Genetic Algorithms in Machine Learning

Jonathan Shapiro

1 Introduction

1.1 What is a Genetic Algorithm

Genetic algorithms are stochastic search algorithms which act on a population of possible solutions. They are loosely based on the mechanics of population genetics and selection. The potential solutions are encoded as 'genes' — strings of characters from some alphabet. New solutions can be produced by 'mutating' members of the current population, and by 'mating' two solutions together to form a new solution. The better solutions are selected to breed and mutate and the worse ones are discarded. They are probabilistic search methods; this means that the states which they explore are not determined solely by the properties of the problems. A random process helps to guide the search. Genetic algorithms are used in artificial intelligence like other search algorithms are used in artificial intelligence — to search a space of potential solutions to find one which solves the problem.

1.2 Genetic Algorithms in Machine Learning

In machine learning we are trying to create solutions to some problem by using data or examples. There are essentially two ways to do this. Either the solution is constructed from the data, or some search method is used to search over a class of candidate solutions to find an effective one. Decision tree induction by ID3 and nearest-neighbour classification are examples of creation of a solution by construction. Use of a gradient-descent algorithm to search over a space of neural network weights to find those which affect a particular input-output mapping is an example of the use of search. Genetic algorithms are stochastic search algorithms which are often used in machine learning applications.

This distinction might not be strict; ID3 might include a search over different prunings, for example. Nonetheless, algorithms like ID3 are fast and computationally simple. In addition, there is usually an explicit simplifying assumption about the nature of the solutions. For example, in ID3 the bias is towards independent attributes; in nearest neighbour methods it is towards similarity of outputs being reflected in input similarity. Using search, however, a wide range of complex potential solutions can be tested against the examples, and thus, much more difficult problems can be tackled. The cost, of course, is in increased computational cost. In addition, any inductive bias is embedded in the algorithm and is thereby more difficult to control, although introducing explicit controls or regularizers is becoming increasingly important. In many respects, genetic algorithms are on the dumb and uncontrolled end of machine learning methods. They rely least on information and assumptions, and most on blind search to find solutions. However, they have been seen to be very effective in a range of problems.

Genetic algorithms are important in machine learning for three reasons. First, they act on discrete spaces, where gradient-based methods cannot be used. They can be used to search rule sets, neural network architectures, cellular automata computers, and so forth. In this respect, they can be used where stochastic hill-climbing and simulated annealing might also be considered. Second, they are essentially reinforcement learning algorithms. The performance of a learning system is determined by a single number, the *fitness*. This is unlike something like back-propagation, which gives different signals to different parts of a neural network based on its contribution to the error. Thus, they are widely used in situations where the only information is a measurement of performance. In that regard, its competitors are Q-learning, TD-learning and so forth. Finally, genetic algorithms involve a population and sometimes what one desires is not a single entity but a group. Learning in multi-agent systems is a prime example.

In artificial intelligence, search is used in reasoning as well as learning, and genetic algorithms are used in this context as well. To make the distinction clear, consider a chess-playing program. Machine learning could be used to acquire the competence of chess-playing. (Most chess-playing programs are not created that way; they are programmed.) However, when the program plays the game it also uses search to find a good move. Other examples include searching over a set of rules to evaluate a predicate. Genetic algorithms have been used for problems which have been in the domain of artificial intelligence, such as finding an effective timetable or schedule within a set of constraints.

1.3 History of Evolutionary Computing

Genetic algorithms are one of a class of approaches often called *evolutionary computation* methods used in adaptive aspects of computation — search, optimisation, machine learning, parameter adjustment, etc. These approaches are distinguished by the fact that they act on a population of potential solutions. Whereby most search algorithms take a single candidate solution and make small changes to that, attempting to improve it, evolutionary algorithms adapt an entire population of candidate solutions to the problem. These algorithms are based on biological populations and include selection operators which increase the number of better solutions in the population and decrease the number of the poorer ones; and other operators which generate new solutions. The algorithms differ in the standard representation of the problems and in the form and relative importance of the operations which introduce new solutions.

Genetic algorithms were first proposed by John Holland in his book "Adaptation in Natural and Artificial Systems" [14]. Holland guessed that an important feature which distinguished natural adaptive systems like the biological systems from artificial ones like perceptrons (this was the early 70's remember) was that in biological systems there is only an effective solution when all of the necessary elements are in place, a property which he deemed "epistatis". (In biology, genes encode proteins which catalyse reactions, and typically all of the reactants must be present. Epistatis is a biological term to describe the situation where many genes contributed to a single trait.) Holland tried to create an artificial procedure which simulated this, one component of which was what is now called a genetic algorithm with crossover. He emphasised crossover, because he believed that this was the ideal way to search over systems containing epistatis. During a similar period, L. Fogel and his collaborators [9] used selection and mutation to search over finite-state automata as solutions to a range of problems. These studies are described in the book, "Artificial Intelligence through Simulated Evolution". Around the same time, Richenberg [26] developed a method called Evolutionary Strategies in which solutions to a problem were represented as real numbers; these were selected and mutated with Gaussian noise. He applied this technique to engineering problems including the design of airfoils. More recently, John Koza [17, 18] developed Genetic Programming, in which genetic algorithms are used to search parse trees of LISP expressions, essentially computer programs.

These are the four so-called evolutionary programming paradigms. Genetic algorithms and their extension to genetic programming are probably the most widely used and most important in machine learning. However, there are still proponents of evolutionary strategies, notably Schwefel, Back, Hoffmeister and others who have achieved interesting results.

2 The Basics of Genetic Algorithms

2.1 Elements of a Simple Genetic Algorithm

Representation In order to use a genetic algorithm to search for solutions to a problem, potential solutions to the problem must be encoded as strings of characters drawn from some alphabet, $A = a_1 a_2 ... a_L$. Often the characters are binary, but it is also possible to draw them from larger alphabets, and real numbers are sometimes used.

In describing genetic algorithms, one often uses jargon taken from the population dynamics. The strings, A are called chromosomes. The components of the string, $a_1, a_2, ...$ are called genes. The values which each gene can take are called the alleles. In the most common case of binary strings, the two possible alleles are 0 and 1. So, for example, the chromosome $A = 11110000$ the first four genes take the allele 1 and the last four take the allele 0.

