Optimization of Neural Networks Using Genetic Algorithms

Yanxin Wang

***Abstract*—Genetic algorithms are a robust adaptive optimization method based on biological principles. Neural networks are a computational paradigm modeled on the human brain. This paper is an overview of combining genetic algorithms with neural network. Three ways of using genetic algorithms to optimize neural networks are introduced, including preprocessing data, training weights and topology optimization. Experiments on training neural networks with genetic algorithms are also described.**

I. Introduction

Genetic algorithms and neural networks are powerful tools for artificial intelligent systems. Mechanisms of them are both inspired by biological systems. Genetic algorithms follow the process of species evolution and heredity while neural networks simulate the structures of biology neural systems. Basically, a great deal of biological neural architecture is determined by gene. Therefore it is no surprise that some neural network researchers explored the idea of conjunction of genetic algorithms and neural networks.

Genetic algorithms have been used in combination with neural networks in three major ways. First, they have been used to train the weights and biases in already set up networks. Most times they are used to enhance or replace gradient descend methods. This can be applied to both supervised learning applications and reinforcement learning applications. Genetic algorithms have also been used to set the learning rates which in turn are used by other types of training methods. Second, genetic algorithms have been used to learn neural network topologies. This includes the problem of specifying how many hidden units a neural network should have and how the nodes are organized. Third, genetic algorithms have been used to select training data and to interpret the output behavior of neural networks.

This paper will introduce these aspects of using genetic algorithms for neural networks. Section II demonstrates using genetic algorithm for preprocessing data. Section III illustrates how to optimization topologies with genetic algorithms, followed by using genetic algorithms to train weights in neural networks in section IV. Section V gives an example of applying genetic algorithms to train neural networks for function approximation.

II. Preprocessing and Interpreting Data with Genetic Algorithm

Genetic algorithms have been used for preprocessing data in other fields like image processing. This kind of preprocess can improve the performance of processing systems effectively. The basic idea of using genetic algorithms to preprocess data is to eliminate those meaningless data so that the problem is easier to be solved. Two examples of using genetic algorithms for preprocessing data is given in the work of Chang and Lippmann[1] and work of Brill, Brown and Martin[2]. Both cases deal with a K nearest neighbor (KNN) classifier with a huge number of inputs. In this case, the chromosome, which carries heredity information in genetic algorithms, can be a simple binary string indicating whether a particular input or combination of inputs can be eliminated from the input set without significantly changing the classification behavior. 120 out of 153 input features in the input set were reduced by genetic algorithms in Chang and Lippmann application. Brill, Brown and Martin not only were capable of reducing the input set to the nearest neighbor classifier but also want to indentify inputs that would also work well for a backpropagation network. The evaluation of a feature set is much faster with KNN classifier than with the backpropagation network since they have totally different mechanisms. Nevertheless, the reduced input set for KNN classifier also worked well for the backpropagation network despite the difference between them.

Genetic algorithms have not only been used to reduce the input data set but also been used to interpret outputs of a neural network. One of the barriers to users' acceptance of neural network applications is the lack of explanation facilities. That is, the neural network system cannot be queried to ascertain how a classification was obtained, or what the differential is between the current classification and anther classification. Genetic algorithms could be used to interpret the output of neural networks. Eberhart and Dobbins [3] used a genetic algorithm to search for features to explain how a neural network classifies novel inputs. They defined the term "codebook vector" as an input pattern that result in a maximum or nearly maximum activation value for a given output neurode. And the term "decision surface" refers to the surface in hyperspace that separates one classification from another. They demonstrated the use of a genetic algorithm to define codebook vectors and points on the decision surfaces for a trained neural network. The genetic algorithms implementation uses a trained backpropagation neural network weight matrix as the genetic algorithms fitness function. Using different combinations of Genesis' run-time options, codebook vectors and decision surfaces are defined for the trained neural network. According to their conclusion, storing a sufficient number of these vectors and points could provide the basis for a neural network explanation facility. When working on a trained network system, given an input pattern, it should be possible to examine additional information, such as the location of nearby decision surfaces, in real-time or near-real-time. Genetic algorithms is much

Yanxin Wang is with University of Waterloo, 200 University Avenue West, Waterloo, Ontario, N2L 3G1, Canada (Phone: 01-226-220-1899, e-mail: y574wang@uwaterloo.ca).

better than attempt to run a neural backwards to look for node activation that yields a particular output because back-propagation can be time consuming when number of nodes is too large.

III. Optimizing Neural Network Topologies with Genetic Algorithms

Early efforts on optimization of neural network topologies were to directly encode network architectures into chromosome string. Assuming that the number of hidden units was bounded, genetic algorithms could then be used to determine what combinations of weights or hidden units yield improved computational behavior within a finite range of architectures. These directly coded network architectures have usually been trained using backpropagation. Miller and Todd [4] have explored these ideas, as have Belew, McInerney and Schraudolf[5]. Whitley, Starkweather and Bogart [6] show that genetic algorithms have the ability to explore possible network architectures far more effectively than a trial and error approach. They used the term "pruning" to refer the optimization techniques but also pointed out that this approach can be used to both reduce and to increase the size of the net. They found a way to reward nets that use fewer connections while at the same time selecting for nets that learn quickly and accurately. As a side effect of their research, it appeared that for some nets with a hidden layer, faster learning can be achieved by adding additional direct connections from the input to the output layer. Several hundred runs on XOR and 2-bit adders suggested that nets with direct input-output connections were somewhat faster and less prone to becoming stuck in local optima. An example of the effort to reduce the network topology for a 2-bit adder is given in Fig.1. Network C was evolved by a genetic algorithm and learned to add in between 8,000 and 9,000 training epochs on 50 out of 50 tests. As shown in Table.I, network A failed to converge on 5 of the 50 tests, and about half of the networks required more than 50,000 training epochs. Network B was created by adding direct input-output connections to nodes of Network A. Network B learned the training set in between 10,000 and 50,000 training epochs on 46 out of 50 tests.

