# 17

# Genetic Algorithms

## 17.1 Coding and operators

Learning in neural networks is an optimization process by which the error function of a network is minimized. Any suitable numerical method can be used for the optimization. Therefore it is worth having a closer look at the efficiency and reliability of different strategies. In the last few years genetic algorithms have attracted considerable attention because they represent a new method of stochastic optimization with some interesting properties [163, 305]. With this class of algorithms an evolution process is simulated in the computer, in the course of which the parameters that produce a minimum or maximum of a function are determined. In this chapter we take a closer look at this technique and explore its applicability to the field of neural networks.

### 17.1.1 Optimization problems

Genetic algorithms evaluate the target function to be optimized at some randomly selected points of the definition domain. Taking this information into account, a new set of points (a new population) is generated. Gradually the points in the population approach local maxima and minima of the function. Figure 17.1 shows how a population of points encloses a local maximum of the target function after some iterations. Genetic algorithms can be used when no information is available about the gradient of the function at the evaluated points. The function itself does not need to be continuous or differentiable. Genetic algorithms can still achieve good results even in cases in which the function has several local minima or maxima.

These properties of genetic algorithms have their price: unlike traditional random search, the function is not examined at a single place, constructing a possible path to the local maximum or minimum, but many different places are considered simultaneously. The function must be calculated for all elements of the population. The creation of new populations also requires additional calculations. In this way the optimum of the function is sought in several directions simultaneously and many paths to the optimum are processed in parallel. The calculations required for this feat are obviously much more extensive than for a simple random search.

However, compared to other stochastic methods genetic algorithms have the advantage that they can be parallelized with little effort. Since the calculations of the function on all points of a population are independent from each other, they can be carried out in several processors [164]. Genetic algorithms are thus inherently parallel. A clear improvement in performance can be achieved with them in comparison to other non-parallelizable optimization methods.

**Fig. 17.1.** A population of points encircles the global maximum after some generations.

Compared to purely local methods (e.g., gradient descent) genetic algorithms have the advantage that they do not necessarily remain trapped in a suboptimal local maximum or minimum of the target function. Since information from many different regions is used, a genetic algorithm can move away from a local maximum or minimum if the population finds better function values in other areas of the definition domain.

In this chapter we show how evolutionary methods are used in the search for minima of the error function of neural networks. Such error functions have special properties which make their optimization difficult. We will discuss the extent to which genetic algorithms can overcome these difficulties.

Even without this practical motivation the analysis of genetic algorithms is important, because in the course of evolution the networking pattern of biological neural networks has been created and improved. Through an evolutionary organization process nerve systems were continuously modified until they attained an enormous complexity. Therefore, by studying artificial neural networks and their relationship with genetic algorithms we can gain further valuable insights for understanding biological systems.

### 17.1.2 Methods of stochastic optimization

Let us look at the problem of the minimization of a real function $f$ of the $n$ variables $x_1, x_2, \ldots, x_n$. If the function is differentiable the minima can often be found by direct analytical methods. If no analytical expression for $f$ is known or if $f$ is not differentiable, the function can be optimized with stochastic methods. The following are some well-known techniques:

#### Random search

The simplest form of random optimization is stochastic search. A starting point $x = (x_1, x_2, \ldots, x_n)$ is randomly generated and $f(x_1, x_2, \ldots, x_n)$ is computed. Then a direction is sought in which the value of the function decreases. To do this a vector $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_n)$ is randomly generated and $f$ is computed at $(x_1 + \delta_1, \ldots, x_n + \delta_n)$. If the value of the function at this point is lower than at $x = (x_1, x_2, \ldots, x_n)$ then $(x_1 + \delta_1, \ldots, x_n + \delta_n)$ is taken as the new search point and the algorithm is started again. If the new function value is greater, a new direction is generated. The algorithm runs (within a predetermined maximum number of attempts) until no further decreasing direction of function values can be found.

The algorithm can be further improved by making the length of the direction vector $\boldsymbol{\delta}$ decrease in time. Thus the minimum of the function is approximated by increasingly smaller steps.

**Fig. 17.2.** A local minimum traps the search process

The disadvantage of simple stochastic search is that a local minimum of the function can steer the search in the wrong direction (Figure 17.2). However, this can be partially compensated by carrying out several independent searches.

### Metropolis algorithm

Stochastic search can be improved using a technique proposed by Metropolis. ([304]). If a new search direction $(\delta_1, \ldots, \delta_n)$ guarantees that the function value decreases, it is used to update the search position. If the function in this direction increases, it is still used with the probability $p$ where

$$p = \frac{1}{1 + \exp\left[\frac{1}{\alpha}\left(f(x_1 + \delta_1, \ldots, x_n + \delta_n) - f(x_1, \ldots, x_n)\right)\right]}$$

The constant $\alpha$ approaches zero gradually. This means that the probability $p$ tends towards zero if the function $f$ increases in the direction $(\delta_1, \ldots, \delta_n)$. In the final iterations of the algorithm only those directions in which the function values decrease are actually taken.

This strategy can prevent the iteration process from remaining trapped in suboptimal minima of the function $f$. With probability $p > 0$ an iteration can take an ascending direction and possibly overcome a local minimum.

### Bit-based descent methods

If the problem can be recoded so that the function $f$ is calculated with the help of a binary string (a sequence of binary symbols), then bit-based stochastic methods can be used. For example, let $f$ be the one-dimensional function $x \mapsto (1-x)^2$. The positive real value $x$ can be coded in a computer as a binary number. The fixed-point coding of $x$ in eight bits

$$x = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$

can be interpreted as follows: the first three bits $b_7$, $b_6$ and $b_5$ code that part of the value of $x$ which is in front of the decimal point in the binary code. The five bits $b_4$ to $b_0$ code the portion of the value of $x$ after the point. Thus only numbers whose absolute value is less than 8 are coded. An additional bit can be used for the sign.

