7

Evolutionary Computing for Architecture Optimization

This chapter introduces the basic concepts and notation of evolutionary algorithms, which are basic search methodologies that can be used for modelling and simulation of complex non-linear dynamical systems. Since these techniques can be considered as general purpose optimization methodologies, we can use them to find the mathematical model which minimizes the fitting errors for a specific problem. On the other hand, we can also use any of these techniques for simulation if we exploit their efficient search capabilities to find the appropriate parameter values for a specific mathematical model. We also describe in this chapter the application of genetic algorithms to the problem of finding the best neural network or fuzzy system for a particular problem. We can use a genetic algorithm to optimize the weights or the architecture of a neural network for a particular application. Alternatively, we can use a genetic algorithm to optimize the number of rules or the membership functions of a fuzzy system for a specific problem. These are two important application of genetic algorithms, which will be used in later chapters to design intelligent systems for pattern recognition in real world applications.

Genetic algorithms have been used extensively for both continuous and discrete optimization problems (Jang, Sun & Mizutani, 1997). Common characteristics of these methods are described next.

- *Derivative freeness*: These methods do not need functional derivative information to search for a set of parameters that minimize (or maximize) a given objective function. Instead they rely exclusively on repeated evaluations of the objective function, and the subsequent search direction after each evaluation follows certain heuristic guidelines.
- *Heuristic guidelines*: The guidelines followed by these search procedures are usually based on simple intuitive concepts. Some of these concepts are motivated by so-called nature's wisdom, such as the evolution.
- *Flexibility*: Derivative freeness also relieves the requirement for differentiable objective functions, so we can use as complex an objective function as a specific application might need, without sacrificing too much in extra coding

and computation time. In some cases, an objective function can even include the structure of a data-fitting model itself, which may be a fuzzy model.

- *Randomness*: These methods are stochastic, which means that they use random number generators in determining subsequent search directions. This element of randomness usually gives rise to the optimistic view that these methods are "global optimizers" that will find a global optimum given enough computing time. In theory, their random nature does make the probability of finding an optimal solution nonzero over a fixed amount of computation time. In practice, however, it might take a considerable amount of computation time.
- *Analytic opacity*: It is difficult to do analytic studies of these methods, in part because of their randomness and problem-specific nature. Therefore, most of our knowledge about them is based on empirical studies.
- *Iterative nature*: These techniques are iterative in nature and we need certain stopping criteria to determine when to terminate the optimization process. Let K denote an iteration count and f_k denote the best objective function obtained at count k; common stopping criteria for a maximization problem include the following:
  (1) Computation time: a designated amount of computation time, or number of function evaluations and/or iteration counts is reached.
  (2) Optimization goal: f_k is less than a certain preset goal value.
  (3) Minimal improvement: $f_k - f_{k-1}$ is less than a preset value.
  (4) Minimal relative improvement: $(f_k - f_{k-1})/f_{k-1}$ is less than a preset value.

Evolutionary algorithms (EAs), in general, and genetic algorithms (GAs), in particular, have been receiving increasing amounts of attention due to their versatile optimization capabilities for both continuous and discrete optimization problems. Moreover, both of them are motivated by so-called "nature's wisdom": EAs are based on the concepts of natural selection and evolution; while GAs consider genetic information in a more simple binary form.

7.1 Genetic Algorithms

Genetic algorithms (GAs) are derivative-free optimization methods based on the concepts of natural selection and evolutionary processes (Goldberg, 1989). They were first proposed and investigated by John Holland at the University of Michigan (Holland, 1975). As a general-purpose optimization tool, GAs are moving out of academia and finding significant applications in many areas. Their popularity can be attributed to their freedom from dependence on functional derivatives and their incorporation of the following characteristics:

- GAs are parallel-search procedures that can be implemented on parallel processing machines for massively speeding up their operations.

- GAs are applicable to both continuous and discrete (combinatorial) optimization problems.
- GAs are stochastic and less likely to get trapped in local minima, which inevitably are present in any optimization application.
- GAs' flexibility facilitates both structure and parameter identification in complex models such as fuzzy inference systems or neural networks.

GAs encode each point in a parameter (or solution) space into a binary bit string called a "chromosome", and each point is associated with a "fitness value" that, for maximization, is usually equal to the objective function evaluated at the point. Instead of a single point, GAs usually keep a set of points as a "population", which is then evolved repeatedly toward a better overall fitness value. In each generation, the GA constructs a new population using "genetic operators" such as crossover and mutation; members with higher fitness values are more likely to survive and to participate in mating (crossover) operations. After a number of generations, the population contains members with better fitness values; this is analogous to Darwinian models of evolution by random mutation and natural selection. GAs and their variants are sometimes referred to as methods of "population-based optimization" that improve performance by upgrading entire populations rather than individual members. Major components of GAs include encoding schemes, fitness evaluations, parent selection, crossover operators, and mutation operators; these are explained next.

Encoding schemes: These transform points in parameter space into bit string representations. For instance, a point (11, 4, 8) in a three-dimensional parameter space can be represented as a concatenated binary string:

$$\underbrace{1011}_{11}\ \underbrace{0100}_{4}\ \underbrace{1000}_{8}$$

in which each coordinate value is encoded as a "gene" composed of four binary bits using binary coding. other encoding schemes, such as gray coding, can also be used and, when necessary, arrangements can be made for encoding negative, floating-point, or discrete-valued numbers. Encoding schemes provide a way of translating problem-specific knowledge directly into the GA framework, and thus play a key role in determining GAs' performance. Moreover, genetic operators, such as crossover and mutation, can and should be designed along with the encoding scheme used for a specific application.

Fitness evaluation: The first step after creating a generation is to calculate the fitness value of each member in the population. For a maximization problem, the fitness value f_i of the ith member is usually the objective function evaluated at this member (or point). We usually need fitness values that are positive, so some kind of monotonical scaling and/or translation may by necessary if the objective function is not strictly positive. Another approach is to use the rankings of members in a population as their fitness values. The advantage of this is that the objective function does not need to be accurate, as long as it can provide the correct ranking information.

Selection: After evaluation, we have to create a new population from the current generation. The selection operation determines which parents participate in producing offspring for the next generation, and it is analogous to "survival of the fittest" in natural selection. Usually members are selected for mating with a selection probability proportional to their fitness values. The most common way to implement this is to set the selection probability equal to:

$$f_i \Big/ \sum_{k=1}^{k=n} f_k \ ,$$

where n is the population size. The effect of this selection method is to allow members with above-average fitness values to reproduce and replace members with below-average fitness values.

Crossover: To exploit the potential of the current population, we use "crossover" operators to generate new chromosomes that we hope will retain good features from the previous generation. Crossover is usually applied to selected pairs of parents with a probability equal to a given "crossover rate". "One-point crossover" is the most basic crossover operator, where a crossover point on the genetic code is selected at random and two parent chromosomes are interchanged at this point. In "two-point crossover", two crossover points are selected and the part of the chromosome string between these two points is then swapped to generate two children. We can define n-point crossover similarly. In general, $(n-1)$-point crossover is a special case of n-point crossover. Examples of one-and two-point crossover are shown in Fig. 7.1.

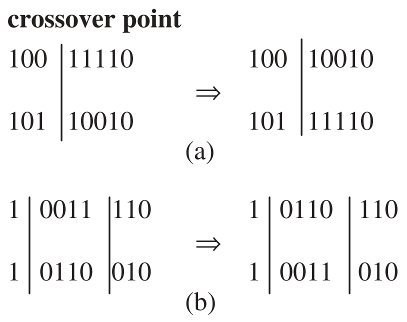


Fig. 7.1. Crossover operators: (**a**) one-point crossover; (**b**) two-point crossover

Mutation: Crossover exploits current gene potentials, but if the population does not contain all the encoded information needed to solve a particular problem, no amount of gene mixing can produce a satisfactory solution. For this reason, a "mutation" operator capable of spontaneously generating new chromosomes is included. The most common way of implementing mutation is to flip a bit with a probability equal to a very low given "mutation rate". A mutation operator can prevent any single bit from converging to a value throughout the entire population and, more important, it can prevent the population from converging and stagnating at any local optima. The mutation rate is usually kept low so good chromosomes obtained from crossover are not lost. If the mutation rate is high (above 0.1), GA performance will approach that of a primitive random search. Figure 7.2 provides an example of mutation.

$$\begin{array}{ccc} & & \textbf{Mutated bit} \\ & & \downarrow \\ 10011110 & \Rightarrow & 10011010 \end{array}$$

Fig. 7.2. Mutation operator

In the natural evolutionary process, selection, crossover, and mutation all occur in the single act of generating offspring. Here we distinguish them clearly to facilitate implementation of and experimentation with GAs.

Based on the aforementioned concepts, a simple genetic algorithm for maximization problems is described next.

Step 1: Initialize a population with randomly generated individuals and evaluate the fitness value of each individual.
Step 2: Perform the following operations:
- (a) Select two members from the population with probabilities proportional to their fitness values.
- (b) Apply crossover with a probability equal to the crossover rate.
- (c) Apply mutation with a probability equal to the mutation rate.
- (d) Repeat (a) to (d) until enough members are generated to form the next generation.

Step 3: Repeat steps 2 and 3 until a stopping criterion is met.

Figure 7.3 is a schematic diagram illustrating how to produce the next generation from the current one.

Lets consider a simple example to illustrate the application of the basic genetic algorithm. Will consider the maximization of the "peaks" function, which is given by the following equation:

$$Z = f(x,y) = 3(1-x)^2 e^{-x2-(y+1)2} - 10(x/5 - x^3 - y^5)e^{-x2-y2} - (1/3)e^{-(x+1)2-y2} \, . \tag{7.1}$$

The surface plot of this function is shown in Fig. 7.4. To use Gas to find the maximum of this function, we first confine the search domain to the square $[-3,3] \times [-3,3]$. We use 8-bit binary coding for each variable. Each generation in our GA implementation contains 30 points or individuals. We use a

