Parallel Computing 14 (1990) 347-361
North-Holland

Genetic algorithms and neural networks: optimizing connections and connectivity

D. WHITLEY, T. STARKWEATHER and C. BOGART
Computer Science Department, Colorado State University, Fort Collins, CO 80523, USA

Abstract. Genetic algorithms are a robust adaptive optimization method based on biological principles. A population of strings representing possible problem solutions is maintained. Search proceeds by recombining strings in the population. The theoretical foundations of genetic algorithms are based on the notion that selective reproduction and recombination of binary strings changes the sampling rate of hyperplanes in the search space so as to reflect the average fitness of strings that reside in any particular hyperplane. Thus, genetic algorithms need not search along the contours of the function being optimized and tend not to become trapped in local minima. This paper is an overview of several different experiments applying genetic algorithms to neural network problems. These problems include
(1) optimizing the weighted connections in feed-forward neural networks using both binary and real-valued representations, and
(2) using a genetic algorithm to discover novel architectures in the form of connectivity patterns for neural networks that learn using error propagation.
Future applications in neural network optimization in which genetic algorithm can perhaps play a significant role are also presented.

Keywords. Optimization, neural network, genetic algorithm.

1. Introduction

Learning neural network connection weights using genetic algorithms might be considered analogous to "evolving" a "hardwired" set of connections. An "evolutionary" approach to learning may not at first seem practical. One might suspect, for example, that genetic recombination produces an extremely slow search process. However, genetic algorithms are capable of a much faster search than might be expected. They also have many advantages. They are fairly simple to develop and represent a global search method. Perhaps the greatest advance to genetic algorithms, however, is that they make almost no assumptions about the problem space that they are searching. When optimizing a neural network, genetic algorithms require no information about how the weights (parameters) that the genetic algorithm is asked to evaluate are actually used. No gradient information is required. This could mean that it is possible to use the same genetic algorithms described in this paper to optimize recurrent networks, for example, which tend to have restrictive storage and computational requirements when trained with true gradient descent learning techniques [28]. Furthermore, it may be possible to use the genetic algorithm to learn networks especially designed to meet restrictive hardware implementation requirements. All of these possible applications are suggested by the results which have already been achieved. This paper
(1) reviews work applying genetic algorithms to neural network connection weight optimization for several test problems and two real world applications,
(2) reviews the foundations of genetic algorithms and research into theoretical problems related to the use of genetic algorithms for neural network weight optimization, and

(3) shows how genetic algorithms can be used to find interesting connectivities for small neural network problems.

The optimization problems posed by neural networks provide a new and challenging test for genetic algorithms. The theoretical foundations of genetic algorithms typically assume that the optimization problem one is trying to solve is represented as a binary encoding so that hyperplanes can explicitly be represented by different combinations of bits [12]. The standard test suite that researchers have used for over a decade to test genetic algorithms is largely made up of problems with encodings of less that 50 bits [6]; neural networks can have encoding with hundreds or thousands of bits. Thus, the results reported here not only address how genetic algorithms can be applied to neural network optimization problems, but also how to improve genetic search to achieve more accurate and consistent performance on problems that are significantly larger than many of those to which genetic algorithms have been applied in the past.

In a standard genetic algorithm the entire population undergoes reproduction in a single generation with offspring displacing parents. In the algorithm used here, which we refer to as the *GENITOR* algorithm, two parents are first selected from the population. Copies of the parents are then recombined, producing two offspring. One of the offspring is randomly discarded, the other is then allowed to replace the lowest ranking string in the population – the offspring does not replace a parent string. This new string is then ranked according to its performance relative to the remainder of the population, so that it may now compete for reproductive opportunities. We have previously reported that *GENITOR* appears to yield superior results when compared to a standard genetic algorithm when optimizing small binary encoded problems (the DeJong test suite [6,25]), but is especially more effective when compared on larger, more difficult optimization problems [24]. We have also found that search can be enhanced by mechanisms that help to sustain genetic diversity.

One way in which we have been able to sustain genetic diversity is by using *adaptive mutation*. Adaptive mutation responds to the level of genetic diversity in the population. During the early stages of search when there is ample diversity in the population, mutation occurs at very low rates. However, as diversity decreases in the population, the mutation rate increases. In this way mutation helps to sustain diversity and thereby helps to sustain the potential of the search.

We have also found that a distributed genetic algorithm can be used to speed up search while at the same time increasing the consistency of the genetic algorithm at finding precise solutions [27]. *GENITOR II* is a distributed version of *GENITOR* which uses several smaller distributed populations in place of a single large population. By occasionally swapping individuals between the subpopulations two complementary effects are achieved. First, the effects of genetic drift in the various subpopulations are countered. Second, the variability that results globally between the subpopulations is actually used at a local level as a source for new, yet high quality genetic material that allows diversity to be sustained in a way that constructively contributes to the search process.

Despite these enhancements to our algorithm, the neural network problems we have been able to solve using binary encodings remain relatively small. However, Montana and Davis [17] report optimizing larger networks using genetic recombination and real valued encodings. We have reproduced similar results on both large and small problems. This raises certain questions, since any real valued encoding can be replicated in binary form and any operators that act on a real encoding can be made to operate with the same effect on the binary encoding. While this issue is far from being resolved, we discuss some of the difficulties inherent in optimizing the connection weights in neural network using genetic algorithms and offer some suggestions for future research aimed at resolving these difficulties.