Fitness Function It is the task of the genetic algorithm to find very good strings. The goodness of a string is represented in the GA by some function of the strings, which I will call the *objective function*, the quantity to be optimised. Another function which is needed is called the *fitness function*. This is a monotonic function of the objective function, and is what determines how the strings will be multiplied or discarded during the algorithm. It is usual to define the fitness function so that the algorithm *increases* the fitness of the strings, so better solutions correspond to more fit strings.

The *genotype* is the string representation. The *phenotype* is what the genotype produces. Often in genetic algorithms, the two are not terribly different, although there is a great difference between a biological organism (the phenotype) and its DNA (the genotype). The objective function is a kind of phenotype. The objective function and the fitness function are often used interchangeably in the GA community, but it is useful to separate them. In the biological terminology, the fitness is the increase due to selection. Thus, the fitness has to do with the particular form of selection, whereas the objective function is a property of the problem to be solved.

Population Dynamics A genetic algorithm works on a population of strings. It starts with a random population and evolves the population to one containing (hopefully) good strings. At the heart of the algorithm are operations which take the population at the present generation, and produce the population at the next timestep in such a way that the overall fitness of the population is increased. These operations are repeated until some stopping criterion is met, such as a given number of strings been processed, a string of given quality has been produced, etc. There are numerous ways of actually implementing a genetic algorithm.

The simple genetic algorithm we will consider works with a population size, N, which is held constant. Three operations take the population at generation t and produce the new population at generation $t+1$: selection, crossover, and mutation. These are described below.

Selection: This operator selects strings from the current population such that better strings are more likely to be chosen. Selection does not introduce any new strings; if applied repeatedly it simply re-proportions the strings in the population, increasing the number of fitter ones and decreasing the number of less fit ones.

Recombination/Crossover: This operation produces two (or some other number) of offspring from a pair of parents by combining parts of each parent. Each offspring contains parts of the strings of each parent.

Mutation: This operation simply changes characters in the offspring string at random. Each character is changed with a small probability.

2.2 A Simple Genetic Algorithm

There are a number of different variations. Here I show the simplest form.

Dynamics

A simple realization is the following,

```
current_population=N random strings.
repeat
   repeat
      Select 2 strings from current population using selection operator.
      Produce 2 offspring using recombination.
      Mutate those 2 offspring.
      Add the 2 offspring to new_population.
   until size_of(new_population) = N.
   current_population = new_population.
until (stopping criterion met)
```

Such an approach is called "generational" because it is divided into distinct generations during which the entire population changes. In addition, offspring do not compete with parents. Alternatives are to change just a few of the strings in each time step, or to generate a large "mating pool" using recombination, including parents, and select N from that to get the next generation.

Fitness Proportional Selection

The simplest implementation of selection is to select in proportion to the fitness. That is, the probability of selecting a string α is

$$p^\alpha = \frac{F^\alpha}{\sum_{\alpha'} F^{\alpha'}} \tag{1}$$

where F^α is the fitness of the string α. Often, this is taken to be the objective function E^α itself. This works if E is positive and is to be maximised. Another approach is called Boltzmann selection, where

$$F(E^\alpha) = \exp(sE^\alpha). \tag{2}$$

This has a selection strength parameter s.

Fitness proportional selection is also called "roulette wheel selection". This is in analogy with a biased roulette wheel, associated with each string is a wedge of the wheel, whose size is proportional to the fitness of the string. Selection is equivalent to a spin of the wheel.

There are are two problems with roulette wheel selection. First, the randomness does not introduce new individuals; it only removes individuals from the population. Although it is most likely that unfit members of the population are removed, it is also possible that fit members are removed by chance. It is often better to remove this randomness and put the strings into the next population deterministically in proportion to their fitness.

The second problem is that as the search evolves, the spread of fitnesses decreases. There is less diversity in the population. This means that selection has less effect later in the search. To counteract this, the fitness is often rescaled to keep the spread constant. For example, one can rescale the selection strength by dividing by the standard deviation of the fitness within the population at each generation [28].

Mutation The mutation operation changes the value of each gene with a mutation probability p_m. I.e. each character is changed to some other character with probability p_m, and left unchanged with probability $1-p_m$. The mutation probability is taken to be small, e.g. $1/L$, for strings of length L. This probability is small so that mutation can explore small changes in the solutions without disrupting good solutions too much.

Crossover Crossover works on pairs of strings. For each pair crossover occurs with probability p_c and with probability $1-p_c$ the offspring are identical to the parents. Typical value is $p_c \approx 0.6$.

A commonly used form is called single-point crossover. In this operation, a crossing site is chosen at random somewhere along the length of the strings. Two new strings are generated by swapping all of the components of the two strings on one side of the crossing site. For example, if the two strings are

$$A_1 = 1111111|1111111$$

$$A_2 = 0000000|0000000$$

and the crossover position is between the 7th and 8th position (as marked), the new strings would be

$$A_1' = 0000000|1111111$$

$$A_2' = 1111111|0000000$$

This operation effectively means that each offspring possesses features from both parents.

Likewise, multi-point crossover uses multiple crossover points. I.e.

$$A_1 = 111|111111|11111$$

$$A_2 = 000|000000|00000$$

would yield

$$A_1' = 000|111111|00000$$

$$A_2' = 111|000000|11111.$$

A most extreme form of crossover is uniform crossover, where each site is independently chosen from parent 1 or parent 2. This is a very effective search operator if the genes don't interact in producing the fitness, but is very disruptive when they do.

Other forms of crossover have been invented for use in specific problems or domains.

Expected behaviour

To see how the genetic algorithm can be useful, consider the effect of mutation and crossover. Mutation is clearly local random search, it takes a solution and generates a small, unguided change. Crossover is more complicated. It takes a pair of parents and produces two offspring between the parents (think of what it does to binary numbers). The two offspring can be near the two parents, or they can be far from them, in this sense it is said to be a global rather than local search algorithm.

One view of what crossover is meant to do is to combine partial solutions to form more complete solutions. To use a ridiculous example, if you are trying to produce a duck and one parent has developed wings, while the other has developed webbed feet, the crossover operation should combine these so that one offspring has *both* webbed feet and wings and the other offspring has neither. Goldberg [10] writes about the *the principle of meaningful building blocks* — the problem should be encoded in such a way that substrings or partial solutions with intermediate fitness can combine under crossover to produce fitter, more complete solutions. Whether this will actually occur depends upon the likelihood of both building blocks being in the population simultaneously, and the likelihood of combining them in that way. Others have argued that crossover is useful because it allows large moves.