With this recoding $f$ is a real function over all binary strings with length eight. The following algorithm can be used: a randomly chosen initial string $b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$ is generated. The function $f$ is then computed for $x = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$. A bit of the string is selected at random and flipped. Let the new string be, for example, $x' = b_7' b_6 b_5 b_4 b_3 b_2 b_1 b_0$. The function $f$ is computed at this new point. If the value of the function is lower than before, the new string is accepted as the current string and the algorithm is started again. The algorithm runs until no bit flip improves the value of the function [103].

Strictly speaking this algorithm is only a special instance of stochastic search. The only difference is that the directions which can be generated are now fixed from the beginning. There are only eight possibilities which correspond to the eight bits of the fixed-point representation. The precision of the approximation is also fixed from the beginning because with 8-bit fixed-point coding only a certain maximum precision can be achieved. The search space of the optimization method and the possible search direction are thus discretized.

The method can be generalized for $n$-dimensional functions by recoding the real values $x_1, x_2, \ldots, x_n$ in $n$ binary strings which are then appended to form a single string, which is processed by the algorithm. The Metropolis strategy can also be used in bit-based methods.

Bit-based optimization techniques are already very close to genetic algorithms; these naive search methods work effectively in many cases and can even outdo elaborate genetic algorithms [103]. If the function to be optimized is not too complex, they reach the optimal minimum with substantially fewer iteration steps than the more intricate algorithms.

### 17.1.3 Genetic coding

Genetic algorithms are stochastic search methods managing a population of simultaneous search positions. A conventional genetic algorithm consists of three essential elements:

- a coding of the optimization problem
- a mutation operator
- a set of information-exchange operators

The *coding* of the optimization problem produces the required discretization of the variable values (for optimization of real functions) and makes their simple management in a population of search points possible. Normally the maximum number of search points, i.e., the *population size*, is fixed at the beginning.

The *mutation operator* determines the probability with which the data structures are modified. This can occur spontaneously (as in stochastic search) or only when the strings are combined to generate a new population of search points. In binary strings a mutation corresponds to a bit flip.

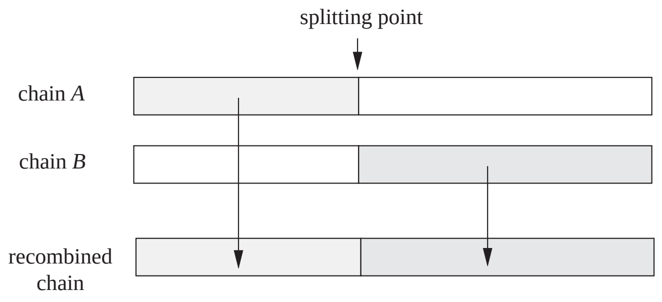

**Fig. 17.3.** An example of crossover

The *information exchange operators* control the recombination of the search points in order to generate a new, better population of points at each iteration step. Before recombining, the function to be optimized must be evaluated for all data structures in the population. The search points are then sorted in the order of their function value, i.e., in the order of their so-called *fitness*. In a minimization problem the points which are placed at the beginning of the list are those for which the function value is lowest. Those points for which the function to be minimized has the greatest function value are placed at the end of the list. Following this sorting operation the points are reproduced in such a way that the data structures at the beginning of the list are selected with a higher probability than the ones at the end of the list. A typical reproduction operator is *crossover*. Two strings $A$ and $B$ are selected as "parents" and a cut-off position for both is selected at random. The new string is formed so that the left side comes from one parent and the right side from the other. This produces an interchange of the information stored in each parent string. The whole process is reminiscent of genetic exchange in living organisms. A favorable interchange can produce a string closer to the minimum of the target function than each parent string by itself. We expect the collective fitness of the population to increase in time. The algorithm can be stopped when the fitness of the best string in the population no longer changes.

For a more precise illustration of genetic algorithms we will now consider the problem of string coding.

Optimization problems whose variables can be coded in a string are suitable for genetic algorithms. To this end an alphabet for the coding of the information must be agreed upon. Consider, for example, the following problem: a keyboard is to be optimized by distributing the 26 letters A to Z over 26 positions. The learning time of various selected candidates is to be minimized. This problem could only be solved by a multitude of experiments. A suitable coding would be a string with 26 symbols, each of which can be one of the 26 letters (without repeats). So the alphabet of the coding consists of 26 symbols, and the search space of the problem contains 26! different combinations. For Asian languages the search space is even greater and the problem correspondingly more difficult [162].

A binary coding of the optimization problem is ideal because in this way the mutation and information exchange operators are simple to implement. With neural networks, in which the parameters to be optimized are usually real numbers, the definition of an adequate coding is an important problem. There are two alternatives: floating-point or fixed-point coding. Both possibilities can be used, whereby fixed-point coding allows more gradual mutations than floating-point coding [221]. With the latter the change of a single bit in the exponent of the number can cause a dramatic jump. Fixed-point coding is usually sufficient for dealing with the parameters of neural networks (see Sect. 8.2.3).

### 17.1.4 Information exchange with genetic operators

Genetic operators determine the way new strings are generated out of existing ones. The mutation operator is the simplest to describe. A new string can be generated by copying an old string position by position. However, during copying each symbol in the string can be modified with a certain probability, the mutation rate. The new string is then not a perfect copy and can be used as a new starting point for a search operation in the definition domain of the function to be optimized.

The crossover operator was described in the last section. With this operator portions of the information contained in two strings can be combined. However an important question for crossover is at which exact places we are allowed to partition the strings.

