Training Feedforward Neural Networks Using Genetic Algorithms

David J. Montana and **Lawrence Davis**
BBN Systems and Technologies Corp.
10 Moulton St.
Cambridge, MA 02138

Abstract

Multilayered feedforward neural networks possess a number of properties which make them particularly suited to complex pattern classification problems. However, their application to some real-world problems has been hampered by the lack of a training algorithm which reliably finds a nearly globally optimal set of weights in a relatively short time. Genetic algorithms are a class of optimization procedures which are good at exploring a large and complex space in an intelligent way to find values close to the global optimum. Hence, they are well suited to the problem of training feedforward networks. In this paper, we describe a set of experiments performed on data from a sonar image classification problem. These experiments both 1) illustrate the improvements gained by using a genetic algorithm rather than backpropagation and 2) chronicle the evolution of the performance of the genetic algorithm as we added more and more domain-specific knowledge into it.

1 Introduction

Neural networks and genetic algorithms are two techniques for optimization and learning, each with its own strengths and weaknesses. The two have generally evolved along seperate paths. However, recently there have been attempts to combine the two technologies. Davis (1988) showed how any neural network can be rewritten as a type of genetic algorithm called a classifier system and vice versa. Whitley (1988) attempted unsuccessfully to train feedforward neural networks using genetic algorithms. In this paper we describe a different genetic algorithm for training feedforward networks. It not only succeeds in its task but it outperforms backpropagation, the standard training algorithm, on a difficult example. This success comes from tailoring the genetic algorithm to the domain of training neural networks. We document the evolution and ultimate success of this algorithm with a series of experiments.

The paper is structured as follows. Sections 2 and 3 give an overview of neural networks and genetic algorithms respectively with a special emphasis on their strengths and weaknesses. Section 4 describes the data on which the experiments were run. Section 5 details the genetic algorithm we used to perform neural network weight optimization. Section 6 describes the experiments we ran and analyzes their results. Section 7 provides conclusions about our work and suggestions for future work.

2 Neural Networks

Neural networks are algorithms for optimization and learning based loosely on concepts inspired by research into the nature of the brain. They generally consist of five components:

1. A directed graph known as the network topology whose arcs we refer to as links.
2. A state variable associated with each node.
3. A real-valued weight associated with each link.
4. A real-valued bias associated with each node.
5. A transfer function for each node which determines the state of a node as a function of a) its bias θ, b) the weights, w_i of its incoming links, and c) the states, x_i, of the nodes connected to it by these links. This transfer function usually takes the form $f(\Sigma w_i x_i - \theta)$ where f is either a sigmoid or a step function.

A feedforward network is one whose topology has no closed paths. Its input nodes are the ones with no arcs to them, and its output nodes have no arcs away from them. All other nodes are hidden nodes. When the states of all the input nodes are set, all the other nodes in the network can also set their states as values propagate through the network. The operation of a feedforward network consists of calculating outputs given a set of inputs in this manner. A layered feedforward network is one such that any path from an input node to an output node traverses the same number of arcs. The nth layer of such a network consists of all nodes which are n arc traversals from an input node. A hidden layer is one which contains hidden nodes. Such a network is fully connected if each node in layer i is connected to all nodes in layer i+1 for all i.

Layered feedforward networks have become very popular for a few reasons. For one, they have been found in practice to generalize well, i.e. when trained on a relatively sparse set of data points, they will often provide the right output for an input not in the training set. Secondly, a training algorithm called backpropagation exists which can often find a good set of weights (and biases) in a reasonable amount of time [Rumelhart 1986a]. Backpropagation is a variation on gradient search. It generally uses a least-squares optimality

criterion. The key to backpropagation is a method for calculating the gradient of the error with respect to the weights for a given input by propagating error backwards through the network.

There are some drawbacks to backpropagation. For one, there is the "scaling problem". Backpropagation works well on simple training problems. However, as the problem complexity increases (due to increased dimensionality and/or greater complexity of the data), the performance of backpropagation falls off rapidly. This makes it infeasible for many real-world problems including the one described in Section 4. The performance degradation appears to stem from the fact that complex spaces have nearly global minima which are sparse among the local minima. Gradient search techniques tend to get trapped at local minima. With a high enough gain (or momentum), backpropagation can escape these local minima. However, it leaves them without knowing whether the next one it finds will be better or worse. When the nearly global minima are well hidden among the local minima, backpropagation can end up bouncing between local minima without much overall improvement, thus making for very slow training.

