11 Genetic Algorithms and Neural Networks

One of the basic tasks in network design is to choose an architecture and weights appropriate for the given problem. The genetic algorithm (GA) [173, 141] is a general optimization method that has been applied to many problems including neural network training. It is appropriate for neural networks because it scales well to large nonlinear problems with multiple local minima.

As the name implies, the genetic algorithm is based on an analogy to natural evolutionary mechanisms. Many variations have been investigated, but the basic idea is competition between alternative solutions and "survival of the fittest." In this case, fitness is measured by a predefined objective function. Individuals in a large population have varying traits that affect their reproductive success in the given environment. Successful individuals live long enough to mate and pass their traits to their offspring. Offspring inherit traits from successful parents so they also have a good chance of being successful. Over many generations, the population adapts to its environment; disadvantageous traits become rare and the average fitness tends to increase over time.

One of the main advantages of the algorithm is that it requires very little problem-specific information. To apply the method to a specific problem, all that is needed is a fitness function that evaluates individual solutions and returns a rating of their quality, or "fitness." The algorithm itself operates on bit strings containing the "genetic code," that is, the parameters specifying a particular solution. Aside from the problem-specific evaluation function, all problems look the same to the algorithm, differing only in the length of the bit string and the number of units.

Because the algorithm needs so little problem-specific information, it is useful for complex problems that are difficult to analyze correctly. In particular, it does not need gradient information and so can be used on discontinuous functions and functions that are described empirically rather than analytically. It can also be used for temporal learning problems in which evaluation comes at the end of a long sequence of actions with no intermediate target values. It is not a simple hill-climbing method so it is not particularly bothered by local maxima. It will also tolerate a certain amount of noise in the evaluation function.

The algorithm has some of the flavor of simulated annealing in that many alternative solutions are examined and the search contains an element of randomness that helps prevent convergence to local maxima. It differs in that many candidate solutions are maintained rather than just one and elements of the better solutions are combined to generate new candidates. Like simulated annealing, it is a general optimization method that has applications beyond neural networks.

The main disadvantage of the method is the amount of computation needed to evaluate and store a large population of candidate solutions and converge to an optimum. Other techniques often converge faster when they can be used.

11.1 The Basic Algorithm

The basic operations are (1) selection based on fitness, (2) recombination of genetic material by crossover, and (3) mutation. The algorithm operates on a population of many units. Each unit has a bit string, its "genetic code," which encodes its solution to the given problem. The user supplies a problem-specific function that decodes the bit string, evaluates the solution, and returns a value which is translated to a fitness score. The fitness score determines which units are selected for mating. High scoring solutions tend to be selected more often than low scoring solutions and thus pass their characteristics to the next generation at a higher rate.

The basic algorithm starts with an initial population of N units with random parameters encoded in a binary bit string. Larger population sizes generally increase the chance of finding a good solution but, of course, require more processing time. The following steps are repeated until a solution is found or patience is exhausted.

Evaluation. Evaluate each unit and assign it a non-negative score (higher=better). Normalize by dividing by the sum of all scores to obtain fitness scores f_i in the range 0–1. If any unit satisfies the goal criteria, discard the other units and stop.

Selection. On N trials, select an individual i with probability f_i and copy it to the mating population. Units can be selected with probability f_i by assigning each unit a segment of the 0–1 interval proportional to its fitness and choosing a uniform random number; if the number falls in the interval assigned to the kth unit then select unit k. In the case of three units with fitness scores 0.1, 0.6, and 0.3, for example, the intervals would be 0–0.1, 0.1–0.7, and 0.7–1.0. Figure 11.1 illustrates this by analogy to a roulette wheel where each unit has a number of slots proportional to its fitness.

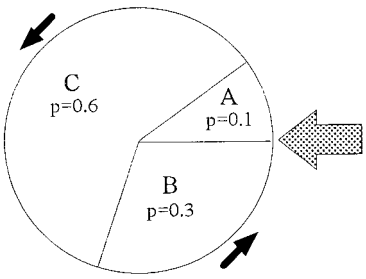

Figure 11.1
Selection. The genetic algorithm selects units for reproduction with probability proportional to their fitness. Units with higher fitness (corresponding to more slots on the wheel) are more likely to be chosen than units with lower fitness, but all units have some chance of being chosen.