Both types of operator can be visualized in the case of function optimization. Assume that the function $x \mapsto x^2$ is to be optimized in the interval $[0, 1]$. The values of the variable $x$ can be encoded with the binary fixed-point coding $x = 0.b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$. Thus there are 1024 different values of $x$ and only one of them optimizes the proposed quadratic function. The code used discretizes the definition space of the function. The minimum distance between consecutive values of $x$ is $2^{-10}$.

A mutation of the $i$-th bit of the string $0.b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$ produces a change of $x$ of $\delta = 2^{-i}$. Thus a new point $x + \delta$ is generated. In this case the mutation corresponds to a stochastic search operation with variable step length.

When the number $x = 0.b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$ is recombined with the number $y = 0.a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$, a cut-off position $i$ is selected at random. The new string, which belongs to the number $z$, is then:

$$z = 0.b_9b_8 \cdots b_ia_{i-1} \cdots a_0.$$

The new point $z$ can also be written as $z = x + \delta$, where $\delta$ is dependent on the bit sequences of $x$ and $y$. Crossover can therefore be interpreted as a further variation of stochastic search. But note that in this extremely simplified example any gradient descent method is much more efficient than a genetic algorithm.

## 17.2 Properties of genetic algorithms

Genetic algorithms have made a real impact on all those problems in which there is not enough information to build a differentiable function or where the problem has such a complex structure that the interplay of different parameters in the final cost function cannot be expressed analytically.

### 17.2.1 Convergence analysis

The advantages of genetic algorithms first become apparent when a population of strings is observed. Let $f$ be the function $x \mapsto x^2$ which is to be maximized, as before, in the interval $[0, 1]$. A population of $N$ numbers in the interval $[0, 1]$ is generated in 10-bit fixed-point coding. The function $f$ is evaluated for each of the numbers $x_1, x_2, \ldots, x_N$, and the strings are then listed in descending order of their function values. Two strings from this list are always selected to generate a new member of a new population, whereby the probability of selection decreases monotonically in accordance with the ascending position in the sorted list.

The computed list contains $N$ strings which, for a sufficiently large $N$, should look like this:

$$\begin{array}{ll} 1 & 0.1{*}{*}{*}{*}{*}{*}{*}{*}{*}{*} \\ 1 & 0.1{*}{*}{*}{*}{*}{*}{*}{*}{*}{*} \\ \vdots & \vdots \\ 1 & 0.0{*}{*}{*}{*}{*}{*}{*}{*}{*}{*} \end{array}$$

The first positions in the list are occupied by strings in which the first bit after the point is a 1 (i.e., the corresponding numbers lie in the interval $[0.5, 1]$). The last positions are occupied by strings in which the first bit after the decimal point is a 0. The asterisk stands for any bit value from 0 to 1, and the zero in front of the point does not need to be coded in the strings. The upper strings are more likely to be selected for a recombination, so that the offspring is more likely to contain a 1 in the first bit than a 0. The new population is evaluated and a new fitness list is drawn up. On the other hand, strings with a 0 in the first bit are placed at the end of the list and are less likely to be selected than the strings which begin with a 1. After several generations no more strings with a 0 in the first bit after the decimal point are contained in the population.

The same process is repeated for the strings with a zero in the second bit. They are also pushed towards extinction. Gradually the whole population converges to the optimal string 0.1111111111 (when no mutation is present).

With this quadratic function the search operation is carried out within a very well-ordered framework. New points are defined at each crossover, but steps in the direction $x = 1$ are more likely than steps in the opposite direction. The step length is also reduced in each reproduction step in which the 0 bits are eliminated from a position. When, for example, the whole population only consists of strings with ones in the first nine positions, then the maximum step length can only be $2^{-10}$.

The whole process strongly resembles simulated annealing. There, stochastic jumps are also generated, whereby transitions in the maximization direction are more probable. In time the temperature constant decreases to zero, so that the jumps become increasingly smaller until the system *freezes* at a local maximum.

John Holland [195] suggested the notion of *schemata* for the convergence analysis of genetic algorithms. Schemata are bit patterns which function as representatives of a set of binary strings. We already used such bit patterns in the example above: the bit patterns can contain each of the three symbols 0, 1 or $*$. The schema $**00**$, for example, is a representative of all strings of length 6 with two zeros in the central positions, such as: 100000, 110011, 010010, etc.

During the course of a genetic algorithm the best bit patterns are gradually selected, i.e., those patterns which minimize/maximize the value of the function to be optimized. Normal genetic algorithms consist of a finite repetition of the three steps:

1. selection of the parent strings,
2. recombination,
3. mutation.

This raises the question: how likely is it that the better bit patterns survive from one generation of a genetic algorithm to another? This depends on the probability with which they are selected for the generation of new child strings and with which they survive the recombination and mutation steps. We now want to calculate this probability.

In the algorithm to be analyzed, the population consists of a set of $N$ binary strings of length $\ell$ at time $t$. A string of length $\ell$ which contains one of the three symbols 0, 1, or $\square$ in each position is a bit pattern or schema. The symbol $\square$ represents a 0 or a 1. The number of strings in the population in generation $t$ which contain the bit pattern $H$ is called $o(H,t)$. The diameter of a bit pattern is defined as the length of the pattern's shortest substring that still contains all fixed bits in the pattern. For example, the bit pattern $**1*1**$ has diameter three because the shortest fragment that contains both constant bits is the substring $1*1$ and its length is three. The diameter of a bit pattern $H$ is called $d(H)$, with $d(H) \geq 1$. It is important to understand that a schema is of the same length as the strings that compose the population.