A second shortcoming of backpropagation is the following. To compute a gradient requires differentiability. Therefore, backpropagation cannot handle discontinuous optimality criteria or discontinuous node transfer functions. This precludes its use on some common node types and simple optimality criteria.

For a more complete description of neural networks, the reader is referred to [Rumelhart 1986b].

3 Genetic Algorithms

Genetic algorithms are algorithms for optimization and learning based loosely on several features of biological evolution. They require five components:

1. A way of encoding solutions to the problem on chromosomes.
2. An evaluation function that returns a rating for each chromosome given to it.
3. A way of initializing the population of chromosomes.
4. Operators that may be applied to parents when they reproduce to alter their genetic composition. Included might be mutation, crossover (i.e. recombination of genetic material), and domain-specific operators.
5. Parameter settings for the algorithm, the operators, and so forth.

Given these five components, a genetic algorithm operates according to the following steps:

1. The population is initialized, using the procedure in C3. The result of the initialization is a set of chromosomes as determined in C2.
2. Each member of the population is evaluated, using the function in C1. Evaluations may be normalized; the important thing is to preserve relative ranking of evaluations.
3. The population undergoes reproduction until a stopping criterion is met. Reproduction consists of a number of iterations of the following three steps:
   (a) One or more parents are chosen to reproduce. Selection is stochastic, but the parents with the highest evaluations are favored in the selection. The parameters of C5 can influence the selection process.
   (b) The operators of C4 are applied to the parents to produce children. The parameters of C5 help determine which operators to use.
   (c) The children are evaluated and inserted into the population. In some versions of the genetic algorithm, the entire population is replaced in each cycle of reproduction. In others, only subsets of the population are replaced.

When a genetic algorithm is run using a representation that usefully encodes solutions to a problem and operators that can generate better children from good parents, the algorithm can produce populations of better and better individuals, converging finally on results close to a global optimum. In many cases (such as the example discussed in this paper), the standard operators, mutation and crossover, are sufficient for performing the optimization. In such cases, genetic algorithms can serve as a black-box function optimizer not requiring their creator to input any knowledge about the domain. However, as illustrated in this paper, knowledge of the domain can often be exploited to improve the genetic algorithm's performance through the incorporation of new operators.

Genetic algorithms should not have the same problem with scaling as backpropagation. One reason for this is that they generally improve the current best candidate monotonically. They do this by keeping the current best individual as part of their population while they search for better candidates. Secondly, genetic algorithms are generally not bothered by local minima. The mutation and crossover operators can step from a valley across a hill to an even lower valley with no more difficulty than descending directly into a valley.

The field of genetic algorithms was created by John Holland. His first book [Holland 1975] was an early landmark. The best introduction for the interested reader is [Goldberg 1988].

4 The Data

The data used in the experiments were from one stage in the processing of passive sonar data from arrays of underwater acoustic receivers. BBN has been building an expert system to detect and reason about interesting signals in the midst of the wide variety of acoustic noise and interference which exist in the ocean. The main inputs to the expert system (as well as to sonar operators) are lofargrams, which (like spectograms) are intensity-based images depicting the distribution of acoustic energy as a function of frequency and time. Narrowband energy appears on the lofargrams as "lines" which tend to be predominantly vertical. A line has certain characteristics which provide clues to sonar operators as to what is producing it. Deriving algorithms which capture all that the operators can see in real lines has never succeeded despite a large amount of work in the area. Therefore, we are attempting to use neural networks in the problem of line characterization.