TABLE. I Distrubution of traning 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 feedforward network

NET B: Standard network with direct input-output connections

NET C: Network pruned by genetic algorithm

Sample size: 50

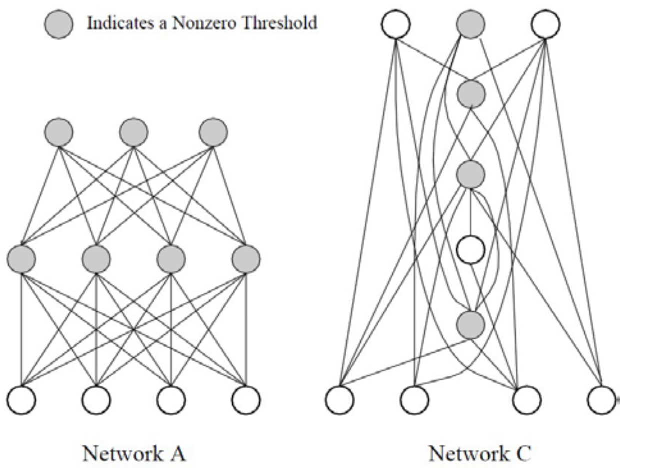

Figure.1 Network A: standard fully connected feedforward network and Network C: Network pruned by genetic algorithm

Whitley, Starkweather and Bogart [6] also pointed out that the optimization of both weights and connectivity using genetic algorithms may increase learning speed and enhance network performance at the same time.

Such early results were encouraging, but the difficulty with directly optimizing network architecture is the high computation cost of each evaluation. Since each evaluation requests to run a backpropagation algorithm or other gradient descent algorithm once, the number of evaluations needed to find improved network architectures quickly becomes computationally prohibitive. The computation cost is too high as to make genetic algorithms impractical except for optimizing small topologies. When finding an architecture-optimized network takes much longer time then training any built ready network, there is no need to apply such search. Therefore, if the architecture is complex, then it is most likely inadequate for genetic search to be effective.

Also, one trend in neural networks that evolving the architecture issue is constructive algorithms such as the Cascade-Correlation Learning Architecture. Instead of just adjusting the weights in a network of fixed topology, Cascade-Correlation begins with a minimal network, then automatically trains and adds new hidden units one by one, creating a multi-layer structure. Thus, genetic algorithm has to be able to handle more complex changes in network architectures other than just fully connected feedforward structure.

Further steps on evolving neural networks require both weights and topologies could be adjusted freely when networks are trained so that network would converge quickly and accurately. In other word, nodes and layers in neural networks should be added or removed while weights between nodes are being adjusted. Just like Whitley predicted, some of the most advanced work during 1990-2000 for using genetic algorithms to develop neural network have focused on growing neural network. Weights and architectures are often developed together. Angeline et al[7]argues that genetic algorithms are inappropriate for network acquisition and describes an evolutionary program, called GNARL, which simultaneously acquires both the structure and weights for recurrent networks. GNARL, which stands for Generalized

Acquisition of Recurrent Links, is an evolutionary algorithm that nonmonotonically constructs recurrent networks to solve a given task. GNARL's empirical acquisition method allows for the emergence of complex behaviors and topologies that are potentially excluded by the artificial architectural constraints imposed in standard network induction methods. Other researchers looked at genetic programming as a way of developing architectures and weights together [8]. In genetic programming, the individuals in the population are computer programs. Koza uses hierarchical compositions of functions and arguments of various sizes and shapes (i.e. LISP symbolic expressions) as individuals. Both the weights and architecture for a one-bit adder network are found by genetic programming in their work. They success in adjusting parameters including the number of layers, the number of processing elements per layer and the connectivity between processing elements.

Grammar based architecture descriptions have been explored by Kitano [9] and Gruau [10]. Since employing direct mapping of chromosomes into network connectivity is the reason for scalability problem, employing more efficient encoding methods might solve that problem. By optimizing grammar structures that generate network architectures instead of directly optimizing architectures, these researches hope to achieve better scalability, and in some sense, reusability of network architectures. In particular, this would allow developers to define neural structures for smaller problems as building blocks that could be reused for solving larger problems.

As an alternative approach to direct encoding, Kitano proposed a graph grammatical encoding that will encode graph generation grammar to the chromosome so that it generates more regular connectivity patterns with shorter chromosome length. Kitano's new scheme provides magnitude of speedup in convergence of neural network design and exhibits desirable scaling property. Later on, Kitano presented a simple model of neurogenesis that is more biological in nature. In this approach, "axons grow while cell metabolisms are being computed."[11]. Cell membranes are also modeled that are capable of chemical transport and diffusion. This work appears to be focused on understanding the emergent properties of this type of system.