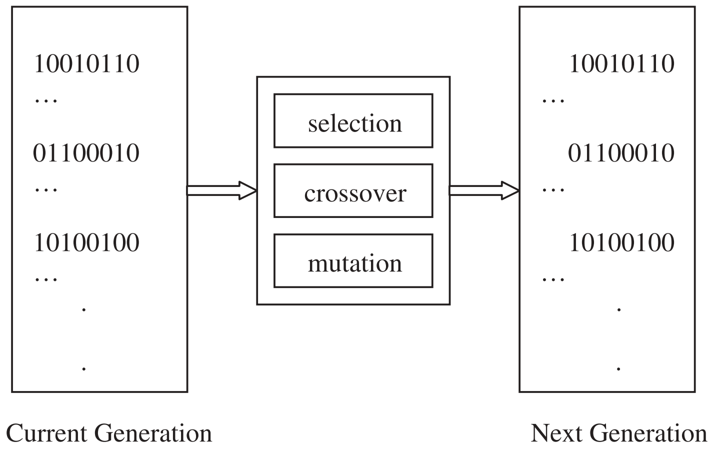


Fig. 7.3. Producing the next generation in GAs

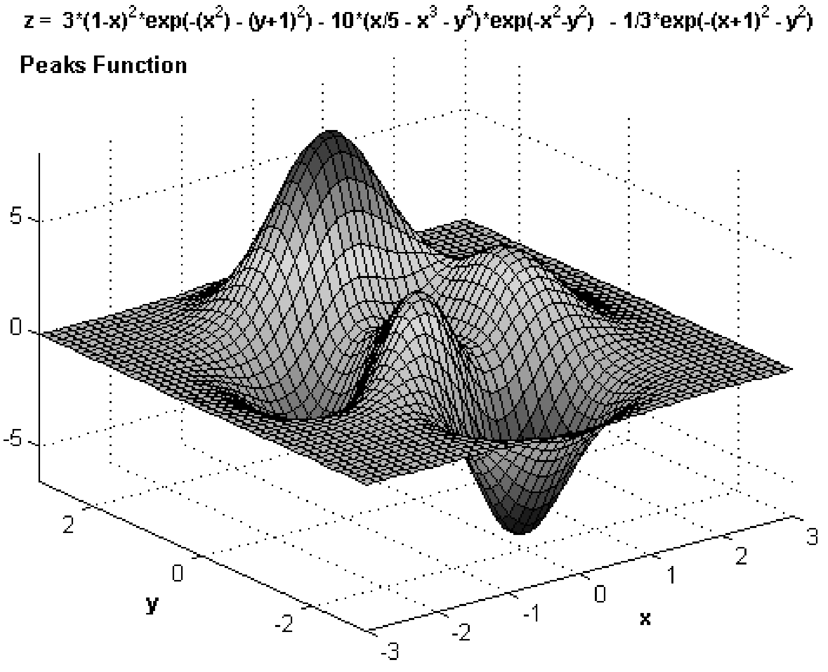


Fig. 7.4. Surface plot of the "peaks" function

simple one-point crossover scheme with crossover rate equal to 0.9. We choose uniform mutation with mutation rate equal to 0.05. Figure 7.5 shows a plot of the best, average, and poorest values of the objective function across 30 generations. Figure 7.6 shows the contour plot of the "peaks" function with the final population distribution after 30 generations. We can appreciate from these figures how the GA is able to find the maximum value of 8.

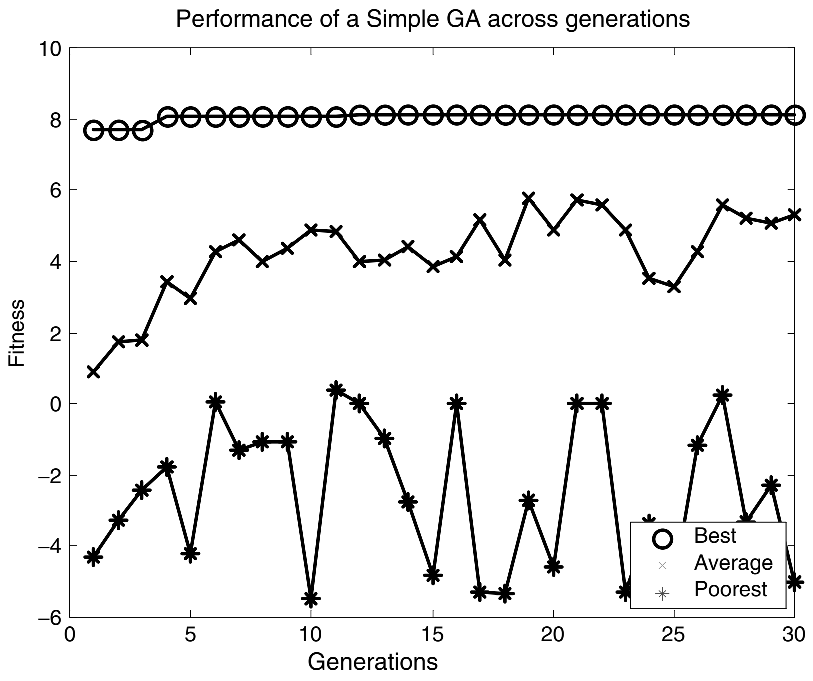


Fig. 7.5. Performance of the genetic algorithm across generations

Now lets consider a more complicated example. We will consider the Rosenbrock's valley, which is a classic optimization problem. The global optimum is inside a long, narrow, parabolic shaped flat valley. To find the valley is trivial, however convergence to the global optimum is difficult and hence this problem has been repeatedly used in assess the performance of optimization algorithms. In this case, the function is given by the following equation

$$f_2 = \sum_{i}^{n-1} 100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2 - 2 \leq x_i \leq 2$$

$$\text{global minimum: } x_i = 1 \quad f(x) = 0\,. \tag{7.2}$$

We show in Fig. 7.7 the plot of Rosenbrock's valley function. We will apply a simple genetic algorithm with the same parameters as in the previous example, i.e. same population size, mutation rate, and crossover rate. The only

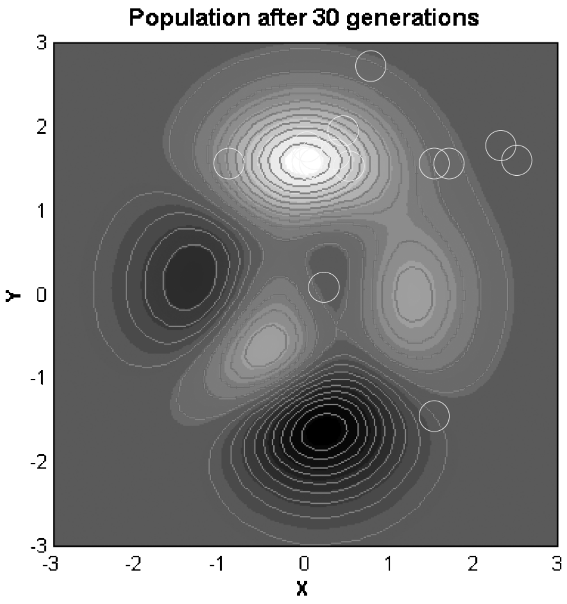


Fig. 7.6. Contour plot of the "peaks" function with the final population

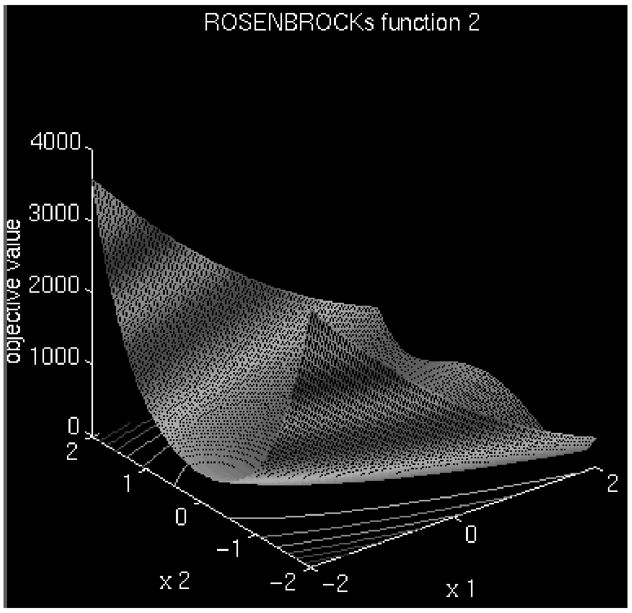


Fig. 7.7. Plot of the Rosenbrock's valley

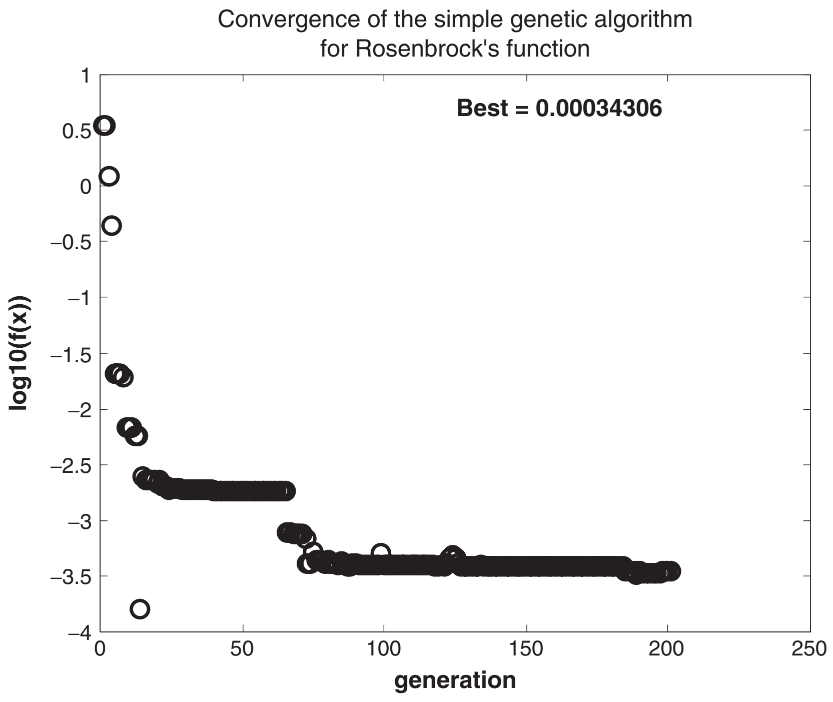


Fig. 7.8. Performance of the genetic algorithm for Rosenbrock's valley

parameter we will change is the maximum number of generations, which will now be of 250. Figure 7.8 shows the performance of the simple genetic algorithm for Rosenbrock's function. We can notice from this figure that the GA is able to converge in about 200 generations to a best value of 0.00034306. We have to mention that the computer program to obtain these results was implemented in the MATLAB programming language.

7.2 Modifications to Genetic Algorithms

The GA mechanism is neither governed by the use of differential equations nor does it behave like a continuous function. However, it possesses the unique ability to search and optimize a solution for complex system, where other mathematical oriented techniques may have failed to compile the necessary design specifications. Due to its evolutionary characteristics, a standard GA may not be flexible enough for a practical application, and an engineering insight is always required whenever a GA is applied. This becomes more apparent where the problem to be tackled is complicated, multi-tasking and conflicting. Therefore, a means of modifying the GA structure is sought in order to meet the design requirements. There are many facets of operational modes that can be introduced.

7.2.1 Chromosome Representation

The problem to be tackled varies from one to the other. The coding of chromosome representation may vary according to the nature of the problem itself. In general, the bit string encoding (Holland, 1975) is the most classic method used by GA researchers because of its simplicity and traceability. The conventional GA operations and theory (scheme theory) are also developed on the basis of this fundamental structure. Hence, this representation is adopted in many applications. However, one minor modification can be suggested in that a Gray code may be used instead of the binary coding. Hollstien (1971) investigated the use of GA for optimizing functions of two variables based on a Gray code representation, and discovered that this works slightly better than the normal binary representation.

More recently, a direct manipulation of real-value chromosomes (Janikow & Michalewicz, 1991; Wright, 1991) raised considerable interest. This representation was introduced especially to deal with real parameter problems. The work currently taking place by Janikow and Michalewicz indicates that the floating-point representation would be faster in computation and more consistent from the basis of run-to-run. At the same time, its performance can be enhanced by special operators to achieve high accuracy (Michalewicz, 1996).

7.2.2 Objective Functions and Fitness

An objective function is a measuring mechanism that is used to evaluate the status of a chromosome. This is a very important link to relate the GA and the system concerned. Since each chromosome is individually going through the same calculations, the range of this value varies from one chromosome to another. To maintain uniformity, the objective value O is mapped into a fitness value with a map Ψ where the domain of F is usually greater than zero.

$$\Psi : O \rightarrow F \tag{7.3}$$

Linear Scaling

The fitness value f_i of chromosome i has a linear relationship with the objective value o_i as

$$f_i = ao_i + b \tag{7.4}$$

where a and b are chosen to enforce the equality of the average objective value and the average fitness value, and cause maximum scaled fitness to be a specified multiple of the average fitness.

