental results, it is believed that the value of the 2DELTA-GANN method is not in the evolution of the rules, but in the usefulness of genetic recombination as it is applied to the delta values themselves.

To demonstrate this, in this work the chromosome is applied to the network in the following way:

```
if Rand() > 0.5
    if Rand() > 0.5 then
            double delta2
    else
            halve delta2
    endif
    add delta2 to delta1
endif
add delta1 to weight (or bias)
```

Even though the rule bits are not being used in the application of the delta values to the weights, crossover and mutation of the delta values follow the same rules: the delta values of any gene whose rule bits would have been affected by crossover will be exchanged; and the deltas for any gene whose rule bits would have been affected by mutation are assigned randomly generated values.

The genetic algorithm for this work is a modified version of the algorithm implemented by Krishnan, and is as follows:

Step 1: Initialise the network weights to random values
Step 2: Create a population with randomly assigned rule bits and delta values
Step 3: Evaluate each member of the population
Step 4: If the best performer is an improvement on the previous best, then
    Step 4.1: Apply the delta values of the best performer to the network weights to calculate new network weights
    Step 4.2: Re-evaluate each member of the population against the new network weights
Step 5: Rank and select members of the population for reproduction
Step 6: Perform crossover
Step 7: Perform mutation
**Step 8:** If not finished goto Step 3

A new fitness function was also coded to allow evaluation of recurrent networks of several different recurrent architectures.

3 Network Architecture

Several feed-forward and recurrent neural networks were tested as part of this work. No attempt was made to modify network topology to avoid the permutations (or competing conventions) problem [14]. While the effect of the permutations problem is a possible lack of diversity in an apparently diverse population and premature convergence of the genetic algorithm, this did not seem to adversely affect the tests conducted as part of this work.

The recurrent architectures tested here include both Simple Recurrent Networks (SRN) [3] and Real Time Recurrent Networks (RTRN) [20]. A modified Jordan network [6, 9] was also tested for sequence generation.

The XOR network shown in figure 2 is an example of the feed-forward architecture tested. The networks shown in figure 3 and figure 4 are examples of the recurrent architectures tested.
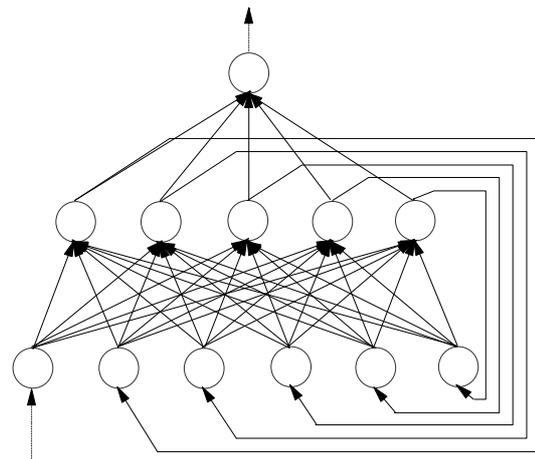
Figure 3 Simple Recurrent Network.
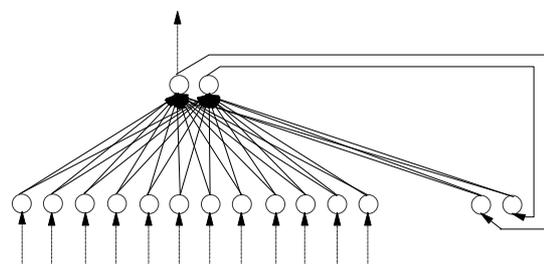
Figure 4 Real Time Recurrent Network.

4

Network		New Method	2DELTA-GANN	Whitley*	Back-Propagation
XOR	Success Rate	90%	92%	100%	90%
	Evaluations	4666	2392	500	2168
	Population	200	200	100	*n/a*
4-2-4 Encoder	Success Rate	100%	92%	100%	100%
	Evaluations	5134	6136	25000	118
	Population	200	200	1000	*n/a*
Digital to Analogue Converter	Success Rate	100%	96%	*n/a*	100%
	Evaluations	39438	23040	*n/a*	222
	Population	1000	1000	*n/a*	*n/a*
Sequence Generator	Success Rate	74%	18%	*n/a*	80%
	Evaluations	115526	89400	*n/a*	40962
	Population	500	500	*n/a*	*n/a*

Table 1 Comparison of methods over 50 experiments (*25 for Whitley).