Gruau [10] used genetic algorithms to generate neural networks that implement Boolean functions. The algorithm that is put forward yields both the architecture and the weight by using chromosomes that encode an algorithmic description based upon a cell rewriting grammar. The developmental process interprets the grammar and develops a neural network according to its interpretation. The representation on the chromosome is abstract and compact. Any chromosome develops a valid phenotype. The developmental process gives modular and interpretable architectures with a powerful scalability property. Since the rewriting is done on cells, which are nodes with ordered connections, the encoding is called Cellular Encoding. Gruau used a genetic algorithm to recombine grammar trees representing cellular encodings and has showed that neural networks for the parity problem and symmetry problem could be found.

IV. Training neural networks with genetic algorithms

The idea of training neural networks with genetic algorithms can be found in Holland' 1975 book *Adaptation in Natural and Artificial Systems*. Most of the actual work in this area is far more recent. Belew, McInernery and Schraudolph[5], Harp, Samad and Guha[12] and Schaffer, Caruana and Eshelman[13] all used genetic algorithm to set the learning and momentum rates for feedforward neural networks.

This tuning was often done in conjunction with other changes to the network, such as weight initialization or changing the network topology. In addition, there have also been several researchers that attempted to train feedforward neural networks for decision problems using genetic algorithms [6][14][15]. Related to this is the use of genetic search in the optimization of Kanerva's sparse distributed memories by Rogers[16] and Wilson's work[17] which learned predicates over input features to construct new higher order inputs to a perceptron.

Rogers [16] has used genetic algorithms to optimize the "location addresses" (i.e. the layer mapping inputs to hidden units) of a sparse distributed memory. Das and Whitley [18] extend the work of Rogers by using a genetic algorithm for "location address" optimization that actively extracts information about multiple local minima based on relative global competitiveness. Each local optimum in this particular definition of the search space represents a different and distinct data pattern that correlates with some output or event of interest. This allows multiple data patterns to be tracked simultaneously, where each pattern corresponds to a different local optimum in location address space.

The application of genetic algorithms to simple weight training for neural networks has been hampered by two factors. First, gradient methods have been developed that are highly effective for weight training in supervised learning applications where input-output training examples are available and where the target network is a simple feed forward network. Second, the problem of training a feedforward Artificial Neural Network may represent an application that is inherently not a good match for genetic algorithms that rely heavily on recombination. Some researchers do not use recombination while others use small populations and high mutation rates in conjunction with recombination. We first look at why optimizing the weights in a neural network may cause problem for algorithms that rely heavily on simple recombination schemes.

A. Competing Conventions Problem

One reason that genetic algorithms may not yield a good approach to optimizing neural network weights is the *Competing Conventions Problem*. Nick Radcliffe [19] has also named this the *Permutations Problem*. The source of the problem is that there can be numerous equivalent symmetric solutions to a neural network weights optimization problem.

Fig.2 illustrates a simple feedforward network. Assume that the vector

$$w_{a1}, w_{a2}, w_{a3}, w_{b1}, w_{b2}, w_{b3}, w_{c1}, w_{c2}, w_{c3}, w_{d1}, w_{d2}, w_{d3}$$

is an arbitrary assignment of weights to this neural network, where w_{xi} passes through hidden node x and $i=1, i=2$ are input connections and $i=3$ is the output connection. Note that every vector of this form there are $4!=24$ equivalent vectors representing exactly the same solution. All permutations over the set of hidden unit indices, $\{a,b,c,d\}$, are equivalent vectors in terms of neural network functionality and in terms of the resulting evaluation function. This is because rearranging the order of the hidden units has no effect on the functionality. Thus, given H hidden units in a simple fully connected feedforward network, there are $H!$ symmetries and up to $H!$ equivalent solutions.

The problem this creates for a genetic algorithm that uses simple recombination is as follows. If one does simple crossover on a permutation such as $[a \quad b \quad c \quad d]$ and $[d \quad a \quad c \quad b]$ then the offspring will duplicate some elements of the permutation and will omit others. Similarly, if different strings try to map functionality of hidden nodes in different ways, then recombining these strings will result in duplication of some hidden units and omission of other hidden units. In this case, using a population-based form of search can be a disadvantage, since different strings in the population may not map functionality to the different hidden units in the same way.

Various solutions have been proposed to the Competing Conventions Problems. Early on, Montana and Davis [15] attempted to identify functional aspects of hidden units during recombination in order to perform a type of intelligent crossover. Radcliffe[19] also suggested a solution whereby a network topology can be described as a multiset of hidden units each of which is specified by its set of external connections. As showed in Fig.3, this is an entirely non-redundant representation which should allow genetic search to proceed efficiently in principle. However, there is an obvious complication. If the node types are considered as atomic and not available for recombination, it will be extremely difficult for the genetic algorithm to make progress. Fortunately, this situation is familiar in genetic search. The solution usually employed is to allow recombination to take place at the sub-parameter level either through employing binary or other low-cardinality encodings or by using recombination operators which make use of knowledge of the high-level meaning of parameters. Hancock [20] has implemented this idea as well as extensions to consider how similar hidden units are; he concludes the permutation problem is not as bad as has often been suggested. More recently, Korning [21] has suggested that the traditional use of the standard quadratic error measurement is part of the problem and suggests the use of other fitness measurements. Korning also suggests "killing off" any offspring that do not meet minimal fitness requirements, which might filter out offspring from incompatible parents. Overall however, it is very difficult to find cases where genetic algorithms have been shown to yield results better than gradient based methods for supervised learning application.