One expects the mean fitness in the population to increase until some form of equilibrium is reached. Equilibrium is were there is a balance between the improving effect of selection and the (generally) deleterious effects of mutation and crossover. Selection tends to make the population better on average, but decreases the variation. The other operators tend to decrease the quality of the population on average, but restore diversity. Figure 1 shows histograms of the fitness distribution in a population during a GA evolution.

One thing that often happens is that a state is reached where there is very little diversity in the population. When this happens, improvement is very slow because crossover has little effect on very similar strings. This is called "premature convergence" and is one of the most important causes of failure of genetic algorithms. A population of very similar strings in the vicinity of a local minimum is very stable. Crossover has very little effect on populations of very similar strings and the time to leave a local minimum via selection and mutation alone can be very long. Exploration is inhibited by competition with strings in the vicinity of the minimum [29].

Figures 1 and 2 show examples of the increase in mean fitness in a population for a short and longer run. The improvements come in levels, or epoch. Had premature convergence occurred, the population would have remained stuck in the first level for a long time.

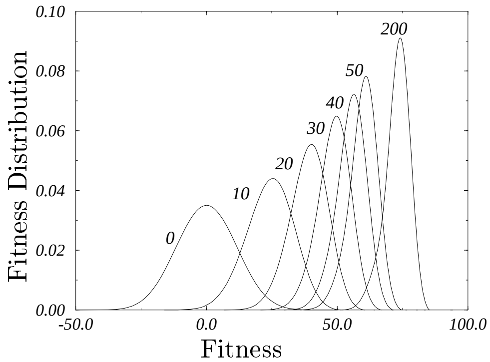

Figure 1: The distribution of fitnesses in a population for different generations of a genetic algorithm. The x axis is the fitness value; the y axis is the fraction of the population with that value. Computed from histograms which have been averaged over many runs of the genetic algorithm. The decrease in variance as the algorithm runs is apparent. It is also the case that more of the variance is on the low fitness side; the distribution has positive skewness.

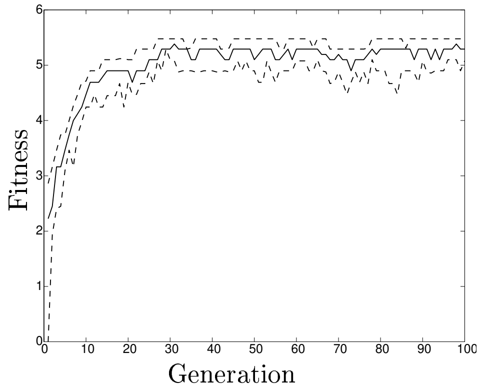

Figure 2: Evolution of the mean fitness and the 10 and 90 percentile on a simple problem. An apparent stable population has been reached.

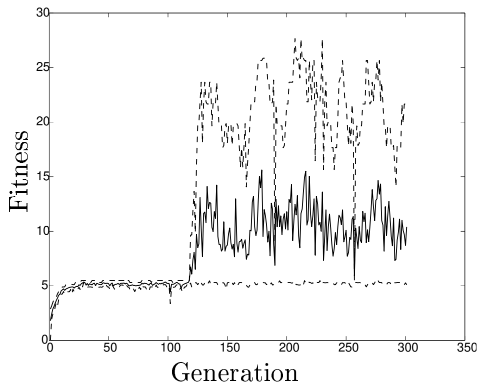

Figure 3: The same as the previous figure for a longer run. Punctuated equilibrium is apparent. Had premature convergence occurred, the population would have been like in the previous figure.

Example – Colouring a Map or Graph

Here is a simple example problem showing how a genetic algorithm is applied to a simple optimisation problem. Suppose you have a map that you want to colour with three colours. By this I mean that you want to assign one of three colours to each country such that adjacent countries are coloured with a different colour. A related problem is graph colouring - you assign colours to vertices of a graph so that adjacent vertices have a different colour. Say the colours are red, green, and blue.

In order to devise a genetic algorithm to solve a problem, there are basically three things you must do:

1. Encode possible solutions to the problem as strings
2. Devise a fitness function which determines how good a solution is. Generally, this should be positive and increasing with the quality of the solution.
3. Decide on the forms of the genetic search operators to be used, selection, mutation and crossover.

Encoding: Let each gene of the string represent one of the countries. Each gene can take three values: red, green, or blue. For example, one possible string might be

$$A = rbrgrbgg.$$

This would be a map in which the first country is coloured red, the second is blue, the third is red, and so forth.

Objective function: The obvious cost function is the number of times that two neighbouring countries have the same colour. This could be written,

$$E = \sum_{ij} W_{ij} d(a_i a_j)$$

where $W_{ij} = 1$ if country i neighbours country j and 0 otherwise; and $d(a_i a_j) = 1$ if $a_i = a_j$ and 0 otherwise.

Fitness Function: However, this does not work as a genetic algorithm fitness function, because we would want to minimise this, while genetic algorithm maximises the fitness function. We also could not use $-E$ since the fitness function should be positive. So, add to $-E$ the smallest value which insures that the fitness is always positive. What works as the fitness function is,

$$F = N(N-1) - \sum_{ij} W_{ij} a_i a_j.$$

Another possibility is to use

$$F = \exp\left(-\beta E\right),$$

where β is a selection strength parameter. This works because exp (anything) is always positive. This is called "Boltzmann selection". It has the advantage that it has a parameter, β which controls the strength of selection.

Genetic Operators: The usual selection, single-point crossover, and mutation could be used.

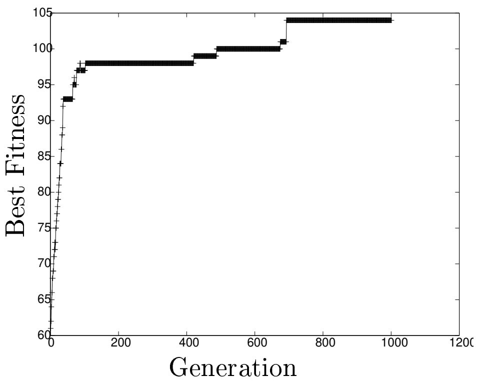

Figure 4: Evolution of the best member of the population for a GA attempting to 2-colour a map. Premature convergence has occurred; the final population contains strings which differ by only a few mutations, are within a few mutations of a local optimum, and are many mutations away from the global optimum.