Let us assume that $f$ has to be maximized. The function $f$ is defined over all binary strings of length $\ell$ and is called the fitness of the strings. Two parent strings from the current population are always selected for the creation of a new string. The probability that a parent string $H_j$ will be selected from $N$ strings $H_1, H_2, \ldots, H_N$ is

$$p(H_j) = \frac{f(H_j)}{\sum_{i=1}^{N} f(H_i)}.$$

This means that strings with greater fitness are more likely to be selected than strings with lesser fitness. Let $f_\mu$ be the average fitness of all strings in the population, i.e.,

$$f_\mu = \frac{1}{N}\sum_{i=1}^{N} f(H_i).$$

The probability $p(H_j)$ can be rewritten as

$$p(H_j) = \frac{f(H_j)}{Nf_\mu}.$$

The probability that a schema $H$ will be passed on to a child string can be calculated in the following three steps:

**i) Selection**

Selection with replacement is used, i.e., the whole population is the basis for each individual parent selection. It can occur that the same string is selected twice. The probability $P$ that a string is selected which contains the bit pattern $H$ is:

$$P = \frac{f(H_1)}{Nf_\mu} + \frac{f(H_2)}{Nf_\mu} + \cdots + \frac{f(H_k)}{Nf_\mu},$$

where $H_1, H2, \ldots, Hk$ represent all strings of the generation which contain the bit pattern $H$. If there are no such strings, then $P$ is zero.

The fitness $f(H)$ of the bit pattern $H$ in the generation $t$ is defined as

$$f(H) = \frac{f(H_1) + f(H_2) + \cdots + f(H_k)}{o(H,t)}.$$

Thus $P$ can be rewritten as

$$P = \frac{o(H,t)f(H)}{Nf_\mu}.$$

The probability $P_A$ that two strings which contain pattern $H$ are selected as parent strings is given by

$$P_A = \left(\frac{o(H,t)f(H)}{Nf_\mu}\right)^2.$$

The probability $P_B$ that from two selected strings only one contains the pattern $H$ is:

$$P_B = 2\frac{o(H,t)f(H)}{Nf_\mu}\left(1 - \frac{o(H,t)f(H)}{Nf_\mu}\right).$$

**ii) Recombination**

For the recombination of two strings a cut-off point is selected between the positions 1 and $\ell - 1$ and then a crossover is carried out. The probability $W$ that a schema $H$ is transmitted to the new string depends on two cases. If both parent strings contain $H$, then they pass on this substring to the new string. If only one of the strings contains $H$, then the schema is inherited at most half of the time. The substring $H$ can also be destroyed with probability $(d(H)-1)/(\ell-1)$ during crossover. This means that

$$W \geq \left(\frac{o(H,t)f(H)}{Nf_\mu}\right)^2 + \frac{2}{2}\frac{o(H,t)f(H)}{Nf_\mu}\left(1-\frac{o(H,t)f(H)}{Nf_\mu}\right)\left(1-\frac{d(H)-1}{\ell-1}\right).$$

The probability $W$ is greater than or equal to the term on the right in the above inequality, because in some favorable cases the bit string is not destroyed by crossover (one parent string contains $H$ and the other parent *part* of $H$). To simplify the discussion we will not examine all these possibilities. The inequality for $W$ can be further simplified to

$$W \geq \frac{o(H,t)f(H)}{Nf_\mu}\left(1-\frac{d(H)-1}{\ell-1}\left(1-\frac{o(H,t)f(H)}{Nf_\mu}\right)\right).$$

### iii) Mutation

When two strings are recombined, the information contained in them is copied bit by bit to the child string. A mutation can produce a bit flip with the probability $p$. This means that a schema $H$ with $b(H)$ fixed bits will be preserved after copying with probability $(1-p)^{b(H)}$. If a mutation occurs the probability $W$ of the schema $H$ being passed on to a child string changes according to $W'$, where

$$W' \geq \frac{o(H,t)f(H)}{Nf_\mu}\left(1-\frac{d(H)-1}{\ell-1}\left(1-\frac{o(H,t)f(H)}{Nf_\mu}\right)\right)(1-p)^{b(H)}.$$

If in each generation $N$ new strings are produced, the expected value of the number of strings which contain $H$ in the generation $t+1$ is $NW'$, that is

$$\langle o(H,t+1)\rangle \geq \frac{o(H,t)f(H)}{f_\mu}\left(1-\frac{d(H)-1}{\ell-1}\left(1-\frac{0(H,t)f(H)}{Nf_\mu}\right)\right)(1-p)^{b(H)}. \tag{17.1}$$

Equation (17.1) or slight variants thereof are known in the literature by the name "schema theorem" [195, 163]. This result is interpreted as stating that in the long run the best bit patterns will diffuse to the whole population.

### 17.2.2 Deceptive problems

The schema theorem has to be taken with a grain of salt. There are some functions in which finding the optimal bit patterns can be extremely difficult for a genetic algorithm. This happens in many cases when the optimal bits in the selected coding exhibit some kind of correlation. Consider the following function of $n$ variables,

$$(x_1,x_2,\ldots,x_n) \mapsto \frac{x_1^2+\cdots+x_n^2}{x_1^2+\varepsilon}+\cdots+\frac{x_1^2+\cdots+x_n^2}{x_n^2+\varepsilon},$$

where $\varepsilon$ is a small positive constant. The minimum of this function is located at the origin and any gradient descent method would find it immediately. However if the $n$ parameters are coded in a single string using 10 bits, any time one of these parameters approaches the value zero, the value of the function increases. It is not possible to approach the origin following the direction of the axes. This means that only *correlated* mutations of the $n$ parameters are favorable, so that the origin is reached through the diagonal valleys of the fitness function. The probability of such coordinated mutations is very small when the number of parameters $n$ and the number of bits used to code each parameter increases. Figure 17.4 shows the shape of the two-dimensional version of this problematic function.