To start, we formed a database of approximately 1200

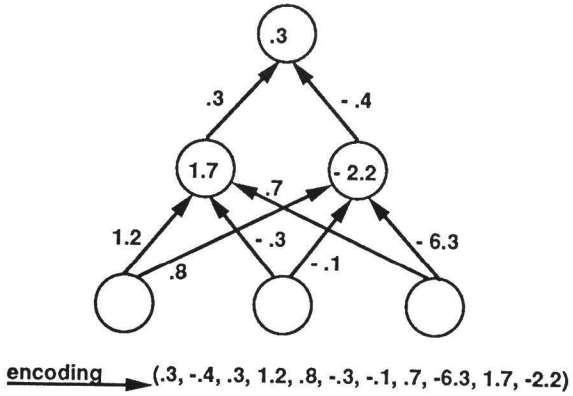

Figure 1: Encoding a Network on a Chromosome

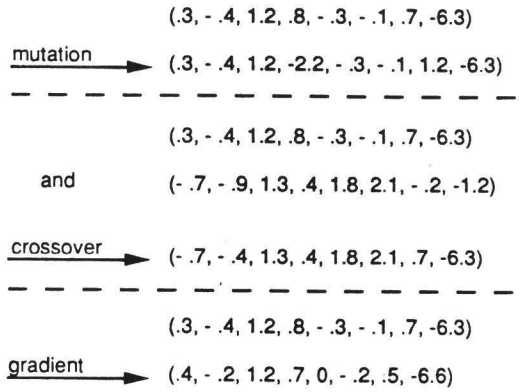

Figure 2: Operation of the Operators

fixed-size rectangular pieces of lofargrams, each centered on a line of type determined by an expert. Around 30% of these were from lines of a type in which we were particularly interested, and around 70% were from lines of a variety of other types. One experiment we ran investigated whether a feedforward network could classify the line pieces as either interesting or not based on four of the parameters which our system presently uses to characterize such pieces. The network had four inputs, one output, and first and second hidden layers of seven and ten nodes respectively for a total of 126 weights. We used a subset of the examples of size 236 as a training set. It is on this network that the comparative runs described in Section 6 were made.

5 Our Genetic Algorithm

We now discuss the genetic algorithm we set up to do neural network weight optimization. We start by describing the five components of the algorithm listed in Section 3.

1) Chromosome Encoding: The weights (and biases) in the neural network are encoded as a list of real numbers (see Figure 1).

2) Evaluation Function: Assign the weights on the chromosome to the links in a network of a given architecture, run the network over the training set of examples, and return the sum of the squares of the errors.

3) Initialization Procedure: The weights of the initial members of the population are chosen at random with a probability distribution given by $e^{-|x|}$. This is different from the initial probability distribution of the weights usually used in backpropagation, which is a uniform distribution between -1.0 and 1.0. Our probability distribution reflects the empirical observation by researchers that optimal solutions tend to contain weights with small absolute values but can have weights with arbitrarily large absolute values. We therefore seed the initial population with genetic material which allows the genetic algorithm to explore the range of all possible solutions but which tends to favor those solutions which are a priori the most likely.

4) Operators: We created a large number of different types of genetic operators. The goal of most of the experiments we performed was to find out how different operators perform in different situations and thus to be able to select a good set of operators for the final algorithm. The operators can be grouped into three basic categories: mutations, crossovers, and gradients (see Figure 2). A mutation operator takes one parent and randomly changes some of the entries in its chromosome to create a child. A crossover operator takes two parents and creates one or two children containing some of the genetic material of each parent. A gradient operator takes one parent and produces a child by adding to its entries a multiple of the gradient with respect to the evaluation function. We now discuss each of the operators individually, one category at a time.

UNBIASED-MUTATE-WEIGHTS: For each entry in the chromosome, this operator will with fixed probability $p = 0.1$ replace it with a random value chosen from the initialization probability distribution.

BIASED-MUTATE-WEIGHTS: For each entry in the chromosome, this operator will with fixed probability $p = 0.1$ add to it a random value chosen from the initialization probability distribution. We expect biased mutation to be better than unbiased mutation for the following reason. Right from the start of a run, parents are chosen which tend to be better than average. Therefore, the weight settings in these parents tend to be better than random settings. Hence, biasing the probability distribution by the present value of the weight should give better results than a probability distribution centered on zero.

