Chapter 12

Evolutionary Learning

In this chapter we are going to start by treating evolution the same way that we treated neuroscience earlier in the book—by cherry-picking a few useful concepts, and then filling in the gaps with computer science in order to make an effective learning method. To see why this might be interesting, you need to view evolution as a search problem. We don't generally think of it in this way, but animals are competing with each other in all kinds of ways—for example, eating each other—which encourages them to try to find camouflage colours, become toxic to certain predators, etc.

Evolution works on a population through an imaginary fitness landscape, which has an implicit bias towards animals that are 'fitter,' i.e., those animals that live long enough to reproduce, are more attractive, and so get more mates, and generate more and healthier offspring. You can find out more from hundreds of books, such as Charles Darwin's "The Origin of Species" (the original book on the topic, still in print and very interesting) and Richard Dawkin's "The Blind Watchmaker."

The genetic algorithm models the genetic process that gives rise to evolution. In particular, it models sexual reproduction, where both parents give some genetic information to their offspring. As is sketched in Figure 12.1, in biological organisms, each parent passes on one chromosome out of their two, and so there is a 50% chance of any gene making it into the offspring. Of the two versions of each gene (one from each parent) *one allele* (variation) is selected. Hence, children have similarities with their parents, and there is lots of genetic inheritance. However, there are also random mutations, caused by copying errors when the chromosome material is reproduced, which mean that some things do change over time. Real genetics is obviously a lot more complicated than this, but we are taking only the things that we want for our model.

The genetic algorithm shows many of the things that are best and worst about machine learning: it is often, but not always, very effective, it has an array of parameters that are crucial, but hard to set, and it is impossible to guarantee that it will find a result that is any good at all. Having said all that, it often works very well, and it has become a very popular algorithm for people to use when they have no idea of any other way to find a reasonable solution.

In the terms that we saw at the end of the previous chapter, genetic al-

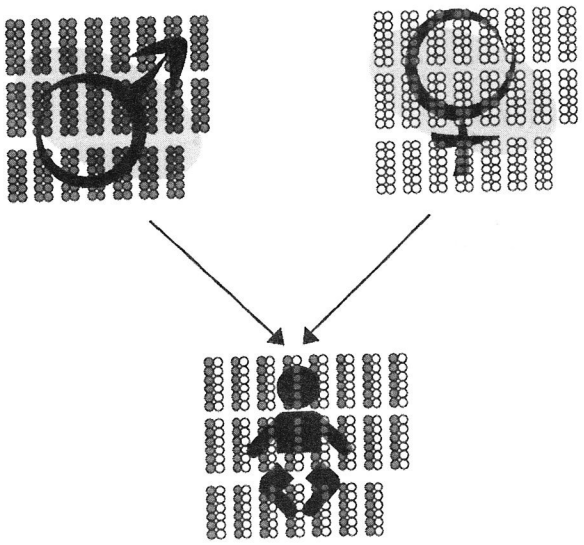

FIGURE 12.1: Each adult in the mating pair passes one of their two chromosomes to their offspring.

gorithms perform both exploitation and exploration, so that they can make incremental improvements to current good solutions, but also find radically new solutions, some of which may be better than the current best.

We will also look briefly at two other topics in this chapter, a variation of the genetic algorithm that acts on trees that represent computer programs that is known as genetic programming, and a set of algorithms that use sampling from a probability distribution rather than an evolving population in order to find better solutions.

12.1 The Genetic Algorithm (GA)

The Genetic Algorithm is a computational approximation to how evolution performs search, which is by altering the genome and thus changing the fitness of individuals. Like another mathematical model that we saw earlier in the book—the neuron—it attempts to abstract away everything except the important parts that we need to understand what evolution does. From this principle, the things that we need to model simple genetics inside a computer and solve problems with it are:

- a method for representing problems as chromosomes

- a way to calculate the fitness of a solution
- a selection method to choose parents
- a way to generate offspring by breeding the parents

These items are all described in the following sections, and the basic algorithm is described. We are going to use an example to describe the methods, which is an NP-complete problem (if you are not familiar with the term NP-complete, its practical implication is that the problem runs in exponential time in the number of inputs) known as the knapsack problem (a knapsack is a rather old name for a rucksack or bag). Sections 12.3.1 and 12.3.4 provide other examples. The knapsack problem is easy to describe, but difficult to solve in general. Here is the version of it that we will use:

Suppose that you are packing for your holidays. You've bought the biggest and best rucksack that was for sale, but there is still no way that you are going to fit in everything you want to take (camera, money, addresses of friends, etc.) and the things that your mum is insisting you take (spare underwear, phrasebook, stamps to write home with, etc.). As a good computer scientist you decide to assign a value to each item, and measure how much space it takes up. Then you want to maximise the value of the items you will take with you, with the constraint that everything has to fit into the bag.

