Neural Network Weight Selection Using Genetic Algorithms

David J. Montana

Bolt Beranek and Newman Inc.
70 Fawcett Street, Cambridge, MA 02138

1 Introduction

Neural networks are a computational paradigm modeled on the human brain that has become popular in recent years for a few reasons. First, despite their simple structure, they provide very general computational capabilities [HORN89]. Second, they can be manufactured directly in VLSI hardware and hence provide the potential for relatively inexpensive massive parallelism [MEAD89]. Most importantly, they can adapt themselves to different tasks, i.e. learn, solely by selection of numerical "weights". How to select these weights is a key issue in the use of neural networks. The usual approach is to derive a special-purpose weight selection algorithm for each neural network architecture. Here, we dicuss a different approach.

Genetic algorithms are a class of search algorithms modeled on the process of natural evolution. They have been shown in practice to be very effective at function optimization, efficiently searching large and complex (multimodal, discontinuous, etc.) spaces to find nearly global optima. The search space associated with a neural network weight selection problem is just such a space. In this chapter, we investigate the utilization of genetic algorithms for neural network weight selection.

2 Overview of the Technologies Involved

2.1 Neural Networks

Neural networks generally consist of five components

1. A directed graph known as the network topology whose nodes represent the neurodes (or processing elements) and whose arcs represent the connections
2. A state variable associated with each neurode
3. A real-valued weight associated with each connection
4. A real-valued bias associated with each neurode
5. A transfer function f for each neurode such that the state of the neurode is $f(\Sigma \omega_i x_i - \beta)$, where β is the bias of the neurode, ω_i are the weights on the incoming connections, x_i are the states of the neurodes on the other end of these connections.

To execute a neural network, one must initialize it by setting the states of a subset of its neurodes. In particular, all the input neurodes, i.e. those neurodes which have no incoming connections, must have their states set at initialization. Then, at successive time ticks (or continuously in the case of analog

$$(\,19.3,\ 0.05,\ -1.2,\ 345,\ 2.0,\ 7.7,\ 68.0\,) \xrightarrow{\text{mutation}} (\,19.3,\ 0.05,\ -1.2,\ 345,\ 2.0,\ 8.2,\ 68.0\,)$$

$$\begin{array}{l} (\,19.3,\ 0.05,\ -1.2,\ 345,\ 2.0,\ 7.7,\ 68.0\,) \\ \\ (\,17.6,\ 0.05,\ -3.5,\ 250,\ 3.0,\ 7.7,\ 70.0\,) \end{array} \xrightarrow{\text{crossover}} (\,17.6,\ 0.05,\ -1.2,\ 345,\ 3.0,\ 7.7,\ 68.0\,)$$

Figure 1: The mutation and crossover operators.

networks), each neurode updates its state as computed by its transfer function based on the states of the other neurodes feeding into this one. After enough time ticks that the network has reached steady-state (i.e., the neurode states have stopped changing significantly), the states of a chosen subset of the neurodes are the network's output. In particular, the state of any output neurode, i.e. a neurode with no outgoing connections, is part of the network output (or else superfluous).

A feedforward neural network is one whose topology has no closed paths. To execute a feedforward network, the states of the input neurodes are set. After n time ticks, where n is the maximum path length from an input neurode to an output neurode, all the neurodes have assumed their steady-state values. Note that the computation from each of these n times ticks can be performed in parallel, and hence feedforward neural networks execute very quickly when fully parallelize.

As mentioned above, many networks are able to adapt to the data, i.e. learn. In most cases, this requires a separate stage called "training". During training, the network topology and/or the weights and/or the biases and/or the transfer functions are selected based on some training data. In many approaches, the topology and transfer functions are held fixed, and the space of possible networks is spanned by all possible values of the weights and biases. It is this problem of weight selection which we discuss in this chapter.

A good introduction to neural networks is [RUME86].

2.2 Genetic Algorithms

Genetic algorithms are algorithms for optimization and machine learning based loosely on several features of biological evolution [HOLL75]. They require five components [DAVI87]:

1. A way of encoding solutions to the problem on chromosomes.

2. An evaluation function which 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. Standard operators are mutation and crossover (see Figure 1).

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. Initialize the population using the initialization procedure, and evaluate each member of the initial population.

2. Reproduce until a stopping criterion is met. Reproduction consists of iterations of the following steps:

(a) Choose one or more parents to reproduce. Selection is stochastic, but the individuals with the hightest evaluations are favored in the selection.

(b) Choose a genetic operator and apply it to the parents.

(c) Evaluate the children and accumulate them into a generation. After accumulating enough individuals, insert them into the population, replacing the worst current members of the population.

When the components of the genetic algorithm are chosen appropriately, the reproduction process will continually improve the population, converging finally on solutions close to a global optimum. Genetic algorithms can efficiently search large and complex (i.e., possessing many local optima) spaces to find nearly global optima.

A key concept for genetic algorithms is that of schemata, or building blocks. A schema is a subset of the fields of a chromosome set to particular values with the other fields left to vary. As originally observed in [HOLL75], the power of genetic algorithms lies in their ability to implicitly evaluate large numbers of schemata simultaneously and to combine smaller schemata into larger schemata.

A good introduction to genetic algorithms is [GOLD88].

3 Introduction to the Application Domain

The fixed-features supervised pattern classification problem is the following. Consider a k-dimensional feature space and M separate classes. Different instances of a particular class generally have different values for the features and hence correspond to different points in feature space. Our only knowledge of how the instances of a particular class are distributed in feature space comes from a finite number of exemplars which are known to be of this class and whose feature vectors are known. The set of exemplars from all classes is called the training set. Then, the problem is: given a training set, select the most likely class for any feature vector not in the training set.