The data presented in table 1 are the results from 50 experiments for each method with each of the networks listed, with the exception of the Whitley GA-NN method which are for 25 experiments. The results presented is:

- the percentage of experiments which were successful.
- the average number of network evaluations for all experiments.
- the population size (for the GA-NN methods).

Network		New Method	Back-Propagation
Sine SRN*	Training Error	0.00278	0.04178
	Evaluations	15000	31630
	Population	100	*n/a*
Sunspot SRN	Training Error	0.119	0.102
	Evaluations	100000	10000
	Population	1000	*n/a*
Sunspot RTRN	Training Error	0.153	0.183
	Evaluations	100000	10000
	Population	1000	*n/a*

Table 2 Comparison of methods over 5 experiments (*10 for Sine network).

The data presented in table 2 are the results from 5 experiments for each method with each of the networks listed, with the exception of the sine network which are for 10 experiments. The results presented are:

- the average normalised training error for all experiments.
- the average number of network evaluations for all experiments.
- the population size (for the GA-NN methods).

4 Results

Several feed-forward networks were tested using the new method. These were the standard XOR network, the 4-2-4 encoder network, and the digital to analogue converter network.

The results for these tests are shown in table 1. Also shown in table 1 are the results for the sequence generator recurrent network. As a comparison with another GA-NN approach, table 1 also shows some results from [18] and [19]. Some of this data is reproduced from [7]. Results were averaged over a 50 experiments for all methods

except Whitley, which are averaged over 25 experiments.

Simple recurrent networks were tested to predict the next number in a series. Two different data series were used: the sine curve and yearly sunspot data. A real time recurrent network was also tested using the sunspot data. Results for these tests are shown in table 2. Results using back-propagation to train these networks are also shown in table 2. The back-propagation results are from [10]. Results were averaged over a 10 experiments for the sine network, and over 5 experiments for the sunspot networks.

For the purpose of comparison with back-propagation, a single evaluation of an individual of the genetic algorithm's population is considered to be computationally equivalent to a single (forward or backward) pass of the back-propagation technique. A generation requires evaluation of each member of the population for the entire training data set, so in terms of back-propagation a generation is typically computationally equivalent to a number of epochs. The number of epochs a generation is equivalent to varies with the size of the population.

For example, for the original 2DELTA-GANN method and a population of 200 networks which takes 15 generations to train, each network will have been evaluated 15 times for any one input pattern, so there will have been $15 \times 200 = 3000$ network evaluations per input pattern. For the new method, which introduces an extra evaluation, each network will have been evaluated twice for each generation, so there will have been $15 \times 2 \times 200 = 6000$ network evaluations per input pattern.

Similarly, for back-propagation, a training period of 175 epochs will have resulted in $175 \times 2 = 350$ network evaluations per input pattern (this takes into consideration the forward and backward passes of the back-propagation technique).

For ease of comparison, the number of epochs for the back-propagation technique and the number of trials for GA-NN techniques are converted to the number of network evaluations per input pattern.

In terms of network accuracy, for the feed-forward networks tested the new method out-performs the original 2DELTA-GANN and the Whitley GA-NN methods for all but the XOR network, and compares very well with back-propagation. In terms of training time, the new method out-performs Whitley for the 4-2-4 encoder network, is slightly behind the original 2DELTA-GANN and

does not compare quite so favourably with back-propagation.

The new method out-performs recurrent back-propagation for the SINE simple recurrent network in terms of both network accuracy and training time. For the sunspot networks, recurrent back-propagation and real time recurrent learning both out-perform the new method in terms of training time, but in terms of network accuracy the new method compares quite favourably.

5 Conclusions

A new method of training both feed-forward and recurrent neural networks was presented and shown to compare very favourably with other GA-NN approaches as well as back-propagation in terms of network accuracy. Training times of the new method compare well with other GA-NN approaches and with back-propagation in some cases. The value of this new method is that it requires no gradient information and is less susceptible than back-propagation to local variations in the error surface.

This new method of training both feed-forward and recurrent neural networks is likely to be most useful for problems which are found to be difficult for the more common gradient descent techniques to solve.