MUTATE-NODES: This operator selects n non-input nodes of the network which the parent chromosome represents. For each of the ingoing links to these n nodes, the operator adds to the link's weight a random value from the initialization probability distribution. It then encodes this new network on the child chromosome. The intuition here is that the ingoing links to a node form a logical subgroup of all the links in terms of the operation of the network. By confining its changes to a small number of these subgroups, it will make its improvements more likely to result in a good evaluation. In our experiments, $n = 2$.

MUTATE-WEAKEST-NODES: The concept of node strength is different from the concept of error used in backpropagation. For example, a node can have zero error if all its output links are set to zero, but such a node is not contributing anything positive to the network and is thus not a strong node. The concept of strength comes from classifier systems and was introduced into neural networks in [Davis

1988]. We define the strength of a hidden node in a feed-forward network as the difference between the evaluation of the network intact and the evaluation of the network with that node lobotomized (i.e. with its output links set to zero). We have devised an efficient way to calculate node strength not discussed here.

The operator MUTATE-WEAKEST-NODES takes the network which the parent chromosome represents and calculates the strength of each hidden node. It then selects the m weakest nodes and performs a mutation on each of their ingoing and outgoing links. This mutation is unbiased if the node strength is negative and biased if the node strength is positive. It then encodes this new network on a chromosome as the child. The intuition behind this operator is that there are some nodes which are not only not very useful to the network but may actually be hurting it. Performing mutation on these nodes is more likely to yield bigger gains than mutation on a node which is already doing a good job. Note that since this operator will not be able to improve nodes which are already doing well it should not be the only source of diversity in the population. In our experiments, m=1.

CROSSOVER-WEIGHTS: This operator puts a value into each position of the child's chromosome by randomly selecting one of the two parents and using the value in the same position on that parent's chromosome.

CROSSOVER-NODES: For each node in the network encoded by the child chromosome, this operator selects one of the two parent's networks and finds the corresponding node in this network. It then puts the weight of each ingoing link to the parent's node into the corresponding link of the child's network. The intuition here is that networks succeed because of the synergism between their various weights, and this synergism is greatest among weights from ingoing links to the same node. Therefore, as genetic material gets passed around, these logical subgroups should stay together.

CROSSOVER-FEATURES: Different nodes in a neural network perform different roles. For a fully connected, layered network, the role which a given node can play depends only on which layer it is in and not on its position in that layer. In fact, we can exchange the role of two nodes A and B in the same layer of a network as follows. Loop over all nodes C connected (by either an ingoing or outgoing link) to A (and thus also to B). Exchange the weight on the link between C and A with that on the link between C and B. Ignoring the internal structure, the new network is identical to the old network, i.e. given the same inputs they will produce the same outputs.

The child produced by the previously discussed crossovers is greatly affected by the internal structures of the parents. The CROSSOVER-FEATURES operator reduces this dependence on internal structure by doing the following. For each node in the first parents' network, it tries to find a node in the second parent's network which is playing the same role by showing a number of inputs to both networks and comparing the responses of different nodes. It then rearranges the second parent's network so that nodes playing the same role are in the same position. At this point, it forms a child in the same way as CROSSOVER-NODES. The greatest improvement gained from this operator over the other crossover operators should come at the beginning of a run before all the members of a population start looking alike.

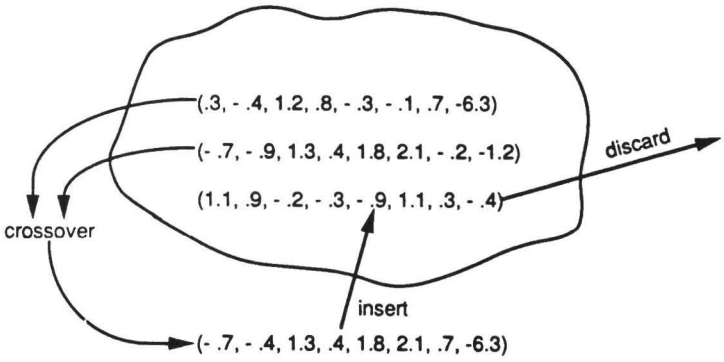

Figure 3: An Iteration of a Genetic Algorithm

HILLCLIMB: This operator calculates the gradient for each member of the training set and sums them together to get a total gradient. It then normalizes this gradient by dividing by the magnitude. The child is obtained from the parent by taking a step in the direction determined by the normalized gradient of size step-size, where step-size is a parameter which adapts throughout the run in the following way. If the evaluation of the child is worse than the parent's, step-size is multipled by the parameter step-size-decay=0.4; if the child is better than the parent, step-size is multiplied by step-size-expand=1.4. This operator differs from back-propagation in the following ways: 1) Weights are adjusted only after calculating the gradient for all members of the training set and 2) The gradient is normalized so that the step size is not proportional to the size of the gradient.

