Chapter 6: Understanding Genetic Algorithms

- Introducing the Genetic Algorithm
- Understanding the Structure of a Genetic Algorithm
- Understanding How a Genetic Algorithm Works
- Implementing the Traveling Salesman Problem
- Neural Network Plays Tic-Tac-Toe

Backpropagation is not the only way to train a neural network. This chapter will introduce genetic algorithms (GAs), which can be used to solve many different types of problems. We will begin by exploring how to use a genetic algorithm to solve a problem independent of a neural network. The genetic algorithm will then be applied to a feedforward neural network.

Genetic Algorithms

Both genetic algorithms and simulated annealing are evolutionary processes that may be utilized to solve search space and optimization problems. However, genetic algorithms differ substantially from simulated annealing.

Simulated annealing is based on a thermodynamic evolutionary process, whereas genetic algorithms are based on the principles of Darwin's theory of evolution and the field of biology. Two features introduced by GAs, which distinguish them from simulated annealing, are the inclusion of a population and the use of a genetic operator called "crossover" or recombination. These features will be discussed in more detail later in this chapter.

A key component of evolution is natural selection. Organisms poorly suited to their environment tend to die off, while organisms better suited to their current environment are more likely to survive. The surviving organisms produce offspring that have many of the better qualities possessed by their parents. As a result, these children tend to be "better suited" to their environment, and are more likely to survive, mate, and produce future generations. This process is analogous to Darwin's "survival of the fittest" theory, an ongoing process of evolution in which life continues to improve over time. The same concepts that apply to natural selection apply to genetic algorithms as well.

When discussing evolution, it is important to note that sometimes a distinction is made between microevolution and macroevolution. Microevolution refers to small changes that occur in the overall genetic makeup of a population over a relatively short period of time. These changes are generally small adaptations to an existing species, and not the introduction of a whole new species. Microevolution is caused by factors such as natural selection and mutation. Macroevolution refers to significant changes in a population over a long period of time. These changes may result in a new species. The concepts of genetic algorithms are consistent with microevolution.

Background of Genetic Algorithms

John Holland, a professor at the University of Michigan, developed the concepts associated with genetic algorithms through research with his colleagues and students. In 1975, he published a book, Adaptation in Natural and Artificial Systems, in which he presents the theory behind genetic algorithms and explores their practical application. Holland is considered the father of genetic algorithms.

Another significant contributor to the area of genetic algorithms is David Goldberg. Goldberg studied under Holland at the University of Michigan and has written a collection of books, including Genetic Algorithms in Search, Optimization, and Machine Learning (1989), and more recently, The Design of Innovation (2002).

Uses for Genetic Algorithms

Genetic algorithms are adaptive search algorithms, which can be used for many purposes in many fields, such as science, business, engineering, and medicine. GAs are adept at searching large, nonlinear search spaces. A nonlinear search space problem has a large number of potential solutions and the optimal solution cannot be solved by conventional iterative means. GAs are most efficient and appropriate for situations in which:

- the search space is large, complex, or not easily understood;
- there is no programmatic method that can be used to narrow the search space;
- traditional optimization methods are inadequate.

Table 6.1 lists some examples.

Table 6.1: Common Uses for Genetic Algorithms

Purpose	Common Uses
Optimization	Production scheduling, call routing for call centers, routing for transportation, determining electrical circuit layouts
Design	Machine learning: designing neural networks, designing and controlling robots
Business applications	Utilized in financial trading, credit evaluation, budget allocation, fraud detection

Many optimization problems are nonlinear in behavior and are too complex for traditional methods. The set of possible solutions for such a problem can be enormous (e.g., determining the optimum route for a traveling salesperson or determining the optimum design for an electrical circuit layout). A genetic algorithm possesses the ability to search large and complex search spaces to efficiently determine near optimal solutions in a reasonable time frame by simulating biological evolution. Now that you have been introduced to some of the uses for genetic algorithms, we must examine how to actually construct one.

Understanding Genetic Algorithms

Genetic algorithms closely resemble the biological model of chromosomes and genes. Individual organisms in a genetic algorithm generally consist of a single chromosome. These chromosomes are composed of genes. By manipulating the genes, new chromosomes are created, which possess different traits. These manipulations occur through crossover and mutation, just as they occur in nature. Crossover is analogous to the biological process of mating, and mutation is one way in which new information can be introduced into an existing population.

Understanding Genes

In a genetic algorithm, genes represent individual components of a solution. This is a very important concept in the analysis of a problem for which a genetic algorithm is to be used. To effectively solve a problem, you must determine a way to break it into its related components, or genes. The genes will then be linked together to form a chromosome. Life forms in this simulation consist of a single chromosome; therefore, each chromosome will represent one possible solution to a problem.