The comparison of the new method with the original 2DELTA-GANN also indicates that the search is being driven by crossover between the real-valued delta values and not by the evolution of the heuristic rules. This result highlights the usefulness of crossover in genetic search.

6 Bibliography

[1] Angeline, P. J., Saunders, G. M., and Pollack, J. B., "An Evolutionary Algorithm that Constructs Recurrent Neural Networks". In *IEEE Transactions on Neural Networks, Vol. 5, No. 1, 1994.*

[2] Belew, R. K., McInerney, J., and Schraudolph, N. N., "Evolving Networks: using the Genetic Algorithm with Connectionist Learning". *CSE Technical Report CS90-174, Computer Science and Engineering Department, University of California at San Diego, California, USA, 1990.*

[3] Elman, J. L., "Distributed Representations, Simple Recurrent Networks, and Grammatical Structure". In *Machine Learning 7, 1991.*

[4] Esparcia-Alcázar, A., and Sharman, K., "Evolving Recurrent Neural Network Architectures by Genetic Programming". In *Genetic Programming 1997: Proceedings of the Second Annual Conference.*

[5] Holland, J. H., "Adaptation in Natural and Artificial Systems". *Ann Arbor: The University of Michigan Press, 1975.*

[6] Jordan, M. I., "Attractor Dynamics and Parallelism in a Connectionist Sequential Machine". In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society. Lawrence Erlbaum, 1986.*

[7] Krishnan, R., "2DELTA-GANN: A New Method of Training Neural Networks Using Genetic Algorithms". *Master of Applied Science thesis, Department of Computer Science, Royal Melbourne Institute of Technology, Victoria, Australia, 1994.*

[8] Krishnan, R., and Ciesielski, V. B., "2DELTA-GANN: A New Approach To Training Neural Networks Using Genetic Algorithms". In *Proceedings of the Fifth Australian Conference on Neural Networks, 1994.*

[9] McClelland, J. L., and Rumelhart, D. E., "Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises". *MIT Press, Cambridge, MA., 1988.*

[10] McCluskey, P. C., "Feedforward and Recurrent Neural Networks and Genetic Programs for Stock Market and Time Series Forecasting". *Master of Applied Science thesis, Department of Computer Science, Brown University, Rhode Island, USA, 1993.*

[11] Miller, G. F., Todd, P. M., and Hegde, S. U., "Designing Neural Networks using Genetic Algorithms". In *Proceedings of the Third International Conference on Genetic Algorithms, 1989 (ICGA-89).*

[12] Montana, D. J., and Davis, L. D., "Training Feedforward Networks using Genetic Algorithms". In *Proceedings of the International Joint Conference on Artificial Intelligence, 1989 (IJCAI-89).*

[13] Moriarty, D., and Miikkulainen, R., "Hierarchical Evolution of Neural Networks". *Technical Report AI96-242, Department of Computer Sciences, The University of Texas at Austin, Texas, USA, 1996.*

[14] Radcliffe, N. J., "Genetic Neural Networks on MIMD Computers". *PhD. thesis, Physics Department, University of Edinburgh, Scotland, 1990.*

[15] Schraudolph, N. N., and Belew, R. K., "Dynamic Parameter Encoding for Genetic Algorithms". *Technical Report LAUR 90-2795, Los Alamos National Laboratory Center for Nonlinear Studies, Los Alamos, NM, 1990.*

[16] Schraudolph, N. N., and Grefenstette, J. J., "A User's Guide to GAUCSD 1.4, July 1992". *Technical Report CS92-249, Computer Science and Engineering Department, University of California, San Diego, California, USA, 1992.*

[17] Spears, W. M., and DeJong, K. A., "On the Virtues of Parameterized Uniform Crossover". In *Proceedings of the Fourth Annual Conference on Genetic Algorithms, 1991 (ICGA-91).*

[18] Whitley, D., and Hanson, T., "Optimizing Neural Networks Using Faster More Accurate Genetic Search". In *Proceedings of the Third International Conference on Genetic Algorithms, 1989 (ICGA-89).*

[19] Whitley, D., Dominic, S. and Das, R., "Genetic Reinforcement Learning with Multilayered Neural Networks". In *Proceedings of the Fourth International Conference on Genetic Algorithms, 1991 (ICGA-91).*

[20] Williams, R. J., and Zipser, D., "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks". In *Neural Computation 1, 1989.*