The other major problem which we address is the use of genetic algorithms to define the connectivity of a neural network. Some work in this area has been done by Harp, Samad and Guha [11]. Wilson [29] has looked at similar ideas for "evolving" perceptrons. Muhlenbien [19] has presented an overview of how neural networks and genetic algorithm methods can be combined at various levels. In particular, his work addresses how genes might specify neural network architecture. On the one hand, the genetic encoding might be developed so as to specify a general, flexible structure that is modified by development; on the other hand it can specify exact structure and functionality [7]. Real genetic systems almost certainly do not specify exact neural connectivities since the number of neurons and connections is larger than the number of genes coding the neural system [19]. However, coding for exact structure is the approach taken in our work and in similar work by Miller, Todd, and Hegde [16]. Our purpose is to show that genetic search has the potential of developing specific structures that affect the learning abilities of artificial neural networks.

The approach we have used is to define a "maximally connected" feed-forward network and then use the genetic algorithm to discover a combination of connections that enhances learning ability. This means that the approach is a kind of pruning algorithm, but it is distinct from other kinds of pruning methods in that connections are not simply removed. The space of "possible connectivities" is explored with connections being both removed and reintroduced. Our results show that it is possible to find connectivities which actually enhance a network's ability to learn.

2. Foundations of genetic algorithms

To understand the theory behind genetic algorithms one must first understand how recombination takes place. Consider the following binary string: 1101001100101101. In general, the string might represent a possible solution to any problem that can be parameterized. This string could represent, for example, a simple neural net with 4 links, where each connection weight is represented by 4 bits. The goal of genetic recombination is to find a set of parameters that yield an optimal or near optimal solution to the problem. Recombination requires two parents. Consider the string 1101001100101101 and another binary string yxyyxyxxyyyxyxxy where x and y are used to represent 0 and 1. Using two "break-points" recombination occurs as follows:

```
11010   01100101   101
yxyyx   yxxyyyxy   xxy
```

Swapping the fragments between the two parents produces the offspring: 11010yxxyyyxy101 and yxyyx01100101xxy.

Genetic algorithms typically start by randomly generating a population of strings representing the encoded parameters. The strings, or "genotypes," are then evaluated to obtain a quantitative measure of how well they perform as possible problem solutions. Reproductive opportunities are allocated such that the best strings receive more opportunities to reproduce than those which have poor performance; this bias need not be great to produce the required selective pressure to allow "artificial selection" to occur.

To understand how recombination on binary strings can be related to hyperspace, consider a "genotype" that is encoded with just 3 bits. With 3 bits the resulting search space is three dimensional and can be represented by a simple cube. Let the points 000, 001, 010 and 011 be the front face of the cube. Note that the front plane of the cube can be characterized as "all points that begin with 0." If * is used as a "don't care" or wild card match symbol, then this plane can also be represented by the similarity template 0**. These similarity templates are referred to as schemata; each schema corresponds to a hyperplane in the search space. All bit strings that match a particular schema lie in its hyperplane. In general, every binary encoding corresponds to a corner in an L-dimensional hypercube and is a member of $2^L - 1$ different hyperplanes, where L is the length of the binary encoding. For example, the string 011 not only samples its own corner in the hypercube (011) but also the hyperplanes represented by the schemata 0**, *1*, **1, 01*, 0*1, *11, and even ***.

The characterization of the search space as a hypercube is not just a way of describing the space; it relates very much to the theoretical foundations of genetic search. Note that each schema represents a different genetic "fragment," a different combination of "alleles." When recombination occurs, strings are actually exchanging hyperplane information. For example, if 10101100 and 11011110 are recombined, the offspring must reside in the hyperplane 1***11*0; the part of the search space that is in contention between the two strings is -010--0- and -101--1-. Booker [4] refers to these as the "reduced surrogates" of the parent strings; Booker notes that recombination can be applied to the reduced surrogates of the parents in such a way as to guarantee that the offspring will not duplicate the parents. It can also insure that each corner in the hypercube between the two parents which can be generated by recombination has an equal chance of being sampled. Thus, using a "reduced surrogate" operator allows the genetic algorithm to obtain new hyperplane samples as often as possible; it also reduces the likelihood of producing duplicate strings in the population.

Note that the offspring resample those hyperplanes (represented by schemata or genetic fragments) inherited from the parents, but they resample them in a new context – from a new corner in the hypercube with a different evaluation. By testing one new genotype, additional information is gained about the $2^L - 1$ hyperplanes that intersect at a corner in the hypercube where that genotype resides.

After a genotype has been tested it is probabilistically given the chance to reproduce at a rate that reflects its "fitness" relative to the remainder of the population. Of course, it is not necessary to resample the point 101 if it has a fixed evaluation. Recombining the "genetic material" from two parents by crossing the binary encodings allows other hyperplanes in the search space to be sampled (or resampled), but these new probes will be biased toward those hyperplanes that have displayed above average performance. If a schema is most often found in genotypes, or strings, that have above average performance, then this indicates that the schema may represent a hyperplane in the search space with above average performance. In other words, the binary strings which lie in this hyperplane on average do a better job at optimizing the target problem than the average genotype in the population. This means that more points in this hyperplane should be checked. Recombining "fragments" does exactly this, and it does it in such a way that many hyperplanes which have displayed above average performance increase their representation in the populations.

For a more thorough discussion of the theoretical foundations of genetic algorithms see John Holland's original treatise "Adaptation in natural and artificial systems" [12] and David Goldberg's "Genetic algorithms in search, optimization and machine learning" [8]. Heinz Muhlenbein [20] also presents an overview of genetic algorithms as well as a discussion of a broader class of algorithms based on an evolutionary approach to combinatorial optimization.