Later in this chapter, we will examine the traveling salesman problem. You will be shown how to decompose the solution for this problem into individual genes. Additionally, in this chapter you will be shown how to make the individual weights in a neural network represent the genes in the chromosome.

It is important to note that there is a finite number of genes that are used. Individual genes are not modified as the organisms evolve. It is the chromosomes that evolve when the order and makeup of their genes are changed.

Understanding Chromosomes

As explained above, each organism in our genetic algorithm contains one chromosome. As a result, each chromosome can be thought of as an “individual” or a solution composed of a collection of parameters to be optimized. These parameters are genes, which you learned about in the previous section. Each chromosome is initially assigned a random solution or collection of genes. This solution is used to calculate a "fitness" level, which determines the chromosome's suitability or "fitness" to survive—as in Darwin's theory of natural selection. If a chromosome has a high level of "fitness," it has a higher probability of mating and staying alive.

How Genetic Algorithms Work

A genetic algorithm begins by creating an initial population of chromosomes that are given a random collection of genes. It then continues as follows:

1. Create an initial population of chromosomes.

2. Evaluate the fitness or "suitability" of each chromosome that makes up the population.

3. Based on the fitness level of each chromosome, select the chromosomes that will mate, or those that have the "privilege" to mate.

4. Crossover (or mate) the selected chromosomes and produce offspring.

5. Randomly mutate some of the genes of the chromosomes.

6. Repeat steps three through five until a new population is created.

7. The algorithm ends when the best solution has not changed for a preset number of generations.

Genetic algorithms strive to determine the optimal solution to a problem by utilizing three genetic operators. These operators are selection, crossover, and mutation. GAs search for the optimal solution until specific criteria are met and the process terminates. The results of the process include good solutions, as compared to one "optimal" solution, for complex problems (such as "NP-hard" problems). NP-hard refers to problems which cannot be solved in polynomial time. Most problems solved with computers today are not NP-hard and can be solved in polynomial time. A polynomial is a mathematical expression involving exponents and variables. A P-Problem, or polynomial problem, is a problem for which the number of steps to find the answer is bounded by a polynomial. An NP-hard problem does not increase exponentially, but often increases at a much greater rate, described by the factorial operator (n!). One example of an NP-hard problem is the traveling salesman problem, which will be discussed later in this chapter.

Initial Population

To recap, the population of a genetic algorithm is comprised of organisms, each of which usually contains a single chromosome. Chromosomes are comprised of genes, and these genes are usually initialized to random values based on defined boundaries. Each chromosome represents one complete solution to the defined problem. The genetic algorithm must create the initial population, which is comprised of multiple chromosomes or solutions.

Each chromosome in the initial population must be evaluated. This is done by evaluating its “fitness” or the quality of its solution. The fitness is determined through the use of a function specified for the problem the genetic algorithm is designed to solve.

Suitability and the Privilege to Mate

In a genetic algorithm, mating is used to create a new and improved population. The “suitability” to mate refers to whether or not chromosomes are qualified to mate, or whether they have the “privilege” to mate.

Determining the specific chromosomes that will mate is based upon each individual chromosome’s fitness. The chromosomes are selected from the old population, mated, and children are produced, which are new chromosomes. These new children are added to the existing population. The updated population is used for selection of chromosomes for the subsequent mating.

Mating

We will now examine the crossover process used in genetic algorithms to accomplish mating. Mating is achieved by selecting two parents and taking a “splice” from each of their gene sequences. These splices effectively divide the chromosomes’ gene sequences into three parts. The children are then created based on genes from each of these three sections.

The process of mating combines genes from two parents into new offspring chromosomes. This is useful in that it allows new chromosomes to inherit traits from each parent. However, this method can also lead to a problem in which no new genetic material is introduced into the population. To introduce new genetic material, the process of mutation is used.

Mutation

Mutation is used to introduce new genetic material into a population. Mutation can be thought of as natural experiments. These experiments introduce a new, somewhat random, sequence of genes into a chromosome. It is completely unknown whether or not this mutation will produce a desirable attribute, and it does not really matter, since natural selection will determine the fate of the mutated chromosome.

If the fitness of the mutated chromosome is higher than the general population, it will survive and likely be allowed to mate with other chromosomes. If the genetic mutation produces an undesirable feature, then natural selection will ensure that the chromosome does not live to mate.

An important consideration for any genetic algorithm is the mutation level that will be used. The mutation level is generally expressed as a percentage. The example program that will be examined later in this chapter will use a mutation level of 10%. Selection of a mutation level has many ramifications. For example, if you choose a mutation level that is too high, you will be performing nothing more than a random search. There will be no adaptation; instead, a completely new solution will be tested until no better solution can be found.