This problem, and variations of it, appear in various disguises in cryptography, combinatorics, applied mathematics, logistics, and business, so it is an important problem. Unfortunately, it is also NP-complete, so finding the optimal solution for interesting cases is computationally impossible. We are going to find approximations to the correct solution using a Genetic Algorithm.

12.1.1 String Representation

The first thing that we need is some way to represent the individual solutions, in analogy to the chromosome. GAs use a string, with each element of the string (equivalent to the gene) being chosen from some alphabet. The different values in the alphabet, which is often just binary, are analogous to the alleles. For the problem we are trying to solve we have to work out a way of encoding the description of a solution as a string. We then create a set of random strings to be our initial population.

It is possible to modify the GA so that the alphabet it uses runs over the real numbers. While purists don't think that this is a GA at all, it is quite popular, because of the number of applications, but it is not as elegant as using a discrete alphabet. It also makes the mutation operator that we will see later less useful.

For the knapsack problem the alphabet is very simple, since we can make it binary. We make the string L units long, where L is the total number of things we would like to take with us, and make each unit a binary digit. We then encode a solution using 0 for the things we will not take and 1 for the

things we will. So if there were four things we wanted to take, then $(0, 1, 1, 0)$ would mean that we take the middle two, but not the first or last.

Note that this does not tell us whether or not this string is possible (that is, whether it will fit into the knapsack), nor whether it is a good string (whether it fills the knapsack). To work these out we need some way to decide how well each string fulfills the problem criteria. This is known as the fitness of the string.

12.1.2 Evaluating Fitness

The fitness function can be seen as an oracle that takes a string as an argument and returns a value for that string. It is the only problem-specific part of the algorithm. It is worth thinking about what we want from our fitness function. Clearly, the best string should have the highest fitness, and the fitness should decrease as the strings do less well on the problem. In general, fitness should always be a positive function—even the least fit strings should have fitness of at least zero. In real evolution, the fitness landscape is not static: there is competition between different species, such as predators and prey, or medical cures for certain diseases, and so the measure of fitness changes over time. We'll ignore that in the genetic algorithm.

For the knapsack problem, we could decide that we want to make the bag as full as possible. So we would need to know the volume of each item that we want to put into the knapsack, and then for a given string that says which things should be taken, and which should not, we can compute the total volume. This is then a possible fitness function. However, it does not tell us anything about whether they will fit into the bag—with this fitness function the optimal solution is to take everything. So we need to check that they will fit, and if they will not, reduce the fitness of that solution. One option would be to set the fitness to 0 if it will not fit. However, suppose that the solution is almost perfect, it is just that there is one thing too many in the knapsack. By setting the fitness to 0 we are reducing the chance of this solution being allowed to evolve and improve during later iterations. For this reason we will make the fitness function be the sum of the values of the items to be taken if they fit into the knapsack, but if they do not we will subtract twice the amount by which they are too big for the knapsack from the size of the knapsack. This allows solutions that are only just over to be considered for improvement, but tries to ensure that they are not the fittest solutions around.

There is an obvious greedy algorithm that finds solutions to the knapsack problem. At each stage it takes the largest thing that hasn't been packed yet and that will still fit into the bag, and iterates that rule. This will not necessarily return the optimal solution (unless each thing is larger than the sum of all the ones smaller than it, in which case it will), but it is very quick and simple. So a GA should be getting a much better solution than the greedy rule in general to be worth all the effort involved in writing and running it.

12.1.3 Population

We can now measure the fitness of any string. The GA works on a population of strings, with the first generation usually being created randomly. The fitness of each string is then evaluated, and that first generation is bred together to make a second generation, which is then used to generate a third, and so on. After the initial population is chosen randomly, the algorithm evolves in such a way that the fitness of individuals in the population increases over the generations.

So for the knapsack problem, we will now create a set of random binary strings of length L by using the random number generator. We'll arbitrarily decide to make 100 strings. We now need to choose parents out of this population, and start breeding them. At every iteration the population stays the same size, something else that is unlike real evolution. Creating the initial population is very easy in NumPy using the uniform random number generator and the `where()` function:

```
pop = random.rand(popSize,stringLength)
pop = where(pop<0.5,0,1)
```

12.1.4 Generating Offspring: Parent Selection

For the current generation we need to select those strings that will be used to generate new offspring. The idea here is that fitness will improve if we select strings that are already relatively fit compared to the other members of the population (following natural selection). This is exploitation of our current population. However, it is also good to allow some exploration in there, which means that we have to allow some possibility of weak strings being considered. The basic idea is that we choose strings proportionally to their fitness, so that fitter strings are more likely to be chosen to enter the 'mating pool.' There are two commonly employed ways to do this, although the second one is better:

Truncation Selection A simple method is just to pick some fraction f of the best strings and ignore the rest. For example, $f = 0.5$ is often used, so the best 50% of the strings are put into the mating pool, and chosen with equal probability. This is obviously very easy to implement, but it does limit the amount of exploration that is done, biasing the GA towards exploitation.