Before crossover												
A	1	1	0	1	0	0	1	1	1	0	0	1
B	0	1	1	0	1	1	1	0	0	1	0	1

After crossover												
C	1	1	0	1	1	1	1	0	0	1	0	1
D	0	1	1	0	0	0	1	1	1	0	0	1

Figure 11.2
Crossover. Crossover mixes genetic codes inherited from two parents by crossing the bit strings at one or two random points. The bit strings encode characteristics of the parents so the offspring receives traits of both parents, but is not identical to either. The crossing point is random, so the mix of characteristics transferred varies with each mating.

Because of the element of chance, the number of times a unit reproduces will not be exactly proportional to its fitness, but, on average, if unit i has twice the fitness of unit j, then it will usually have about twice the offspring. Units with very low fitness ratings will rarely reproduce and face extinction.

Crossover. Divide the mating population into pairs and mix their genetic information by crossing the bit strings at one or two random points. Figure 11.2 illustrates the operation. If units A and B have parameter strings 110100111001 and **011011100101**, for example , then they would produce offspring 0110**00111001** and **1101**11100101 if the crossing point is after position 4. The probability that crossover will occur is set by a parameter p_c. For $p_c < 1$, there is some chance that the parents simply survive in the next generation unaltered by crossover. This helps to preserve good solutions since some copies are likely to survive unchanged. Typical values are $p_c = 0.6$ to 0.9.

Mutation. For each unit in the new set, flip each bit with some small probability; for example, $p_m \leq 0.001$. The number of mutations should be small to prevent deterioration of the algorithm into random search. The main purpose of mutation is to maintain population diversity. In general, p_m should be chosen so that mutations occur in only a small percentage of the population—in only one or two units for moderately sized populations.

Replacement. Copy the newly created units to the working population. In some variations, new units replace their parents. In others, they replace the least fit units.

Many variations of the algorithm have been proposed. In the basic algorithm, all units have a chance to reproduce and large portions of parameter strings are exchanged during reproduction so it is possible for good solutions to be lost. One remedy is to allow only the most successful fraction of the population to mate, with their offspring replacing the less

successful part of the population. Since the offspring do not replace their parents, this helps to preserve good solutions.

Other variations extend the biological analogy further by incorporating features such as paired chromosomes, dominance, inversion, and niche specialization. Some versions are Lamarckian, allowing adaptations made in the lifetime of a parent to be passed on to the offspring. Some vary the number of units that reproduce at each cycle. Some allow the population size to fluctuate and some maintain several subpopulations with only limited mixing. Goldberg [141] reviews many of these cases.

11.1.1 Effects of Crossover

By some accounts, crossover is responsible for most of the adaptive power of the algorithm. Crossover selects parameters from two good solutions and mixes them to create new combinations. The parent solutions were successful enough to be selected for reproduction so they presumably contain good parameter sets. Ideally, the offspring will inherit the best parameters from both parents and produce a new combination which is better than either.

Crossover is different from random search in that with crossover the offspring are in some sense intermediate between the two parents; they inherit some attributes from parent A and some from parent B but are identical to neither. This tends to confine the search to new combinations of parameters that have already proven useful. Random search, in contrast, is unguided and might create new units anywhere in the parameter space.

A *schemata* theory [141] has been developed to study how parameter strings evolve. A particular template of 1s, 0s and $*$s (don't cares) in a bit string, for example, $011***$ $*01**$, is called a *schema* (plural: schemata). Each bit in the string is simultaneously a component of many different schemata and each string simultaneously contains many overlapping schemata. Likewise, a single schema may be present in many strings in the population. The core idea is that a string containing a bit combination that is strongly correlated with good solutions is likely to be reproduced in the next generation. The *defining length* of a schema is the distance between its most separated defining bits. The distance between the leading 0 and the final 1 of $011****01**$, for example, is 8 bits. Schemata with long defining lengths contain widely separated significant bits and are more likely to be broken during crossover and thus less likely to survive than shorter schemata. Depending on how parameters are encoded in the bit string, this tends to make the algorithm favor low order, less complex, solutions over high order ones—usually a desirable feature for a learning algorithm.