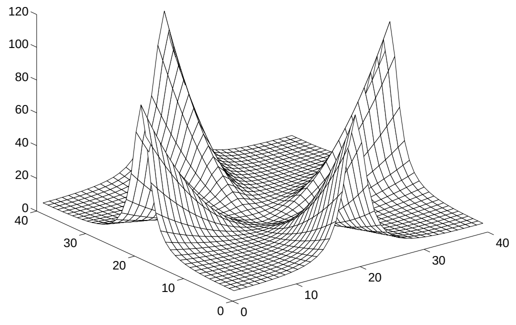

**Fig. 17.4.** A deceptive function

Functions which "hide" the optimum from genetic algorithms have been called *deceptive functions* by Goldberg and other authors. They mislead the GA into pursuing false leads to the optimum, which in many cases is only reached by pure luck.

There has been much research over the last few years on classifying the kinds of function which should be easy for a genetic algorithm to optimize and those which should be hard to deal with. Holland and Mitchell defined so-called "royal road" functions [314], which are defined in such a way that several parameters $x_1, x_2, \ldots, x_n$ are coded contiguously and the fitness function $f$ is just a sum of $n$ functions of each parameter, that is,

$$f(x_1, x_2, \ldots, x_n) = f_1(x_1) + f_2(x_2) + \cdots + f_n(x_n).$$

When these functions are optimized, genetic algorithms rapidly find the necessary values for each parameter. Mutation and crossover are both beneficial. However, mere addition of a further function $f'$ of the variables $x_1$ and $x_2$ can slow down convergence [141]. This happens because the additional term tends to select contiguous combinations of $x_1$ and $x_2$. If the contribution from

$f'(x_1, x_2)$ to the fitness function is more important than the contribution from $f_3(x_3)$, some garbage bits at the coding positions for $x_3$ can become attached to the strings with the optimum values for $x_1$ and $x_2$. They then get a "free ride" and become selected more often. That is why they are called *hitch-hiking* bits [142].

Some authors have argued in favor of the *building block hypothesis* to explain why GAs do well in some circumstances. According to this hypothesis a GA finds building blocks which are then combined through the generations in order to reach the optimal solution. But the phenomena we just pointed out, and the correlations between the optimization parameters, sometimes preclude altogether the formation of these building blocks. The hypothesis has received some strong criticism in recent years [166, 141].

### 17.2.3 Genetic drift

Equation (17.1) giving the expected number of strings which inherit a schema $H$ shows the following: schemata with above average fitness and small diameter will be selected more often, so that we expect them to diffuse to the population at large. The equation also shows that when the mutation rate is too high ($p \approx 1$) schemata are destroyed. The objective of having mutations is to bring variability into the population and extend it over the whole definition domain of the fitness function. New unexplored regions can be tested. This tendency to explore can be controlled by regulating the magnitude of the mutation rate. So-called *metagenetic algorithms* encode the mutation rate or the length of stochastic changes to the points in the population in the individual bit strings associated with each point. In this way the optimal mutation rate can be sought simultaneously with the optimum of the fitness function to accelerate the convergence speed.

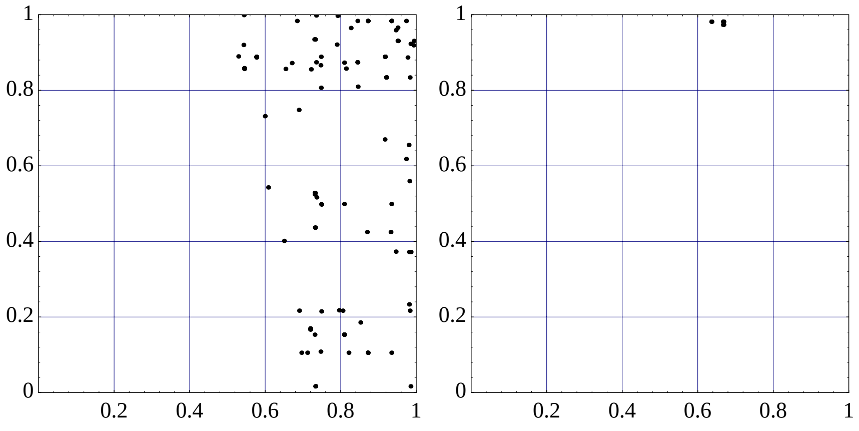

**Fig. 17.5.** Genetic drift in a population after 1000 and 3000 generations

Equation (17.1) also shows that when a schema $H$ is overrepresented in a population, it can diffuse even when its fitness $f(H)$ is not different from the average fitness $f_\mu$. In this case

$$\left(1-\frac{d(H)-1}{\ell-1}\left(1-\frac{o(H,t)f(H)}{Nf_\mu}\right)\right)=\left(1-\frac{d(H)-1}{\ell-1}\left(1-\frac{o(H,t)}{N}\right)\right).$$

Schemata with a large factor $o(H,t)/N$ will be less disturbed by crossover. The algorithm then converges to the schema $H$ without any good reason other than the fact that the schema is already over-represented in the population. In biology this is called *genetic drift* because it represents a *random walk* in search space. Figure 17.5 shows the result of an experiment in which a population of two-dimensional points was randomly generated in the domain $[0,1]\times[0,1]$. A constant fitness function was used and no mutations were allowed, only recombinations. One could think that under these circumstances the population would expand and occupy random positions in the whole definition domain. After 100 generations the population had moved to just one side of the square and after 3000 generations it had merged to just three points. This symmetry breaking comes from an initial bias in the bit combinations present in the original population.

Losing bit patterns during the run of the GA is what a GA is all about, otherwise the population would never reach a consensus about the optimal region to be explored. This loss of information becomes problematic when the function to be optimized exhibits many flat regions. Bit patterns that would be needed for later convergence steps can be lost. Mutation tries to keep the balance between these two contradictory objectives, exploration and convergence. Finding the right mix of both factors depends on the particular problem and has to be left to the practitioner. Some alternatives are the metagenetic algorithms already mentioned or a broader set of recombination operators, such as crossover with the simultaneous inversion of one or both of the inherited string pieces. This corresponds to a massive mutation of the coded string and brings new variability into the population.