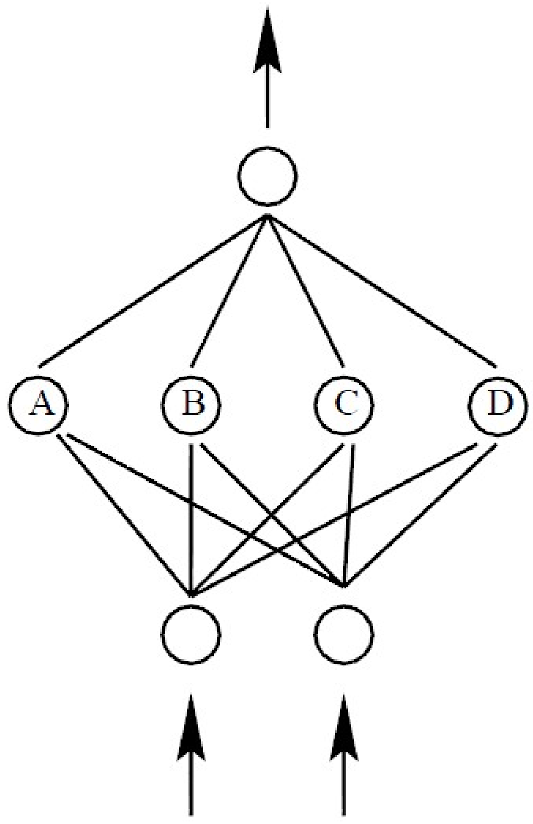

Figure.2 A Simple Feedforward Neural Network

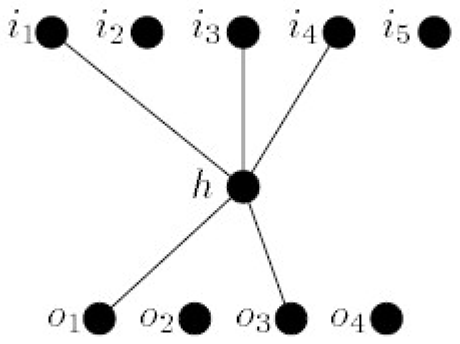

Figure.3 A hidden node h, which can be described by the set of external nodes to which it is connected, $\{i_1, i_3, i_4, o_1, o_3\}$. This can also be conveniently described by the binary string 101101010 where a one indicates the presence of a connection and a zero the absence, and where the nodes are laid out in order with the input nodes preceding the output nodes.

One recent report returns to a theme initially put forward by Belew el al [5]. Part of the traditional wisdom which has grown up around genetic algorithms is that a genetic algorithm is good at roughly characterizing the structure of a search space and finding regions of good average fitness, but not adept at exploiting local features of the search space. One way to use a genetic algorithm then is to use it to find an initial set of good weights and then to turn the search over to a gradient based method. Skinner and Broughton[22] have reported good results with this kind of approach and suggest this method is better than using gradient methods alone for complex problem involving large weight vectors.

B. *Genetic and Evolutionary Algorithms for Reinforcement Learning*

Another strategy is to use genetic and evolutionary algorithms for weight optimization in domains where gradient methods cannot be directly applied, or where gradient methods are less effective than in simple supervised learning applications. One such application is the use of evolutionary algorithms to train neural networks for reinforcement learning problems and neurocontrol applications. Some results suggest that evolutionary algorithms can be quite competitive against other algorithms that are applicable to reinforcement learning problems. For reinforcement learning applications the set of target outputs that correspond to some set of inputs used to train the net are not known a prior. Rather, the evaluation of the network is performance based. Most existing algorithms attempt to convert the reinforcement learning problem to a supervised learning problem by indirectly or heuristically generating a target output for each input. Some approaches compute an inverse of a system model. The system model maps inputs (current state and control actions) to outputs (the subsequent state). Given a target state, the inverse of the system model can be used to generate actions, which can then be used as an output target (the appropriate action) for a separated controller. This general description is applicable to methods such as “Back Propagation Through Time”. “Adaptive Critic” methods use a separate evaluation net that learns to predict or evaluate performance at each time step. The prediction can then be used to heuristically generate output targets for an “action” net which controls system behavior. Note, however, that both of these methods compute target outputs and the resulting gradients either indirectly or heuristically.

Genetic algorithms can be directly applied to reinforcement learning problems because genetic algorithms do not use gradient information, but rather only a relative measure of performance for each set of weight vectors that is evaluated. Genetic algorithms and evolutionary algorithms have been successfully applied to training neural nets to controlling an inverted pendulum. Weiland[23] for example, trained recurrent networks to balance two inverted pendulums of different lengths at the same time, as well as jointed pendulum. These algorithms often use smaller population sizes and higher mutation rates to cope with the “Competing Conventions Problems”. Whitley et al[24][25] compared a genetic hill-climber to the well known work of Anderson[26] which uses the “temporal difference method” [27] to train an “Adaptive Heuristic Critic” (AHC) which in turn is used to generate target outputs for doing reinforcement backpropagation. The results suggest that the genetic algorithms produced training times comparable to the AHC with reinforcement backpropagation, while generalization was better for the genetic algorithm.