5) Parameter Settings: There are a number of parameters whose values can greatly influence the performance of the algorithm. Except where stated otherwise, we kept these constant across runs. We now discuss some of the important parameters individually.

PARENT-SCALAR: This parameter determines with what probability each individual is chosen as a parent. The second-best individual is PARENT-SCALAR times as likely as the best to be chosen, the third-best is PARENT-SCALAR times as likely as the second-best, etc. The value was usually linearly interpolated between 0.92 and 0.89 over the course of a run.

OPERATOR-PROBABILITIES: This list of parameters determines with what probability each operator in the operator pool is selected. Usually, these values were initialized so that the operators all had equal probabilities of selection. An adaptation mechanism changes these probabilities over the course of a run to reflect the performance of the operators, increasing the probability of selection for operators that are doing well and decreasing it for operators performing poorly. This saves the user from having to hand-tune these probabilities.

POPULATION-SIZE: This self-explanatory parameter was usually set to 50.

An example of an iteration of the genetic algorithm is shown in Figure 3.

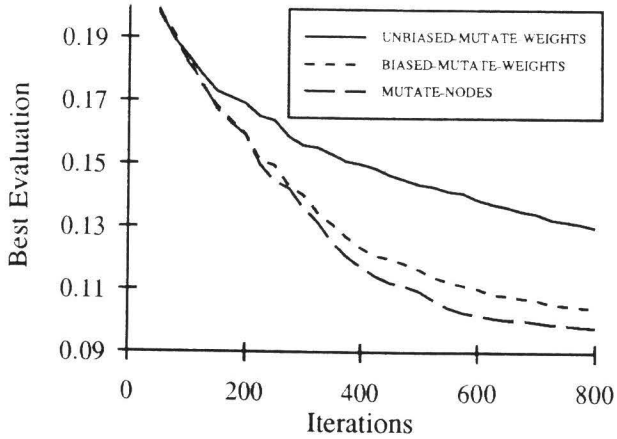

Figure 4: Results of Experiment 1

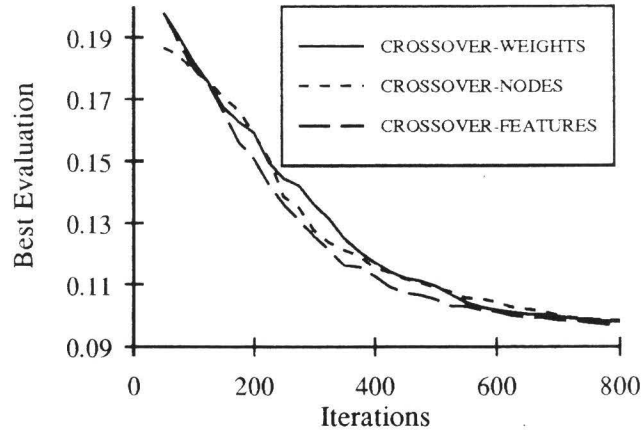

Figure 5: Results of Experiment 2

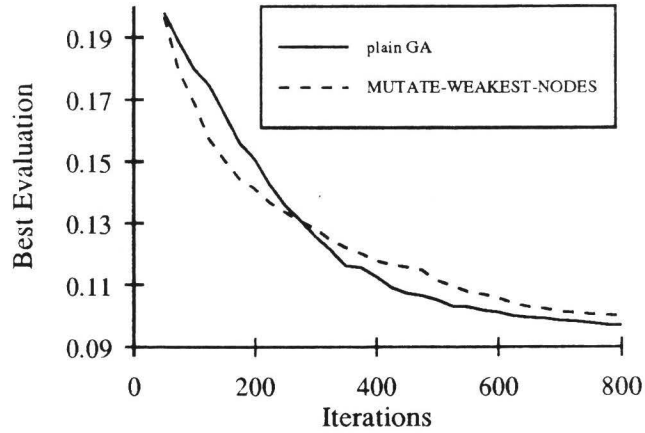

Figure 6: Results of Experiment 3

6 Experiments

In this section we discuss a series of experiments which led us to our final version of the genetic algorithm and one more experiment comparing the genetic algorithm to backpropagation. To evaluate the algorithm with a given set of operators and parameter settings, we performed a series of ten independent runs recording the evaluation of the best individual as a function of the number of iterations. Afterwards, we averaged the results so as to reduce the variations introduced by the stochastic nature of the algorithm. Even after averaging some variations remain, and so it is a judgment call to separate random variations from significant differences between averaged runs.

Experiment 1: This experiment was designed to compare the performances of three different versions of mutation: UNBIASED-MUTATE-WEIGHTS, BIASED-MUTATE-WEIGHTS and MUTATE-NODES. To do this we performed three series of runs, each with the same parameter settings and with CROSSOVER-WEIGHTS as one of the operators. The only difference between runs was the version of mutation used. The results are pictured in Figure 4. As predicted, there is a clear ordering in terms of performance: 1) MUTATE-NODES, 2) BIASED-MUTATE-WEIGHTS and 3) UNBIASED-MUTATE-WEIGHTS.