The loss of variability in the long run can be compared to the controlled lowering of the temperature constant in *simulated annealing*. In a less variable population, parent and child strings are very similar and searching is mostly done locally. Crossover by itself leads to this kind of controlled convergence to a region in search space.

### 17.2.4 Gradient methods versus genetic algorithms

Genetic algorithms offer some interesting properties to offset their high computational cost. We can mention at least three of them: a) GAs explore the domain of definition of the target function at many points and can thus escape from local minima or maxima; b) the function to be optimized does not need to be given in a closed analytic form – if the process being analyzed is too complex to describe in formulas, the elements of the population are used to run some experiments (numerical or in the laboratory) and the results are interpreted as their fitness; c) since evaluation of the target function is independent for each element of the population, the parallelization of the GA is quite straightforward. A population can be distributed on several processors and the selection process carried out in parallel. The kinds of recombination operator define the type of communication needed between the processors.

Straightforward parallelization and the possibility of their application in ill-defined problems makes GAs attractive. De Jong has emphasized that GAs are not function optimizers of the kind studied in numerical analysis [105]. Otherwise their range of applicability would be very narrow. As we already mentioned, in many cases a naive hill-climber is able to outperform complex genetic algorithms. The hill-climber is started $n$ times at $n$ different positions and the best solution is selected. A combination of a GA and hill-climbing is also straightforward: the elements in the population are selected in "Darwinian" fashion from generation to generation, but can become better by modifying their parameters in "Lamarckian" way, that is, by performing some hill-climbing steps before recombining. Davis [103] showed that many popular functions used as benchmarks for genetic algorithms can be optimized with a simple *bit climber* (stochastic bit-flipping). In six out of seven test functions the bit climber outperformed two variants of a genetic algorithm. It was three times faster than an efficient GA and twenty-three times faster than an inefficient version. Ackley [8, 9] arrived at similar results when he compared seven different optimization methods. Depending on the test function one or the other minimization strategies emerged victorious. These results only confirm what numerical analysts have known for a long time: there is no optimal optimization method, it all depends on the problem at hand. Even if this is so, the attractiveness of easy parallelization is not diminished. If no best method exists, we can at least parallelize those we have.

## 17.3 Neural networks and genetic algorithms

Our interest in genetic algorithms for neural networks has two sources: is it possible to use this kind of approach to find the weights in a network? And even more important: is it possible to let networks evolve so that they find an optimal topology? The question of network topology is one of those problems for which no closed-form fitness function over all possible configurations can be given. We just propose a topology and let the network run, change the topology again and look at the results. Before going into the details we have to look at some specific characteristics of neural networks which seem to preclude the use of genetic algorithms.

### 17.3.1 The problem of symmetries

Before genetic algorithms can be used to minimize the error function of neural networks, an appropriate code must be designed. Usually the weights of the network are floating-point numbers. Assume that the $m$ weights $w_1, w_2, \ldots, w_m$ have been coded and arranged in a string. The target function for the GA is the error of the network for a given training set. The code for each parameter could consist of 20 bits, and in that case all network parameters could be represented by a string of $20m$ bits. These strings are then processed in the usual way.

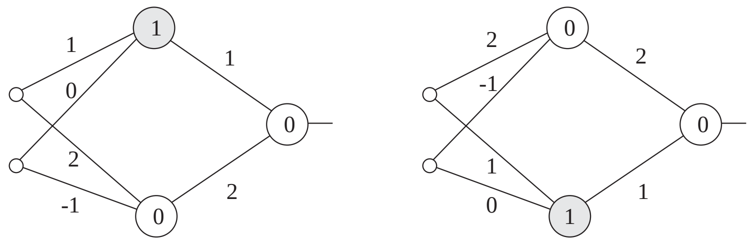

**Fig. 17.6.** Equivalent networks (weight permutation)

However, the target function exhibits some "deceptive" characteristics in this case. The basic problem is the high number of symmetries of the error function, which arise because the parameters of multilayered networks can be permuted without affecting the network function. The two networks shown in Figure 17.6 are equivalent, since they produce the same output for the same inputs. In both networks the functionality of the hidden units was interchanged. However, the arrangement of the coded parameters in a string looks very different. Because of the symmetries of the error function the number of local minima is high. The population of network parameters then disperses over different regions and examines different incompatible combinations. Crossover under these circumstances will almost certainly lead nowhere or converge very slowly when the population has drifted to a definite region of weight space.

Some authors' experiments have also shown that coding of the network parameters has an immediate effect on the convergence speed [317]. They have found that fixed-point coding is usually superior to floating-point coding of the parameters. Care must also be taken not to divide the strings through the middle of some parameter. It is usually better to respect the parameter boundaries so that crossover only combines the parameters but does not change the bit representation of each one.

Helping the genetic algorithm to respect good parameter combinations for a computing unit is an important problem. The simplest strategy is to find a good enumeration of the weights in a network that keeps related weights near each other in the coding string. When crossover is applied on such strings

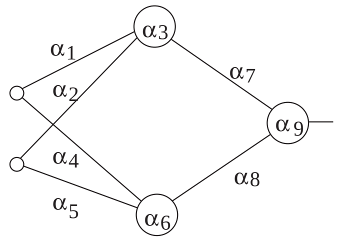

**Fig. 17.7.** Sequencing of the network parameters

there is a higher probability that related parameters are held together in the new string. The weights of the incoming edges into a unit constitute a logical entity and an enumeration like the one shown in Figure 17.7 can help a GA to converge on a good solution. The subindex of each weight shows which place it occupies in the coding string. This simple idea can be used with any network topology, but has obvious limits.