3. Optimizing neural network weights

Two very different kinds of experiments have been conducted with the goal of optimizing the connection weights in feed-forward neural networks. One set of experiments is largely built on the theoretical foundations of genetic algorithms. Another set of experiments was motivated by previous research carried out by Lawrence Davis and his collaborators. The experiments and the results are very different – and perhaps say as much about the current state of genetic

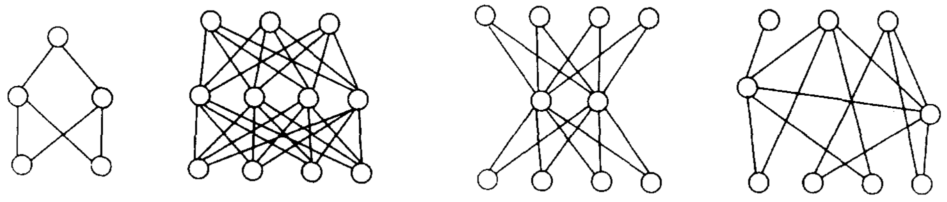
Fig. 1. Four small nets optimized using genetic recombination.

algorithm research as they do about the application of genetic algorithms to neural network problems.

GENITOR has been used to optimize the connection weights in four neural net problems using a binary encoding. The test problems are
(1) the exclusive-or (Xor) problem,
(2) a 424-encoder, and
(3)two versions of an adder problem [15].
The nets for these problems are given in Fig. 1. The adder problem involves the addition of 2 2-bit numbers. These first experiments largely conform to the prevailing theory of genetic algorithms – they use a binary encoding and primarily rely on recombination to drive the search.

To optimize neural net connection weights, the problem is first encoded as a binary string. It is assumed that the connectivity of the net has already been defined and that each connection weight can take on a predefined finite range of settings. Eight bits were used to represent each connection weight. These eight bits were interpreted as a signed magnitude number ranging between -127 and $+127$ with 0 occurring twice.

The genetic algorithm generates a population of binary strings, each of which represents one set of connection weights for an entire net. Thus, each connection weight is viewed as a separate parameter and each binary string is simply the concatenated connection weight parameters for a single net. Initially a population is randomly generated. After a new binary string is created, an interpreter then takes this binary data and uses it to set the connection weights in the network. The network is then run in a feed-forward fashion for each training pattern just as if one were going to use backpropagation. The sum of the squared error is then accumulated; this value represents the "performance" or "fitness value" of the binary genotype. In the *GENITOR* approach, this performance is used to rank the string, and the ranking is then used to determine how to allocate reproductive opportunities.

The transfer function used in the nodes of all of our nets is the fairly standard logistic function. Since it does not use gradient information, the genetic algorithm does not necessarily require that the transfer function be a sigmoid. However, it is useful that the error on the output nodes be continuous, not just discrete, since it allows the genetic algorithm to better discriminate between the performance of different strings representing different solutions.

In our initial experiments we emphasized population size as a means of obtaining the genetic diversity needed to solve the various problems. *GENITOR* found solutions for the Xor problem, the 424 encoder problem, and the minimal 2-bit adder problem, but had difficulty with the fully connected 2-bit adder problem. We will largely limit our discussion to the minimal 2-bit adder and the fully connected 2-bit adder.

The minimally connected 2-bit adder has only two hidden units and allows direct connections between the input and output. The GENITOR algorithm correctly and completely solved this problem every time using a population of 3000, using between 60000 and 90000 recombinations. No mutation was used. The net was encoded using 144 bits. Every solution reduced error to between 10^{-10} and 10^{-30}. Clearly the genetic algorithm is able to solve this problem by merely changing sampling rates of the hyperplanes in the population.

The fully connected 2-bit adder problem has four hidden nodes and three output nodes. The layers are fully connected and the encoding uses 280 bits. This problem was actually more difficult for the genetic algorithm to solve.

With the *GENITOR* algorithm there emerged a very consistent pattern relating population size, accuracy of optimization, and the number of recombinations needed to achieve optimization. Increasing population size consistently slows down the learning rate (more recombinations are needed as population size increases) but larger populations converge to a more accurate solution. We have found it often necessary to double the population size for an incremental increase in performance. Unfortunately, search time (as measured in number of recombinations) tends to double as well [24].

3.1. Adaptive mutation

The experiments with population size conveyed a clear message: population diversity is critical to sustaining search. It seems reasonable that some form of mutation might be used to sustain diversity. Introducing a constant rate of mutation improves performance somewhat, but not enough to dramatically affect search speed or the accuracy of results.

Since the purpose of mutation is to sustain genetic diversity, we looked for ways to monitor population diversity with the intent of invoking mutation in response to increasing genetic homogeneity in the population. These results showed marked improvements over the static mutation settings. Population homogeneity was indirectly monitored by measuring the hamming distance between the two parent strings during reproduction. The more similar the two parents, the more likely that mutation would be applied to bits in the offspring created by recombination. While the population is still diverse, mutation is nearly "turned off," letting the population converge toward an optimum. With an increased occurrence of homogeneity between parents, mutation increases to levels that provoke a search for new, yet competitive, genetic material. This does not disrupt the existing population since it is maintained in rank order; a new genotype must be competitive before it can become a stable member of the population. A more detailed discussion of the types of functions used to implement adaptive mutation is presented elsewhere [27].

We have ran 30 experiments using a population of 5000 and allowing 2 million recombinations. With adaptive mutation we were able to find solutions for the fully connected 2-bit adder on 17 out of 30 runs (56%) using adaptive mutation.