2.3 Extensions to the Basic Genetic Algorithm

There are many different ways of implementing a genetic algorithm, and researchers rarely use the simple form described above. Below some extensions are outlined.

2.3.1 Alternative Selection Schemes

Most alternative selection schemes are designed to avoid premature convergence. Here is a list:

Elitism: The best members of the population are put into the next generation at each timestep. The best individuals cannot be lost due to random sampling, and selection strength used to pick the other members of the population can be weaker to maintain diversity.

Tournament Selection: Two members of the population are chosen at random. The least fit is removed and replaced by a copy of the most fit.

Steady-state selection: Rather than changing all members of the population at each stage, only a few members of the population are changed at each timestep.

Fitness Sharing: In order to maintain diversity in the population, the fitness of an individual is shared among all other individuals which have similar genotypes. So a string which is very different from others in the population has its fitness enhanced, while strings which are very similar to many others have their fitness decreased.

A related method is called "implicit fitness sharing". This can only be applied in applications where the task is to perform well on a set of examples, such as in classification tasks. Fitness sharing is implemented by having the fitness shared among all of the solutions which classify a given example correctly, summed over all examples. Two strings which perform well on a overlapping set of examples would have their fitness reduced, while a string which performs well on examples not correctly dealt with by other strings would have a high fitness.

For a comparison of fitness sharing to implicit fitness sharing, see [7].

Niching and Island Models: This is similar to the previous; it is a method of maintaining diversity in a population and preventing premature convergence. Here a population is divided into several subpopulations, each evolving independently for periods of time. If the subpopulations converge, it is likely to be to different optima. Occasionally, some members of the subpopulations are exchanged. This injects diversity into the subpopulations. A useful feature of this approach is that it can be implemented on parallel computers. Different subpopulations evolve on different processors, and there is only occasional need for inter-processor communication.

2.3.2 Alternative Crossover Operators

Cutting the solutions up into chunks and recombining them may not be the best way of creating offspring. The method of recombination should reflect the structure of the problem. This has been discussed in a formal sense by [25]. A well-studied example is that of the Travelling Salesman Problem. In this case, a potential solution is a permutation, e.g. a potential solution might be 364521 which means visit city 3 first, 6 second, etc. Single point crossover between two such lists will fail to produce even a valid tour, much less a good one. To see more about special operators for the travelling salesman problem, see for example, [8].

2.3.3 Using the entire population

In most uses of genetic algorithms, the "answer" is the best member of the population; the other members of the population are used to help in the search but are discarded at the end. However, there are some exceptions.

In the early work on genetic algorithms in machine learning, Holland developed the Learning Classifier System [15]. In this, each string of the genetic algorithm was one "classifier", an if-then rule. The population was the system of classifiers. This approach is called the "Michigan" approach. The alternative approach, where each member of the population is a complete solution to the problem, as in DeJong et. al.'s GABIL system, is deemed the "Pittsburgh" approach.

Another approach in which the entire population is used rather as a solution is that proposed by Yao and Liu [34]. Here a genetic algorithm is used to train a neural network. It has been known in the neural network community for some time that if there exist a number of classifiers to solve the same problem, a classifier consisting of a committee of all of the classifiers can give better expected generalisation than any one of the classifiers taken alone (see, for example, subsection 9.6 of [6]). Yao and Liu propose using the final population of neural networks as the committee, and propose four different methods for combining the output of the neural networks. They use implicit fitness sharing, so that different learners are encouraged to learn different examples.

3 Examples of Genetic Algorithms in Machine Learning

3.1 Learning Rule sets

A simple example of the use of a genetic algorithm to induce a set of rules was described by DeJong et. al. and is called GABIL. The genetic algorithm learns a boolean concept by searching for a disjunctive set of propositional rules. To represent a rule, a string contains a bit for each value that each variable can take, and a bit for the concept value. For example, if v_1 can take the values V_{11}, V_{12}, V_{13}, and v_2 takes the values V_{21}, V_{22}, and these are the only two variables, then the string

```
110 01 1
```

is used to denote the rule

$$\text{IF } (v_1 = V_{12}) \vee (v_1 = V_{13}) \wedge (v_2 = V_{21}) \text{ THEN } c = T, \tag{3}$$

where c is the concept. So if the substring associated with a particular variable is all 1's, that variable can take any value.

A set of rules is a long string of rules concatenated together. In order to allow for a variable number of rules, a modified form of crossover is used. Two crossover points are selected at random in the first parent. The crossover points in the second parent can be in any rule, but must be in the analogous place in the rule as in the rule of the first parent. For example, if parent one consists of three rules as represented by the following string and the crossover points are as shown,

```
v_1   v_2 c    v_1 v_2 c    v_1   v_2 c
00|1  11   1    111 0|1  0    110 10  1
```

then possible crossover points in the second parent must be between the first and second attribute bit in a substring associated with v_1 in any rule, and the first and second attribute bit in a substring associated with v_2 in any rule. For example,

```
v_1   v_2 c    v_1 v_2 c
10|1  1|1  1    001 01  0
```

would yield as offspring,

```
v_1   v_2 c    v_1 v_2 c
001   11  0     110 10  1
```

and

```
v_1   v_2 c    v_1 v_2 c    v_1   v_2 c
101   11  1    111 01  1    001    01 0
```

This allows varying lengths of rule sets to be explored, while preserving the structure of the rules.

It is worth noting that bits with more 1's are more general rules, and bits with more 0's are more specific. Thus, a mutation operator which is biased towards adding more 1's than 0's will generate more general rule sets. DeJong et. al. exploited this fact by experimenting with operators which changed a 0 to a 1 in a variable substring. An additional operator changed all of the bits to 1's in a substring associated with a variable in a rule (thereby exploring the possibility that that variable is irrelevant). The objective function was taken to be the square of the number of examples correctly classified by the rule set. In experiments comparing GABIL to over rule induction algorithms such as AQ14 and ID5R and C4.5, comparable results were achieved. Significant improvements were found by introducing the biased mutation operators mentioned in the previous paragraph.

Another example of use of genetic algorithms to evolve rule sets is Holland's Learning Classifier System [15].