This method can reduce the effect of genetic drift by producing an extraordinarily good chromosome. However, it may introduce a negative fitness value, which must be avoided. Hence, the choice of a and b are dependent on the knowledge of the range of the objective values.

Sigma Truncation

This method avoids the negative fitness value and incorporates the problem dependent information into the scaling mechanism. The fitness value f_i of chromosome i is calculated according to

$$f_i = o_i - (\tilde{o} - c\sigma) \tag{7.5}$$

where c is small integer, $\tilde{o}$ is the mean of the objective values, σ is the standard deviation in the population.

To prevent negative values of f, any negative result $f < 0$ is arbitrarily set to zero. Chromosomes whose fitness values are less than c (a small integer from the range 1 and 5) standard deviations from the average fitness value are not selected.

Power Law Scaling

The actual fitness value is taken as a specific power of the objective value, o_i

$$Fi = o_i^k \tag{7.6}$$

where k is in general problem dependent or even varying during the run (Gillies, 1985).

Ranking

There are other methods that can be used such as the Ranking scheme (Baker, 1987). The fitness values do not directly relate to their corresponding objective values, but to the ranks of the objective values.

Using this approach can help the avoidance of premature convergence and speed up the search when the population approaches convergence. On the other hand, it requires additional overheads in the in the GA computation for sorting chromosomes according to their objective values.

7.2.3 Selection Methods

To generate good offspring, a good parent selection mechanism is necessary. This is a process used for determining the number of trials for one particular individual used in reproduction. The chance of selecting one chromosome as a parent should be directly proportional to the number of offspring produced.

Baker (1987) presented three measures of performance of the selection methods: Bias, Spread, and Efficiency. *Bias* defines the absolute difference between individuals in actual and expected probability for selection. Optimal zero bias is achieved when an individual's probability equals its expected number of trials. *Spread* is a range in the possible number of trials that an individual may achieve. If $g(i)$ is the actual number of trials due to each individual i, then the "minimum spread" is the smallest spread that theoretically permits zero bias, i.e.

$$G(i) \in \{\lfloor et(i) \rfloor, \lceil et(i) \rceil\} \tag{7.7}$$

where $et(i)$ is the expected number of trials of individual i, $\lfloor et(i) \rfloor$ is the floor and $\lceil et(i) \rceil$ is the ceiling. Thus the spread of a selection method measures its consistency. *Efficiency* is related to the overall time complexity of the algorithms.

The selection method should thus be achieving a zero bias whilst maintaining a minimum spread and not contributing to an increased time complexity of the GA.

Many selection techniques employ the "roulette wheel mechanism". The basic roulette wheel selection method is a stochastic sampling with replacement (SSR) technique. The segment size and selection probability remain the same throughout the selection phase and the individuals are selected according to the above procedures. SSR tends to give zero bias but potentially inclines to a spread that is unlimited.

Stochastic Sampling with Partial Replacement (SSPR) extends upon SSR by resizing a chromosome's segment if it is selected. Each time a chromosome is selected, the size of its segment is reduced by a certain factor. If the segment size becomes negative, then it is set to zero. This provides an upper bound on the spread of $\lfloor et(i) \rfloor$ but with a zero lower bound and a higher bias. The roulette wheel selection methods can generally be implemented with a time complexity of the order of $NlogN$ where N is the population size.

Stochastic Universal Sampling (SUS) is another single-phase sampling method with minimum spread, zero bias and the time complexity is in the order of N (Baker, 1987). SUS uses an N equally spaced pointer, where N is the number of selections required. The population is shuffled randomly and a single random number in the range $[0, F_{\text{sum}}/N]$ is generated, ptr, where F_{sum} is the sum of the individuals' fitness values. An individual is thus guaranteed to be selected a minimum of $\lfloor et(i) \rfloor$ times and no more than $\lceil et(i) \rceil$, thus achieving minimum spread. In addition, as individuals are selected entirely based on their position in the population, SUS has zero bias.

7.2.4 Genetic Operations

Crossover

Although the one-point crossover method was inspired by biological processes, it has one major drawback in that certain combinations of schema cannot be combined in some situations (Michalewicz, 1996).

A multi-point crossover can be introduced to overcome this problem. As a result, the performance of generating offspring is greatly improved. Another approach is the uniform crossover. This generates offspring from the parents, based on a randomly generated crossover mask. The resulting offspring contain a mixture of genes from each parent. The number of effective crossing points is not fixed, but will be averaged at $L/2$ (where L is the chromosome length).

The preference for using which crossover techniques is still arguable. However, De Jong (1975) concluded that a two-point crossover seemed to be an optimal number for multi-point crossover. Since then, this has been contradicted by Spears and De Jong (1991) as a two-point crossover could perform poorly if the population has largely being converged because of any reduced crossover productivity. A general comment was that each of these crossover operators was particularly useful for some classes of problems and quite poor for others, and that the one-point crossover was considered a "loser" experimentally.

Crossover operations can be directly adopted into the chromosome with real number representation. The only difference would be if the string is composed of a series of real numbers instead of binary numbers.

Mutation

Originally, mutation was designed only for the binary-represented chromosome. To adopt the concept of introducing variations into the chromosome, a random mutation (Michalewicz, 1996) has been designed for a real number chromosome:

$$g = g + \psi(\mu, \sigma) \tag{7.8}$$

where g is the real number gene; ψ is a random function which may be Gaussian or normally distributed; μ is the mean and σ is the variance of the random function.

Operational Rates Settings

The choice of an optimal probability operation rate for crossover and mutation is another controversial debate for both analytical and empirical investigations. The increase of crossover probability would cause the recombination of building blocks to rise, and at the same time, it also increases the disruption of good chromosomes. On the other hand, should the mutation probability increase, this would transform the genetic search into a random search, but would help to reintroduce the lost genetic material.

7.2.5 Parallel Genetic Algorithm

Considering that the GA already possesses an intrinsic, parallelism architecture, there is no extra effort to construct a parallel computational framework. Rather, the GA can be fully exploited in its parallel structure to gain the required speed for practical applications.

There are a number of GA-based parallel methods to enhance the computational speed (Cantú-Paz, 1995). The methods of parallelization can be classified as Global, Migration and Diffusion. These categories reflect different ways in which parallelism can be exploited in the GA as well as the nature of the population structure and recombination mechanisms used.

Global GA

Global GA treats the entire population as a single breeding mechanism. This can be implemented on a shared memory multiprocessor or distributed memory computer. On a shared memory multiprocessor, chromosomes are stored in the shared memory. Each processor accesses the particular assigned chromosome and returns the fitness values without any conflicts. It should be noted that there is some synchronization needed between generation to generation. It is necessary to balance the computational load among the processors using a dynamic scheduling algorithm.

On a distributed memory computer, the population can be stored in one processor to simplify the genetic operators. This is based on the farmer-worker architecture. The farmer processor is responsible for sending chromosomes to the worker processors for the purpose of fitness evaluation. It also collects the results from them, and applies the genetic operators for producing the next generation.

Migration GA

This is another parallel processing method for computing the GA. The migration GA divides the population into a number of sub-populations, each of which is treated as a separate breeding unit under the control of a conventional GA. To encourage the proliferation of good genetic material throughout the whole population, migration between the sub-populations occurs from time to time. The required parameters for successful migration are the “migration rate” and the “migration interval”. The migration rate governs the number of individuals to be migrated. The migration interval affects the frequency of migrations. The values of these parameters are intuitively chosen rather than based on some rigorous scientific analysis. In general, the occurrence of migration is usually set at a predetermined constant interval that is governed by migration intervals. We illustrate the concept of migration in Fig. 7.9.

The migration GA is well suited to parallel implementation on Multiple Instruction Multiple Data (MIMD) machines. The architecture of hypercubes

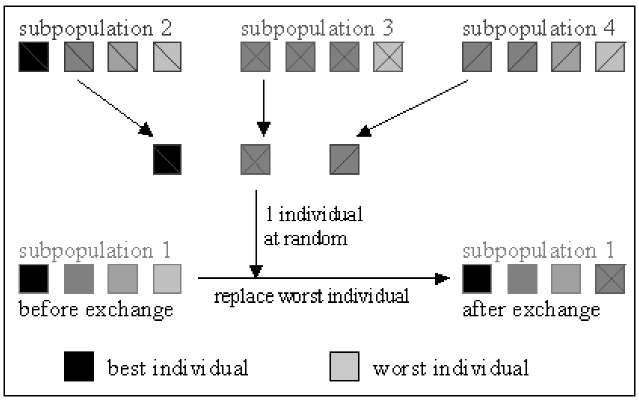


Fig. 7.9. Detailed description of the migration concept

and rings are commonly used for this purpose. Given the range of possible population topologies and migration paths between them, efficient communication networks should thus be possible on most parallel architectures. This applies to small multiprocessor platforms or even the clustering of networked workstations.

Diffusion GA

The Diffusion GA considers the population as a single continuous structure. Each individual is assigned to a geographic location on the population surface and usually placed in a two-dimensional grid. This is because of the topology of the processing element in many massively parallel computers that are constructed in this form. The individuals are allowed to breed with individuals contained in a small local neighborhood. This neighborhood is usually chosen from immediately adjacent individuals on the population surface and is motivated by the practical communication restrictions of parallel computers.

We will illustrate the ideas of parallel genetic algorithms with the well known harvest optimization problem. We will apply a migration GA to solve this optimization problem. The harvest system is a one-dimensional equation of growth with one constraint:

$$x(k+1) = ax(k) - u(k) \; k = 1, \ldots, N, \quad \text{such that } x(0) = x(N) \, . \tag{7.9}$$

The objective function for minimization is therefore defined as:

$$F(u) = -\sum_{k=1}^{N} [u(k)]^{1/2} \, . \tag{7.10}$$

We will apply a multi-population genetic algorithm with migration and real-valued representation of the individuals. We used 20 decision variables, a

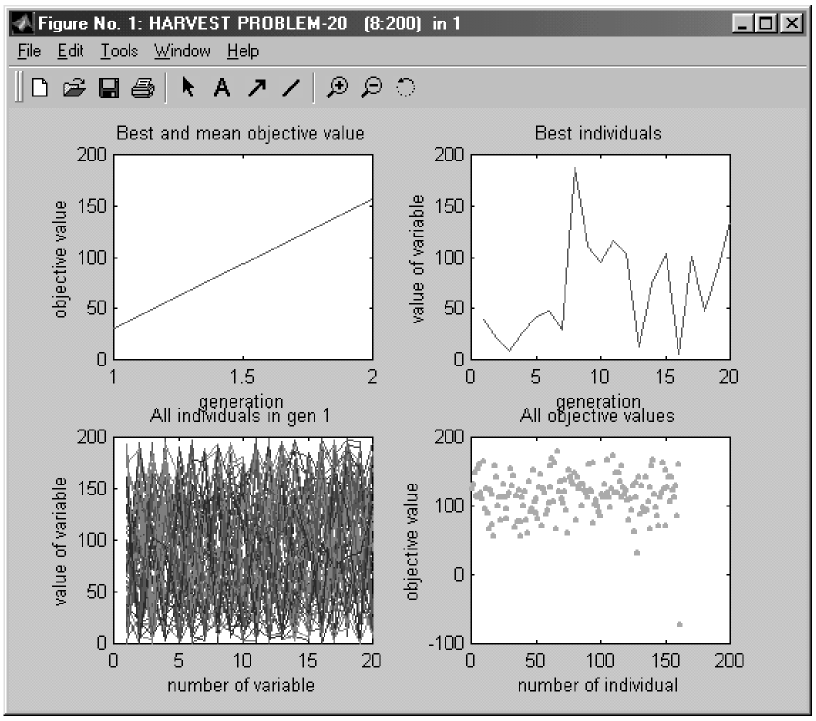


Fig. 7.10. Initial population and subpopulations for the migration GA

crossover rate of 1.0 and a mutation rate of 1/number of variables. We used 8 subpopulations with 20 individuals each, and a migration rate of 0.2. The maximum number of generations was specified at 200. We show in Fig. 7.10 the initial population of individuals. In Fig. 7.11 we show the simulation results after 200 generations of the GA.