Training consists of selecting a network which minimizes some error criterion (often taken to be the sum of the squares of the errors on the training data) which predicts how well the network will perform on test data (i.e., data not in the training set). Hence, training is an optimization problem, requiring search through the space of possible networks for an optimal network. When the topology and transfer functions are fixed, this is optimization over $\Re^n$, where n is the number of weights and biases left to vary.

We now discuss two different neural network architectures and their approach to pattern classification.

3.1 Sigmoidal Feedforward Networks

The most common neural network is a fixed-topology feedforward sigmoidal network. A sigmoidal network is one such that the transfer functions of all its neurodes are sigmoids of the form $f : x \mapsto \frac{1}{1+e^{-x}}$. Such a network is picture in Figure 2.

For a pattern classification problem with k features and M classes, a feedforward network must have k input neurodes, M output neurodes, and an arbitrary number and configuration of hidden neurodes, where hidden neurodes are those which have both incoming and outgoing connections. When the states of the input neurodes are set to particular feature values and the states are propagated through the network to the output neurodes, the values of the output neurodes are the likelihoods for each class. The evaluation function optimized during training is often the sum over the training exemplars of the output errors squared. These networks are usually trained using the backpropagation algorithm, which is a variant on gradient search and is described in [RUME86b]. Below, we examine an alternate training algorithm.

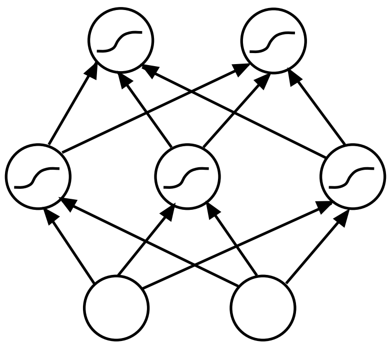

Figure 2: A feedforward sigmoidal network

3.2 Weighted Probabilistic Neural Networks (WPNN)

WPNN [MONT92] is a pattern classification algorithm which is an extension of Probabilistic Neural Networks (PNN) [SPEC90] and which works as follows. Let the exemplars from class i be the k-vectors $\vec{x}^i_j$ for $j = 1, ..., N_i$. Then, the likelihood function for class i is defined to be

$$L_i(\vec{x}) = \frac{1}{N_i(2\pi)^{k/2}(\det \Sigma)^{1/2}} \sum_{j=1}^{N_i} e^{-(\vec{x}-\vec{x}^i_j)^T \Sigma^{-1} (\vec{x}-\vec{x}^i_j)} \tag{1}$$

where the covariance matrix Σ is a positive-definite kxk symmetric matrix whose entries are the free parameters. The conditional probability for class i is

$$P_i(\vec{x}) = L_i(\vec{x}) / \sum_{j=1}^{M} L_j(\vec{x}) \tag{2}$$

Note that the class likelihood functions are sums of identical (generally) anisotropic Gaussians centered at the exemplars.

Training WPNN consists of selecting the entries of Σ. This is equivalent to selecting a particular distance metric in feature space. Hence, our work is similar to that of [ATKE91] and [KELL91], which are other nearest-neighbor-like algorithms which use modified metrics. Like [ATKE91] and [KELL91], we have so far restricted ourselves to diagonal covariances, and we can hence think of the inverse of each of the diagonal entries of Σ as a weight on the corresponding feature. We therefore refer to these entries as "feature weights". Selection of feature weights is done by a genetic alorithm as described in Section 6.

The neural network implementation of the WPNN algorithm is as follows. There are k neurodes in the input layer. There are M neurodes in the output layer, each with a normal (i.e., mean 0 and variance 1) Gaussian transfer function and a zero bias. There are kN neurodes in the single hidden layer, where N is the total number of exemplars, each with a linear transfer function. The $(ik + j)^{th}$ hidden neurode (i) is connected to the j^{th} input neurode with weight 1.0, (ii) has bias equal to the j^{th} feature of the i^{th} example, and (iii) is connected to the l^{th} output neurode, where l is the class of the i^{th} exemplar, with weight f_j, where f_j is the selected feature weight for the j^{th} feature. Such a network is pictured in Figure 3.

4 The Rationale for Hybrid Processing

There are a few reasons why it can be beneficial to use genetic algorithms for training neural networks. With regard solely to the problem of weight (and bias) selection for networks with fixed topologies and transfer

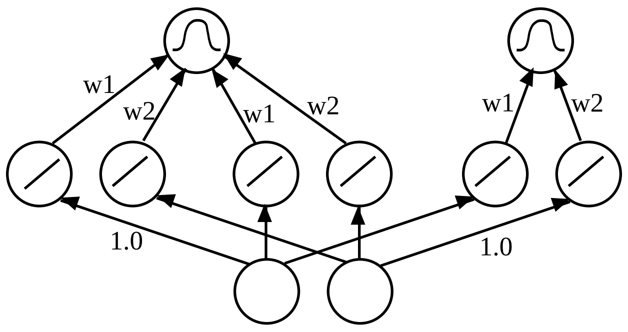

Figure 3: A WPNN network corresponding to a pattern classification problem with two features, two classes, and three exemplars, two of class 1 and one of class 2.

functions (which is all that we address in this chapter), there is the issue of local optima. As the number of exemplars and the complexity of the network topology increase, the complexity of the search space also increases insofar as the error function obtains more and more local minima spread out over a larger portion of the space. Gradient-based techniques often have problems getting past the local optima to find global (or at least nearly global) optima. However, genetic algorithms are particularly good at efficiently searching large and complex spaces to find nearly global optima. As the complexity of the search space increases, genetic algorithms present an increasingly attractive alternative to gradient-based techniques such as backpropagation. Even better, genetic algorithms are an excellent complement to gradient-based techniques such as backpropagation for complex searches. A genetic algorithm is run first to get to the "right hill" (i.e., a part of the space in a neighborhood of a nearly global optimum) and the gradient-based technique climbs the hill to its peak [MONT89, BELE90].