3.2. Distributed genetic search

A distributed genetic algorithm provides an alternative means of sustaining genetic diversity. Using several smaller populations in parallel allows information to be aggressively exploited within a subpopulation without totally exhausting the genetic diversity of the entire population. Each subpopulation will, due to inherent sampling biases, exploit its genetic material in a slightly different way and converge to a slightly different, but similarly competitive solution. Muhlenbein [20] has reported that a fine grain parallel genetic algorithm implementation produces superior results to a serial algorithm for a large Traveling Salesman Problem; the search finds a best known solution for a 442 city problem. Although our distributed genetic implementation is somewhat different, we have also matched best known solutions on Traveling Salesman Problems and have demonstrated improved search on other problems that include neural network optimization problems as well as "deceptive" optimization problems specifically designed to mislead genetic search [9,27]. Tanese also reports results using a distributed genetic algorithm that appear to be consistent with our findings [21].

By occasionally swapping individuals new yet high quality genetic material can be introduced into the evolving subpopulations. While these are only rough guidelines, we have found that with a subpopulation of size X, swapping after $5X$ to $10X$ recombinations (in each subpopulation) works well. When a swap occurs, each subpopulation sends its best to one of its neighbors. In our implementation we view the subpopulations as a circle. On swap 1, position 0 passes 5 of its best strings to its nearest neighbor at position 1. All other subpopulations do the same. On swap 2, position 0 passes its best strings to position 2; position 1 is now passing its best to position 3. On swap N, position X passes its best to position $X + N$. In this way, it is possible to distribute a string to each of the N subpopulations in $\log(N) + 1$ swaps.

Swapping should not occur too often or the effect will be similar to simply having a single population. On the other hand, swapping must occur often enough to prevent the subpopulations from converging toward incompatible solutions. This is particularly true in a problem such as neural network optimization; there are typically multiple ways in which to find an appropriate set of weighted connections and recombining disparate solutions may result in a disfunctional net.

Using a combination of adaptive mutation and a distributed genetic algorithm, we were able to solve the fully connected 2-bit adder problem on 28 out of 30 runs (93%). The distributed *GENITOR II* used 10 populations of 500 each running in parallel for up to 200 000 recombinations. The amount of search done by the serial search (with adaptive mutation) and the distributed results can be compared since 10 subpopulations of 500 performing 200 000 recombinations represents a comparable number of string evaluations as that used with a single population of 5000 running for 2 million recombinations. (The distributed algorithm actually does a comparable amount of work. The distributed algorithm has the overhead of swapping strings, but does less work sorting the 10 subpopulations than the serial genetic algorithm which must sort a single population that is 10 times as large.) Thus the 93% convergence rate for the distributed genetic algorithm compare well with the 56% convergence rate of the serial search, as well as a 90% convergence rate for backpropagation. (The backpropagation data is presented in Table 1 later in this paper.)

Despite the success of the distributed algorithm, the search times involved are excessively long. In the next section, we discuss why neural net optimization poses difficulties for genetic recombination and how those difficulties can be resolved.

3.3. The problem with ANN

Why is it difficult to optimize one artificial neural network (ANN) and not another? A simple answer might be that genetic algorithms do not scale up well. It is well known that a bias exists against schemata that are highly separated in the binary encoding. As the string increases in length a greater proportion of the sampled hyperplanes will span more than half the string, and thus will be more likely to be disrupted by crossover. We assume this contributes to the difficulty of longer problems. But it is probably not the main problem.

A more likely cause of difficulty is that there are often a number of different solutions to any one neural net problem [17]. Consider the following simplied example: assume that to solve some neural net problem we need 3 hidden nodes that perform tasks A, B and C. When all the hidden nodes have the same connectivity pattern it is possible in one parent that hidden node 1 may be doing task A, while 2 is doing B and 3 is doing task C. In another parent, node 1 maybe doing task B, node 2 task C and node 3 task A. We will use the phase "structural/functional mapping" to refer to how a neural net associates a particular functionality with a specific node. As illustrated in Fig. 2, crossing parent nets with different structural/functional mappings can result in a loss of functionality in the offspring. In general, when two or more hidden units have identical sets of connections (to the input and output layer, for example) then the structural/

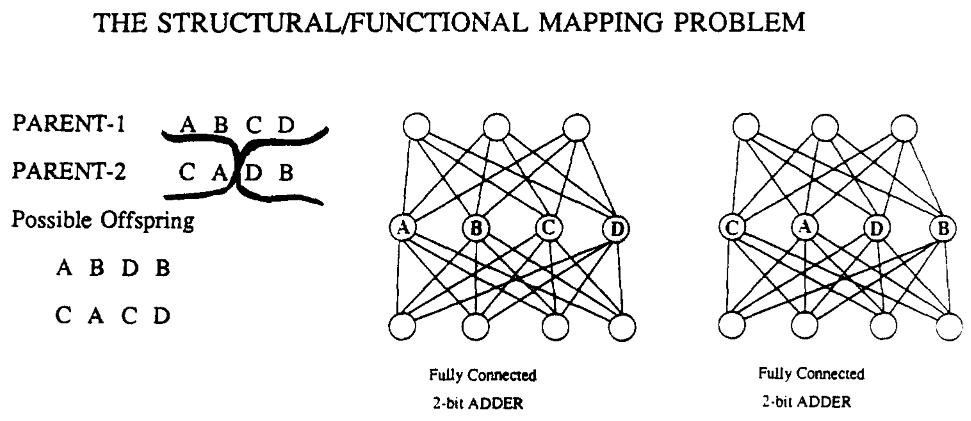
THE STRUCTURAL/FUNCTIONAL MAPPING PROBLEM
PARENT-1 A B C D
PARENT-2 C A D B
Possible Offspring
A B D B
C A C D
A B C D
Fully Connected 2-bit ADDER
C A D B
Fully Connected 2-bit ADDER

Fig. 2. Assume that the hidden nodes in some problem each perform a distinct function. With 4 hidden nodes, these functions can he labeled A, B, C and D. But since each hidden node can take on any functionality, it is possible to distribute functionality differently in different nets. Recombining dissimilar nets can result in disfunctional strings which produces inconsistent hyperplane samples.

functional mapping will be arbitrary: these nodes can swap their functionality without altering the functionality of the net as a whole.