Modifications of the individual weights can be produced only by mutation. The mutations can be produced by copying not the value $\alpha$ to the child string but $\alpha + \varepsilon$, where $\varepsilon$ is a random number of small absolute value. This produces the necessary variability of the individual parameters. The mutations produce a random walk of the network weights in weight space. The selection method of the GA separates good from bad mutations and we expect the population to stabilize at a local minimum.

### 17.3.2 A numerical experiment

Figure 17.8 shows a network for the encoding-decoding problem with 8 input bits. The input must be reproduced at the output, and only one of the input bits is a one, the others are zero. The hidden layer is the bottleneck for transmission of input to output. One possible solution is to encode the input in three bits at the hidden layer, but other solutions also exist.

The 48 weights of the network and the 11 bits were encoded in a string with 59 floating-point numbers. Crossover through the middle of parameters was avoided and in this case a fixed-point coding is not really necessary. Mutation was implemented as described above, not by bit flips but by adding some stochastic deviation. Figure 17.9 shows the evolution of the error curve. After 5350 generations a good parameter combination was found, capable of keeping the total error for all 8 output lines under 0.05. The figure also shows the average error for the whole population and the error for the best parameter combination in each generation.

This is an example of a small network in which the GA does indeed converge to a solution. Some authors have succeeded in training much larger networks [448]. However, as we explained before, in the case of such well-defined numerical optimization problems direct hill climbing methods are usually better.

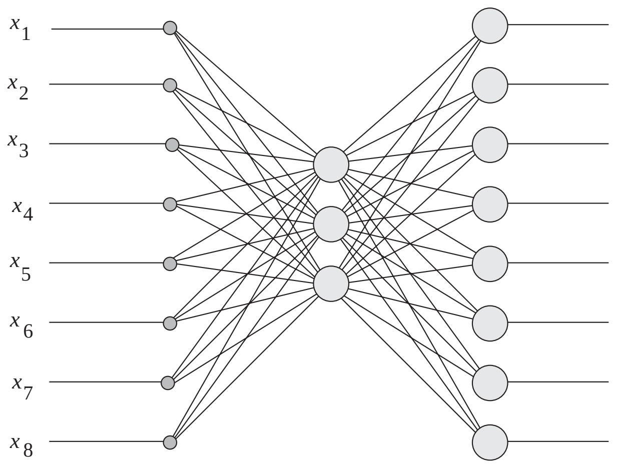

**Fig. 17.8.** An 8-bit encoder-decoder

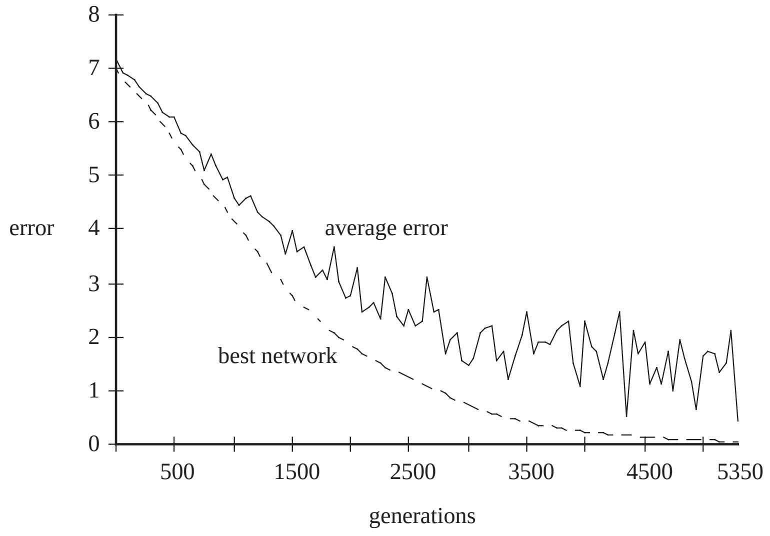

**Fig. 17.9.** Error function at each generation: population average and best network

Braun has shown how to optimize the network topology using genetic algorithms [69]. He introduces order into the network by defining a fitness function $f$ made of several parts: one proportional to the approximation error, and another proportional to the *length* of the network edges. The nodes are all fixed in a two-dimensional plane and the network is organized in layers. The fitness function is minimized using a genetic algorithm. Edges with small weights are dropped stochastically. Smaller networks are favored in the course of evolution. The two-dimensional mapping is introduced to avoid destroying good solutions when reproducing the population, as happens when the network symmetries are not considered.

### 17.3.3 Other applications of GAs

One field in which GAs seem to be a very appropriate search method is game theory. In a typical mathematical game some rules are defined and the reward for the players after each move is given by a certain pay-off function. Mathematicians then ask what is the optimal strategy, that is, how can a player maximize his pay-off. In many cases the cumulative effect of the pay-off function cannot be expressed analytically and the only possibility is to actually play the game and compare the results using different strategies. An unknown function must be optimized and this is done with a computer simulation. Since the range of possible strategies is so large only a few variants are tested.

This is where GAs take over. A population of "players" is generated and a tournament is held. The strategy of each player must be coded in a data structure that can be recombined and mutated. Each round of the tournament is used to compute the pay-off for each strategy and the best players provide more genes for the next generation. Axelrod [36] did exactly this for the problem known as the *prisoner's dilemma*. In this game between two players, each one decides independently whether he wants to cooperate with the other player or betray him. There are four possibilities each time: a) both players cooperate; b) the first cooperates, the second betrays; c) the first betrays, the second cooperates; and c) both players betray each other. The pay-off for each of the four combinations is shown in Figure 17.10. The letters C and B stand for "cooperation" and "betrayal" respectively.