7.3 Applications of Genetic Algorithms

The genetic algorithms described in the previous sections are very simple, but variations of these algorithms have been used in a large number of scientific and engineering problems and models (Mitchell, 1996). Some examples follow.

- Optimization: genetic algorithms have been used in a wide variety of optimization tasks, including numerical optimization and such combinatorial optimization problems as circuit layout and job-shop scheduling.
- Automatic Programming: genetic algorithms have been used to evolve computer programs for specific tasks, and to design other computational structures such as cellular automata and sorting networks.

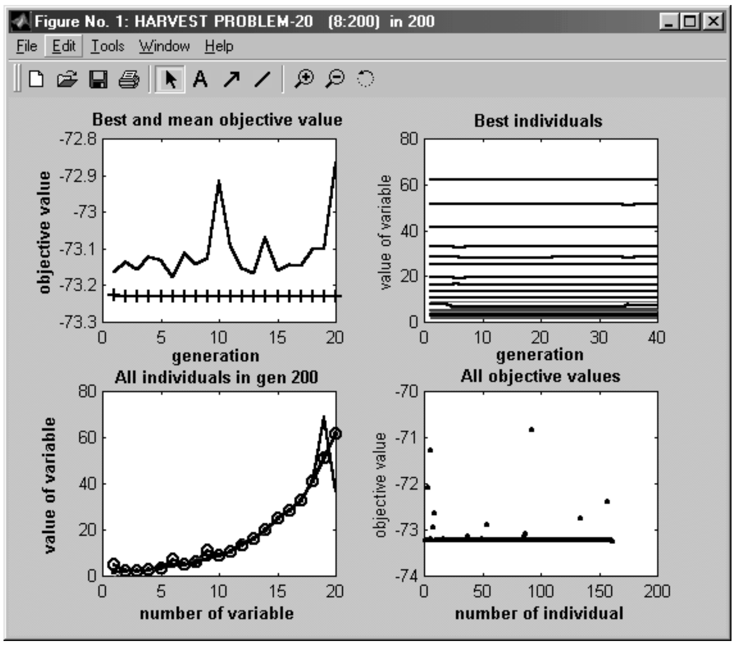


Fig. 7.11. Final population for the migration GA after 200 generations

- Machine Learning: genetic algorithms have been used for many machine learning applications, including classification and prediction tasks, such as the prediction of weather or protein structure. Genetic algorithms have also been used to evolve aspects of particular machine learning systems, such as weights for neural networks, rules for learning classifier systems or symbolic production systems, and sensors for robots.
- Economics: genetic algorithms have been used to model processes of innovation, the development of bidding strategies, and the emergence of economic markets.
- Immune Systems: genetic algorithms have been used to model various aspects of natural immune systems, including somatic mutation during an individual's lifetime and the discovery of multi-gene families during evolutionary time.
- Ecology: genetic algorithms have used to model ecological phenomena such as biological arms races, host-parasite co-evolution, symbiosis, and resource flow.
- Social Systems: genetic algorithms have been used to study evolutionary aspects of social systems, such as the evolution of social behavior in insect colonies, and, more generally, the evolution of cooperation and communication in multi-agent systems.

This list is by no means exhaustive, but it gives the flavor of the kinds of things genetic algorithms have been used for, both in problem solving and in scientific contexts. Because of their success in these and other areas, interest in genetic algorithms has been growing rapidly in the last several years among researchers in many disciplines.

We will describe bellow the application of genetic algorithms to the problem of evolving neural networks, which is a very important problem in designing the particular neural network for a problem.

7.3.1 Evolving Neural Networks

Neural Networks are biologically motivated approaches to machine learning, inspired by ideas from neuroscience. Recently, some efforts have been made to use genetic algorithms to evolve aspects of neural networks (Mitchell, 1996).

In its simplest feedforward form, a neural network is a collection of connected neurons in which the connections are weighted, usually with real-valued weights. The network is presented with an activation pattern on its input units, such as a set of numbers representing features of an image to be classified. Activation spreads in a forward direction from the input units through one or more layers of middle units to the output units over the weighted connections. This process is meant to roughly mimic the way activation spreads through networks of neurons in the brain. In a feedforward network, activation spreads only in a forward direction, from the input layer through the hidden layers to the output layer. Many people have also experimented with "recurrent" networks, in which there are feedback connections between layers.

In most applications, the neural network learns a correct mapping between input and output patterns via a learning algorithm. Typically the weights are initially set to small random values. Then a set of training inputs is presented sequentially to the network. In the backpropagation learning procedure, after each input has propagated through the network and an output has been produced, a "teacher" compares the activation value at each output unit with the correct values, and the weights in the network are adjusted in order to reduce the difference between the network's output and the correct output. This type of procedure is known as "supervised learning", since a teacher supervises the learning by providing correct output values to guide the learning process.

There are many ways to apply genetic algorithms to neural networks. Some aspects that can be evolved are the weights in a fixed network, the network architecture (i.e., the number of neurons and their interconnections can change), and the learning rule used by the network.

Evolving Weights in a Fixed Network

David Montana and Lawrence Davis (1989) took the first approach of evolving the weights in a fixed network. That is, Montana and Davis were using the genetic algorithm instead of backpropagation as a way of finding a good set of weights for a fixed set of connections. Several problems associated with the backpropagation algorithm (e.g., the tendency to get stuck at local minima, or the unavailability of a "teacher" to supervise learning in some tasks) often make it desirable to find alternative weight training schemes.

Montana and Davis were interested in using neural networks to classify underwater sonic "lofargrams" (similar to spectrograms) into two classes: "interesting" and "not interesting". The networks were to be trained from a database containing lofargrams and classifications made by experts as to whether or not a given lofargram is "interesting". Each network had four input units, representing four parameters used by an expert system that performed the same classification. Each network had one output unit and two layers of hidden units (the first with seven units and the second with ten units). The networks were fully connected feedforward networks. In total there were 108 weighted connections between units. In addition, there were 18 weighted connections between the non-input units and a "threshold unit" whose outgoing links implemented the thresholding for each of the non-input units, for a total of 126 weights to evolve.

The genetic algorithm was used as follows. Each chromosome was a list of 126 weights. Figure 7.12 shows (for a much smaller network) how the encoding was done: the weights were read off the network in a fixed order (from left to right and from top to bottom) and placed in a list. Notice that each "gene" in the chromosome is a real number rather than a bit. To calculate the fitness of a given chromosome, the weights in the chromosome were assigned to the links in the corresponding network, the network was run on the training set (here 236 examples from the database), and the sum of the squares of the errors was returned. Here, an "error" was the difference between the desired output value and the actual output value. Low error meant high fitness in this case.

An initial population of 50 weights vectors was chosen randomly, with each weight being between −1.0 and +1.0. Montana and Davis tried a number of different genetic operators in various experiments. The mutation and crossover operators they used for their comparison of the genetic algorithm with backpropagation are illustrated in Figs. 7.13 and 7.14.

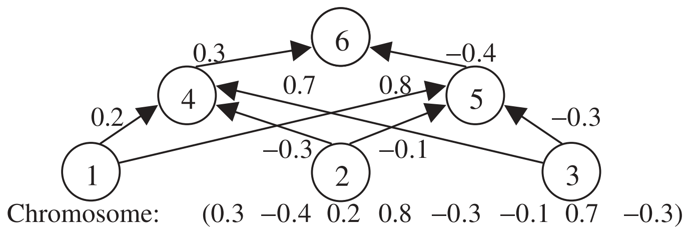


Fig. 7.12. Encoding of network weights for the genetic algorithm

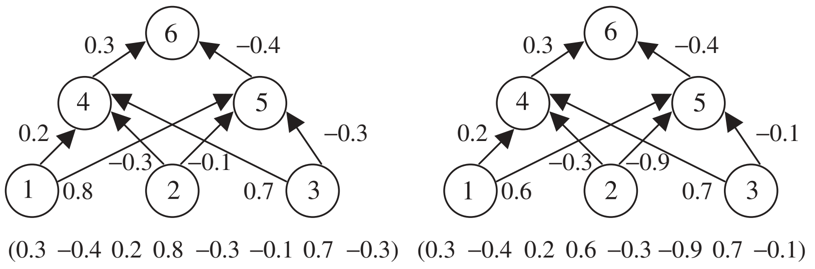


Fig. 7.13. Illustration of the mutation method. The weights on incoming links to unit 5 are mutated

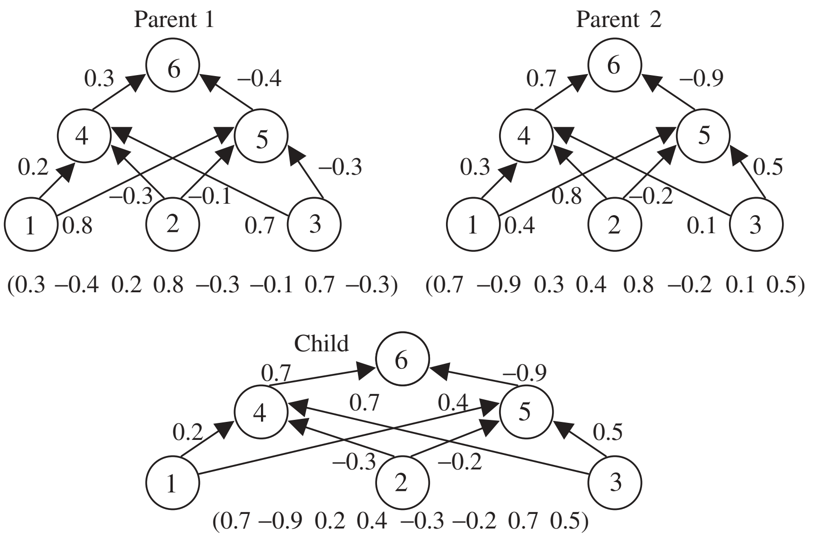


Fig. 7.14. Illustration of the crossover method. In the child network shown here, the incoming links to unit 4 come from parent 1 and the incoming links 5 and 6 come from parent 2

The mutation operator selects n non-input units, and for each incoming link to those units, adds a random value between −1.0 and +1.0 to the weight on the link. The crossover operator takes two parent weight vectors, and for each non-input unit in the offspring vector, selects one of the parents at random and copies the weights on the incoming links from that parent to the offspring. Notice that only one offspring is created.

The performance of a genetic algorithm using these operators was compared with the performance of a backpropagation algorithm. The genetic algorithm had a population of 50 weight vectors, and a rank selection method was used. The genetic algorithm was allowed to run for 200 generations. The backpropagation algorithm was allowed to run for 5000 iterations, where one iteration is a complete epoch (a complete pass through the training data). Montana and Davis found that the genetic algorithm significantly outperforms backpropagation on this task, obtaining better weight vectors more quickly.