A second advantage of genetic algorithms is their generality. With only minor changes to the algorithm, genetic algorithms can be used to train all different varieties of networks. They can select wieghts for recurrent networks, i.e. networks whose topologies have closed paths. They can train networks with different transfer functions than sigmoids, such as the Gaussians used by WPNN or even step transfer functions [KOZA91, COLL91, BORN91], which are discontinuous and hence not trainable by gradient techniques. Furthermore, genetic algorithms can optimize not just weights and biases but any combination of weights, biases, topology, and transfer functions. Possibly the most important type of generalization allowed by genetic algorithms is the use of an arbitrary evaluation function. This allows potentially the use of discontinuous evaluation functions based on the principle of Minimum Description Length (MDL) [RISS89] to make the trained network generalize better. This ability to use arbitrary evaluation function becomes increasingly important as we attempt to expand neural networks beyond pattern classification and function approximation and towards domains such as control logic for autonomous vehicles or early visual processing, where performance is not measured as a simple sum of squares of the error.

A third reason to study genetic algorithms for learning neural networks is that this is an important method used in nature. While it is not clear that evolutionary learning methods are used in individual organisms (although some such as [EDEL87] would claim so), it is relatively certain that evolutionary learning at a species level is a major method of "wet" neural network training. For example, in parts of the brain such as the early visual processing component, the neural network weights (in addition to the topology and transfer functions) are essentially "hard-wired" genetically and hence were "learned" via the process of natural selection.

5 Survey of Related Hybrid Systems

Genetic algorithms have been used together with neural networks in a variety of ways, each with their own body of literature. One such way is to use genetic algorithms for the preprocessing of neural network data, usually using the genetic algorithm to perform feature selection [CHAN91, DECK93]. A second use of genetic algorithms with neural networks is to use the genetic algorithm to select the neural network topology [HARP89, SCHA90]. A third such application is to employ genetic algorithms for selecting the weights of a genetic algorithm [MONT89, MONT92]. Some approaches optimize both the weights and the topology simultaneously [KOZA91, GRUA92, COLL91, BORN91]. A fourth application is to use genetic algorithms to learn the neural network learning algorithm [HARP89, BELE90, SCHA90]. We do not attempt to give a survey of the whole field. Entire papers have been devoted to this endeavor, and interested readers are referred to [SCHA92] as a good overview. In this chapter, we concentrate solely on the problem of weight selection.

There are two basic differences between the different approaches to using genetic algorithms for weight selection. The first difference is that the weight selection is performed on different architectures. Examples of the differenct network architectures to which genetic weight selection algorithms have been applied are: feedforward sigmoidal [MONT89, WHIT90, BELE90], WPNN [MONT92], cascade-correlation [KARU92, POTT92], recurrent sigmoidal [WIEL91] recurrent linear threshold [TORR91], feedforward with step functions [KOZA91], and feedback with step function [COLL91, BORN91]. The second difference, and the one on which this chapter concentrates, is the difference in the genetic algorithms. However, we do not discuss these differences in this section; instead, in Section 6.1 we will compare and contrast each of the five components of our genetic algorithms with other genetic algorithms after we discuss each component.

6 Case Studies

In this section, we describe how we have applied genetic algorithms to the problem of weight selection for the two different neural network architectures discussed in Section 3. Section 6.1 details the five components of the genetic algorithm used for each architecture and compares and contrasts them with other approaches. Section 6.2 gives some experimental results.

6.1 The Genetic Algorithms

6.1.1 Representation

WPNN: The genetic algorithm we used with WPNN employs a "logarithmic" representation. A chromosome is a fixed-length string of integers with values between 1 and K, with each location in the string corresponding to a particular feature weight. The map from an integer n in the chromosome to its corresponding feature weight values is $n \mapsto B^{n-k_0}$. Here, K, k_0 and B are parameters we choose depending on the desired range and resolution for the feature weight values. We use a logarithmic representation because it provides large dynamic range with proportional resolution (and hence proportional representation in a randomly selected population) at all scales.

Feedforward Sigmoidal: The genetic algorithm we used with feedforward sigmoidal neural networks does not use any encoding scheme for the networks but rather uses the networks themselves as the chromosomes. This allows us to define genetic operators such as MUTATE-NODES and CROSSOVER-NODES (see below), which utilize the structure of the network when generating children. Note that when we use the weight-based genetic operators such as BIASED-MUTATE-WEIGHTS and CROSSOVER-WEIGHTS (see below) rather than node-based operators, the representation we use is equivalent to using a real-valued fixed-length string representation.

Other Approaches: Some other genetic algorithms for weight selection use fixed-length binary string representations [BELE90, WIEL91, WHIT89]. Their philosophy, which is common in the field, is that using a binary representation maximizes the number of schemata and hence the amount of implicit parallelism in the search. While this philosophy is valid when we have no a priori knowledge of what the good schemata are, for the current problem we know that the bits which represent a single weight are inextricably linked and hence are much more likely than random bits to form a good schema. Forcing the genetic algorithm to learn what we already know makes the search much less efficient. [WHIT90] provides experimental evidence of this theoretical argument.

We have taken this concept of treating as an indivisible unit elements which we know a priori to form such a unit one step further by using the operators Mutate-Nodes and Crossover-Nodes (see below). These operators treat all the incoming weights to a node as an indivisible unit, which makes sense because it is only the values of these weights relative to each other which matter. The experimental results discussed in Section 6.2 validate this reasoning.