Fitness Proportional Selection The better option is to select strings probabilistically, with the probability of a string being selected being proportional to its fitness. The function that is generally used is (for string α):

$$p^\alpha = \frac{F^\alpha}{\sum_{\alpha'} F^{\alpha'}}, \tag{12.1}$$

where F^α is the fitness. This probabilistic interpretation is the reason why fitness should be positive. If they aren't guaranteed positive, then Boltzmann selection can be used to make them so (where s is the selection strength, a parameter, and you might recognise the equation as the softmax activation from Chapter 3):

$$p^\alpha = \frac{\exp(sF^\alpha)}{\sum_{\alpha'} \exp(sF^{\alpha'})}. \tag{12.2}$$

There is an implementation issue here. We want to pick each string with probability proportional to its fitness, but if we only have one copy of each string, then the probability of picking each string is the same. One way around this is to add more copies of the fitter strings, so that they are more likely to get chosen. This is sometimes called 'roulette selection,' because if you imagine that each string gets an area on a roulette wheel, then the larger the area associated to one number, the more likely it is that the ball will land there. You can then just randomly pick strings from this larger set. A method of doing this is shown in the following code snippet, which uses the `kron()` function. We've seen this before (in Section 10.5); it is a NumPy function that multiples each element of its first array argument by every element of the second, putting all of the results together into one multi-dimensional output array. It is useful here in order to populate the new and much larger `newPopulation` array, which contains multiple copies of each string.

```
# Put in repeated copies of each string according to fitness
# Deal with strings with very low fitness
j=0
while round(fitness[j])<1:
 j = j+1

newPop = kron(ones((round(fitness[j]),1)),pop[j,:])

# Add multiple copies of strings into the newPop
for i in range(j+1,self.popSize):
 if round(fitness[i])>=1:
  newPop = concatenate((newPop,kron(ones((round(fitness[i]),1))
  ,pop[i,:])),axis=0)

# Shuffle the order (note that there are still too many)
indices = range(shape(newPop)[0])
```

```
random.shuffle(indices)
newPop = newPop[indices[:popSize],:]
return newPop
```

However we select the strings to put into the mating pool, the next operation is to put them into pairs. Since the order that they are in is random, we can simply pair up the strings so that each even-indexed string takes the following odd-indexed one as its mate.

12.2 Generating Offspring: Genetic Operators

Having selected our breeding pairs, we now need to decide how to combine their two strings to generate the offspring, which is the genetics part of the algorithm. There are two genetic operators that are generally used, and they are discussed now. There are others, but these were the original choices, and are far and away the most common.

12.2.1 Crossover

In biology, organisms have two chromosomes, and each parent donates one of them. Members of our GA population only have one chromosome-equivalent, the string. Thus, we generate the new string as part of the first parent and part of the second. The most common way of doing this is to pick one point at random in the string, and to use parent 1 for the first part of the string, up to the crossover point and parent 2 for the rest. We actually generate two offspring, with the second one consisting of the first part of parent 2 and the second part of parent 1. This scheme is known as single point crossover, and the extension to multi-point crossover is hopefully obvious. The most extreme version is known as uniform crossover and consists of independently selecting each element of the string at random from the two parents. The three types of crossover are shown in Figure 12.2.

Crossover is the operator that performs global exploration, since the strings that are produced are radically different to both parents. The hope is that sometimes we will take good parts of both solutions and put them together to make an even better solution. A nice picture example is to imagine a bird that has webbed feet for good swimming, but that cannot fly, breeding with a bird that can fly, but not swim. The offspring? A duck! Obviously, this is not biologically plausible, but it is a good picture of how crossover works. One interesting feature of the GA that obviously isn't true in real genetics is that in addition to the duck the algorithm would produce the bird that can't fly or

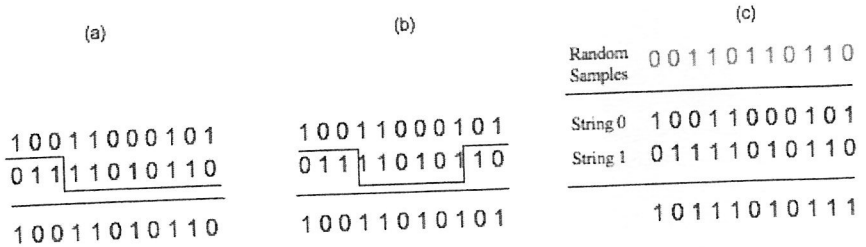

FIGURE 12.2: The different forms of the crossover operator. (a) Single point crossover. A position in the string is chosen at random, and the offspring is made up of the first part of parent 1 and the second part of parent 2.(b) Multi-point crossover. Multiple points are chosen, with the offspring being made in the same way. (c) Uniform crossover. Random numbers are used to select which parent to take each element from.

swim, although it is unlikely to last long since its fitness will presumably not be high. In fact, there are exceptions to this, such as the great New Zealand Kiwi, which can neither swim nor fly, but is happily not extinct.