Experiment 2: This experiment was designed to compare the performances of three different versions of crossover: CROSSOVER-WEIGHTS, CROSSOVER-NODES and CROSSOVER-FEATURES. To do this we performed three series of runs, each with the same parameter settings and with MUTATE-NODES as one of the operators. The only difference between runs was the version of crossover used. The results are pictured in Figure 5. They indicate that there is little performance difference between the different types of crossover.

Experiment 3: This experiment attempted to determine the effectiveness of the special-purpose operator MUTATE-WEAKEST-NODES. To do this, it compared an averaged run with just the two operators MUTATE-NODES and CROSSOVER-FEATURES to an averaged run with these two operators plus MUTATE-WEAKEST-NODES. The results are pictured in Figure 6. Having MUTATE-WEAKEST-NODES improved the performance at the beginning of the runs but diminished the performance in the middle and at the end. This could indicate that we need to change our definition of node strength.

Experiment 4: This experiment investigated the effectiveness of a hill-climbing mode for use at the end of a genetic algorithm run. The hill-climbing mode has HILL-CLIMB as its only operator and has the parameter PARENT-SCALAR=0.0 (so that the best individual is always chosen as the parent). The experiment compared an averaged run of 800 iterations with MUTATE-NODES and CROSSOVER-NODES to an averaged run of 500 iterations with these two operators and 300 iterations in hill-climbing mode. The results are shown in Figure 7. Note the rapid progress immediately after entering hill-climbing followed quickly by a period of little or no progress. This reflects the speed of hill-climbing mode at climbing local hills and its inability to go anywhere once it hits the peak. It is clear that hill-climbing mode carries with it a big risk of local minima and therefore should only be used when it is relatively certain that a global minimum is near.

Experiment 5: This experiment compared the performance of standard backpropagation with our genetic algorithm for training our feedforward network. We used the backpropagation algorithm described in [Rumelhart 1986a] with a learning rate (i.e. gain) of 0.5. The genetic algorithm