11.1.2 Effects of Mutation

Mutation plays a rather small part in the standard algorithm. If the mutation rate is too large, the algorithm tends to degenerate into an inefficient random search. When all the

units are very similar however, as in the final stages of convergence, crossover creates few new solutions and mutation becomes more important. Because all defining bits of a schema must survive mutation for the schema to survive, schemata with fewer defining bits are more likely to survive mutation than those with many defining bits. If simple solutions have representations in terms of small numbers of parameters (few bits), then this favors simpler and presumably more robust solutions. Overall, the combination of fitness selection, crossover, and mutation favors schemata with above average fitness, short defining length, and low order.

11.1.3 Fitness Scaling

Because the user-supplied evaluation function can be chosen arbitrarily, it is useful to scale the raw scores to obtain normalized fitness scores. If all units receive raw scores in the range from 1000 to 1005, for example, the best solutions would have very little advantage over the worst and the search would be essentially random. This might occur in late stages of the algorithm when most units are clustered around a good solution. Similarly, in early generations most units may have low raw scores and a unit that makes a significant (but not decisive) improvement may get a much higher score, allowing it to dominate the next generation and cause premature loss of population diversity. This effect is more important when populations are small.

Scaling of the raw scores helps prevent these problems. A linear transformation is often used to map the raw scores f to fitness values f'

$$f' = af + b.$$

In choosing a and b, it is desirable that $f_{avg} \rightarrow f'_{avg}$ so that one expects each average unit to produce one offspring. The number of offspring for the best unit is controlled by ensuring $f'_{max} = C_{mult} f_{avg}$, where C_{mult} is the desired number of offspring for the best unit. For small populations ($n = 50$ to 100), values of $C_{mult} = 1.2$ to 2 are suggested [141]. If this scaling results in negative scores, set them to 0. Other methods of fitness scaling are discussed by Goldberg [141].

11.2 Example

A very simple example illustrates the mechanics of the algorithm. Figure 11.3 shows the function

$$J(x) = 64 - (x - 7)^2.$$

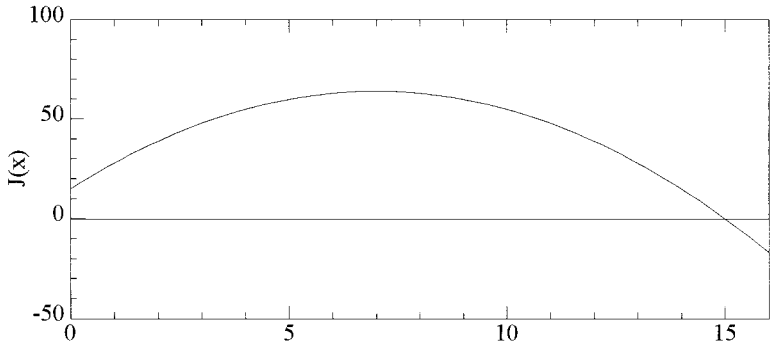

Figure 11.3
The function considered in the example.

for $0 \leq x < 16$. Let the population consist of four units A, B, C, and D with solutions x encoded in 4-bit strings. (A simple function and small population were chosen to provide a clear example. Normal functions are not so simple and populations are larger.)

Units are initialized with random values for the first generation.

Generation 1

unit	x	bits	J	f_i
A	1	0 0 0 1	28	0.1854
B	9	1 0 0 1	60	0.3973
C	15	1 1 1 1	0	0
D	6	0 1 1 0	63	0.4742

After random selection weighted by fitness, the population is A, B, D, D. Unit C with fitness 0 has died and unit D, with the highest fitness, is selected twice.