The following code snippet shows a NumPy implementation of single point crossover. The extension to multi-point and uniform crossover is not particularly difficult.

```
def spCrossover(pop):
    newPop = zeros(shape(pop))
    crossoverPoint = random.randint(0,stringLength,popSize)
    for i in range(0,self.popSize,2):
  newPop[i,:crossoverPoint[i]] = pop[i,:crossoverPoint[i]]
  newPop[i+1,:crossoverPoint[i]] = pop[i+1,:crossoverPoint[i]]
  newPop[i,crossoverPoint[i]:] = pop[i+1,crossoverPoint[i]:]
  newPop[i+1,crossoverPoint[i]:] = pop[i,crossoverPoint[i]:]
    return newPop
```

Crossover is not always useful, depending upon the problem; for example, in the Travelling Salesman Problem that we talked about in Chapter 11, the strings that are generated by crossover might not even be valid tours. However, when it is useful, it is often the more powerful of the genetic operators, and has led to the building block hypothesis of how GAs work. The idea is that GAs work well on problems where the solution comes from putting together lots of little solutions, so that different strings assemble each separate building block, and then crossover puts those substrings together to make the final solution.

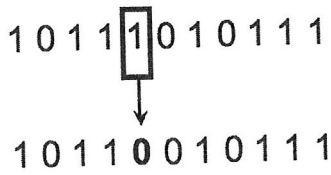

FIGURE 12.3: The effects of mutation on a string.

12.2.2 Mutation

The exploitation of the current best strings is performed by the mutation operator, which effectively performs local random search. The value of each element of the string is changed with some (usually low) probability p. For our binary alphabet in the knapsack problem, mutation causes a bit-flip, as is shown in Figure 12.3. For chromosomes with real values, some random number is generally added or subtracted from the current value. Often, $p \approx 1/L$ where L is the string length, so that there is approximately one mutation in each string. This might seem quite high, but it is often found to be a good choice given that the mutation rate has to trade off doing lots of local search with the risk of disrupting the good solutions.

12.2.3 Elitism, Tournaments, and Niching

One problem with the standard GA is that the best fitness can decrease as well as increase in the next generation. This happens because the best strings in the current population are not copied into the next generation, and sometimes none of their offspring are as good. One way around this is called elitism, and it is the simple idea of copying the best strings in the current population into the next population without any change. Another solution is to implement a tournament, where the two parents and their two offsprings compete, with the two fittest out of the four being put into the new population.

The implementation of these functions continues along the same lines as the previous ones; the `argsort()` function returns the indices of the array that sorts them into order, but does not actually sort the array. It returns an array the same size as the one that is sorted, which is why the `squeeze()` function is needed to reduce the array to the right size.

```
def elitism(oldPop,pop,fitness):
    best = argsort(fitness)
    best = squeeze(oldPop[best[-nElite:],:])
    indices = range(shape(pop)[0])
    random.shuffle(indices)
    pop = pop[indices,:]
    pop[0:nElite,:] = best
```

The Basic Genetic Algorithm

- **Initialisation**
  - generate N random strings of length L with our chosen alphabet
- **Learning**
  - repeat:
    - create an (initially empty) new population
    - repeat:
      - select two strings from current population by fitness
      - recombine them to produce two new strings
      - mutate the offspring
      - either add the two offspring to the population or use elitism or tournaments
      - keep track of the best string in the population
    - until N strings for the new population are generated
    - replace the current population with the new population
  - until stopping criteria met

12.3 Using Genetic Algorithms

12.3.1 Map Colouring

Graph colouring is a typical discrete optimisation problem. We want to colour a graph using only k colours, and choose them in such a way that adjacent regions have different colours. It has been mathematically proven that any two-dimensional planar graph can be coloured with four colours, which was the first ever proof that used a computer program to check the cases. Even though it might be impossible, we are going to try to solve the three-colour problem using a genetic algorithm, we just won't be upset if the solution isn't perfect (this is a good idea with a GA anyway, of course). With all problems where you want to apply a genetic algorithm, there are three basic tasks that need to be performed:

Encode possible solutions as strings For this problem, we'll choose our alphabet to consist of the three possible shades (black (b), dark (d), and light (l), say). So for a six-region map, a possible string is $\alpha = \{bdblbb\}$. This says that the first region is black, the second dark grey, etc. We

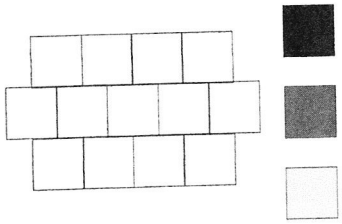

FIGURE 12.4: A sample map that we wish to colour using the three colours shown, without any two adjacent squares having the same colour.