One possibility for the poor results on the fully connected 2-bit adder may be that we were more likely to recombine nets that have different structural/functional maps and thus to produce non-functional offspring.

Our ability to easily solve the minimally connected 2-bit adder may be related to its structural/functional mapping. The minimally connected adder does not have the same connectivity pattern between the two hidden units. Thus, the connectivity forces the two hidden units to always assume the same functional/structural mapping. (One hidden node monitors the carry bit in the first bit position, the second monitors carries in the second bit position.) This means that recombining encodings for this problem may produce more consistent evaluations. In a problem where there is potential conflict in the structural/functional mapping, the evaluations will tend to be "better" if the two parents are similar but "worse" if they are dissimilar. This kind of problem produces hyperplane samples that are not particularly useful to the genetic algorithm because of their high variance. The probability of different structural/functional mappings increases as the number of hidden units increases if the hidden units all have identical connection patterns.

Hyperplane samples which show considerable variance in their fitness values makes the search space more difficult to explore because it provides inconsistent and conflicting feedback about whether a hyperplane represents a good region in the search space or not [3]. Our results suggest this problem can be overcome, but it certainly slows the rate of convergence and makes genetic search more difficult.

3.4. Using real-valued encodings

Despite the success achieved using the distributed algorithm and adaptive mutation, search time on the binary-encoded problems can sometime be rather long and does not compare well with backpropagation. However, Montana and Davis [17] report excellent results using a genetic algorithm that uses a real-valued encoding. Their application was to optimize a neural network for the classification of sonar data. They describe several different experiments. We chose to duplicate those experiments that differ the least in implementation from the experiments we have presented. Since they employed a *GENITOR* style algorithm (one-at-a-time reproduction and a rank based allocations of trials) only three major differences exist. First, as noted, a real-valued encoding is used. This means that each parameter (weight) is presented by a single real-value and that recombination can only occur between parameters. Second, a much higher level of mutation is used and that mutation is such that a random value is added to (or subtracted from) the existing weight rather than replacing it. Third, they used a small population (50 individuals). The small population may be important in part because our empirical data indicates that only one solution typically will exist in a population of this size. However, this empirical observation has other implications as will be explained.

In our experiments we have obtained convergence in 90% of our runs on the fully connected 2-bit adder problem using a population of fifty. The average convergence time (for those that converged) was 42 500 training cycles. We also applied the same code (population of 50, real-value encodings, a high rate of mutation) to a real world application that involves the optimization of a neural network for signal detection [22]. The training data is made up of 300 signal samples. This network has over 300 connections and is difficult to train. The error, when mapped as a function of training time, asymptotically approaches zero. We have never obtained zero error when training this net with backpropagation. We also did not reach zero error using the genetic algorithm, but did obtain our lowest error to date (detection is 100% correct while agreement in the signal period is 99.5% correct) and we achieved this value faster than on any of our runs using back propagation.

Why does this approach work so readily? Our analyses indicate that in these small populations only a single solution is being pursued by the algorithm. Therefore the problem of recombining disparate solutions does not arise. However, the empirical evidence also indicates the search being performed by this kind of genetic algorithm is different from that which is characterized by the fundamental theorem of genetic algorithms – the "schema theorem." Theoretically, genetic search proceeds by changing the sampling rate of hyperplanes in the population. But our analysis of these searches indicates this is not the primary mechanism at work. In these small populations, the diversity quickly goes to zero. In our experiments on the 2-bit fully connected adder using a population of 50, the diversity (as represented by the performance difference between the best and worst individuals in the population) is reduced to less than 0.001 by the time the search had carried out 30 000 evaluations in each of 25 cases that we checked. Diversity remains low throughout the search. Diversity this low would completely stall a standard genetic algorithm which primarily uses recombination to drive the search.

We tried reducing the population size down to 5 individuals on the 2-bit adder problem. The algorithm then found a solution in 80% of the runs (40 out of 50), but the average convergence time for those that converged was reduced to 24 000 data presentations – almost half of what it had been using a population of 50. In a population of 5 strings, diversity was always reduced to below 0.001 within 5000 evaluations and in 11 out of 25 of the cases we checked, diversity was reduced to within 0.001 before 1000 evaluations. This data strongly suggests that stochastic hill-climbing is the dominant search mechanism, not intelligent hyperplane sampling.

This is not to say that we should not use a genetic algorithm that works in a way other than that which corresponds to the dominant theoretical paradigm. Perhaps it suggests that the prevailing theory is too restrictive. This is also not to say that recombination or the use of a genetic algorithm is irrelevant – although we strongly suspect (and are current testing the idea) that hill-climbing from a single solution will produce good results in this particular domain. If the 5 population and 50 population tests are any indication, we might expect hill-climbing from a single string to be faster in most cases, but it might also fail to converge more often.

To be fair, we should note that there are differences between our implementation and that used by Davis and his collaborators which could impact on our observations (Davis, personal communication). We allow duplicates – which allows diversity to decay to zero. If we did not allow duplicates, then in a population of size N we would always retain N unique strings. This could perhaps allow recombination to have more impact on the direction of search.

It is surprising that an algorithm that relies so strongly on stochastic hill-climbing should do so well. This is particularly notable if one considers that one evaluation requires less than half the computation of a backpropagation cycle. (An evaluation typically involves one (often partial) new feed-forward evaluation for the genetic hillclimbing algorithm versus a complete feed-forward and a complete error propagation pass for backpropagation.) This perhaps suggests that much simpler training algorithms exist for neural network optimization than the ones we are currently using.