When the weights are optimized simultaneously with the topology, then the number of weights is not fixed and the representations are generally much less standard. For example, [KOZA91] represents a neural network as a tree of variable size and shape. Similarly, [GRUA92] represents a network using a constrained syntactic structure.

6.1.2 Evaluation Function

WPNN: To evaluate the performance of a particular set of features weights on the training set, we use a technique called "leaving-one-out", which is a special form of cross-validation [STON74]. One exemplar at a time is withheld from the training set and classified using WPNN with those feature weights and the reduced training set. The evaluation is the sum of the errors squared on the individual exemplars. Formally, for the exemplar $\vec{x}^i_j$, let $\tilde{L}_q(\vec{x}^i_j)$ for $q = 1, ..., M$ denote the class likelihoods obtained upon withholding this exemplar and applying Equation 1, and let $\tilde{P}_q(\vec{x}^i_j)$ be the probabilities obtained from these likelihoods via Equation 2. Then, we define the performance as

$$E = \sum_{i=1}^{M}\sum_{j=1}^{N_i}((1 - \tilde{P}_i(\vec{x}^i_j))^2 + \sum_{q \neq i}(\tilde{P}_q(\vec{x}^i_j))^2) \tag{3}$$

We have incorporated two heuristics to quickly identify covariances which are clearly bad and give them a value of ∞, the worst possible score. This greatly speeds up the optimization process because many of the generated feature weights can be eliminated this way. The first heuristic identifies covariances which are too "small" based on the condition that, for some exemplar $\vec{x}^i_j$ and all $q = 1, ...M$, $\tilde{L}_q(\vec{x}^i_j) = 0$ to within the precision of IEEE double-precision floating-point format. In this case, the probabilities $\tilde{P}_q(\vec{x}^i_j)$ are not well-defined. The second heuristic identifies covariances which are too "big" in the sense that too many exemplars contribute significantly to the likelihood functions. Empirical observations and theoretical arguments show that PNN (and WPNN) work best when only a small fraction of the exemplars contribute significantly. Hence, we reject a particular Σ if, for any exemplar $\vec{x}^i_j$,

$$\sum_{\vec{x} \neq \vec{x}^i_j} e^{-(\vec{x}-\vec{x}^i_j)^T \Sigma^{-1} (\vec{x}-\vec{x}^i_j)} > (\sum_{i=1}^{M} N_i)/P \tag{4}$$

Here, P is a parameter which we chose for our experiments to equal four.

Feedforward Sigmoidal: We used the same evaluation function as is standard for use with backpropagation, the sum of the squares of the errors on the training set. We used the same evaluation function to allow us to experimentally compare the genetic algorithm to backpropagation.

Other Approaches: While the evaluation function for the problem of pattern classification is similar across different systems, the evaluation function can in general be as varied as the problems to which the neural networks are applied. One such example is the application of neural networks to control using reinforcement learning [WHIT91]. Here, the evaluation function is a score which summarized the feedback from the environment.

6.1.3 Initialization Procedure

WPNN: The entries for each individual are integers chosen randomly with uniform distribution over the range 1, ..., K, where K is defined above.

Feedforward Sigmoidal: The weights and biases of each network in the initial population are real numbers randomly chosen from the two-sided exponential distribution $e^{-|x|}$. This distribution is an attempt to reflect the actual distribution of weights in typical neural networks; the majority of weights are between -1 and 1, but the weights can be of arbitrary magnitude.

Other Approaches: Most approaches use some variety of random seeding of the initial population.

6.1.4 Genetic Operators

WPNN: For training WPNN, we used the two standard genetic operators: mutation and crossover. The mutation operator is standard, randomly selecting a fraction of the alleles (i.e., locations in the string) and replacing their values with values randomly selected from the (finite) set of possible values. The crossover operator is uniform crossover, i.e. it selects from which parent to take the value for a particular allele independent of its choices for other alleles. This is in contrast to the point-based crossover operators (most commonly, 1-point and 2-point crossover), which except at a small number of cut points will select the values of neighboring alleles from the same parent. Point-based crossover can provide an advantage when the low-order schemata tend to be formed from neighboring alleles, as is the case when real numbers are encoded using a binary representation. However, since we are using a real-valued representation and since there is no particular ordering to the feature weights, uniform crossover is the better choice.

Feedforward Sigmoidal: For training feedforward sigmoidal networks, we experimented with eight different operators: three varieties of mutation (Unbiased-Mutate-Weights, Biased-Mutate-Weights, and Mutate-Nodes), three varieties of crossover (Crossover-Weights, Crossover-Nodes, and Crossover-Features), and two special-purpose operators (Mutate-Weakest-Nodes and Backprop). While some proved more effective than others in our limited experiments (see Section 6.2), they all illustrate important concepts and all hold potential for the future. We now discuss each of them.

Unbiased-Mutate-Weights: This operator replaces the weights of randomly selected connections in the parent network with values chosen randomly from the same probability distribution used for initialization.

Biased-Mutate-Weights: This operator does the same as Unbiased-Mutate-Weights except that, instead of replacing the old values with the randomly selected values, it adds the randomly selected values to the old values. So, the probability distribution for the new weight values is a two-sided exponential centered at the original value. The reasoning is that any network selected as a parent (especially as the run progresses) is significantly better than a randomly selected network, and so its individual weights are usually better than randomly selected ones. Hence, a mutation which uses the current weights and starting points will outperform a mutation which ignores the current weights. (This concept is similar to Davis' Creep operator [COOM87].) The experiments (see below) bears this out.