This experiment shows that in some situations the genetic algorithm is a better training method for neural networks than simple backpropagation. This does not mean that the genetic algorithm will outperform backpropagation in all cases. It is also possible that enhancements of backpropagation might help it overcome some of the problems that prevented it from performing as well as the genetic algorithm in this experiment.

Evolving Network Architectures

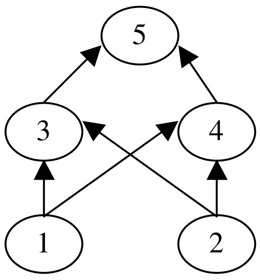
Neural network researchers know all too well that the particular architecture chosen can determine the success or failure of the application, so they would like very much to be able to automatically optimize the procedure of designing an architecture for a particular application. Many believe that genetic algorithms are well suited for this task (Mitchell, 1996). There have been several efforts along these lines, most of which fall into one of two categories: direct encoding and grammatical encoding. Under direct encoding a network architecture is directly encoded into a genetic algorithm chromosome. Under grammatical encoding, the genetic algorithm does not evolve network architectures; rather, it evolves grammars that can be used to develop network architectures.

Direct Encoding

The method of direct encoding is illustrated in work done by Geoffrey Miller, Peter Todd, and Shailesh Hedge (1989), who restricted their initial project to feedforward networks with a fixed number of units for which the genetic algorithm was used to evolve the connection topology. As is shown in Fig. 7.15, the connection topology was represented by a $N \times N$ matrix (5×5 in Fig. 7.15) in which each entry encodes the type of connection from the "from unit" to the "to unit". The entries in the connectivity matrix were either "0" (meaning no connection) or "L" (meaning a "learnable" connection). Figure 7.15 also shows how the connectivity matrix was transformed into a chromosome for the genetic algorithm ("0" corresponds to 0 and "L" to 1) and how the bit string was decoded into a network. Connections that were specified to be learnable were initialized with small random weights.

Miller, Todd, and Hedge used a simple fitness-proportionate selection method and mutation (bits in the string were flipped with some low probability). Their crossover operator randomly chose a row index and swapped the corresponding rows between the two parents to create two offspring. The intuition behind that operator was similar to that behind Montana and Davis's crossover operator-each row represented all the incoming connections to a single unit, and this set was thought to be a functional building block of the

From unit		1	2	3	4	5
To unit	1	0	0	0	0	0
	2	0	0	0	0	0
	3	L	L	0	0	0
	4	L	L	0	0	0
	5	0	0	L	L	0



Chromosome: 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 0 0 0 0 0 1 1 0

Fig. 7.15. Illustration of Miller, Todd, and Hedge's representation scheme

network. The fitness of a chromosome was calculated in the same way as in Montana and Davis's project: for a given problem, the network was trained on a training set for a certain number of epochs, using backpropagation to modify the weights. The fitness of the chromosome was the sum of the squares of the errors on the training set at the last epoch. Again, low error translated to high fitness. Miller, Todd, and Hedge tried their genetic algorithm on several problems with very good results. The problems were relatively easy for multilayer neural networks to learn to solve under backpropagation. The networks had different number of units for different tasks; the goal was to see if the genetic algorithm could discover a good connection topology for each task. For each run the population size was 50, the crossover rate was 0.6, and the mutation rate was 0.005. In all cases, the genetic algorithm was easily able to find networks that readily learned to map inputs to outputs over the training set with little error. However, the tasks were too easy to be a rigorous test of this method-it remains to be seen if this method can scale up to more complex tasks that require much larger networks with many more interconnections.

Grammatical Encoding

The method of grammatical encoding can be illustrated by the work of Hiroaki Kitano (1990), who points out that direct encoding approaches become increasingly difficult to use as the size of the desired network increases. As the network's size grows, the size of the required chromosome increases quickly, which leads to problems both in performance and in efficiency. In addition, since direct encoding methods explicitly represent each connection in the network, repeated or nested structures cannot be represented efficiently, even though these are common for some problems.

The solution pursued by Kitano and others is to encode networks as grammars; the genetic algorithm evolves the grammars, but the fitness is tested only after a "development" step in which a network develops from the grammar. A grammar is a set of rules that can be applied to produce a set of structures (e.g., sentences in a natural language, programs in a computer language, neural network architectures).

Kitano applied this general idea to the development of neural networks using a type of grammar called a "graph-generation grammar", a simple example of which is given in Fig. 7.16(a). Here the right-hand side of each rule is a 2×2 matrix rather than a one-dimensional string. Each lower-case letter from a through p represents one of the 16 possible 2×2 arrays of ones and zeros. There is only one structure that can be formed from this grammar: the 8×8 matrix shown in Fig. 7.16(b). This matrix can be interpreted as a connection matrix for a neural network: a 1 in row i and column i means that unit i is present in the network and a 1 in row i and column, $i \neq j$, means that there is connection from unit i to unit j. The result is the network shown in Fig. 7.16(c) which, with appropriate weights, computes the Boolean function XOR.

Kitano's goal was to have a genetic algorithm evolve such grammars. Figure 7.17 illustrates a chromosome encoding the grammar given in Fig. 7.16(a). The chromosome is divided up into separate rules, each of which consists of five elements. The first element is the left-hand side of the rule; the second through fifth elements are the four symbols in the matrix on the right-hand side of the rule. The possible values for each element are the symbols $A - Z$ and $a - p$. The first element of the chromosome is fixed to be the start symbol, S; at least one rule taking S into a 2×2 matrix is necessary to get started in building a network from a grammar.

The fitness of a grammar was calculated by constructing a network from the grammar, using backpropagation with a set of training inputs to train the resulting network to perform a simple task, and then, after training, measuring the sum of the squares of the errors made by the network on either the training set or a separate test set. The genetic algorithm used fitness-proportionate selection, multi-point crossover, and mutation. A mutation consisted of replacing one symbol in the chromosome with a randomly chosen symbol from the $A - Z$ and $a - p$ alphabets. Kitano used what he called "adaptive mutation": the probability of mutation of an offspring depended on the Hamming distance (number of mismatches) between the two parents. High distance resulted in low mutation, and vice versa. In this way, the genetic algorithm tended to respond to loss of diversity in the population by selectively raising the mutation rate.

Kitano (1990) performed a series of experiments on evolving networks for simple "encoder/decoder" problems to compare the grammatical and direct encoding approaches. He found that, on these relatively simple problems, the performance of a genetic algorithm using the grammatical encoding method consistently surpassed that of a genetic algorithm using the direct encoding method, both in the correctness of the resulting neural networks and in the speed with which they were found by the genetic algorithm. In the

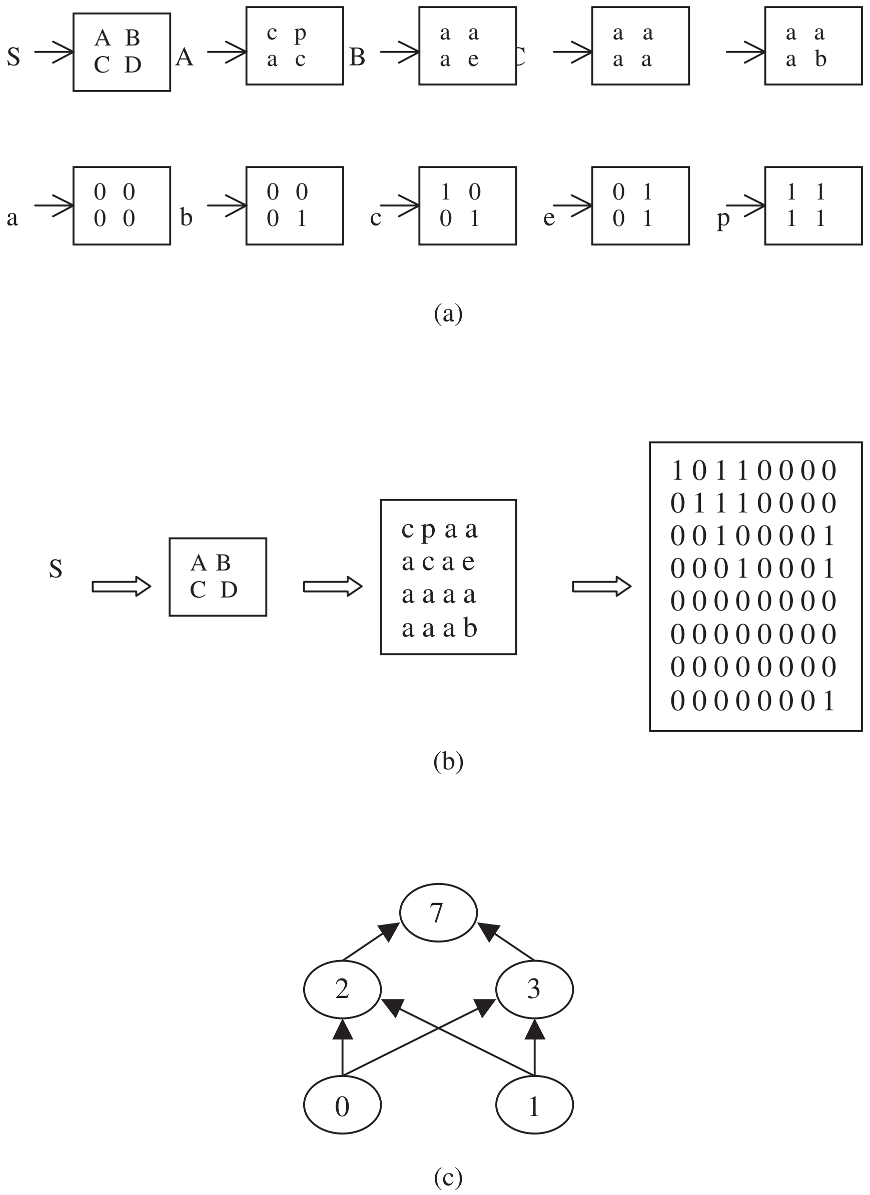


Fig. 7.16. Illustration of Kitano's graph generation grammar for the XOR problem. (**a**) Grammatical rules. (**b**) A connection matrix is produced from the grammar. (**c**) The resulting network

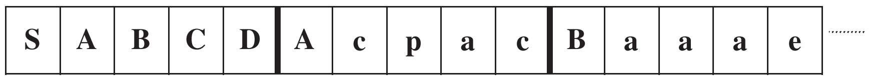


Fig. 7.17. Illustration of a chromosome encoding a grammar

grammatical encoding runs, the genetic algorithm found networks with lower error rate, and found the best networks more quickly, than in direct encoding runs. Kitano also discovered that the performance of the genetic algorithm scaled much better with network size when grammatical encoding was used.

What accounts for the grammatical encoding method's apparent superiority? Kitano argues that the grammatical encoding method can easily create "regular", repeated patterns of connectivity, and that this is a result of the repeated patterns that naturally come from repeatedly applying grammatical rules. We would expect grammatical encoding approaches, to perform well on problems requiring this kind of regularity. Grammatical encoding also has the advantage of requiring shorter chromosomes, since the genetic algorithm works on the instructions for building the network (the grammar) rather that on the network structure itself. For complex networks, the later could be huge and intractable for any search algorithm.

Evolving a Learning Rule

David Chalmers (1990) took the idea of applying genetic algorithms to neural networks in a different direction: he used genetic algorithms to evolve a good learning rule for neural networks. Chalmers limited his initial study to fully connected feedforward networks with input and output layers only, no hidden layers. In general a learning rule is used during the training procedure for modifying network weights in response to the network's performance on the training data. At each training cycle, one training pair is given to the network, which then produces an output. At this point the learning rule is invoked to modify weights. A learning rule for a single layer, fully connected feedforward network might use the following local information for a given training cycle to modify the weight on the link from input unit i to output unit j:

a_i: the activation of input i
o_j: the activation of output unit j
t_j: the training signal on output unit j
w_{ij}: the current weight on the link from i to j

The change to make in the weight w_{ij} is a function of these values:
$\Delta w_{ij} = f(a_i, o_j, t_j, w_{ij})$.