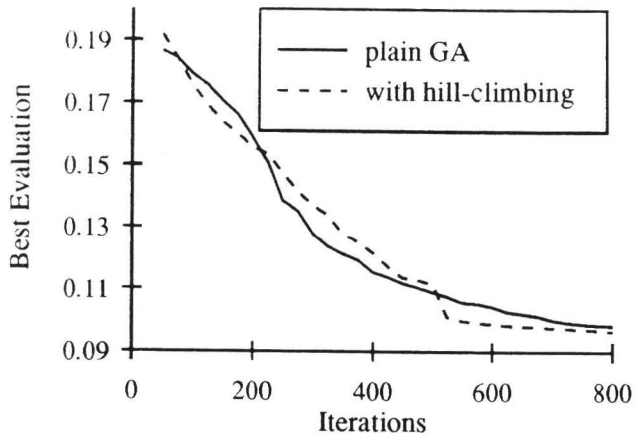

Figure 7: Results of Experiment 4

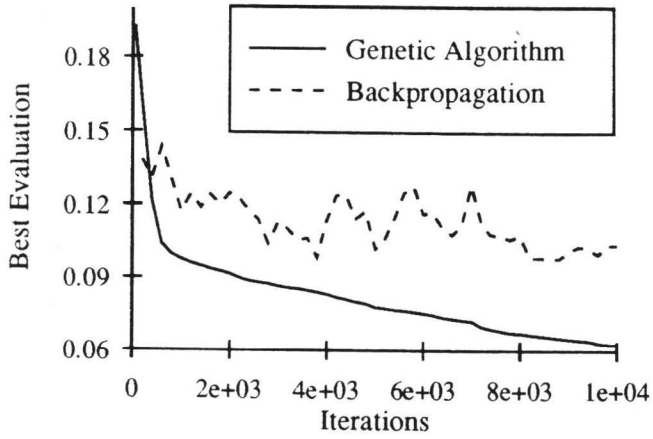

Figure 8: Results of Experiment 5

had two operators, MUTATE-NODES and CROSSOVER-NODES, and was thus a variation on a standard genetic algorithm.

When comparing them, we considered two iterations of the genetic algorithm to be equivalent to one iteration (i.e. one cycle through all the training data) of backpropagation. To see why, observe that backpropagation consists of looping through all training data doing 1) forward propagation and calculation of errors at the outputs and 2) error backward propagation and adjusting of weights. The second step requires more computation in our network and almost all other networks of interest. The evaluation function of the genetic algorithm performs the same calculations as step 1). The operators MUTATE-NODES and CROSSOVER-NODES do very little computation. Hence, one iteration of backpropagation requires more than twice as much computation as one iteration of the genetic algorithm.

The runs consisted of 10000 iterations of the genetic algorithm and 5000 iterations of backpropagation. The results are shown in Figure 8. Clearly, the genetic algorithm outperformed backpropagation.

7 Conclusions and Future Work

We have accomplished a number of things with our work on using genetic algorithms to train feedforward networks. In the field of genetic algorithms, we have demonstrated a real-world application of a genetic algorithm to a large and complex problem. We have also shown how adding domain-specific knowledge into the genetic algorithm can enhance its performance. In the field of neural networks, we have introduced a new type of training algorithm which on our data outperforms the backpropagation algorithm. Our algorithm has the added advantage of being able to work on nodes with discontinuous transfer functions and discontinuous error criteria.

The work described here only touches the surface of the potential for using genetic algorithms to train neural networks. In the realm of feedforward networks, there are a host of other operators with which one might experiment. Perhaps most promising are ones which include backpropagation as all or part of their operation. Another problem is how to modify the genetic algorithm so that it deals with a stream of continually changing training data instead of fixed training data. This requires modifying the genetic algorithm to handle a stochastic evaluation function. Finally, as a general-purpose optimization tool, genetic algorithms should be applicable to any type of neural network (and not just feedforward networks whose nodes have smooth transfer functions) for which an evaluation function can be derived. The existence of genetic algorithms for training could aid in the development of other types of neural networks.

References

[Davis 1988] L. Davis, "Mapping Classifier Systems into Neural Networks," to appear in *Proceedings of the 1988 Conference on Neural Information Processing Systems*, Morgan Kaufmann.

[Goldberg 1988] D. Goldberg, *Genetic Algorithms in Machine Learning, Optimization, and Search*, Addison-Wesley (1988).

[Holland 1975] J. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press (1975).

[Rumelhart 1986a] D.E. Rumelhart, G.E. Hinton and R.J. Williams, "Learning Representations by Back-Propagating Errors," *Nature* **323**, pp. 533-536 (1986).

[Rumelhart 1986b] D.E. Rumelhart and J.L. McClelland (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, MIT Press (1986).

[Whitley 1988] D. Whitley, "Applying Genetic Algorithms to Neural Network Problems," *International Neural Network Society*, p. 230 (1988).