Mutate-Nodes: This operation randomly selects nodes from the set of all hidden and output nodes and performs a biased mutation on all the incoming weights for the selected nodes. As mentioned above, we know a priori that the set of incoming weights for a particular node forms a low-order schemata, or building block. A mutation which disrupts as few of these schemata as possible by concentrating its changes in a small number of them should outperform a muation which disrupts many of these schemata. The experiments (see below) confirms this reasoning. [Aside: While the genetic algorithm literature discusses extensively the problem of schema disruption due to crossover, there is little discussion of schema disruption due to mutation. It is perhaps the widespread use of binary representations together with maximally disruptive mutation operators which have cause the genetic algorithm community to devalue mutation.]

Crossover-Weights: This operator performs uniform crossover (described above) on the weights and biases of the two parent networks. Note that if we were to encode the weights in a string so that the incoming weights for any node were all adjacent, point-based crossover would probably be better than uniform crossover because it would disrupt less schemata. However, since we know where these schemata are, an even better approach is the following.

Crossover-Nodes: This operator performs uniform crossover with the basic units exchanged being the set of all incoming weights of a particular node. This preserves what we know a priori to be some of the low-order schemata. As discussed in Section 6.2, our experiments were inconclusive about whether Crossover-Nodes was better than Crossover-Weights because the dominant operator was mutation.

Crossover-Features: This operator attempts to compensate for the problem of "competing conventions," a term first used in [WHIT90] for a problem discussed earlier in [MONT89]. This problem arises because hidden nodes which are connected to the same set of other nodes are interchangeable, i.e. if two such neurodes exchange all their weights then the output of the network will still always be the same. However, interchangeable neurodes generally play different roles in a network. If two parent networks have assigned neurodes from a set of interchangeable neurodes to different roles, then the two types of crossover discussed above will not be effective. The Crossover-Features operator first attempts to rearrange the hidden nodes of one of the parent networks so that all the hidden neurodes are playing the same role before performing crossover at the node level.

Mutate-Weakest-Nodes: This operator is a version of node-based mutation which differs from Mutate-Nodes in the following ways. First, instead of randomly selecting which nodes to mutate, it mutates those nodes which contribute the least to the network. The contribution of a node is defined as the total error of the network with the node deleted minus the total error of the network with the node included. (We have devised an efficient method for calculating these contributions.) Second, instead of mutating just the incoming weights, it also mutates the outgoing weights. Third, if the contribution of the nodes is negative, it performs an unbiased mutation rather than a biased mutation. The intuition behind this operator is that performing mutation on nodes that are contributing the least to the network (and especially on those that are contributing negatively) will yield bigger gains on the average. As shown in Section 6.2, Mutate-Weakest-Nodes does in fact yield large payoffs when at the beginning of a genetic algorithm run, before the population has started converging to good solutions and hence while there are still many nodes making negative contributions. However, it becomes ineffective after the population has converged and all the neurodes are making positive contributions because the size of a neurode's positive contribution is a factor of both the role it plays as well as how well it plays it. Note that since this operator will never change neurodes which are already performing well, it should not be the only source of diversity in the population.

Hillclimb: This operator creates a child by taking a single step in the direction of the gradient of the parent of a variable step size. (See [MONT89] for more details.) This operator is detrimental at the beginning of a run because it yields only incremental improvements to the parents at the same time that it acts to greatly reduce diversity. However, it is beneficial at the end of a run in order to help climb the final hill (see Section 6.2).

Other Approaches: Some genetic algorithms leave out the crossover operator and just use mutation [PORT90]. The work of [WHIT91] suggests that this can be a good idea.

Some genetic algorithms use an additional operator, called the reproduction operator, which just copies a selected parent without change into the next generation. Because we use a steady-state genetic algorithm (see the next section), such an operator is superfluous.

Those genetic algorithms which have representations based on syntactically constrained structures [KOZA91, GRUA92] need special mutation and crossover operators which respect these syntactic constraints and which are similar in some respects to out node-based operators.

6.1.5 Parameter Values

WPNN: Some of the key parameter values are as follows:

Population-Size: We used a population size of 1600. We needed such a large population size because roughly 90of the initial population evaluated to ∞ (i.e., did not pass the heuristic screener). Hence, only around 160 members of the initial population had any information at all, and this was about the minimum needed to provide sufficient diversity for reproduction.

Generation-Size: This parameter is the number of new individuals generated and old individuals discarded from the population during one iteration of the reproduction process. There are two big issues in selecting the value of GENERATION-SIZE. The first issue is whether to choose it to be small relative to POPULATION-SIZE or to choose it equal to or slightly less than POPULATION-SIZE. The former is known as the "steady-state" approach, while the latter is "generational replacement" [SYSW89]. Assuming that the evaluation function does not change with time (which is the case with our problem), the only argument in favor of the generational replacement approach is that it converges slower than the steady-state approach and hence is more likely to find a nearly global optimum. However, controling convergence rate by proper selection of PARENT-SCALAR and POPULATION-SIZE is seemingly a much more efficient way to protect against premature convergence than throwing away good individuals. The second issue is whether to choose GENERATION-SIZE to be one or a small number greater than one. As discussed in [MONT91], this depends on whether the evaluations are being performed by a single CPU or by multiple CPU's. When there is a single CPU performing evaluations, the only effect of a generation size greater than one is that it imposes a delay between when good individuals are found to be good and when they can be used as parents. Hence, the optimal GENERATION-SIZE when using a single CPU is one. However, when using multiple CPU's, it is usually necessary to have GENERATION-SIZE greater than one in order to utilize the potential parallelism. Because we did our experiments with a single CPU, all the experiments except those evaluating full generational replacement used a GENERATION-SIZE of one.