4. Defining network connectivity

Our experiments now turn to a different application of genetic algorithms to neural networks. Defining the connectivity of an artificial neural network so as to enhance learning speed or to use only "necessary" connections is a difficult task that is not well understood. The general approach discussed here is similar to one previously used by Miller et al. [16]. Other work along these lines has also been carried out by Wilson [29] and by Harp et al. [11].

We define a net that is as large or larger than necessary to do the job, and then use a genetic algorithm to discover some combination of connections which are sufficient to quickly and accurately learn to perform some target task using backpropagation. This approach is similar to "pruning" techniques that have been developed for neural networks. In fact we have investigated the use of more traditional pruning methods in the context of the signal detection problem – including node pruning [18,5], weight pruning (e.g. methods similar to the "optimal brain damage" approach [13]) and the use of weight decay [10]. While our work with the application of genetic algorithms for this task is preliminary, we have produced interesting results.

We have used *GENITOR* to "prune" the connections of a feed-forward neural network. Unlike other pruning methods, the genetic algorithm does not remove one connection at a time, but rather explores the interaction of possible connections to find a combination of connections with some desired property. Connections can be both removed and reintroduced through recombination. As our results indicate, this approach can be used not only to reduce the size of the net, but also to discover topologies with enhanced learning abilities. This appears to be in contrast to other pruning methods that appear to increase the number of local optima and to make learning more difficult [10,18].

To use a genetic algorithm to define network connectivity there needs to be some explicit mechanism for rewarding nets that use fewer connections. This is a non-trivial problem, since directly rewarding or penalizing a net based on the number of connections used can give a "selective advantage" to nets that are not able to learn; in the extreme case, a net might try to gain reward or avoid penalties by pruning all of its connections. We have found a way to reward nets that use fewer connections while at the same time selecting for nets that learn quickly and accurately.

Second, on larger nets, the amount of time required to find a net that learns quickly and accurately is quite significant, since it means evaluating a population of strings, where each evaluation involves running backpropagation on a single net. For a simple net which learned to compute the exclusive-or (Xor) of its two inputs, Miller et al. used enough backpropagation cycles that they could reasonably expect to find a net that would be able to learn the task completely. But this is not practical on larger problems because it means training the same net over and over again.

One alternative is to use a reduced number of backpropagation cycles. The difficulty with this approach is that the resulting nets would not be able to completely learn the problem during evaluation. Furthermore, a fast initial reduction in error does not guarantee that the net will continue to learn. To avoid this, we trained the net before pruning, using the resulting weights to initialize the various nets generated by the genetic algorithm. While this might seem at odds with the objective of finding a net that learns quickly, having to do backpropagation from scratch 2000 times (or even 100 times) is a false economy. The real issues, we think, are the following.

(1) This approach finds much smaller nets that can learn the task. In particular this has important implications for problems where a hardware implementation is the ultimate goal.
(2) Pruning has an impact on generalization and noise tolerance.
(3) Since pruning does introduce error, there is selective pressure in our approach for nets that relearn quickly.

As a side effect of this research we have arrived at the following conjecture: *It appears that for some nets with a hidden layer, faster learning can be achieved adding additional direct connections from the input layer to the output layer.* At this point, this suggestion should only be considered a hypothesis and only rigorous testing can confirm or refute this hypothesis, but there is sufficient evidence to warrant further research. Several hundred runs on Xor and 2-bit adders (and a more limited number of runs on the signal detection problem) suggest that nets with direct I/O connections are somewhat faster and less prone to becoming stuck in local optima.

4.1. The encoding, evaluation function and reward scheme

The problem encoding for defining neural net connectivity is simple. Consider the following binary string: 1101001100101101. This string would represent, in the current problem, a net which originally was solved using 16 links. The potential solution represented by this string uses only 9 of those sixteen links. Evaluation in this case involves running backpropagation on the net using the connections indicated by the binary encoding. In our approach, the initial net has already solved the problem, so we are actually pruning a developed net and finding which nets are least affected or best able to relearn given a specified number of backpropagation cycles. The more links that are removed, the greater the probability that the functionality of the net will suffer. Thus, if the goal is to prune the net, there must be some compensation for nets that use fewer links.

The reward scheme used here is simple but effective. We start with some baseline number of backpropagation cycles, B. For each link that is pruned from the net, we increase the number of backprop cycles the net is allowed by some delta, D. Thus, if the genetic algorithm defines a net that has N number of links pruned, then we allow $(ND + B)$ backprop cycles. This means that nets with fewer links are given more learning opportunities, but they are not actually rewarded unless they are able to exploit the opportunity. At this point, we are using the weights that have been learned for the already trained fully connected net (we refer to these as the "starting weights") to initialize the pruned nets.

4.2. Experiments and results

We have tested our approach by pruning a fully connected 2-bit adder and Xor. On Xor, we begin with a standard net with two hidden units, with the addition of direct connections between the input layer and the output layer. The genetic algorithm pruned this to the standard net with one hidden unit. The adder net has four input nodes, four hidden nodes, and three output nodes; in addition, a "true" node was used for learning the bias, or activation

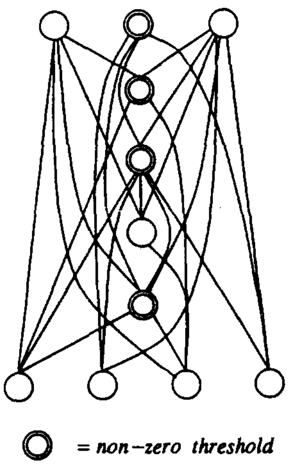

Fig. 3. The net pruned by *GENITOR* for the 2-bit adder.

thresholds [2]. To begin with, all input units were connected to all hidden units, and all hidden units were connected to the output units. The true node is connected to all hidden units and output units. In addition to these connections, all input units were given a direct connection to each output unit. Finally, because the minimal adder described by Rumelhart et al. [15] has connections between members of the hidden layer, we allowed hidden unit 1 to feed to hidden units 2, 3, and 4, hidden unit 2 to feed to hidden units 3 and 4, and hidden unit 3 to feed to hidden unit 4. Thus, with 53 connections the net is "maximally connected;" it has all the possible connections that could be used and still have a feed-forward network.