3.2 Evolving Computer Programs

3.2.1 Genetic Programming

Genetic programming [17, 18] is a form of evolutionary computation in which computer programs are evolved using a genetic algorithm. John Koza is credited with its invention. The representation of the program is in terms of trees rather than strings. Each tree represents the parse tree of a LISP expression. The leaves (or end nodes) of the tree represents terminals — objects which take no input. The other nodes represent functions which take a number of arguments, where each argument is either a terminal or another function. In order to apply this to any problem, the appropriate functions and terminals must be chosen. So, for example, in order to represent the absolute value function, the function set could be `IF`, which takes three arguments and evaluates the second (third) argument if the first argument evaluates to TRUE (FALSE), and `GT`, which takes two arguments and returns TRUE if the first argument is greater than the second. The terminal set would be real variables x and real constants. The LISP expression,

```
(IF (GT (x) (0))
    (x)
    (-x))
```

would be represented by the tree shown in figure 5.

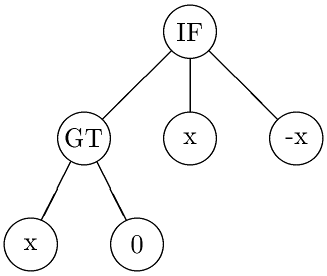

Figure 5: A parse tree representation of the absolute value function.

As the representation is in terms of trees, a tree-based crossover and mutation operator is required. The crossover is done by replacing one randomly chosen subtree from one parent by a subtree from the other parent. This is shown in figure 6. Mutation occurs by taking a randomly chosen subtree and replacing it by a randomly generated subtree. Unlike standard genetic algorithms, this process generates varying sized structures. Some mechanisms might be required to regulate the size of the trees generated, and to insure that objects of the appropriate type go into the functions.

Koza [17] uses fitness proportional selection with 90% of the trees generated from the crossover operator and the remaining unmodified from the previous generation. Koza considered mutation to be of minor importance and often did not use it.

Example - A simulated ant A simple example described in [17]. A simulated ant can detect food directly in front of it and can move forward, or turn left or right on a square grid. There is a trail of food along the grid, with some gaps in the trail. The ant has to collect as much food as possible. Where the trail of food is continuous, the ant must learn to follow the trail. Since there are gaps in the trail, the ant must learn some exploration behaviour, so as to navigate past the gaps and find the trail again. The complexity of the exploration required depends upon the size of the gaps.

To solve this using genetic programming, Koza used three terminals corresponding to the actions of the ant, `MOVE`, `LEFT`, and `RIGHT`, meaning: move forward, turn left, and turn right respectively. `IF-FOOD-AHEAD` is a function with two arguments. The first is evaluated if food is detected ahead and the second is evaluated if food is not. For example, the program

```
(IF-FOOD-AHEAD (MOVE) (LEFT))
```

moves forward if food is detected and turns left if it is not. Two other functions were used. `PROG2` takes two arguments, evaluates the first and then evaluates the second. So the program

```
(PROG2 (MOVE) (LEFT))
```

simply moves then turns left. A similar function, `PROG3`, takes three arguments and evaluates them in turn.

The fitness of a program is the number of pieces of food found by the ant in a given number of time steps (to prevent random exploration solutions). Each program is evaluated fully, then re-executed repeatedly until the maximum amount of time is used up. A population of 500 was used.

Koza found a solution which worked perfectly on one particular trail used for training after 21 generations. It is

```
(IF-FOOD-AHEAD (MOVE)
               (PROGN3 (LEFT)
                       (PROGN2 (IF-FOOD-AHEAD (MOVE)
                                              (RIGHT))
                               (PROGN2 (RIGHT)
```