Whitley et al[24][25] have argued that comparisons of algorithms for reinforcement learning should not only consider learning time but also generalization. Algorithms that learn very quickly can potentially fail to produce an adequate generalized model of the process being learned. Thus, fast learning is not in and of itself a good measurement for evaluating a training algorithm. Generalization is also effected by how the evaluation function is constructed. In reinforcement learning and control problems, the number of possible initial states can be intractable. Thus, evaluation involves sampling the set of possible start states. Evaluation based on a single fixed start state can result in fast learning, but very poor generalization. Evaluation based on a single random start state is somewhat better, but the resulting evaluation is noisy and is difficult to compare the evaluation of one string against another. Evaluation based on a set of start states that uniformly samples the input space would appear to be the best strategy.

V. Genetic Algorithm as An Alternative to Gradient Descent

This section illustrates an easy model of utilizing genetic algorithm to train neural networks. Genetic algorithms are used as alternative to gradient descent methods in searching weights for connections and biases for neurons. Though this idea is basic and has problems mentioned in previous section, it clearly demonstrates how to use genetic algorithms for neural networks.

A. *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 a simple neural network 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 yxyyxyxxyyyyxyxxy 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 string or “genotypes” are 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 string 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 comer 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 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 re-sampled), 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.

B. Genetic algorithms

Genetic algorithms require five components:

C1. A way of encoding solutions to the problem on chromosomes.

C2. An evaluation function (or fitness function) that returns a rating to each chromosome given to it.

C3. A way of initializing the population of chromosomes.

C4. Operators that may be applied to parents when they reproduce to alter their genetic composition. This includes mutation, cross-over and other operators.

C5. Parameter settings for the algorithm

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

S1. The population is initialized, using the procedure in C3. The result of the initialization is a set of chromosomes as determined in C2.

S2. Each member of the population is evaluated, using the function in C1. Evaluations may be normalized; the important thing is to preserve relative ranking of evaluations.

S3. The population undergoes reproduction until a stopping criterion is met. Reproduction consists of a number of iterations of following three steps:

1) One or more parents are chosen to reproduce. Selection is stochastic, but the parents with the highest evaluations are favored in the selection. Parameters of C5 can influence the selection process.

2) The operators of C4 are applied to the parents to produce children. The parameters of C5 help determine which operators to use.

3) The children are evaluated and inserted into the population. In some versions of the genetic algorithm, the entire population is replaced in each cycle of reproduction. In others, only subsets of the population are replaced.

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

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

Genetic algorithms also have some limitation such as premature convergence and low global convergence speed etc. Poorly chosen fitness functions may generate bad chromosome blocks in spite of the fact that only good chromosome block cross-over. There is no absolute assurance that a genetic algorithm will find a global optimum. It happens very often when the populations have a lot of subjects. And like other artificial intelligence techniques, the genetic algorithm cannot assure constant optimization response times. Even more, the difference between the shortest and longest optimization response time is much larger than with conventional gradient methods.

C. *Optimizing neural networks weights*

A neural network as function approximation approach is tested in this experiment. Target function is $x\cos(x)$ where $x\in[0,10]$. The neural network (Fig.4) has one input, one output and two hidden layers each of five nodes for a total of 47 weights. Now genetic algorithm is used to do neural network weight optimization:

1) Chromosome Encoding: The weights and biases in the neural network are directly encoded as a list of real numbers
2) Evaluation Function: Assign the weights and biases on the chromosome to the network structure, run the network over the training set of examples, and return the sum of the squares of the errors.
3) Genetic Algorithm Operators: Matlab Global Optimization Toolbox is used, including default initialization, crossover and mutation functions.
4) Genetic Algorithm parameters: Genetic search stops if run over 1000 times or change in fitness function is less than 10^{-8} for 20 times in sequence.

D. *Experiments and results analysis*

This section includes a series of experiments which led us to the final version of the genetic algorithm comparing to backpropagation. Gradient descend with adaptive learning rate backpropagation is chosen first.

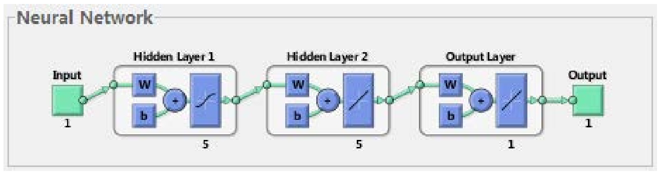

Figure.4 Network structure

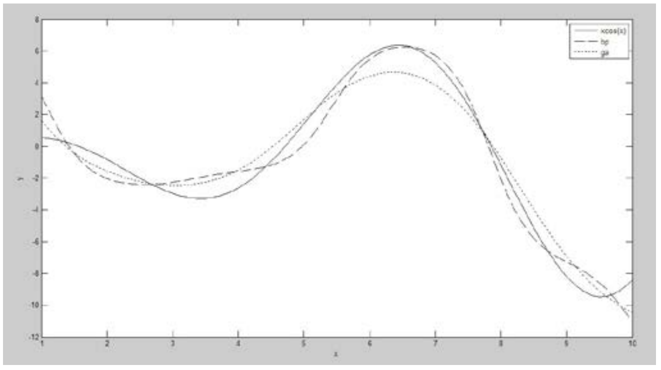

Figure.5 Result comparison between target function, output of network trained with backpropagation and output of network trained with genetic algorithm.