Parent-Scalar: This is the parameter which determines the parent selection probabilities, i.e. the probability for each member of the population that it will be selected as a parent the next time one is needed for reproduction. We use (as provided by OOGA) an exponential distribution of parent selection probabilities in which these proabilities depend only on relative ranking withing the population and the probability of selecting the $(n+1)^{st}$ best individual divided by the probability of selecting the n^{th} best is the same for all n. This ratio is the parameter Parent-Scalar. The value of Parent-Scalar (in addition to Population-Size) controls the convergence rate of the genetic algorithm; making Parent-Scalar smaller (i.e., closer to zero) makes it converge faster, while making Parent-Scalar large (i.e., closer to one) makes it converge slower. We were able to find good solutions relatively quickly with Parent-Scalar set to 0.9.

Feedforward Sigmoidal:

Population-Size: We used a population size of 50. Using such a small population meant that the space was very coarsely sampled by the initial population and forced the run to converge quickly. After convergence, the genetic algorithm essentially became a hill climber using mutations to drive its ascent. It is not clear whether this is the desired behavior for the genetic algorithm, but we probably should have at least experimented with larger populations and slower convergences.

Generation-Size: We used a generation size of one for the reasons given above.

Parent-Scalar: We used a value for Parent-Scalar which varied linearly from 0.93 to 0.89 as the run progressed. In this manner, we were able to do more exploration at the beginning of a run and more exploitation at the end of a run.

Other Approaches: Population sizes vary widely, but our two genetic algorithms with population sizes of 1600 and 50 are close to the extremes. [WHIT91] describes experiments with a population size of 1, although this is correctly referred to as stochastic hillclimbing rather than a true genetic algorithm.

Many genetic algorithms use full generational replacement rather than steady-state replacement.

There are a variety of approaches to parent selection, including tournament selection and fitness-proportional selection, and there is not yet any clearly superior approach.

6.2 Experimental Results

6.2.1 WPNN

Because WPNN was a new and untested algorithm, our experiments with WPNN centered on the overall performance rather than on the training algorithm. To test the performance of WPNN, we constructed a sequence of four datasets designed to illustrate both the advantages and shortcomings of WPNN. Dataset 1 is a training set we generated during an effort to classify simulated sonar signals. It has ten features, five classes, and 516 total exemplars. Dataset 2 is the same as Dataset 1 except that we supplemented the ten features of Dataset 1 with five additional features, which were random numbers uniformly distributed between zero and one (and hence contained no information relevant to classification), thus giving a total of 15 features. Dataset 3 is the same as Dataset 2 except with ten (rather than five) irrelevant features added and hence a total of 20 features. Like Dataset 3, Dataset 4 has 20 features. It is obtained from Dataset

Algorithm \ Dataset	1	2	3	4
Backprop	11 (69)	16 (51)	20 (27)	13 (64)
PNN	9	94	109	29
WPNN	10	11	11	25
CART	14	17	18	53

Figure 4: Performance on the four datasets of backprop, CART, PNN and WPNN (parenthesized quantities are training set errors)

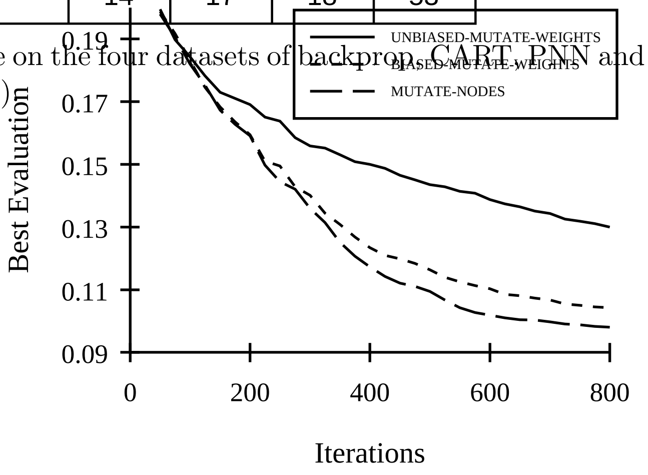

Figure 5: Comparison of mutations.

3 as follows. Pair each of the true features with one of the irrelevant features. Call the feature values of the i^{th} pair f_i and g_i. Then, replace these feature values with the values $0.5(f_i + g_i)$ and $0.5(f_i - g_i + 1)$, thus mixing up the relevant features with the irrelevant features via linear combinations. The results of the experiments are shown in Figure 4.

Because WPNN fully met its theoretical expectations, the training set must have found good sets of weights. It was able to do this by evaluating relatively few points: about 300 per feature weight (i.e., 3000 for Dataset 1, 4500 for Dataset 2 and 6000 for Datasets 3 and 4) with about half of these eliminated quickly via the two heuristics.

6.2.2 Feedforward Sigmoidal

We have run a series of experiments discussed in detail in [Montana, 1989]. We summarize the results here. First, comparing the three general-purpose mutations resulted in a clear ordering: (1) MUTATE-NODES, (2) BIASED-MUTATE-WEIGHTS, and 3) UNBIASED-MUTATE-WEIGHTS (see Figure 6.2.2). Second, a comparison of the three crossovers produced no clear winner (see Figure 6.2.2). Third, when MUTATE-WEAKEST-NODE was added to a mutation and crossover operator, it improved performance only at the

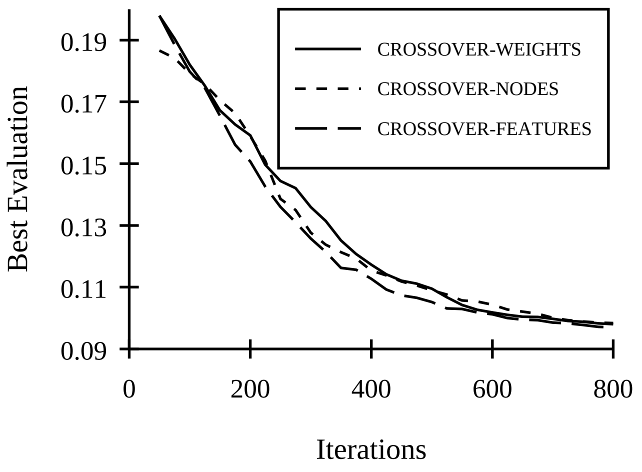

Figure 6: Comparison of crossovers.