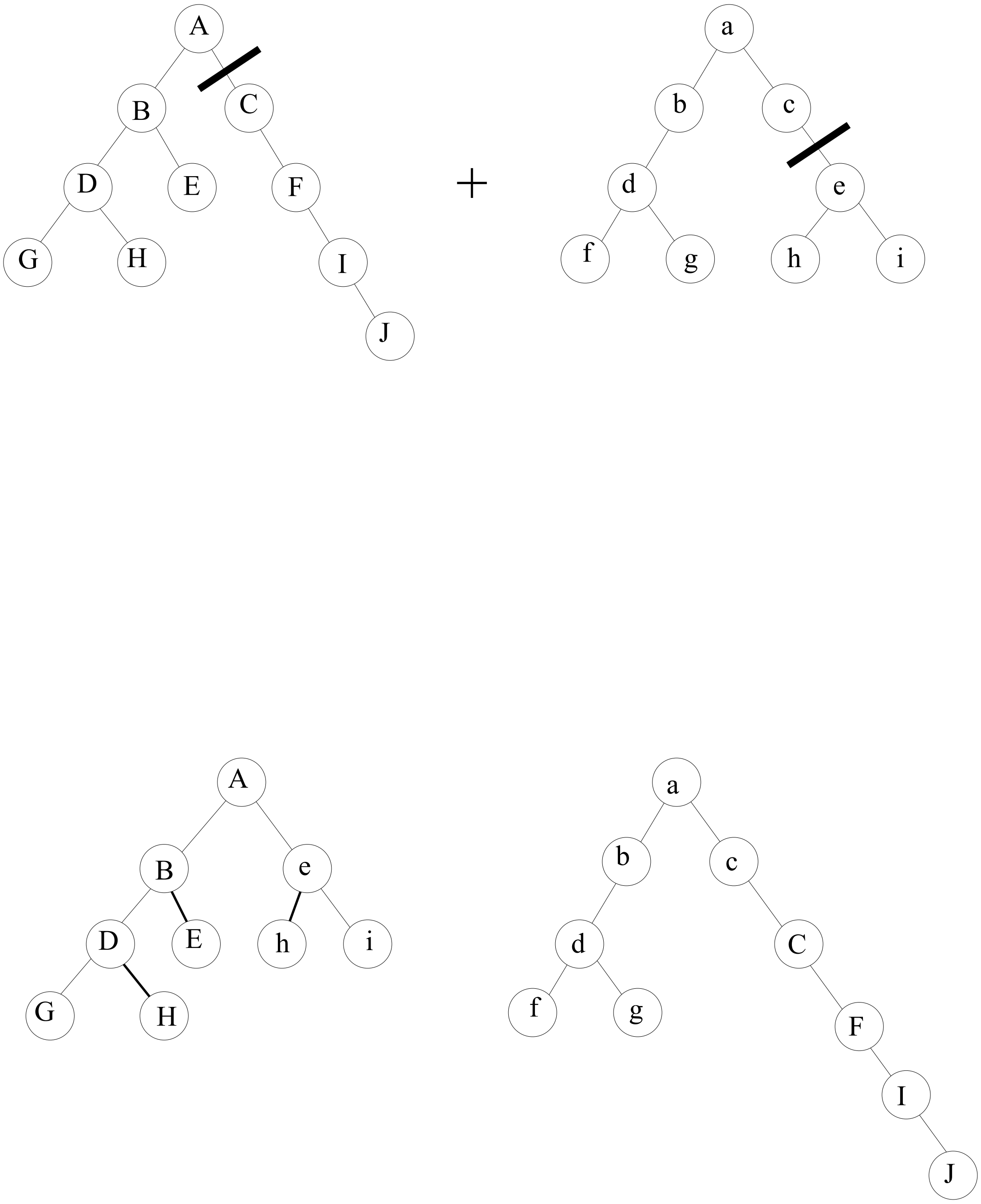

Figure 6: An example of crossover on trees. The two parents are on the top of the figure; the dark line shows the crossover points. The two offspring are below.

```
                              (PROGN2 (LEFT)
                                      (RIGHT))))
               (PROGN2 (IF-FOOD-AHEAD (MOVE)
                                      (LEFT))
                       (MOVE))))
```

If food is detected ahead the ant moves to it. Otherwise it turns left and looks for food. If no food is there, it turns back to the right and the right again, and so looks to the right of its original position. If no food is there, it just moves in the direction it was originally heading. There is also a pointless but harmless step in which it moves to the left and then to the right resulting in no change in heading. There is nothing about this rule induction system which prevents redundant rules like this.

Genetic programming has been used in a wide range of applications, including the design of circuits [19], and the prediction of complex time series.

3.3 Genetic Algorithms and Neural Networks

Artificial neural networks are very successful methods in machine learning. In their simplest form, they produce a mapping between an input space and an output space. This mapping is not represented in terms of rules, but in terms of non-linear functions of real-valued parameters called "weights". The mapping is determined by the structure of the non-linear functions, and given that structure by the values of the weight parameters. Two learning tasks are: find the structure of the neural network, and find the weight values of the neural network to make it perform a particular task.

Often neural networks are used for *supervised learning*. In this situation, the correct outputs are known for some example inputs. Then the learning problem is to find the best neural network given these examples. By "best neural network" we could mean the neural network which does the best job of producing the correct outputs on the example inputs, or some other criterion could be used. For example, one often wants to balance performance on the example set with other criteria, such as model simplicity, in order to improve generalisation performance. A standard approach is to choose the network structure by hand, and then use some real-valued optimisation method using gradient information (e.g. a conjugate gradient method) to search for the weights which optimise the chosen criterion.

Numerous authors have used genetic algorithms to search over the structure space and/or over the weight space to train neural networks to perform different tasks. For reviews, see Schaffer et. al. [27] or Yao [35].

3.3.1 Genetic Algorithms to Train Fixed Structures

A number of authors have explored replacing the standard, gradient-based, algorithm with a genetic algorithm to search for the weights which optimise neural network performance on an example set for a fixed structure neural network.

This appears to be a questionable enterprise for several reasons. First, it is a long-held belief in the optimisation community that if you have gradient information you should use it [24]. Second, it is worth noting that genetic algorithms turn all learning problems into reinforcement learning tasks. The performance of the network is defined by a single number, the fitness (unless a multi-objective approach is used, which has not been tried to the best of my knowledge). If there are multiple outputs, a gradient-based approach can focus the search on those parameters which contribute to the outputs which are wrong, and if learning is done after each pattern, the changes can be made which are associated with the patterns which the network gets wrong. Experience with reinforcement learning as compared to supervised learning using standard algorithms supports the view that reinforcement learning requires longer search, so it seems questionable to turn the problem into a reinforcement learning problem.

The argument in favour of evolutionary algorithms over gradient-based ones is that evolutionary algorithms are global search algorithms, they can in principle search the entire space eventually. Local search algorithms can get stuck in local minima. Thus, in search problems dominated by many local minima, global search methods such as simulated annealing or evolutionary algorithms are used.

An early example of this approach was described by Montana and Davies [22]. They found that a genetic algorithm using simple mutation and uniform crossover on the weight vectors for an underwater sonar classification task outperformed the gradient descent algorithm. However, it did not outperform a simple modified algorithm (quickprop) [27], and it is likely to seriously under-perform compared to more sophisticated versions of gradient descent.

3.3.2 Genetic Algorithms to search for Neural Network Structure

A well known problem in using neural networks is that one does not know what structure to use. A structure which is too complex may easily perform well on the example set without giving good generalisation; a structure which is too simple may be unable to perform the desired task at all. Although methods of automatically growing or pruning network architectures have been known for some time [6], they are not widely used (cascade correlation may be an exception). A more standard approach is to train networks of different architectures, and use cross-validation to determine the expected generalisation of networks of different structures. Then one of the networks from the class with the best expected structure is tested and used. This evolves training an ensemble of networks of different structure (if cross-validation is used to estimate the expected generalisation, many networks of each structure are trained as well). Either only one of these is used, or a committee is formed from a set of them[6, 30].

An alternative approach is to use an evolutionary algorithm to search over different network architectures. In this approach, the weight adaptation is done using a gradient method; the evolutionary search is adapting the structure of the network, either the number of neurons or the connection topology between them. A straight-forward approach is described in [35], and called EPNet. A population of neural networks learns weights using a hybrid adaptive gradient-descent/simulated annealing algorithm. The population undergoes mutations of various types; no crossover. The mutations grow and prune the network. Each generation consists of a fixed application of the hybrid algorithm for a fixed number of computation steps, and then one of four mutations — delete a node, delete a connection, add node, add connection. The mutations are all tried, the first one to lead to improvement is chosen. Deletions are tried first to bias the system towards simple networks. Three example sets are used. The first is used by the hybrid algorithm to train the weights; the second is a validation set used to compete the fitness of the neural networks in the population for selection. The final one is used to select the member of the final population to use as *the* network.