The first observed phenomenon is that genetic algorithm took longer time than backpropagation, which is a limitation of genetic algorithm. Backpropagation usually takes several seconds to do the training while genetic algorithm needs several minutes.

We then run this training process multiple times and record their results. One of the best results is shown in Fig.5. Square error in Backpropagation is 0.8762 and in Genetic Algorithm is 0.9201.

Actually, even in this case, the network is not readily trained because the square error is still huge comparing to our goal which is 10^{-28}. For most of the time, square errors for both backpropagation and genetic algorithm stop at some values, like between 1 and 10, which are much larger than expectation. This is caused by different reasons. For backpropagation (Fig.6), gradient descend process stuck at some point (i.e. local minima). For genetic algorithm, duplicates in individuals decrease the diversity of population leading to a situation where highest ranking individuals reproduce the same children and evolution stops even though fitness function is not optimized.

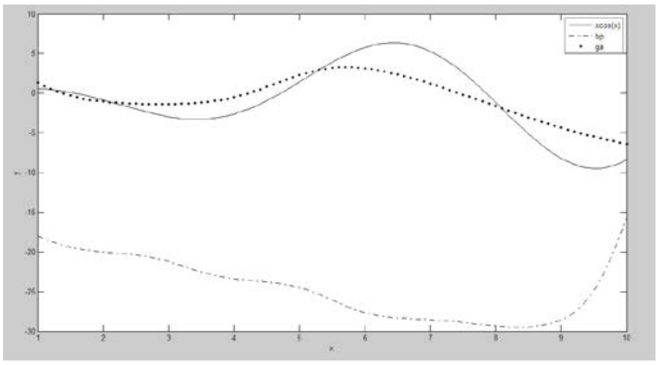

Figure.6 Example of backpropagation failure

However, if we replace Gradient descend with adaptive learning rate backpropagation with Levenberg-Marquardt backpropagation, which is the best gradient descend method in Matlab toolbox, genetic algorithm performs much worse than back propagation (Fig.7). This improvement in gradient descend method inspires us that if applied some techniques this genetic algorithm may overcome its disadvantage.

As mentioned before, genetic algorithm used here consists of default initialization, mutation, cross-over and selection functions. Since these default options may have negative effects on training, we change these parameters and functions to form better algorithms.

The default initialization gives a population of 20, which may be insufficient in this case. So we increase the population to 40 and run network training multiple times, we find out that errors appear to decrease and stabilize. For the first time, in comparison with adaptive backpropagation, genetic algorithm has smaller error (Fig.8). In this case, square error for genetic algorithm is 0.4724 while backpropagation is larger than 1. In other case, we do observe 11.5332 for genetic algorithm but only occasionally. Continue this experiment by increasing population to 80. Average error becomes even lower. We run this approach 10 times and results are compared with population 40 in Table.II. As we obtained, genetic algorithm with population of 80 trained network much better than algorithm with population of 40. This can be explained as that larger population keeps the evolution process going by assuring diversity within generations.

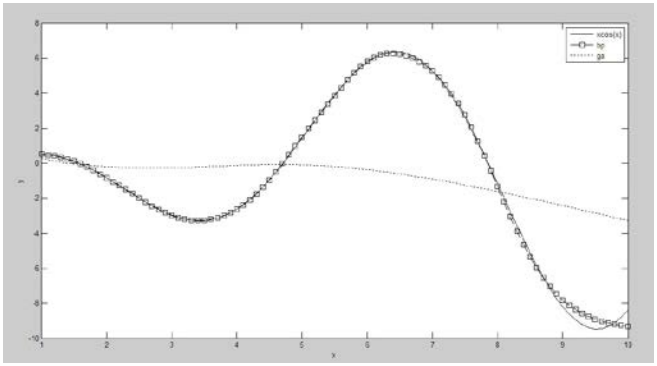

Figure.7 Example of genetic algorithm failure

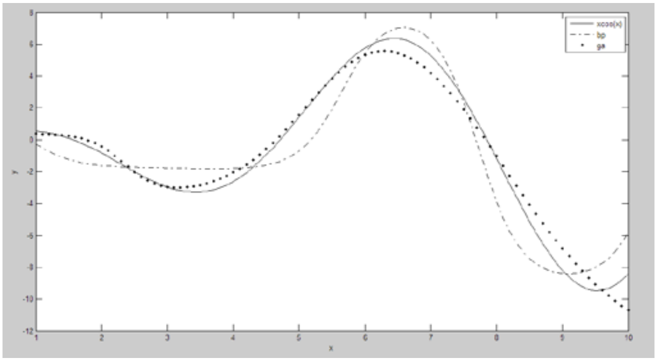

Figure.8 Genetic algorithm with population of 40

TABLE. II Square error in different populations

Time sequence	Population 40	Population 80
1	0.438	0.145
2	1.787	0.347
3	1.120	0.349
4	1.157	0.436
5	1.684	0.701
6	0.209	0.267
7	0.664	0.190
8	0.908	0.631
9	7.778	0.418
10	0.996	0.479
Average	1.674	0.396

We did not try different initial population though Montana proposed a probability distribution random population. We did not do the same initialization because: (i) Montana's attempt only reflects empirical observation by researchers; (ii) genetic material which allows the genetic algorithm to explore the range of all possible solutions could be in any initial individual no matter how they are generated.