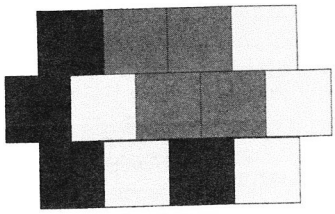

FIGURE 12.5: A possible colouring with several adjacent squares having the same colour.

choose an order to record the regions in and stick to it for all the strings, and now we can encode any way of colouring in those six regions. An example problem and a colouring are given in Figures 12.4 and 12.5.

Choose a suitable fitness function The thing that we want to minimise (a cost function) is the number of times that two adjacent regions have the same colour. We could count these up fairly simply, but it is not a fitness function, because the best solution has the lowest number, not the highest. One easy way to turn it into a fitness function would be to use the Boltzmann selection described earlier (Equation (12.2)), or to count the total number of lines between regions and subtract off the number where the two regions on either side of the line have the same colour. However, we could also just count the number of correct edges. The example in Figure 12.5 has 16 out of the 26 boundaries correct (where a boundary is the intersection between any two squares), so its fitness is 16.

Choose suitable genetic operators We'll use the standard genetic operators for this, since this example makes the operations of crossover and mutation clear. The way that they are used is shown in Figures 12.6 and 12.7. In general, people just use the standard operators for most problems, but if they don't work well, it can be worth putting some effort into thinking of new ones.

Having made those choices, we can let the GA run on the problem, with a possible population and their offspring shown in Figure 12.8, and look at the best solutions after some preset number of iterations. The GA produces good solutions to this problem, and implementing it for yourself is one of the suggested exercises for this chapter.

FIGURE 12.6: The way that mutation is performed on a colour, changing it into one of the other colours.

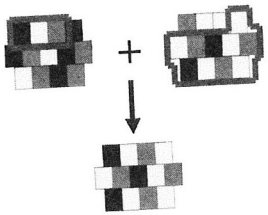

FIGURE 12.7: The effects of crossover on a map.

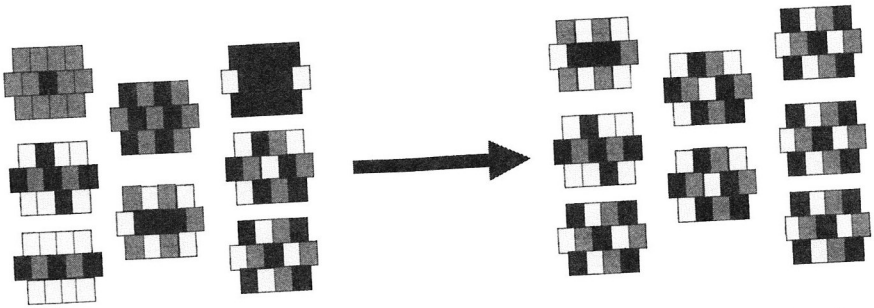

FIGURE 12.8: One generation of the GA working on the map colouring problem.

12.3.2 Punctuated Equilibrium

For a long time, one thing that creationists and others who did not believe in evolution used as an argument against it was the problem of the lack of intermediate animals in the fossil record. The argument runs that if humans evolved from apes, then there should be some evidence of a whole set of intermediary species that existed during the transition phase, and there aren't. Interestingly, GAs demonstrate one of the explanations why this is not correct, which is that the way that evolution actually seems to work is known as punctuated equilibrium. There is basically a steady population of some species for a long time, and then something changes and over a very short (in evolutionary terms... still hundreds or thousands of years) period, there is a big change, and then everything settles down again. So the chance of finding fossils from the intermediary stage is quite small. There is a graph showing this effect in Figure 12.9.

12.3.3 Example: The Knapsack Problem

We used the knapsack problem as an example while we were looking at components of the GA. It is now time to see it being solved. Before we do that, we can use some of the methods from Section 11.4 to solve it. We've

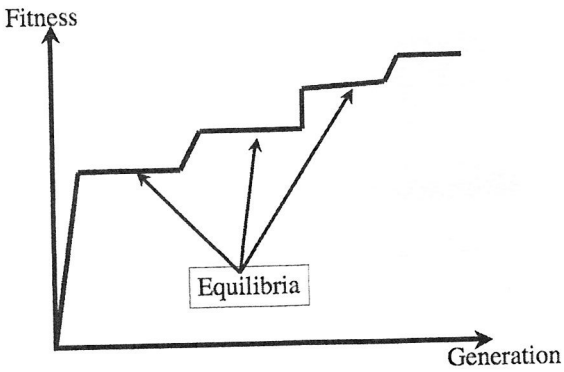

FIGURE 12.9: A graph showing punctuated equilibrium in a genetic algorithm. There is an effectively steady state where fitness does not improve, followed by rapid improvements in fitness until another steady state is reached.

already mentioned the greedy algorithm solution, and we can of course use exhaustive search, as well, or any of the other methods we discussed in the last chapter, such as simulated annealing or hillclimbing.