New Population

unit	
A	0 0 0 1
B	1 0 0 1
D	0 1 1 0
D	0 1 1 0

A mates to *D* and *B* mates to *D*, both with crossover after the 3rd bit.

Mating results

unit	
a	0 0 0 **0**
b	1 0 0 **0**
c	0 1 1 **1**
d	0 1 1 **1**

Mutation flips the 3rd bit in *a*.

Mutation results

unit	
a	0 0 **1** 0
b	1 0 0 0
c	0 1 1 1
d	0 1 1 1

The resulting population after one generation is

Generation 2

unit	x	bits	J	f_i
a	2	0 0 1 0	39	0.1696
b	8	1 0 0 0	63	0.2739
c	7	0 1 1 1	64	0.2782
d	7	0 1 1 1	64	0.2782

Units *c* and *d* have already reached the maximum of the function at $x = 7$ and the average score has increased from 37.75 to 57.75.

11.3 Application to Neural Network Design

The genetic algorithm is a general purpose optimization algorithm with applications beyond neural network design. It can be applied to network design in a number of ways—from simply determining a few weights in a predetermined network to choosing the entire architecture: the number of layers and nodes, connections, weight values and node

functions. Because it does not need gradient information, it can be used on networks with binary units and/or quantized weights.

Several things should be considered in applying the algorithm to neural networks. One of the more important factors is the representation—how problem parameters are encoded in the bit string. Neural networks often have many weights so the parameter string may be long. Because crossover breaks the string, it is desirable to put related parameters near each other in the bit string as much possible. Network weights tend to be strongly interrelated, however, with values of weights in one layer depending on the values of many weights in other layers. This means that the useful schemata often have high order and are easily broken by mutation and crossover. This interdependence leads some [102] to eliminate the crossover operation; this is not typical however.

The following paragraphs describe some applications of the algorithm to neural network design.

Training and Evaluation Some implementations [209] include a small amount of training in the fitness evaluation function. The idea is that one set of weights, A, might be worse that another set B, but set A might be much better than B after a small amount of training because of differences in the local terrain of the error space. In this case, unit A is closer to the solution than B even though it is initial performance is worse. Without training, the algorithm learns only the fitness of the initial point in the space. With training, each unit attempts to find the best spot reachable from its initial position; its fitness reflects the quality of a small area around the initial point so the algorithm searches more of the parameter space.

As in nature, the fitness of each unit is evaluated based on its performance after adaptation, but reproduction transmits the original parameters rather than the adapted parameters. The performance of a unit (in a sense) reflects the nearness of good solutions, and reproduction will tend to produce more offspring near good units, so the algorithm will tend to converge to good solutions.

Only a small amount of training is allowed, typically 5 to 10% of what would be required to train a random net to completion. If all units were trained to convergence, they could converge to the same or equivalent solutions and all units would have the same reproductive fitness regardless of the quality of their genes. In [209], it was found useful to allow each unit a different number of training cycles. Over a number of generations, this effectively measures the sensitivity of the gene to learning and rewards units which improve quickly with a small amounts of additional learning.

Subpopulations In nature, isolation of subpopulations is one factor that contributes to the development of new species. The use of several subpopulations with limited mixing

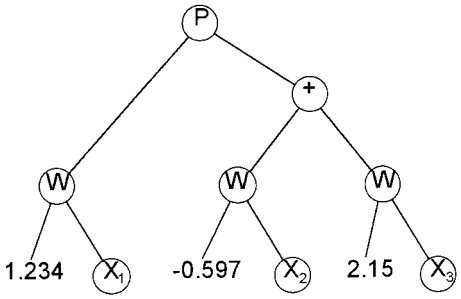

Figure 11.4
Representation of a neural network as a LISP expression. W represents a multiplication operation by an input weight and P represents a summation and the node nonlinearity.

allows aggressive optimization within each subpopulation while preserving diversity in the total population [398]. This is said to be more effective in preserving diversity than simply increasing the population size. One-at-a-time reproduction is used with fitness based on rank. The offspring do not replace their parents; they replace the worst units so good solutions are certain to survive.