Another important element in genetic algorithm is the mutation function. Mutation functions specify how the genetic algorithm makes small random changes in the individuals in the population to create mutation children. By keep an adequate ratio of mutation, new "gene" is injected into the population without disturbing the original dominating "gene". Such mechanism provides genetic diversity and enables the genetic algorithm to search a broader space other than trapped in a local minima. Here in Matlab, the default mutation function, Gaussian, adds a random number taken from a Gaussian distribution with mean 0 to each entry of the parent vector. Will it improve out comings if we use different mutation functions? We then chose another function namely uniform mutation, which is a two-step process. First, the algorithm selects a fraction of the vector entries of an individual for mutation, where each entry has a probability rate of being mutated. The default value of rate is 0.01. In the second step, the algorithm replaces each selected entry by a random number selected uniformly from the range for that entry. We tried this mutation method several times and find it inefficient in such a small population. Then we tune up the mutation rate to 0.1 and later 0.5, which is a horrible rate in nature world. Uniform-mutation genetic algorithms still fail to train the network appropriately. Results of these experiments are shown in Table.III. According to these data, if individuals are uniformly mutated, genetic search only runs less than 100 generations and cannot find appropriate "gene" to reduce the fitness function, which is square error in this case.

The last operator that plays a crucial role in genetic algorithm is cross-over function. Cross-over specifies how the genetic algorithm combines two individuals, or parents, to form a crossover child for the next generation. Scattered, the default crossover function, creates a random binary vector and selects the genes where the vector is a 1 from the first parent,

TABLE. III Square error and generation of uniform mutation(0.01,0.1&0.5)

mutation rate	**0.01**		**0.1**		**0.5**	
	error	***Generation***	***error***	***Generation***	***error***	***Generation***
1	21.45	61	21.36	55	21.32	67
2	21.47	51	21.34	78	21.30	79
3	21.40	63	21.32	87	21.32	74
4	21.48	81	21.33	60	21.32	57
5	21.53	72	21.30	76	21.31	53

and the genes where the vector is a 0 from the second parent, and combines the genes to form the child.For example, if P_1 and P_2 are the parents:

$$P_2 = \begin{bmatrix} a & b & c & d & e & f & g & h \end{bmatrix}$$

$$P_2 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{bmatrix}$$

and the binary vector is $\begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$, the function returns the following child:

$$Child_1 = \begin{bmatrix} a & b & 3 & 4 & e & 6 & 7 & 8 \end{bmatrix}$$

While single point cross-over chooses a random integer n between 1 and number of variables and then select vector entries numbered less than or equal to n from the first parent and vector entries numbered greater than n from the second parent. Finally these entries form a child vector. For example, if P_1 and P_2 are still parents and the cross-over point is 3, the function returns the following child:

$$Child_2 = \begin{bmatrix} a & b & c & 4 & 5 & 6 & 7 & 8 \end{bmatrix}$$

Similar to this, two point cross-over selects two random integer m and n. The function then selects vector entries numbered less than or equal to m from the first parent, vector entries from m+1 to n from the second parent and vector entries numbered greater than n from the first parent. These genes form a single gene as a child. For example, if P_1 and P_2 are parents and the cross-over points are 3 and 6, the function returns following child:

$$Child_3 = \begin{bmatrix} a & b & c & 4 & 5 & 6 & g & h \end{bmatrix}$$

We implement these single point and two point cross-over to our genetic algorithm with 80 populations and find out these mechanisms do not bring performances to a better level. As shown in Table.IV, both single point and two point cross-over lead to greater errors than scattered cross-over. This result is reasonable since the "gene" is directly encoded. The front part of "gene" represents weights for connections followed by codes representing biases. If cut from the boarder of these two part, neither weights nor biases are flexible enough to make any improvement. Furthermore, even a "gene" fraction of two bits will object the evolution because other mechanisms have little chance to change this combination and therefore these two bits are "fixed". In other word, once they are initialized, they only exchange with other fractions in a whole unless mutations take place in them. Consequently this fraction stops evolution ahead of others since lack in diversity. This phenomenon, in turn, reflects to network weights. Some weights in network finish training before fully tuned.

TABLE. IV Square error and generation of different cross-over functions

Single point		**Two point**		**Scattered**	
error	***Generation***	***error***	***Generation***	***error***	***Generation***
4.22	117	2.72	177	0.14	163
3.81	174	5.75	132	0.35	382
2.10	182	2.37	291	0.35	238
3.89	117	0.72	162	0.44	211
1.51	203	5.31	303	0.70	338

VI. CONCLUSION

A literature review of using genetic algorithms to optimize neural networks is given. Preprocessing input data, interpreting output data, training weights and tuning topologies with genetic algorithms are introduced. Then some experiments of using genetic algorithm as an alternative to gradient descent method are accomplished in a feedforward neural network. Direct encoding is used and different operators are tested. Compared with backpropagation, genetic algorithm with large population outperforms gradient descend. The work described here only touches the surface of the potential for using genetic algorithms to train neural networks. In the realm of feedforward networks, there are a host of other operators with which one might experiment. Perhaps most promising are ones which include backpropagation as all or part of their operation.