The website has a simple example with 20 different packages, which have a total size of 2436.77 and a maximum knapsack size of 500. The greedy algorithm finds a solution of 487.47, while the optimal solution is eventually found by the exhaustive search as 499.98. The question is how well the GA does on the same problem. We will use the fitness function that was described in Section 12.1.2, where solutions that are too large are penalised by having twice the amount they are over subtracted from the maximum size. Figure 12.10 shows a graph of the output when the GA is run on this problem for 100 iterations. The GA rapidly finds a near-optimal solution (of 499.94) to this relatively simple problem, although in this run it did not find the global optimum.

12.3.4 Example: The Four Peaks Problem

The four peaks is a toy problem (that is, simple problem that isn't useful itself, but is good for testing algorithms) that is quite often used to test out GAs and various developments of them. It is an invented fitness function that rewards strings with lots of consecutive 0s at the start of the string, and lots of consecutive 1s at the end. The fitness consists of counting the number of 0s at the start, and the number of 1s at the end and returning the maximum of them as the fitness. However, if both the number of 0s and the number of 1s are above some threshold value T then the fitness function gets a bonus of 100 added to it. This is where the name 'four peaks' comes from: there are two small peaks where there are lots of 0s, or lots of 1s, and then there

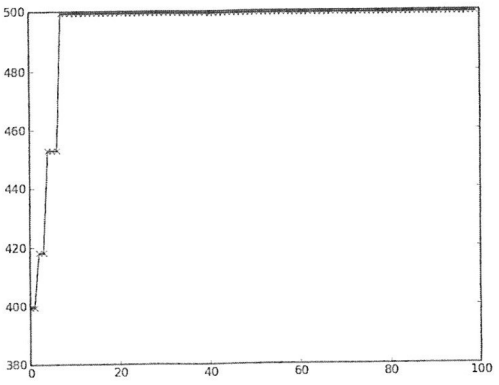

FIGURE 12.10: Evolution of the solution to the knapsack problem. The GA finds a very good solution to this simple problem within a few iterations, but never finds the optimal solution.

are two larger peaks, where the bonus is included. The GA should find these larger peaks for a successful run.

In NumPy the four peaks fitness function can be written as:

```
def fourpeaks(pop,T=10):

    start = zeros((shape(pop)[0],1))
    finish = zeros((shape(pop)[0],1))

    fitness = zeros((shape(pop)[0],1))

    for i in range(shape(pop)[0]):
        s = where(pop[i,:]==1)
        f = where(pop[i,:]==0)
        if size(s)>0:
            start = s[0][0]
        else:
            start = 0

        if size(f)>0:
            finish = shape(pop)[1] - f[-1][-1] -1
        else:
            finish = 0

        if start>T and finish>T:
            fitness[i] = maximum(start,finish)+100
```

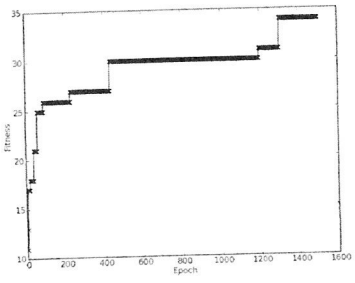

FIGURE 12.11: Evolution of a solution to the four peaks problem. The solution never reaches the bonus score in the fitness function.

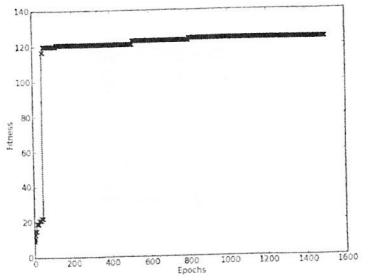

FIGURE 12.12: Another solution to the four peaks problem. This solution does reach the bonus score, but does not get the global maximum.

```
    else:
        fitness[i] = maximum(start,finish)

fitness = squeeze(fitness)
return fitness
```

Figures 12.11 and 12.12 show the outputs of two runs for a chromosome length of 100 and with $T = 15$. In the second the GA reaches the bonus point, while in the first it does not. Both of these runs used a mutation rate of 0.01, which is $1/L$, and single point crossover. They also used elitism.

12.3.5 Limitations of the GA

Genetic algorithms can be very slow. The main problem is that once a local maximum has been reached, it can often be a long time before a string is produced that escapes from the local maximum and finds another, higher, maximum. In addition, because we do not know anything about the fitness landscape, we can't see how well the GA is doing.

A more basic criticism of genetic algorithms is that it is very hard (read basically impossible) to analyse the behaviour of the GA. We expect that the mean fitness of the population will increase until an equilibrium of some kind is reached. This equilibrium is between the selection operator, which makes the population less diverse, but increases the mean fitness (exploitation), and the genetic operators, which usually reduce the mean fitness, but increase the diversity in the population (exploration). However, proving that this is guaranteed to happen has not been possible so far, which means that we cannot

guarantee that the algorithm will converge at all, and certainly not to the optimal solution. This bothers a lot of researchers. That said, genetic algorithms are widely used when other methods do not work, and they are usually treated as a black box—strings are pushed in one end, and eventually an answer emerges. This is risky, because without knowledge of how the algorithm works it is not possible to improve it, nor do you know how cautiously you should treat the results.