The chromosomes in the genetic algorithm population encoded such functions.

Chalmers made the assumption that the learning rule should be a linear function of these variables and all their pairwise products. That is, the general form of the learning rule was

$$\begin{aligned}\Delta w_{ij} = k_0(&k_1 w_{ij} + k_2 a_i + k_3 o_j + k_4 t_j + k_5 w_{ij} a_i + k_6 w_{ij} o_j + k_7 w_{ij} t_j \\ &+ k_8 a_i o_j + k_9 a_i t_j + k_{10} o_j t_j) \ .\end{aligned}$$

The $k_m(1 < m < 10)$ are constant coefficients, and k_0 is a scale parameter that affects how much the weights can change on any cycle. Chalmer's assumption about the form of the learning rule came in part from the fact that a known good learning rule for such networks is the "delta rule". One goal of Chalmer's work was to see if the genetic algorithm could evolve a rule that performs as well as the delta rule.

The task of the genetic algorithm was to evolve values for the km's. The chromosome encoding for the set of km's is illustrated in Fig. 7.18. The scale parameter k_0 is encoded as five bits, with the zeroth bit encoding the sign (1 encoding + and 0 encoding −) and the first through fourth bits encoding an integer $n : k_0 = 0$ if $n = 0$; otherwise $|k_0| = 2^{n-9}$. Thus k_0 can take on the values $0, + -1/256, + -1/128, \ldots, + -32, + -64$. The other coefficients km are encoded by three bits each, with the zeroth bit encoding the sign and the first and second bits encoding an integer n. For $i = 1, \ldots, 10, km = 0$ if $n = 0$; otherwise $|k_m| = 2^{n-1}$.

The fitness of each chromosome (learning rule) was determined as follows. A subset of 20 mappings was selected from the full set of 30 linear separable mappings. For each mapping, 12 training examples were selected. For each of these mappings, a network was created with the appropriate number of input

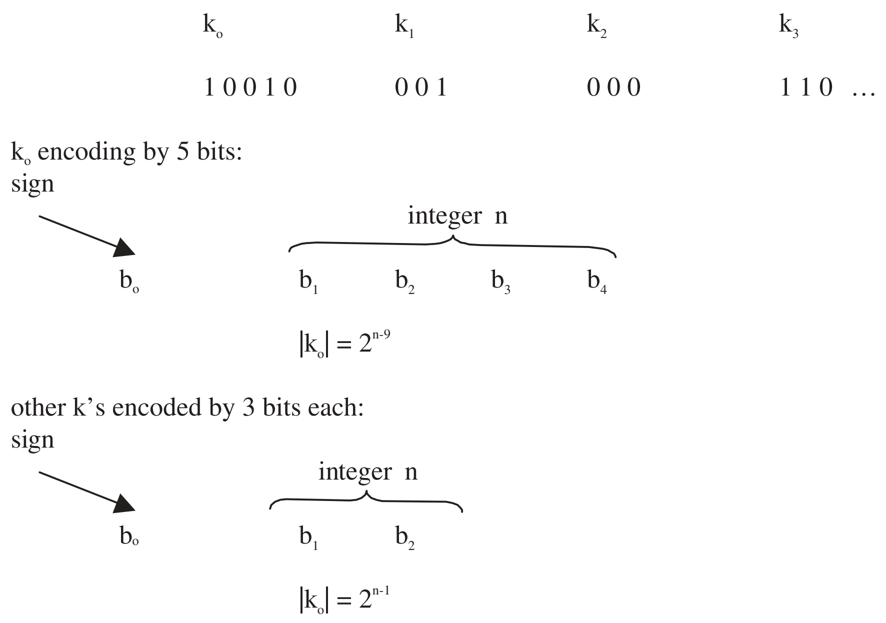


Fig. 7.18. Illustration of the method for encoding the km's

units for the given mapping. The network's weights were initialized randomly. The network was run on the training set for some number of epochs (typically 10), using the learning rule specified by the chromosome. The performance of the learning rule on a given mapping was a function of the network's error on the training set, with low error meaning high performance. The overall fitness of the learning rule was a function of the average error of 20 networks over the chosen subset of 20 mappings. This fitness was then transformed to be a percentage, where a high percentage meant high fitness.

Using this fitness measure, the genetic algorithm was run on a population of 40 learning rules, with two-point crossover and standard mutation. The crossover rate was 0.8 and the mutation rate was 0.01. Typically, over 1000 generations, the fitness of the best learning rules in the population rose from between 40% and 60% in the initial generation to between 80% and 98%. The fitness of the delta rule is around 98%, and on one out of a total of ten run the genetic algorithm discovered this rule. On three of the ten runs, the genetic algorithm discovered slight variations of this rule with lower fitness.

These results show that, given a somewhat constrained representation, the genetic algorithm was able to evolve a successful learning rule for simple single layer networks. The extent to which this method can find learning rules for more complex networks remains an open question, but these results are a first step in that direction. Chalmers suggested that it is unlikely that evolutionary methods will discover learning methods that are more powerful than backpropagation, but he speculated that genetic algorithms might be a powerful method for discovering learning rules for unsupervised learning paradigms or for new classes of network architectures.

Example of Architecture Optimization for a Modular Neural Network

We show the feasibility of the proposed approach of using genetic algorithms for architecture optimization with a problem of system identification. We consider the non-linear function given by the following equation:

$$y = 2e^{(-0.2X)}|\sin(x)| \tag{7.11}$$

We will use a modular neural network for system identification, as in Chap. 6, but now the genetic algorithm will automatically optimize the number of neurons in each of the modules. We will again use as range for the function the [0, 9.45] interval with a 0.01 step size, which gives us 945 points. The data is divided in the three modules as follows:

Module 1: from 0 to 3.15
Module 2: from 3.15 to 6.30
Module 3: from 6.30 to 9.45

The idea behind this data partition is that learning will be easier in each of the modules, i.e. a simple NN can learn more easily the behavior of the

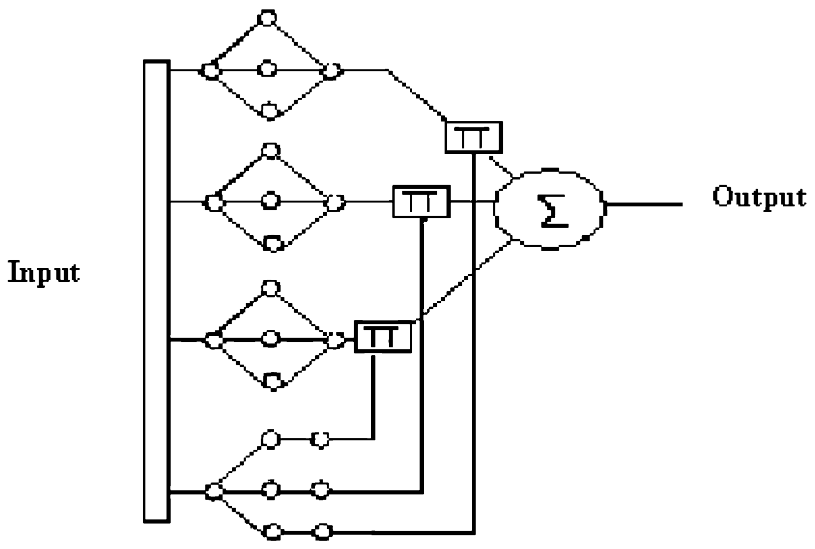


Fig. 7.19. General architecture of the modular neural network

function in one of the regions. We used three-layer feed-forward NNs for each of the modules with the Levenberg-Marquardt training algorithm. We show in Fig. 7.19 the general architecture of the MNN used in this paper.

Regarding the genetic algorithm for MNN evolution, we used a hierarchical chromosome for representing the relevant information of the network. First, we have the bits for representing the number of layers of each of the three modules and then we have the bits for representing the number of nodes of each layer. This scheme basically gives us the chromosome shown in Fig. 7.20.

CM1	M2	CM3	NC1M1	NC2M1	NC3M1	NC1M2	NC2M2	NC3M2	NC1M3	NC1M3	NC1M3
3BITS	3BITS	3BITS	8BITS	8BITS	8BITS	8BITS	8BITS	8BITS	8BITS	8BITS	8BITS

Fig. 7.20. Basic structure of the chromosome containing the information of the MNN

The fitness function used in this work combines the information of the error objective and also the information about the number of nodes as a second objective. This is shown in the following equation.

$$f(z) = \left(\frac{1}{\alpha * \text{Ranking}(ObjV1) + \beta * ObjV2} \right) * 10 \tag{7.12}$$

The first objective is basically the average sum of squared of errors as calculated by the predicted outputs of the MNN compared with real values of the function. This is given by the following equation.

$$f_1 = \frac{1}{N}\sum_{i=1}^{N}(Y_i - y_i) \tag{7.13}$$

The parameters of the genetic algorithm are as follows:

Type of crossover operator: Two-point crossover
Crossover rate: 0.8
Type of mutation operator: Binary mutation
Mutation rate: 0.05
Population size per generation: 10
Total number of generations: 100

We show in Fig. 7.21 the topology of the final evolved modules of the neural network for the problem of function identification. As we can appreciate, from this figure, module 2 is the smallest one and module 3 is the largest one. The result of MNN evolution is a particular architecture with different size of the modules (neural networks).

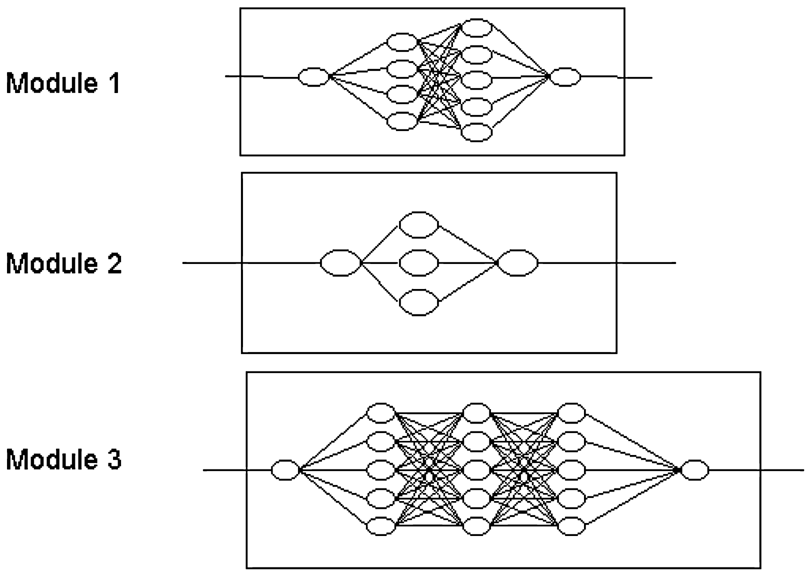


Fig. 7.21. Topology of the final evolved MNN for function identification

The MNN architecture shown in Fig. 7.21 is the best one for the specific problem of system identification. It is worthwhile noting that this network topology is difficult to design manually, for this reason the HGA approach is a good choice for neural network design and optimization. Finally, we show in Fig. 7.22 the evolution of the HGA for MNN topology optimization. From this figure, we can notice that the evolutionary approach is achieving the goal of MNN design.