Finally, as a general-purpose optimization tool, genetic algorithms should be applicable to any type of neural network for which an evaluation function can be derived.

REFERENCES

[1] 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.

[2] F.Z. Brill, D.E. Brown and W.N. Martin, "Fast genetic selection of features for neural network classifiers,"IEEE Transaction on neural networks, 3(2), pp. 324-328, 1992.

[3] R.C. Eberhart and R.W. Dobbins, "Designing neural network explanation facilities using genetic algorithms, "IEEE international joint conference on neural networks, pp. 1758-1763, 1991.

[4] G. Miller, P. Todd and S. Hedge, "Designing neural networks using genetic algorithm", 3rd international conference on genetic algorithms, 1989.

[5] R. Belew, J. McInerney and N. Schraudolph, "Evolving networks: using the genetic algorithms with connectionist learning," CSE Technical Report CS90-174, Computer Science, UCSD, 1990.

[6] D. Whitley, T. Starkweather and C. Bogart, “Genetic Algorithms and Neural Networks: Optimizing Connections and Connectivity,” Parallel Computing, 14, pp. 347-361, 1990.

[7] P.J. Angeline, G.M. Saunders and J.B. Pollack, “An evolutionary algorithm that constructs recurrent neural networks,” IEEE Transaction on Neural networks, 5(1), pp. 54-64, 1994.

[8] J.R. Koza and J.P. Rice, “Genetic generation of both the weights and architecture for a neural network,” In International Joint Conference on Neural Networks, Seattle 92, 1991.

[9] H. Kitano, “Designing neural network using genetic algorithm with graph generation system,” Complex Systems, 4, pp. 461-476. 1990.

[10] F. Gruau, “Genetic synthesis of Boolean neural networks with a cell rewriting developmental prcess,” In Combination of Genetic Algorithms and Neural Networks, D. Whitley and J.D. Schaffer, eds, IEEE Computer Society Press, 1992

[11] H. Kitano, “A simple model of neurogenesis and cell differentiation based on evolutionary large-scale chaos,” Artificial Life, 2, pp. 79-99.

[12] S.A. Harp, T. Samad and A. Guha, Towards the genetic wynthesis of neural networks, In J.D. Schaffer (Ed.), 3rd International conference on genetic algorithms, pp. 360-369, 1989.

[13] J.D. Schaffer, R.A. Caruana and L.J. Eshelman, “Using genetic search to exploit the emergent behavior of neural networks,” In S.Forrest (Ed.), Emergent computation, pp. 244-248, 1990.

[14] D. Whitley and T. Hanson, “Optmizing neural networks using faster, more accurate genetic search,” In J.D. Schaffer (Ed.), 3rd international conference on genetic algorithms, pp. 391-396, 1989.

[15] D.J. Montana and L. Davis, “Training feedforward neural networks using genetic algorithms,” In Proceeding of eleventh international joint conference on artificial intelligence, pp. 762-767, 1989.

[16] D. Rogers, “Predicting Weather Using a Genetic Memory,” Advances in Neural Information Processing 2.

[17] S.W. Wilson, “Perception redux: Emergence of structure,” Physica D: Nonlinear Phenomena, Vol. 42, Issues 1–3, pp. 249–256, June 1990.

[18] R. Das and D. Whitley, “Genetic Sparse Distributed Memories,” Combinations of Genetic Algorithms and Neural Networks, D. Whitley and J.D. Schaffer (Eds.) IEEE Computer Society Press.

[19] N.J. Radcliffe, “Genetic Set Recombination and its Application to Neural Network Topology Optimization,” Neural Computing and Applications, Springer-Verlag, 1:1, pp. 67-90, 1993.

[20] P.J.B. Hancock, “Genetic Algorithms and permutation problems: a comparison of recombination operators for neural net structure specification,” Combinations of Genetic Algorithms and Neural Networks, D. Whitley and J.D. Schaffer (Eds.) IEEE Computer Society Press.

[21] P.G. Korning, “Training of neural networks by means of genetic algorithm working on very lone chromosomes,” Technical Report, Computer Science Department, Aarhus C, Denmark.

[22] A. Skinner and J.Q. Broughton, “Neural Networks in Computational Material Science: Training Algorithms,” Modeling and Simulation in Materials Science and Engineering, 3, pp. 371-390, 1995.

[23] A.P. Weiland, “Evolving controls for unstable systems,” In D.D Touretsky, J.L. Elman, T.J Sejnowski and G.E. Hinton(Eds.), Proceeding of the 1990 connectionist models summer school, San Mateo, CA: Morgan Kaufmann, pp. 91-102, 1990.

[24] D. Whitley, S. Dominic and R. Das, “Genetic Reinforcement Learning with Multilayered Neural Networks,” Proc. 4th International conference on genetic algorithms, 1991.

[25] D. Whitley, S. Dominic, R. Das and C. Anderson, “Genetic Reinforcement Learning for Neurocontrol Problems,” Machine learning, 13, pp. 259-284, 1993.

[26] C.W. Anderson, “Learning to control an inverted pendulum using neural networks,” IEEE Control Systems Magazine, 9, pp. 31-37, 1989.

[27] R. Sutton, “Learning to predict by the methods of temporal differences,” Machine Learning, 3, pp. 9-44, 1988.