The above approach does not include crossover. Once crossover is included, a string representation of the network architecture is required. There are two approaches to this. The first is called *direct encoding*, where the structure of the neural network is represented directly in the strings. The alternative is to encode in the strings some dynamical process which produces the strings, such as a *grammatical encoding*, in which the genetic algorithm evolves a grammar which is then used to produce a network topology [16]. Another example is a genetic string which produces a set of developmental instructions [21]. These have been mostly applied to toy and test problems.

There is a problem to this approach, which is sometimes called the problem of competing conventions. There are a number of structures by which a neural network can solve a problem, many related by symmetries. For example permutation of the labels of the neurons in a layer of a neural network leaves the network functionally unchanged. A population can consist of networks which differ in the symmetry of the solution they are learning towards. Crossover between such neural networks is very disruptive.

3.4 Genetic Algorithms and Reinforcement Learning

Reinforcement learning deals with those situations when the information the learner gets about its performance is evaluative, rather then instructive. In other words, it only knows how well it is doing, not what it should or should not be doing, and perhaps the performance information is available only sometimes. Learning to talk might be an example of this. No teacher is available to instruct on how to move the muscles of the mouth to make a particular word, but if the child can successfully produce the word “up”, it gets a reward (picked up by its mother). If it fails, it gets no information about which part of the sequence of movements it made was wrong. Reinforcement learning is important in adaptive control problems, such as learning to control a mobile robot, learning to play games, and modelling learning in biological systems. The simulated ant discussed in section 3.2.1 was an example of a reinforcement learning problem, since the only feedback provided to the ant is the amount of food picked up after a long sequence of actions.

As mentioned previously, genetic algorithms turn learning problems into reinforcement learning problems, and is perhaps one of the simplest ways to implement a reinforcement learning method. Thus, reinforcement learning is one of the most important applications of genetic algorithms in machine learning. Examples of this include studies of reinforcement learning in simulated agents [1], and training of robots using evolutionary algorithms.

4 Interaction of Evolution and Learning

4.1 Combining Genetic Algorithms with Local Search Algorithms

Many authors have used a local search algorithm combined with an evolutionary algorithm to improve search. The idea is that the evolutionary steps move members of the population near locally optimal solutions, and the local search algorithm takes it the rest of the way. See, for example Ackley and Littman [1] and Belew [4]. Alternatively, one could use the local algorithm as the main search element of the population, adding occasional crossovers as a global element to move individuals away from locally optimal states.

Once local search is added to the genetic algorithm a question arises, should the result of the local search change the strings or not. In other words, should evolution be Lamarckian or Darwinian? Lamarck believed in inheritance of acquired traits, e.g. the giraffe has a long neck because its ancestors stretched their necks to reach the tall leaves, and their offspring inherited the longer necks. Weismann [32] argued that this is impossible for biological organisms; there was no way for information to pass from the somatic cells which make up the body of the organism to the germ cells which carry the inheritable genetic material (Darwin argued that this was unnecessary to understand evolution, but did not rule it out).

Although it is impossible for biological organisms, computer programs can be Lamarckian. There is some evidence that they should be. Hart and Belew [11] used a genetic algorithm, a variety of local optimisation approaches, Lamarckian combinations of genetic algorithms with local optimisation and non-Lamarckian combinations of genetic algorithms with local search on a test optimisation problem containing a large number of local optima. They found that the Lamarckian methods were significantly better than all other methods in most of the trials.

Their interpretation of the results was that initially the GA population is distributed across the search space. A normal GA, however, would become centred around one good optimum as the population converged. Combining the GA with a high number of local search steps per generation prevents that, as all members of the population are centred around optima. The Lamarckian aspect enhances this. However, if the population starts to converge, the opposite is true, the Darwinian algorithm is better. Hart and Belew found that for infrequent local searches, the Darwinian algorithm performs better. Why this might be is shown in figure 7. When the population has converged around a local optimum, genetic algorithm search is particularly slow, as discussed in [29].

However, using local searches without modifying the genotype gives all of the points within a basin of attraction of an optimum effectively the same fitness. This allows a more diverse population which can move between optima much more readily. This idea has been explored for simulated annealing in the *basin hopping algorithm* and is discussed in [31].

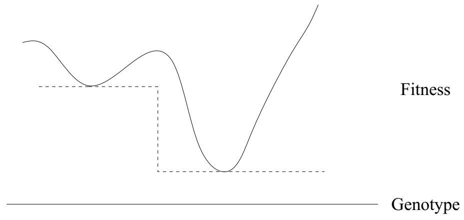

Figure 7: A fitness function (solid line) and the effective fitness (dashed line) when a genetic algorithm is combined with local search in a Darwinian manner. Every point within the basin of attraction of a minimum has the same effective fitness.

Other studies of this effect are presented by Whitley et. al. [33].

4.2 The Baldwin Effect

One effect of the interaction of learning with evolution was first proposed by J. M. Baldwin in 1896 [3]. He argued that acquired characteristics could be indirectly inherited. This occurs in two steps. First, plasticity can increase fitness of partial mutation which is otherwise useless. For example, an organism which lives in a hot climate may be born with structures which protect it from the sun, or it could have skin pigments which can react by tanning. Likewise, an organism can be born with the ability to hunt, or it can be born with the ability to learn to hunt. The tanning reaction and learning are both examples of plasticity which impart increased fitness.

However, plasticity has a cost. While an organism is learning to hunt, it may go hungry or even die. So the second step of the Baldwin effect is genetic assimilation. The adaptation becomes genetically encoded and innate. Thus, through this two step process, acquired characteristics are inherited, but without violating the impossibility of information transfer back to the genetic material.

A model of this was proposed by Hinton and Howlan [13] The genotype is a string of 20 characters drawn from the alphabet, $\{0, 1, ?\}$. The phenotype is a neural network with weight values of 0 or 1 assigned from the genotype. The weights associated with the ?'s are plastic, random search assigns 1 or 0 to those over 1000 trials. The ideal phenotype has all weight values equal to 1, any other phenotype is not useful.

The fitness of the individual is

$$F = 20 - \frac{19}{1000} \min\left(\text{number of trials to find ideal, } 1000\right). \tag{4}$$

So if you are born with the ideal phenotype, the fitness is 20. The fitness decreases linearly with the time for learning to find the ideal phenotype, and is 1 if learning fails to find the solution in 1000 trials. Learning has a cost.