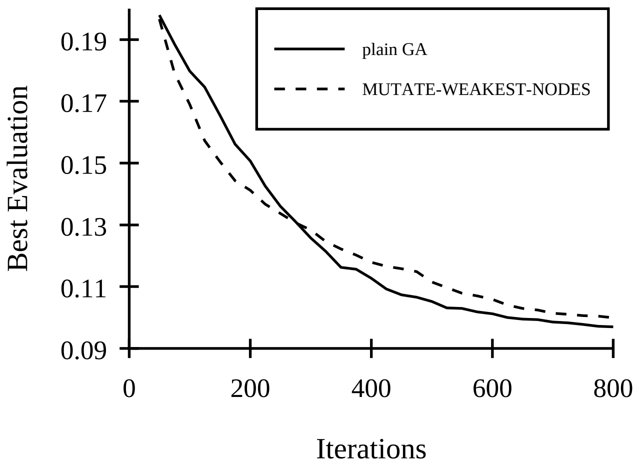

Figure 7: Effect of MUTATE-WEAKEST-NODES.

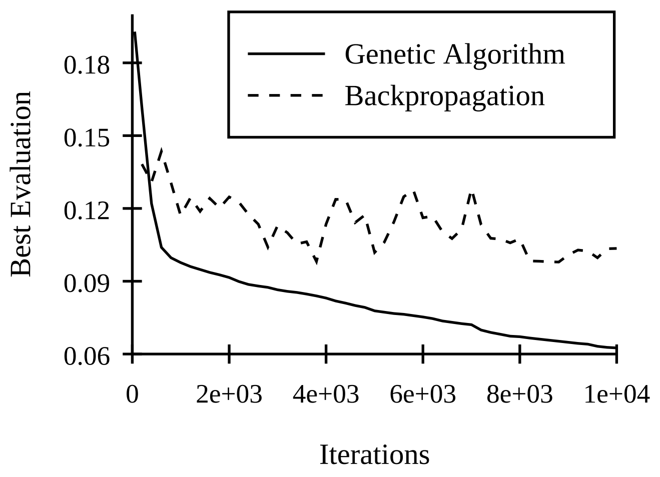

Figure 8: Comparison of genetic algorithm with backpropagation.

beginning; after a certain small amount of time, it decreased performance (see Figure 6.2.2). Fourth, a hill-climbing mode with just a HILLCLIMB operator and a population of size two gave temporarily better performance than MUTATE-NODES and CROSSOVER-NODES but would quickly get stuck at a local minimum. Finally, a genetic training algorithm with operators MUTATE-NODES and CROSSOVER-NODES outperformed backpropagation on our problem (see Figure 6.2.2).

6.2.3 Others

A variety of experimental evidence from other work supports some of the main conclusions of our work: first, genetic algorithms can find nearly globally optimal weight values and are especially useful when used in conjunction with a local hill climber [BELE90, DECK93], and second, genetic algorithms can train network architectures for which other training algorithms do not exist [TORR91, KOZA91].

7 Future Directions

The application of genetic algorithms to neural network weight selection, and more generally neural network training, is a rapidly growing area of research with a few different trends. The first trend is towards more rigorous experimentation on "standard" datasets. In early work, researchers often used their own datasets on which gradient-based techniques did not work well (and hence which required a different type of training algorithm). However, it is now time to compare genetic algorithms against gradient-based techniques on well-known and well-characterized datasets on which gradient-based techniques have achieved success.

The second trend is towards applying genetic algorithms to new neural network architectures. In particular, whole new architectures, such as WPNN, are being developed because it is known that genetic algorithms will be able to train them.

The third, and probably the most important, trend is towards the use of more complex evaluations functions. The most interesting potential consequence of this is the acceleration of neural network development in areas other than pattern recognition and function approximation. While neural networks have provided small improvements over previous algorithms for these two tasks, neural networks will not reach their potential for making computers "smarter" until they can perform tasks such as early visual processing and control of autonomous vehicles. Without use of genetic algorithms, researchers have had to hand-design neural networks to perform such tasks, which cannot be evaluated as the sum of the errors squared. Genetic algorithms provide a potential method to learn neural networks for such tasks and hence to simplify the development of such networks and to make them more robust.

Acknowledgments

The work on training sigmoidal networks was done with Dave Davis using an early version of his OOGA code [DAVI91].

References

[ATKE91] C. G. Atkeson, "Using locally weighted regression for robot learning," *Proceedings of the 1991 IEEE Conference on Robotics and Automation*, pp. 958–963, 1991.

[BELE90] R. K. Belew, J. McInerney, and N. N. Schraudolph, "Evolving Networks: Using the Genetic Algorithm with Connectionist Learning," *Artificial Life II,* pp. 511-547, 1992.

[BORN91] S. Bornholdt and D. Graudenz, *General Asymmetric Neural Networks and Structure Design by Genetic Algorithms,* Deutsches Electronen-Synchrotron, 1991.

[CHAN91] E. J. Chang and R. P. Lippmann, "Using Genetic Algorithms to Improve Pattern Classification Performance," *Advances in Neural Information Processing 3*, pp. 797–803, 1991.

[COLL91] R. Collins and D. Jefferson, "An Artificial Neural Network Representation for Artificial Organisms," *Parallel Problem Solving from Nature,* 1991.

[COOM87] S. Coombs and L. Davis, "Genetic Algorithms and Communication Link Speed Design: Constraints and Operators," *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications,* 1987.

[DAVI87] L. Davis (ed.), *Genetic Algorithms and Simulated Annealing,* Pitman, London, 1987.