The genetic algorithm pruned the net to 29 connections. Fig. 3 gives the net developed by the genetic algorithm. Since the ability of a neural net to learn quickly can very much depend on what weights it is initialized with, we took the "pruned" net and ran a number of learning experiments from small random initial weights in order to collect statistics about the net's learning ability. The results were averaged over 50 runs. Surprisingly, the pruned net not only learned faster, but it was very consistent in the number of backpropagation cycles its required to learn the problem (Table 1, Net C). In fact, the pruned net learned more consistently than Table 1 indicates, learning every time with between 8200 and 8700 presentations of the training

Table 1
Distribution of training times for a 2-bit adder

Number of training repetition	Percentage of nets that had completed training		
	NET A	NET B	NET C
8,000– 9,000	4%	0%	100%
9,000– 10,000	4%	0%	
10,000– 20,000	10%	62%	
20,000– 50,000	34%	32%	
50,000–100,000	24%	2%	
Over 100,000	14%	0%	
Never Converged	10%	4%	

NET A: Standard fully connected net
NET B: Standard net with direct I/O
NET C: Net pruned by *GENITOR*
Sample size: 50

data. Also note that the pruned net uses several connections which go directly from the input layer to the output layer. Miller et al. found the same thing in their exclusive-or networks. This led us to also compare a net that is fully connected with direct connections between the input and output layers (Table 1, Net B) to a net that is fully connected between the input layer and the hidden layer, and between the hidden layer and the output layer, but which does not allow a direct connection between the input and output layer (Table 1, Net A). Our results, averaged over fifty runs, suggest that nets which have links that directly go from the input to the output layer do learn somewhat faster. The nets with direct I/O connections also require a more consistent number of data presentations and were less likely to fail to converge to a solution. This suggests that they are less likely to become trapped in local minima. The pruned net found by the genetic algorithm never became stuck. As noted previously, this result appears to be different from the behavior of other one-at-a-time connection pruning methods that increase the difficulty of the learning task.

The next step in this line of research is to combine our "genetic hillclimbing algorithm" with a search to define the connectivity. Both would occur simultaneously. We think this will be more successful than using backpropagation because it will promote a closer, more continuous relationship between the evolving topology and the weights.

5. Conclusions and future directions

The potential for "evolving" neural networks that are customized for specific applications is one of those ideas that sparkles the imagination and conjures up visions of artificial life. At this point in time however, genetic algorithms, or artificial genetic systems, are, much like artificial neural systems, still a very idealized and simplified model when compared to biological reality. Nevertheless, we have shown that it is possible to optimize both the connection weights and the connectivity itself in small nets. The work presented here has also raised questions that must be addressed by the genetic algorithm research community. Besides some work, by Ackley [1], there has been very little research looking at algorithms that explicitly integrate hillclimbing abilities into a genetic search.

The results of Montana and Davis [17], and our replication of similar results on a different set of test problems, indicate that search speeds that are very competitive with backpropagation can be achieved. This is exciting because genetic search does not display the kind of restrictions that limit backpropagation. For example, there would appear to be no reason why genetic search could not be used to optimize recurrent networks, since genetic algorithms makes no assumptions about how the values (parameters) that the genetic algorithm is asked to evaluate are actually used. Genetic search also makes no assumptions about what kind of transfer functions are used, or whether the nodes in the network are distributed over 3 or 9 layers. As noted in the introduction, it may be possible to use genetic algorithms to develop networks especially designed to meet restrictive hardware implementation requirements. Mesard and Davis [14] have also used a hybrid algorithm that combines the genetic algorithm with backpropagation.

There is also a very real potential for defining networks in which the weights and connectivity are simultaneously defined. Suggestions for work along these lines has already begun appearing in the "neuroevolution" research community; reports on experiments that attempt to optimize both weights and connectivity will no doubt appear in the next year. By defining connectivity and weights simultaneously it may be possible to actually further increase learning speed and at the same time enhance network performance. To do this the genetic algorithm, while searching for an appropriate set of weights, would simultaneously search for a connectivity that would make the problem space easier to search. If these two processes can be made to complement one another, then the result should be smaller networks that learns faster. This would be done, not by running backpropagation on hundreds or even thousands of different nets, but by optimizing the connectivity in a single net while also optimizing the weights at the same time.

In the short term, genetic algorithms can give us the ability to explore possible network architectures far more effectively than a trial and error approach. Finding several nets in several different domains that have enhanced learning capabilities could provide a set of "enhanced" nets that could then be studied in order to determine if there are general characteristics that can be more directly reproduced when building other neural net systems. The long term potential of using genetic algorithms for neural network research is a much more open area of research. Much of the work that has been done so far has been carried out by only a handful of researchers. Nevertheless, the results suggest that the application of genetic algorithms to neural network optimization problems has the potential to impact the way that artificial neural systems are developed and optimized.

Note

Portions of this paper have appeared in brief conference papers dealing with results using a distributed genetic algorithm [26] and the use of a genetic algorithm to define connectivity [23].