Without learning, the genetic search is a needle in a haystack problem, and a solution is never found. With learning, a solution is found and the fitness of the population increases over time. One also sees that the number of ?'s in the population decays slowly over time; genetic assimilation does occur, albeit slowly. Thus, learning guides evolution. With plasticity, evolution can work even where only a perfect solution imparts fitness advantage. This is important, because it was often hard to see how evolution could work otherwise to evolve structures which are useless unless complete. This model has been studied further by Harvey [12] and Mayley [20].

5 Conclusions

Genetic algorithms are general purpose search algorithms which act on a population of candidate solutions. They can be applied to search on discrete spaces, so can be used to search rule sets, representations of computer programs or computer structures, neural network architectures, and so forth. When applied to learning, genetic algorithms turn learning tasks into reinforcement learning problems; it is in this domain where they are often used. Genetic algorithms combine quite successfully with local search algorithms or local machine learning methods.

Good introductory texts on genetic algorithms include books by Goldberg [10] and Mitchell [23]. The standard introductory books on genetic programming are those by Koza [17, 18].

References

[1] D. Ackley and M. Littman, "Interaction between learning and evolution". In *Artificial Life II* C. Langton, editor, Addison-Wesley, 1991.

[2] H. Adeli and S. Hung, *Machine Learning: Neural Networks, Genetic Algorithms and Fuzzy Systems* John Wiley and Sons, 1995.

[3] J. Baldwin, "A new factor in evolution", *American Naturalist* 30: 441–451, 1896.

[4] R. Belew, "When both individuals and populations search: adding simple learning to the genetic algorithm". In *Proceedings of the Third International Conference on Genetic Algorithms* J. D. Schaffer, editor. Morgan-Kaufmann, 1989.

[5] R. Belew and M. Mitchell, *Adaptive Individuals in Evolving Populations* Santa Fe Institute Studies in the Sciences of Complexity Volume XXVI Addison-Wesley, 1996.

[6] C. M. Bishop, *Neural Networks for Pattern Recognition* Oxford University Press, 1996.

[7] P. Darwen and X. Yao, "Every Niching Method has its Niche: Fitness Sharing and Implicit Sharing Compared", preprint.

[8] L. Davis, *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991.

[9] L. Fogel, and A. Owens, and M. Walsh, *Artificial Intelligence through Simulated Evolution*, Wiley, 1966.

[10] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

[11] W. Hart and R. Belew, "Optimization of Genetic Algorithm Hybrids that Use Local Search". In [5].

[12] I. Harvey, "The puzzle of the persistent question marks: a case study of genetic drift." In *Proceedings of the fifth international conference on genetic algorithms*, S. Forrest, editor. 1993.

[13] G. E. Hinton and S. J. Nowlan, "How learning can guide evolution". *Complex Systems* 1: 495–502, 1987.

[14] J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975. (Second edition: MIT Press, 1992.)

[15] J. H. Holland, "Escaping brittleness: The possibilities of general purpose learning algorithms applied to parallel rule-based systems". In *Machine Learning II*, R. Michalski, J. Carbonell, T. M. Mitchell, editors, Morgan Kaufmann, 1986.

[16] H. Kitano, "Designing neural networks using genetic algorithms with graph generation system". *Complex Systems* 4:461–476, 1990.

[17] J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

[18] J. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press: 1994.

[19] J. Koza and F. Bennett III, and D. Andre and M. Keane, "Four problems for which a computer program evolved by genetic programming is competitive with human performance", *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation* IEEE Press, 1996.

[20] G. Mayley, "Landscapes, Learning Costs, and Genetic Assimilation", *Evolutionary Computation* 4(3), 231–234, 1996.

[21] O. Miglino and S. Nolfi, and D. Parisi, "Discontinuity in Evolution: How Different Levels of Organization Imply Preadaptation" In [5].

[22] D. Montana and L. Davis, "Training feedforward networks using genetic algorithms", *Proceedings of International Joint Conference on Artificial Intelligence*, Morgan Kaufman, 1989.

[23] M. Mitchell, *An Introduction to Genetic Algorithms*, MIT Press, 1996.

[24] W. H. Press and S. A. Teukolsky and W. T. Vetterling and B. P. Flannery, *Numerical Recipes in C* Cambridge University Press, 1992.

[25] N. J. Radcliffe, "Equivalent Class Analysis of Genetic Algorithms", *Complex Systems* 5(2) 183–205, 1991.

[26] I. Richenberg, "Cybernetic Solution Path of an Experimental Problem. Ministry of Aviation, Royal Aircraft Establishment (U.K.), 1965.

[27] J. D. Schaffer, D. Whitley, and L. J. Eshelman, "Combinations of genetic algorithms and neural networks: a survey of the state of the art" *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks*, (D. Whitley and J. D. Schaffer, editors), pp. 1–37. IEEE Computer Society Press, Los Alamitos, Ca. 1992.

[28] J. L. Shapiro and A. Prügel-Bennett, "A Maximum Entropy Analysis of Genetic Algorithms", *Lecture Notes in Computer Science* 993, 14–24, 1995.

[29] J. L. Shapiro and A. Prügel-Bennett, "Genetic Algorithm Dynamics in a Two-well Potential", in *Foundations of Genetic Algorithms 4* R. Belew and M. Vose, editors. Morgan Kaufmann, 1997.

[30] L. Tarassenko, *A Guide to Neural Computing Applications* Arnold Publishers, 1998.

[31] D. Wales and J. Doye, "Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lenard-Jones Clusters Containing up to 110 Atoms", *J. Phys. Chem. A.* 101, 5111–5116, 1997.

[32] A. Weismann, *The germ-plasm: A theory of heredity*, Scribners, 1893.

[33] D. Whitley and V. Gordon, and K. Mathias, "Lamarckian Evolution, the Baldwin Effect, and function optimization", in *Parallel Problem Solving From Nature III*, Y. Davidor, H. Schwefel and R. Männer, editors. Springer-Verlag, 1994.

[34] X. Yao and Y. Liu and P. Darwen, "How to Make Best Use of Evolutionary Learning" Published in *Complex Systems — From Local Interactions to Global Phenomena* R. Stocker (ed.) IOS Press, Amsterdam 229–242, 1996.

[35] X. Yao, "Evolutionary artificial neural networks" Published in *Encyclopedia of Computer Science and Technology*, A. Kent and J. G. Williams, editors. Volume 33, pages 137–170, Marcel Dekker, Inc. 1995.