12.3.6 Training Neural Networks with Genetic Algorithms

We trained our neural networks, most notably the MLP, using gradient descent. However, we could encode the problem of finding the correct weights as a set of strings, with the fitness function measuring the sum-of-squares error. This has been done, and with good reported results. However, there are some problems with this approach. The first is that we turn all the local information from the targets about the error at each output node of the network into just one number, the fitness, which is throwing away useful information, and the second is that we are ignoring the gradient information, which is also throwing away useful information.

A more sensible use for GAs with neural networks is to use the GA to choose the topology of the network. Previously, we chose the structure in a completely ad hoc way by trying out different structures and choosing the one that worked best. We can use a GA for this problem, although the crossover operator doesn't make a great deal of sense, so we just consider mutation. However, we allow for four different types of mutation: delete a neuron, delete a weight connection, add a neuron, add a connection. The deletion operators bias the learning towards simple networks. Making the GA more complicated by adding extra mutation operators might make you wonder if you can make it more complicated again. And you can; one example of where this can lead is discussed next.

12.4 Genetic Programming

One extension of genetic algorithms that has had a lot of attention is the idea of genetic programming. This was introduced by John Koza, and the basic idea is to represent a computer program as a tree (imagine a flow chart of the code). For certain programming languages, notably LISP, this is actually a very natural way to represent a program, but it doesn't work very well in Python, so we will have a quick look at the idea, but not get into writing any explicit algorithms for the method.

Tree-based variants on mutation and crossover are defined (replace sub-

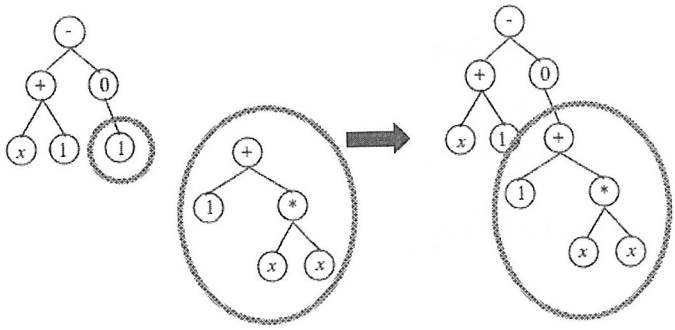

FIGURE 12.13: Example of a mutation in genetic programming.

trees by other subtrees, either randomly generated (mutation, Figure 12.13) or swapped from another tree (crossover, Figure 12.14)), and then the genetic program runs just like a normal genetic algorithm, but acting on these program trees rather than strings. Figure 12.15 shows a set of simple trees that perform arithmetic operations, and some possible developments of them, made using these operators.

Genetic programming has been used for many different tasks, from recognising skin melanomas to circuit design, and lots of very impressive results have been claimed for it. However, the search space is unbelievably large, and the mutation operator not especially useful, and so a lot depends upon the initial population. A set of possibly useful subtrees are usually chosen by the system developer first in order to give the system a head start. There are a couple of places where you can find more information on genetic programming in the Further Reading section.

12.5 Combining Sampling with Evolutionary Learning

The last machine learning method in this chapter is an interesting variation on the theme of evolutionary learning, combined with probabilistic models of the type that are described in Chapter 15, namely Bayesian networks. They are often known as estimation of distribution algorithms (EDA).

The most basic version is known as Population-Based Incremental Learning (PBIL), and it is amazingly simple. It works on a binary alphabet, just like the basic GA, but instead of maintaining a population, instead it keeps a probability vector p that gives the probability of each element being a 0 or 1. Initially, each value of this vector is 0.5, so that each element has equal

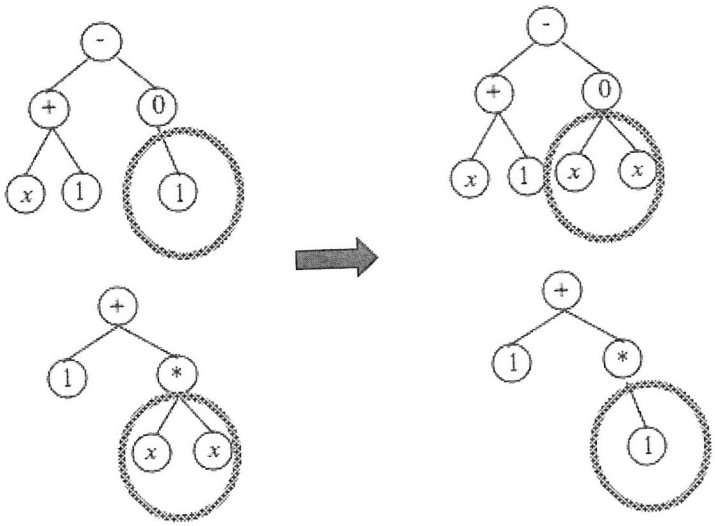

FIGURE 12.14: Example of a crossover in genetic programming.