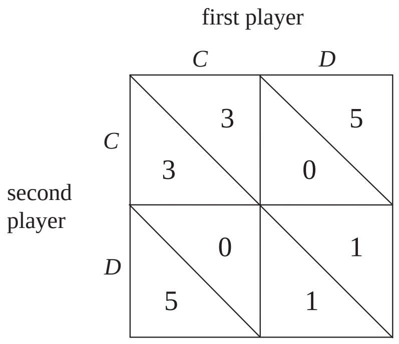

**Fig. 17.10.** Pay-off matrix for the prisoner's dilemma

The pay-off matrix shows that there is an incentive to commit treachery. The pay-off when one of the players betrays the other, who wants to cooperate, is 5. Moreover, the betrayed player does not get any pay-off at all. But if both players betray, the pay-off for both is just 1 point. If both players cooperate each one gets 3 points.

If the game is played only once, the optimal strategy is betraying. From the viewpoint of each player the situation is the following: if the other player cooperates, then the pay-off can be maximized by betraying. And if the other player betrays, then at least one point can be saved by committing treachery too. Since both viewpoints are symmetrical, both players betray.

However, the game becomes more complicated if it is repeated an indefinite number of times in a tournament. Two players willing to cooperate can reap larger profits than two players betraying each other. In that case the players must keep a record of the results of previous games with the same partner in order to adapt to cooperative or uncooperative adversaries. Axelrod and Hamilton [35] held such a tournament in which a population of players operated with different strategies for the iterated prisoner's dilemma. Each player could store only the results of the last three games against each opponent and the players were paired randomly at each iteration of the tournament. It was surprising that the simplest strategy submitted for the tournament collected the most points. Axelrod called it "tit for tat" (TFT) and it consists in just repeating the last move of the opponent. If the adversary cooperated the last time, cooperation ensues. If the adversary betrayed, he is now betrayed. Two players who happen to repeat some rounds of cooperation are better off than those that keep betraying. The TFT strategy is initialized by offering cooperation to a yet unknown player, but responds afterwards with vengeance for each betrayal. It can thus be exploited no more than once in a tournament.

For the first tournament, the strategies were submitted by game theorists. In a second experiment the strategies were generated in the computer by evolving them over time [36]. Since for each move there are four possible outcomes of the game and only the last three moves were stored, there are 64 possible recent histories of the game against each opponent. The strategy of each player can be coded simply as a binary vector of length 64. Each component represents one of the possible histories and the value 1 is interpreted as cooperation in the next move, whereas 0 is interpreted as betrayal. A vector of 64 ones, for example, is the coding for a strategy which always cooperates regardless of the previous game history against the opponent. After some generations of a tournament held under controlled conditions and with some handcrafted strategies present, TFT emerged again as one of the best strategies. Other strategies with a tendency to cooperate also evolved.

## 17.4 Historical and bibliographical remarks

At the end of the 1950s and the beginning of the 1960s several authors independently proposed the use of evolutionary methods for the solution of optimization problems. Goldberg summarized this development from the American perspective [163]. Fraser, for example, studied the optimization of polynomials, borrowing his methods from genetics. Other researchers applied GAs in the 1960s to the solution of such diverse problems as simulation, game theory, or pattern recognition. Fogel and his coauthors studied the problems of mutating populations of finite state automata and recombination, although they stopped short of using the crossover operator [139].

John Holland was the first to try to examine the dynamic of GAs and to formulate a theory of their properties [195]. His book on the subject is a classic to this day. He proposed the concept of schemata and applied statistical methods to the study of their diffusion dynamics. The University of Michigan became one of the leading centers in this field through his work.

In other countries the same ideas were taking shape. In the 1960s Rechenberg [358] and Schwefel [394] proposed their own version of evolutionary computation, which they called *evolution strategies*. Some of the problems that were solved in this early phase were, for example, hard hydrodynamic optimization tasks, such as finding the optimal shape of a rocket booster. Schwefel experimented with the evolution of the parameters of the genetic operators, maybe the first instance of metagenetic algorithms [37]. In the 1980s Rechenberg's lab in Berlin solved many other problems such as the optimal profile of wind concentrators and airplane wings.

Koza and Rice showed that it is possible to optimize the topology of neural networks [260]. Their methods make use of the techniques developed by Koza to breed Lisp programs, represented as trees of terms. Crossover exchanges whole branches of two trees in this kind of representation. Belew and coauthors [51] examined the possibility of applying evolutionary computation to connectionist learning problems.

There has been much discussion recently on the merits of genetic algorithms for static function optimization [105]. Simple minded hill-climbing algorithms seem to outperform GAs when the function to be optimized is fixed. Baluja showed in fact that for a large set of optimization problems GAs do not offer a definitive advantage when compared to other optimization heuristics, either in terms of total function evaluations or in quality of the solutions [40]. More work must be still done to determine under which situations (dynamic enviroments, for example) GAs offer a definitive advantage over other optimization methods.

The prisoner's dilemma was proposed by game-theoreticians in the 1950s and led immediately to many psychological experiments and mathematical arguments [350]. The research of Axelrod on the iterated prisoner's dilemma can be considered one of the milestones in evolutionary computation, since it opened this whole field to the social scientists and behavior biologists, who by the nature of their subject had not transcended the level of qualitative descriptions [104, 453]. The moral dilemma of the contradiction between altruism and egoism could now be modeled in a quantitative way, although Axelrod's results have not remained uncontested.

## Exercises

1. How does the schema theorem (17.1) change when a crossover probability $p_c$ is introduced? This means that with probability $p_c$, crossover is used to combine two strings, otherwise one of the parent strings is copied.
2. Train a neural network with a genetic algorithm. Let the number of weights increase and observe the running time.
3. Propose a coding method for a metagenetic algorithm that lets the mutation rate and the crossover probability evolve.
4. The prisoner's dilemma can be implemented on a cellular automaton. The players occupy one square of a two-dimensional grid and interact with their closest neighbors. Simulate this game in the computer and let the system evolve different strategies.