[DAVI91] L. Davis, *Handbook of Genetic Algorithms,* Von Nostrand Reinhold, New York, 1991.

[DECK93] D. Decker and J. Hintz, "A Genetic Algorithm and Neural Network Hybrid Classification Scheme," *Proceedings of AIAA Computers in Aerospace 9 Conference,* 1993.

[EDEL87] G. M. Edelman, *Neural Darwinism,* Basic Books, New York, 1987.

[GOLD88] D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning,* Addison-Wesley, Redwood City, CA, 1988.

[GRUA92] F. Gruau, "Genetic Synthesis of Boolean Neural Networks with a Cell Rewriting Development Process," *Proceedings of the IEEE Workshop on Combinations of Genetic Algorithms and Neural Networks,* 1992.

[HARP89] S. A. Harp, T. Samad and A. Guha, "Towards the Genetic Synthesis of Neural Networks, *Proceedings of the Third International Conference on Genetic Algorithms,* pp. 360–369, 1989.

[HOLL75] J. H. Holland, *Adaptation in Natural and Artificial Systems,* University of Michigan Press, 1975.

[HORN89] K. Hornik, M. Stinchombe, and H. White, "Multilayer Feedforward Networks are Universal Approximators," *Neural Networks,* vol. 2, pp. 359–366, 1989.

[KARU92] N. Karunanithi, R. Das, and D. Whitley, "Genetic Cascade Learning for Neural Networks," *Proceedings of the IEEE Workshop on Combinations of Genetic Algorithms and Neural Networks,* 1992.

[KELL91] J. D. Kelly, Jr. and L. Davis, "Hybridizing the genetic algorithm and the k nearest neighbors classification algorithm," *Proceedings of the Fourth Internation Conference on Genetic Algorithms*, pp. 377–383, 1991.

[KOZA91] J. R. Koza and J. P. Rice, "Genetic Generation of Both the Weights and Architecture for a Neural Network," *Proceedings of the IEEE Joint Conference on Neural Networks,* pp. 397–404, 1991.

[MEAD89] C. A. Mead, *Analog VLSI and Neural Systems,* Addison-Wesley, Reading, MA, 1989.

[MONT89] D. J. Montana and L. Davis, "Training Feedforward Neural Networks Using Genetic Algorithms," *Proceedings of the International Joint Conference on Artificial Intelligence,* pp. 762–767, 1989.

[MONT91] D. J. Montana, "Automated Parameter Tuning for Interpretation of Synthetic Images," in [DAVI91], pp. 282-311.

[MONT92] D. J. Montana, "A Weighted Probabilistic Neural Network," *Advances in Neural Information Processing Systems 4,* pp. 1110–1117, 1992.

[PORT90] V. W. Porto and D. B. Fogel, "Neural Network Techniques for Navigation of AUVs," *Proceedings of the IEEE Symposium on Autonomous Underwater Vehicle Technology,* pp. 137–141, 1990.

[POTT92] M. A. Potter, "A Genetic Cascade-Correlation Learning Algorithm," *Proceedings of the IEEE Workshop on Combinations of Genetic Algorithms and Neural Networks,* 1992.

[RISS89] J. Rissanen, "Stochastic Complexity in Statistical Inquiry", *World Scientific,* N.J., 1989.

[RUME86] D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing, vol. 1,* MIT Press, Cambridge, MA, 1986.

[RUME86b] D. E. Rumelhart G. E. Hinton and R. J. Williams, "Learning Internal Representations by Error Propagation," in *Parallel Distributed Processing, vol. 1,* MIT Press, Cambridge, MA, 1986.

[SCHA90] J. D. Schaffer, R. A. Caruana and L. J. Eshelman, "Using Genetic Search to Exploit the Emerging Behavior of Neural Networks," in S.— Forrest (ed.) *Emergent Computation,* pp. 102–112, 1990.

[SCHA92] J. D. Schaffer, D. Whitley, and L. J. Eshelman, "Combinations of Genetic Algorithms and Neural Networks: A Survey of the State of the Art," *Proceedings of the IEEE Workshop on Combinations of Genetic Algorithms and Neural Networks,* 1992.

[SPEC90] D. F. Specht, "Probabilistic neural networks," *Neural Networks*, vol. 3, no. 1, pp. 109–118, 1990.

[STON74] M. Stone, "Cross-validatory choice and assessment of statistical predictions," *Journal of the Royal Statistical Society*, vol. 36, pp. 111–147, 1974.

[SYSW89] G. Syswerda, “Uniform Crossover in Genetic Algorithms,” *Proc. Third International Conference on Genetic Algorithms,* pp. 2–9, 1989.

[TORR91] J. Torreele, “Temporal Processing with Recurrent Networks: An Evolutionary Approach,” *Proc. Fourth International Conference on Genetic Algorithms,* pp. 555–561, 1991.

[WHIT89] D. Whitley, “Applying Genetic Algorithms to Neural Network Learning, *Proceedings of the Seventh Conference of the Society of Artificial Intelligence and Simulation of Behavior,* pp. 137–144, 1989.

[WHIT90] D. Whitley, T. Starkweather and C. Bogart, “Genetic Algorithms and Neural Networks: Optimizing Connections and Connectivity,” *Parallel Computing,* vol. 14, pp. 347–361.

[WHIT91] D. Whitley, S. Dominic and R. Das, “Genetic Reinforcement Learning with Multilayer Neural Networks,” *Proceedings of the Fourth International Conference on Genetic Algorithms,* pp. 562–569, 1991.

[WIEL91] A. P. Wieland, “Evolving Neural Network Controllers for Unstable Systems,” *IEEE Joint Conference on Neural Networks,* pp. 667–673, 1991.