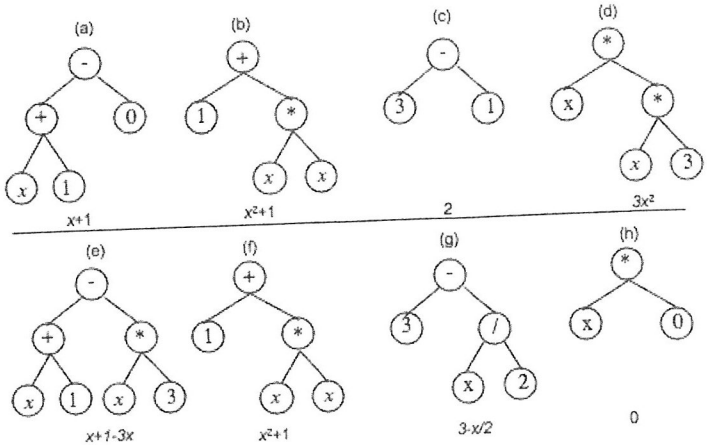

FIGURE 12.15: *Top:* Four arithmetical trees. *Bottom:* Example developments of the four trees: (e) and (h) are a possible crossover of (a) and (d), (f) is a copy of (b), and (g) is a mutation of (c).

chance of being 0 or 1. A population is then constructed by sampling from the distribution specified vector, and the fitness of each member of the population is computed. A subset of this population (typically just the two fittest vectors) is chosen to update the probability vector, using a learning rate η, which is often set to 0.005 (where `best` and `second` represent the best and second-best elements of the population):

$$p = p \times (1 - \eta) + \eta(\text{best} + \text{second})/2. \tag{12.3}$$

The population is then thrown away, and a new one sampled from the updated probability vector. The results of using this simple algorithm on the four-peaks problem with $T = 11$ are shown in Figure 12.16 using strings of length 100 with 200 strings in each population. This is directly comparable with Figure 12.12.

The centre of the algorithm is simply the code to find the strings with the two highest fitnesses and use them to update the vector. Everything else is directly equivalent to the genetic algorithm.

```
# Pick best
best[count] = max(fitness)
bestplace = argmax(fitness)
fitness[bestplace] = 0
secondplace = argmax(fitness)

# Update vector
p  = p*(1-eta) + eta*((pop[bestplace,:]+pop[secondplace,:])/2)
```

The probabilistic model that is used in PBIL is very simple: it is assumed that each element of the probability vector is independent, so that there is no interaction. However, there is no reason why more complicated interactions between variables cannot be considered, and several methods have been developed that do exactly this. The first option is to construct a chain, so that each variable depends only on the one to its left. This might involve sorting the order of the probability vector first, but then the algorithm simply needs to measure the mutual information (see Section 6.2.1) between each pair of neighbouring variables. This use of mutual information gives the algorithm its name: MIMIC. There are also more complicated variants using full Bayesian networks, such as the Bayesian Optimisation Algorithm (BOA) and Factorised Distribution Algorithm (FDA).

The power of these developments of the GA is that they use probabilistic models and are therefore more amenable to analysis than normal GAs, which have steadfastly withstood many attempts to better understand their behaviour. They also enable the algorithm to discover correlations between input variables, which can be useful if you want to understand the solution rather than just apply it.

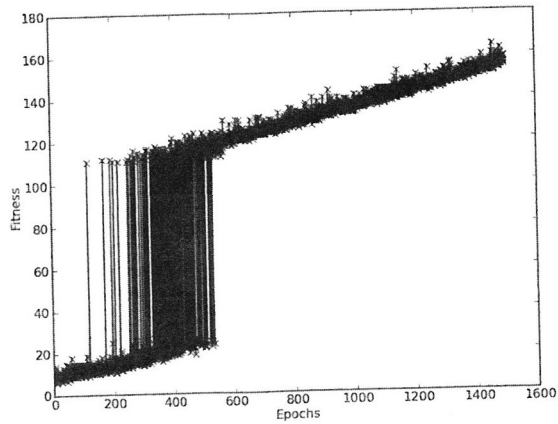

FIGURE 12.16: The evolution of the best fitness using PBIL on the four peaks problem.

It is important to remember that there is no guarantee that a genetic algorithm will find a good solution, although it often will, and certainly no guarantee that it will find the optimum. The vast majority of applications of genetic algorithms and the other algorithms described in this chapter do not consider this, but use the algorithms as a way to avoid having to understand the problem. Recall the No Free Lunch theorem of the last chapter—there is no universally good solution to the search problem—before using the GA or genetic program as the only search method that you use. Having said that, providing that you are prepared to accept the long running time and the fact that there are no guarantees of a good solution, they are frequently very useful methods.

Further Reading

There are entire books written about genetic algorithms, including:

- J.H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* Michigan University Press, Michigan, USA, 1975.
- M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, USA, 1996.