GA Pruning of Neural Networks The genetic algorithm can also be used to prune neural networks [397]. Typically, the parameter string contains a bit for each weight in the original network; the bit is 1 if the connection is retained and 0 if it is pruned. The result of pruning is networks that are smaller, learn faster, and may generalize better.

Genetic Programming A related algorithm is *genetic programming* [221, 222, 223]. Koza and Rice [224] describe an algorithm to determine both the weights and connection architecture of a neural net. The network response function is encoded in a tree-structured LISP expression (figure 11.4) and the crossover operation exchanges subtrees between two parents. The major use of this representation has been outside of neural networks. It has been used for evolutionary adaptation of functions or programs whose statements have syntactic structure; it is useful because the subtree crossover operation preserves syntactic correctness.

11.3.1 Incompatible Genomes

A difficulty with application of the standard genetic algorithm to neural network optimization is the problem of incompatible genomes. In general, two successful individuals do not always yield a successful offspring when they mate; their bit-strings might represent points on two different local maxima and the combination might fall in a valley between

them, for example. In a neural network with H hidden units, there are $H!$ equivalent solutions obtainable by shuffling the order of the units in the hidden layer. There are another 2^H equivalent solutions obtainable by changing the signs of all weights into and out of any combination of hidden units (which leaves their function unchanged). Thus, two networks might compute identical input-output functions using different internal representations in which case the network obtained by mixing their weights would be very different from either and probably a poor solution. Neural networks also tend to be underconstrained; there are often many different input-output functions that satisfy the objective function equally well so equally successful networks may not even compute a similar input-output function.

The effect is that little progress is made until a significant fraction of the population compute similar functions with similar internal representations. Once a cluster of compatible networks develops, they have a higher probability of mating successfully and may grow to dominate the population at the expense of possibly better isolated solutions. The system then does local hill-climbing and is unlikely to explore very different solutions. This produces a strong tendency to converge to local maxima since the particular solution that comes to dominate initially is a nearly random selection. In theory, this doesn't have to happen, but it's likely if parameters are chosen for fast convergence (e.g., small populations and aggressive culling). Variations have been proposed to avoid these problems but they complicate the algorithm.

11.4 Remarks

The genetic algorithm is a general stochastic search method that has been used successfully in a number of ways for neural network design [3, 4, 50, 52, 66, 68, 108, 109, 155, 156, 304, 319]. Its main advantages are that it requires very little problem-specific information and is relatively insensitive to local maxima (minima). The algorithm itself is relatively easy to implement. Unlike some other search techniques, it does not require detailed problem-specific knowledge in order to generate new search candidates. Gradients are not required so the algorithm can be applied to discontinuous functions or functions that are defined empirically rather than analytically. It is applicable to mixed problems containing both continuous and discrete variables.

Its main disadvantage is the amount of processing required to evaluate and store a large number of different network configurations. Although the actual bit manipulations take very little time, the user-supplied objective function must be evaluated many times and this can be very slow with large networks and large training sets. It is worth noting, however, that the candidate solutions can be evaluated independently so N parallel processors should give close to a factor of N reduction in computation time.

Another caveat is that although the algorithm itself needs little problem-specific information, the efficiency of the search and the quality of the results depend heavily on how parameters are represented in the bit string. This is quite problem dependent and use of the algorithm may involve experimenting with several different representations to find one that works well.

Although there are claims of convergence to global maxima of the fitness function, convergence to local maxima is possible with small populations and aggressive culling of less successful solutions. Also, there are many parameters to be selected (population size, mutation rate, crossover method, fitness scaling, etc.) and it is not obvious how these affect the convergence properties.

Because the main theoretical advantage of the algorithm is its global optimization property and its main disadvantage is its inefficiency, it may be useful to use the algorithm in conjunction with more efficient local search methods. For example, the genetic algorithm might be used to do a coarsely quantized search to find the region containing the global minimum and then more efficient methods used to fine tune the solution in this region.