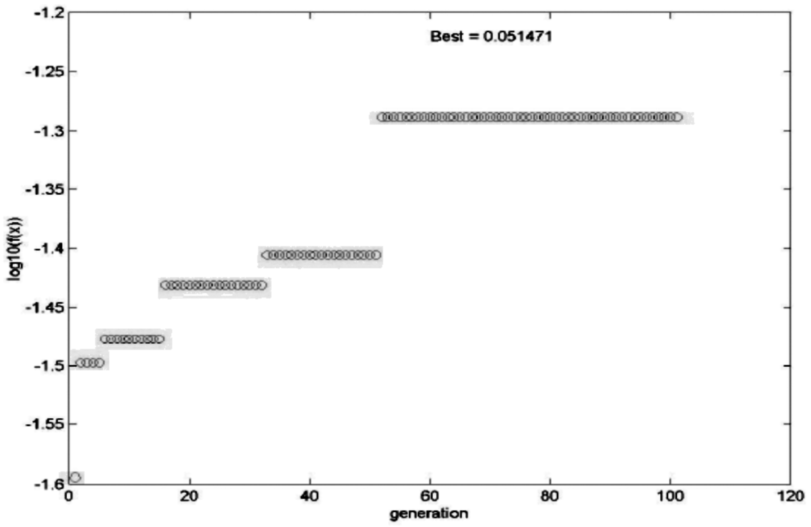


Fig. 7.22. Plot of the HGA performance for MNN evolution

We described in section our hierarchical genetic algorithm approach for modular neural network topology design and optimization. The proposed approach was illustrated with a specific problem of function identification. The best MNN is obtained by evolving the modules (single NNs) according to the error of identification and also the complexity of the modules. The results for the problem of function identification are very good and show the feasibility of the HGA approach for MNN topology optimization.

7.3.2 Evolving Fuzzy Systems

Ever since the very first introduction of the fundamental concept of fuzzy logic by Zadeh in 1973, its use in engineering disciplines has been widely studied. Its main attraction undoubtedly lies in the unique characteristics that fuzzy logic systems possess. They are capable of handling complex, non-linear dynamic systems using simple solutions. Very often, fuzzy systems provide a better performance than conventional non-fuzzy approaches with less development cost.

However, to obtain an optimal set of fuzzy membership functions and rules is not an easy task. It requires time, experience, and skills of the operator for the tedious fuzzy tuning exercise. In principle, there is no general rule or method for the fuzzy logic set-up. Recently, many researchers have considered a number of intelligent techniques for the task of tuning the fuzzy set.

Here, another innovative scheme is described (Man, Tang & Kwong, 1999). This approach has the ability to reach an optimal set of membership functions and rules without a known overall fuzzy set topology. The conceptual idea of

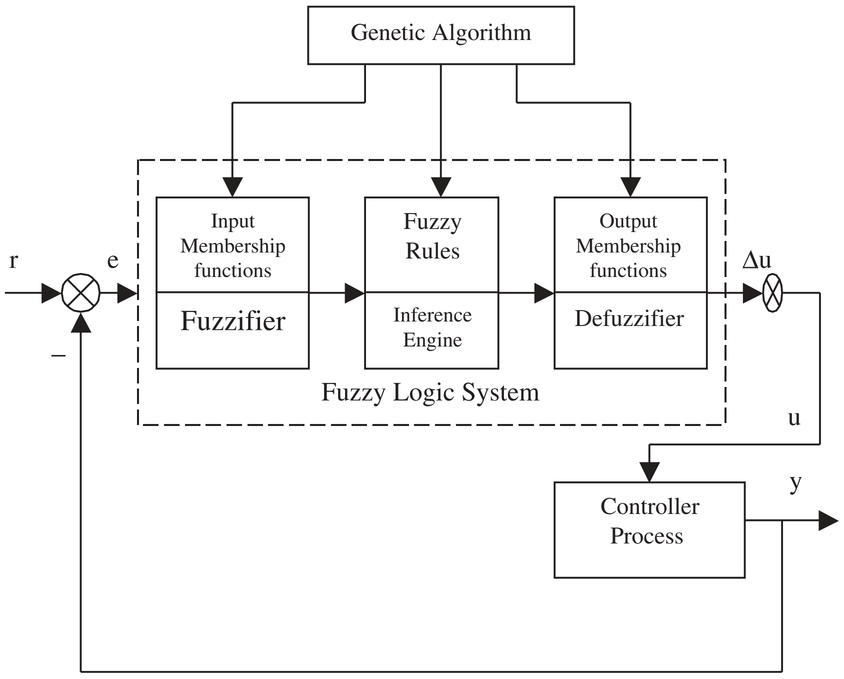


Fig. 7.23. Genetic algorithm for a fuzzy control system

this approach is to have an automatic and intelligent scheme to tune the membership functions and rules, in which the conventional closed loop fuzzy control strategy remains unchanged, as indicated in Fig. 7.23.

In this case, the chromosome of a particular is shown in Fig. 7.24. The chromosome consists of two types of genes, the control genes and parameter genes. The control genes, in the form of bits, determine the membership function activation, whereas the parameter genes are in the form of real numbers to represent the membership functions.

To obtain a complete design for the fuzzy control system, an appropriate set of fuzzy rules is required to ensure system performance. At this point it should be stressed that the introduction of the control genes is done to govern the number of fuzzy subsets in the system.

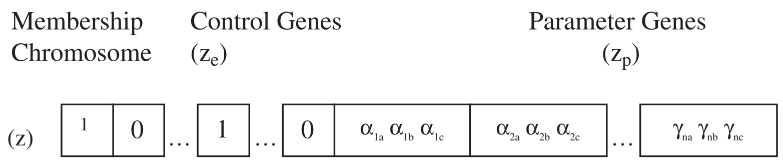

Membership Chromosome
Control Genes
(z_e)
Parameter Genes
(z_p)
(z)
1
0
…
1
…
0
α_{1a} α_{1b} α_{1c}
α_{2a} α_{2b} α_{2c}
…
γ_{na} γ_{nb} γ_{nc}

Fig. 7.24. Chromosome structure for the fuzzy system

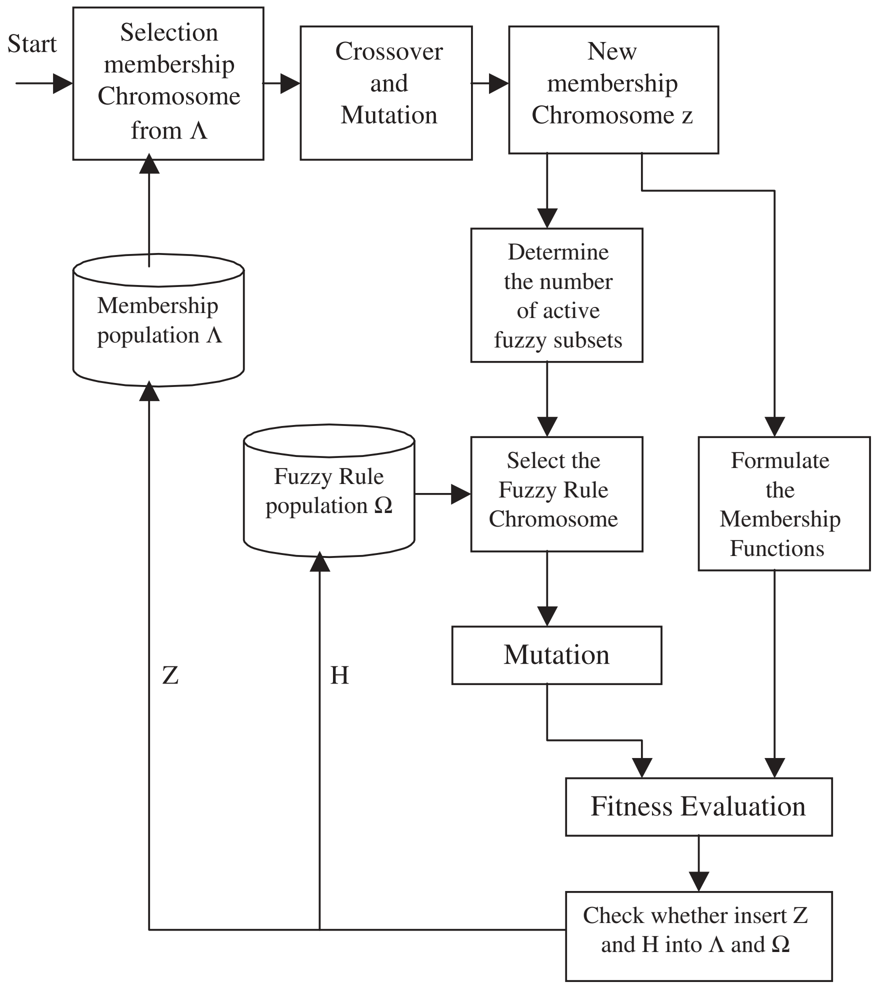

Start
Selection membership Chromosome from Λ
Crossover and Mutation
New membership Chromosome z
Determine the number of active fuzzy subsets
Membership population Λ
Fuzzy Rule population Ω
Select the Fuzzy Rule Chromosome
Formulate the Membership Functions
Mutation
Z
H
Fitness Evaluation
Check whether insert Z and H into Λ and Ω

Fig. 7.25. Genetic cycle for fuzzy system optimization

Once the formulation of the chromosome has been set for the fuzzy membership functions and rules, the genetic operation cycle can be performed. This cycle of operation for the fuzzy control system optimization using a genetic algorithm is illustrated in Fig. 7.25.

There are two population pools, one for storing the membership chromosomes and the other for storing the fuzzy rule chromosomes. We can see this, in Fig. 7.26, as the membership population and fuzzy rule population, respectively. Considering that there are various types of gene structure, a number of different genetic operations can be used. For the crossover operation, a one point crossover is applied separately for both the control and parameter genes of the membership chromosomes within certain operation rates. There is no

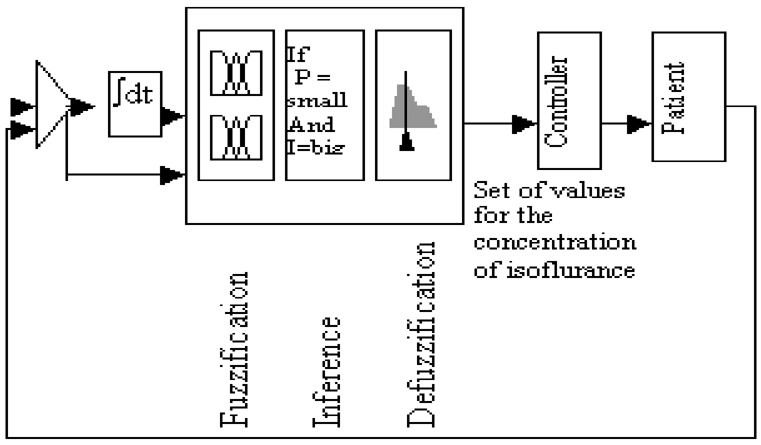


Fig. 7.26. Architecture of the fuzzy control system

crossover operation for fuzzy rule chromosomes since only one suitable rule set can be assisted.

Bit mutation is applied for the control genes of the membership chromosome. Each bit of the control gene is flipped if a probability test is satisfied (a randomly generated number is smaller than a predefined rate). As for the parameter genes, which are real number represented, random mutation is applied.

The complete genetic cycle continues until some termination criteria, for example, meeting the design specification or number of generation reaching a predefined value, are fulfilled.