Acknowledgements

This research was supported in part by a grant from the Colorado Institute of Artificial Intelligence (CIAI). CIAI is sponsored in part by the Colorado Advanced Technology Institute (CATI), an agency of the State of Colorado. CATI promotes advanced technology education and research at universities in Colorado for the purpose of economic development.

References

[1] D. Ackley, *A Connectionist Machine for Genetic Hillclimbing* (Kluwer, Boston, 1987).

[2] D. Ackley, G. Hinton and T. Sejowski, A learning algorithm for Boltzmann machines, Cognitive Sci. **9** (1985) 147–169.

[3] J. Baker and J. Grefenstette, How genetic algorithms work: a critical look at implicit parallelism, in: *Proc. Third Internat. Conf. Genetic Algorithms* (Morgan Kaufmann, San Mateo, CA, 1989).

[4] L. Booker, Improving search in genetic algorithms, in: Lawrence Davis, ed., *Genetic Algorithms and Simulated Annealing* (Morgan Kaufmann, San Mateo, CA, 1987).

[5] Y. Chauvin, A back-propagation algorithm with optimal use of hidden units, in: D. Touretzky, ed., *Advances in Neural Network Information Processing Systems* (Morgan Kaufmann, San Mateo, CA, 1989).

[6] K. DeJong, An analysis of the behavior of a class of genetic adaptive systems, Ph.D. Thesis, University of Michigan, 1975.

[7] C. Dolan and M. Dyer, Towards the evolution of symbols, in: *Proc. Third Internat. Conf. Genetic Algorithms* (Morgan Kaufmann, San Mateo, CA, 1989).

[8] D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning* (Addison-Wesley, Reading, MA, 1989).

[9] D. Goldberg, B. Korb and K. Deb, Messy genetic algorithms: Motivation, analysis, and first results, TCGA Report No. 89003 (1989) to appear in *Complex Systems*.

[10] S. Hanson and L. Pratt, Comparing biases for minimal network construction with back-propagation, in: D. Touretzky, ed., *Advances in Neural Network Information Processing Systems* (Morgan Kaufmann, San Mateo, CA, 1989).

[11] S. Harp, T. Samad and A. Guha, Towards the genetic synthesis of neural networks, in: *Proc. Third Internat. Conf. Genetic Algorithms* (Morgan Kaufmann, San Mateo, CA, 1989).

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

[13] Y. Le Cun, D. Denker, S. Solla, R. Howard and L. Jackel, Optimal brain damage, in: *1989 IEEE Conf. Neural Information Processing.*

[14] W. Mesard and L. Davis, Back propagation in a genetic search environment, in: *1989 IEEE Conf. Neural Information Processing.*

[15] D. Rumelhart, G. Hinton and R. Williams, Learning Interanl representations by error propagation, in: D. Rumelhart and J. McClelland, eds., *Parallel Distributed Processing, Vol. I* (MIT Press, Cambridge, MA, 1986) 318-362.

[16] G. Miller, P. Todd and S. Hegde, Designing neural networks using genetic algorithms, in: *Proc. Third Internat. Conf. Genetic Algorithms* (Morgan Kaufmann, San Mateo, CA, 1989).

[17] D. Montana and L. Davis, Training feedforward neural networks using genetic algorithms *Proc. 1989 Internat. Joint Conf. Artificial Intelligence.*

[18] M. Mozer and P. Smolensky, Skeletonization: A technique for trimming the fat from a network via relevance assessment, in: D. Touretzky, ed., *Advances in Neural Network Information Processing Systems* (Morgan Kaufmann, San Mateo, CA, 1989).

[19] H. Muhlenbein, The dynamics of evolution and learning – toward genetic neural networks, in: R. Pfeifer, ed., *Connectionism in Perspective* (Elsevier, Amsterdam, 1989) 173-198.

[20] H. Muhlenbein, M. Gorges-Schleuter and O. Kramer, Evolution algorithms in combinatorial optimization, *Parallel Comp.* 7 (1988) 65-85.

[21] R. Tanese, Distributed genetic algorithms, in: *Proc. Third Internat. Conf. Genetic Algorithms* (Morgan Kaufmann, San Mateo, CA, 1989).

[22] D. Whitley, Signal detection using neural networks and genetic algorithms, Report to The Colorado Institute for Artificial Intelligence (1989).

[23] D. Whitley and C. Bogart, The evolution of connectivity: Pruning neural networks using genetic algorithms, in: *Proc. 1990 Internat. Joint Conf. Neural Networks* (Lawrence Erlbaum, Hillsdale, NJ, 1990).

[24] D. Whitley, and T. Hanson, Optimizing neural networks using faster, more accurate genetic search, in: *Proc. Third Internat. Conf. Genetic Algorithms* (Morgan Kaufmann, San Mateo, CA, 1989).

[25] D. Whitley and J. Kauth, *GENITOR*: A different genetic algorithm, in: *Proc. Rocky Mountain Conf. Artificial Intelligence,* Denver, CO (1988).

[26] D. Whitley and T. Starkweather, Optimizing small neural networks using a distributed genetic algorithm, in: *Proc. 1990 Internat. Joint Conf. Neural Networks* (Lawrence Erlbaum, Hillsdale, NJ, 1990).

[27] D. Whitley and T. Starkweather, GENITOR II: A distributed genetic algorithm, submitted to: *J. Experimental Theoretical Artificial Intelligence.*

[28] R. Williams and D. Zipser, Experimental analysis of the real-time recurrent learning algorithm, *Connection Sci.* **1** (1989) 87-111.

[29] S. Wilson, Perceptron redux: Emergence of structure, in: *Proc. Conf. Emergent Computation* (Los Alamos National Laboratory, Los Alamos, NM, 1989).