Application to the Optimization of a Fuzzy Controller

We describe in this section the application of a Hierarchical Genetic Algorithm (HGA) for fuzzy system optimization (Man et al., 1999). In particular, we consider the problem of finding the optimal set of rules and membership functions for a specific application (Yen & Langari, 1999). The HGA is used to search for this optimal set of rules and membership functions, according to the data about the problem. We consider, as an illustration, the case of a fuzzy system for intelligent control.

Fuzzy systems are capable of handling complex, non-linear and sometimes mathematically intangible dynamic systems using simple solutions (Jang et al., 1997). Very often, fuzzy systems may provide a better performance than conventional non-fuzzy approaches with less development cost (Procyk & Mamdani, 1979). However, to obtain an optimal set of fuzzy membership functions and rules is not an easy task. It requires time, experience and skills of the designer for the tedious fuzzy tuning exercise. In principle, there is no general rule or method for the fuzzy logic set-up, although a heuristic and iterative procedure for modifying the membership functions to improve performance has been proposed. Recently, many researchers have considered a number of intelligent schemes for the task of tuning the fuzzy system. The noticeable Neural Network (NN) approach (Jang & Sun, 1995) and the Genetic Algorithm (GA) approach (Homaifar & McCormick, 1995) to optimize either the membership functions or rules, have become a trend for fuzzy logic system development.

The HGA approach differs from the other techniques in that it has the ability to reach an optimal set of membership functions and rules without a known fuzzy system topology (Tang et al., 1998). During the optimization phase, the membership functions need not be fixed. Throughout the genetic operations (Holland, 1975), a reduced fuzzy system including the number of membership functions and fuzzy rules will be generated. The HGA approach has a number of advantages:

(1) An optimal and the least number of membership functions and rules are obtained
(2) No pre-fixed fuzzy structure is necessary, and
(3) Simpler implementing procedures and less cost are involved.

We consider in this section the case of automatic anesthesia control in human patients for testing the optimized fuzzy controller. We did have, as a reference, the best fuzzy controller that was developed for the automatic anesthesia control (Karr & Gentry, 1993), and we consider the optimization of this controller using the HGA approach (Castillo & Melin, 2003). After applying the genetic algorithm the number of fuzzy rules was reduced from 12 to 9 with a similar performance of the fuzzy controller (Lozano, 2004). Of course, the parameters of the membership functions were also tuned by the genetic algorithm. We did compare the simulation results of the optimized fuzzy controllers obtained with the HGA against the best fuzzy controller that was obtained previously with expert knowledge, and control is achieved in a similar fashion.

The fitness function in this case can be defined in this case as follows:

$$f_i = \Sigma |y(k) - r(k)| \tag{7.14}$$

where Σ indicates the sum for all the data points in the training set, and $y(k)$ represents the real output of the fuzzy system and $r(k)$ is the reference output. This fitness value measures how well the fuzzy system is approximating the real data of the problem.

We consider the case of controlling the anesthesia given to a patient as the problem for finding the optimal fuzzy system for control (Lozano, 2004). The complete implementation was done in the MATLAB programming language. The fuzzy systems were build automatically by using the Fuzzy Logic Toolbox, and genetic algorithm was coded directly in the MATLAB language. The fuzzy systems for control are the individuals used in the genetic algorithm, and these are evaluated by comparing them to the ideal control given by the experts.

In other words, we compare the performance of the fuzzy systems that are generated by the genetic algorithm, against the ideal control system given by the experts in this application. We give more details below.

The main task of the anesthesist, during and operation, is to control anesthesia concentration. In any case, anesthesia concentration can't be measured directly. For this reason, the anesthesist uses indirect information, like the heartbeat, pressure, and motor activity. The anesthesia concentration is controlled using a medicine, which can be given by a shot or by a mix of gases. We consider here the use of isoflurance, which is usually given in a concentration of 0 to 2% with oxygen. In Fig. 7.26 we show a block diagram of the controller.

The air that is exhaled by the patient contains a specific concentration of isoflurance, and it is re-circulated to the patient. As consequence, we can measure isoflurance concentration on the inhaled and exhaled air by the patient, to estimate isoflurance concentration on the patient's blood. From the control engineering point of view, the task by the anesthesist is to maintain anesthesia concentration between the high level W (threshold to wake up) and the low level E (threshold to success). These levels are difficult to be determined in a changing environment and also are dependent on the patient's condition. For this reason, it is important to automate this anesthesia control, to perform this task more efficiently and accurately, and also to free the anesthesist from this time consuming job. The anesthesist can then concentrate in doing other task during operation of a patient.

The first automated system for anesthesia control was developed using a PID controller in the 60's. However, this system was not very successful due to the non-linear nature of the problem of anesthesia control. After this first attempt, adaptive control was proposed to automate anesthesia control, but robustness was the problem in this case. For these reasons, fuzzy logic was proposed for solving this problem. An additional advantage of fuzzy control is that we can use in the rules the same vocabulary as the medical Doctors use. The fuzzy control system can also be easily interpreted by the anesthesists.

The fuzzy system is defined as follows:

(1) Input variables: Blood pressure and Error
(2) Output variable: Isoflurance concentration
(3) Nine fuzzy if-then rules of the optimized system, which is the base for comparison
(4) 12 fuzzy if-then rules of an initial system to begin the optimization cycle of the genetic algorithm.

The linguistic values used in the fuzzy rules are the following: PB = Positive Big, PS = Positive Small, ZERO = zero, NB =Negative Big, NS = Negative Small

We show below a sample set of fuzzy rules that are used in the fuzzy inference system that is represented in the genetic algorithm for optimization.

if Blood pressure is NB and error is NB
    then conc_isoflurance is PS
if Blood pressures is PS
    then conc_isoflurance is NS
if Blood pressure is NB
    then conc_isoflurance is PB
if Blood pressure is PB
    then conc_isoflurance is NB
if Blood pressure is ZERO and error is ZERO
    then conc_isoflurance is ZERO
if Blood pressure is ZERO and error is PS
    then conc_isoflurance is NS
if Blood pressure is ZERO and error is NS
    then conc_isoflurance is PS
if error is NB
    then conc_isoflurance is PB
if error is PB
    then conc_isoflurance is NB
if error is PS
    then conc_isoflurance is NS
if Blood pressure is NS and error is ZERO
    then conc_isoflurance is NB
if Blood pressure is PS and error is ZERO
    then conc_isoflurance is PS.

The general characteristics of the genetic algorithm that are the following:

NIND = 40; % Number of individuals in each subpopulation.

MAXGEN = 300; % Maximum number of generations allowed.

GGAP = .6; %"Generational gap", which is the percentage from the complete population of new individuals generated in each generation.

PRECI = 120; % Precision of binary representations.

SelCh = select("rws", Chrom, FitnV, GGAP); % Roulette wheel method for selecting the indivuals participating in the genetic operations.

SelCh = recombin("xovmp", SelCh, 0.7); % Multi-point crossover as recombination method for the selected individuals.

ObjV = FuncionObjDifuso120_555(Chrom, sdifuso); Objective function is given by the error between the performance of the ideal control system given by the experts and the fuzzy control system given by the genetic algorithm.

In Table 7.1 we show the chromosome representation, which has 120 binary positions. These positions are divided in two parts, the first one indicates the number of rules of the fuzzy inference system, and the second one is divided again into fuzzy rules to indicate which membership functions are active or inactive for the corresponding rule.

Table 7.1. Binary Chromosome Representation

Bit assigned	Representation
1 a 12	Which rule is active or inactive
13 a 21	Membership functions active or inactive of rule 1
22 a 30	Membership functions active or inactive of rule 2
...	Membership functions active or inactive of rule...
112 a 120	Membership functions active or inactive of rule 12

We now describe the simulation results that were achieved using the hierarchical genetic algorithm for the optimization of the fuzzy control system, for the case of anesthesia control. The genetic algorithm is able to evolve the topology of the fuzzy system for the particular application. We used 300 generations of 40 individuals each to achieve the minimum error. The value of the minimum error achieved with this particular fuzzy logic controller was of 0.0064064, which is considered a small number in this application.

In Fig. 7.27 we show the simulation results of the fuzzy logic controller produced by the genetic algorithm after evolution. We used a sinusoidal input signal with unit amplitude and a frequency of 2 radians/second, with a transfer function of [1/(0.5s+1)]. In this figure we can appreciate the comparison of the outputs of both the ideal controller (1) and the fuzzy controller optimized by the genetic algorithm (2). From this figure it is clear that both controllers are very similar and as a consequence we can conclude that the genetic algorithm was able to optimize the performance of the fuzzy logic controller.

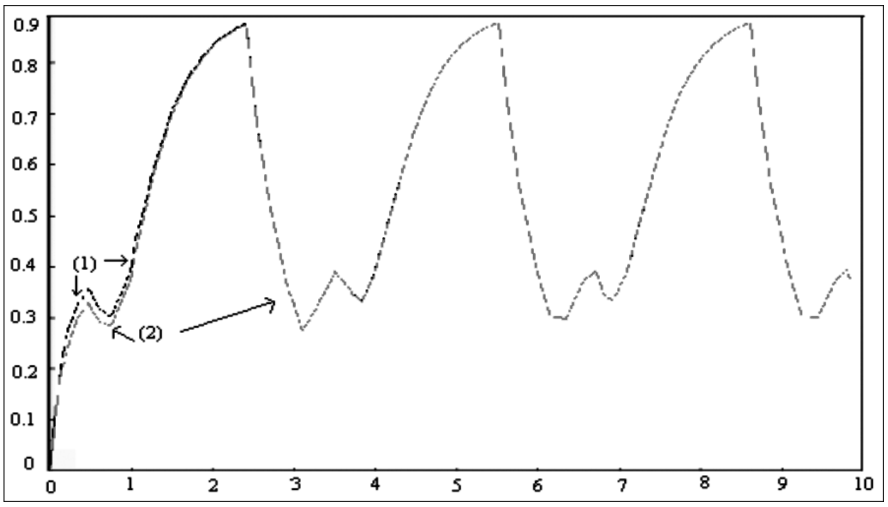


Fig. 7.27. Comparison between outputs of the ideal controller (1) and the fuzzy controller produced with the HGA (2)

We consider in this section the case of automatic anesthesia control in human patients for testing the optimized fuzzy controller. We did have, as a reference, the best fuzzy controller that was developed for the automatic anesthesia control, and we consider the optimization of this controller using the HGA approach. After applying the genetic algorithm the number of fuzzy rules was reduced from 12 to 9 with a similar performance of the fuzzy controller. Of course, the parameters of the membership functions were also tuned by the genetic algorithm. We did compare the simulation results of the optimized fuzzy controllers obtained with the HGA against the best fuzzy controller that was obtained previously with expert knowledge, and control is achieved in a similar fashion. Since simulation results are similar, and the number of rules was reduced, we can conclude that the HGA approach is a good alternative for designing fuzzy systems.

7.4 Summary

We have presented in this chapter the basic concepts of genetic algorithms and their applications. These optimization methodologies are motivated by nature's wisdom. Genetic algorithms emulate the process of evolution in nature. We have presented classical examples of the application of these optimization techniques. We have also presented the application of genetic algorithms to the problems of optimizing neural networks and fuzzy systems. Genetic algorithms can then be viewed as a technique for efficient design of intelligent systems, because they can be used to optimize the weights or architecture of the neural network, or the number of rules in a fuzzy system. In later chapters we will make use of this fact to design intelligent systems for pattern recognition in real world applications.