Implementation of a Generic Genetic Algorithm

Now that you understand the general structure of a genetic algorithm, we will examine a common problem to which they are often applied. To implement a generic genetic algorithm, several classes have been created:

- Chromosome
- GeneticAlgorithm
- MateWorker

The **`Chromosome`** class implements a single chromosome. The **`GeneticAlgorithm`** class is used to control the training process. It implements the **`Train`** interface and can be used to train feedforward neural networks.

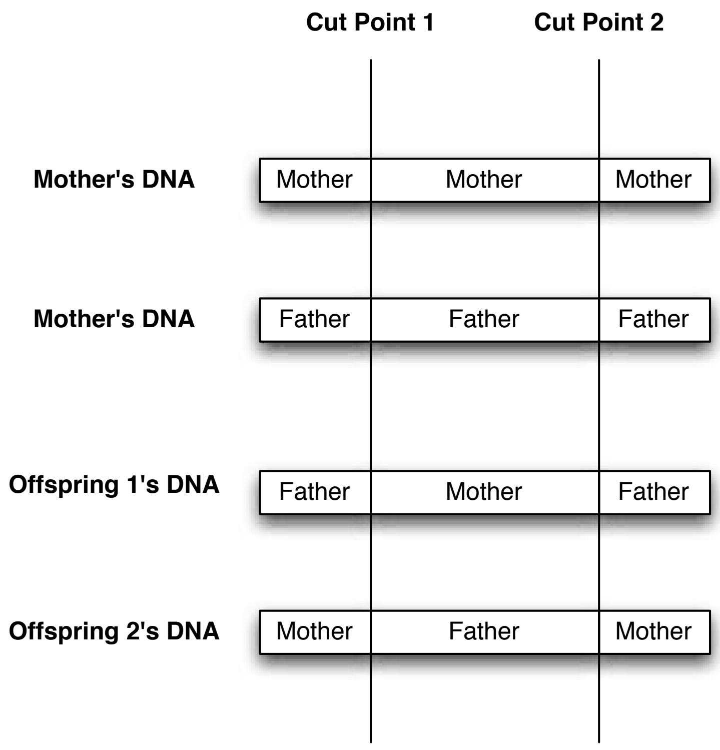
Mating

Mating is performed either in the base **`Chromosome`** class or in a subclass. Subclasses can implement their own **`mate`** methods if specialization is needed. However, the examples in this book do not require anything beyond the base **`Chromosome`** class's **`mate`** method. We will now examine how the **`mate`** method works. The signature for the **`mate`** method of the **`Chromosome`** class is shown here:

```
public void mate(
  final Chromosome<GENE_TYPE, GA_TYPE> father,
  final Chromosome<GENE_TYPE, GA_TYPE> offspring1,
  final Chromosome<GENE_TYPE, GA_TYPE> offspring2)
throws NeuralNetworkError
```

The mating process treats the genes of a chromosome as a long array of elements. For this neural network, these elements will be **`double`** variables taken from the weight matrix. For the traveling salesman problem, these elements will represent cities at which the salesman will stop; however, these elements can represent anything. Some genes will be taken from the mother and some will be taken from the father. Two offspring will be created. Figure 6.1 shows how the genetic material is spliced by the **`mate`** method.

Figure 6.1: Mating two chromosomes.



As you can see, two cut-points are created between the father and the mother. Some genetic material is taken from each region, defined by the cut-points, to create the genetic material for each offspring.

First, the length of the gene array is determined.

```
final int geneLength = getGenes().length;
```

Two cut-points must then be established. The first cut-point is randomly chosen. Because all "cuts" will be of a constant length, the first cut-point cannot be chosen so far in the array that there is not a sufficiently long section left to allow the full cut length to be taken. The second cut-point is simply the cut length added to the first cut-point.

```
final int cutpoint1 = (int) (Math.random() * (geneLength -
      getGeneticAlgorithm().getCutLength()));
final int cutpoint2 = cutpoint1 + getGeneticAlgorithm().
      getCutLength();
```

Two arrays are then allocated that are big enough to hold the two offspring.

```
final Set<GENE_TYPE> taken1 = new HashSet<GENE_TYPE>();
final Set<GENE_TYPE> taken2 = new HashSet<GENE_TYPE>();
```

There are three regions of genetic material that must now be considered. The first is the area between the two cut-points. The other two are the areas before and after the cut-points.

```
for (int i = 0; i < geneLength; i++) {
  if ((i < cutpoint1) || (i > cutpoint2)) {
  } else {
    offspring1.setGene(i, father.getGene(i));
    offspring2.setGene(i, this.getGene(i));
    taken1.add(offspring1.getGene(i));
    taken2.add(offspring2.getGene(i));
  }
}
```

Now that the middle section has been handled, the two outer sections must be addressed.

```
            // handle outer sections
            for (int i = 0; i < geneLength; i++) {
                  if ((i < cutpoint1) || (i > cutpoint2)) {
                        if (getGeneticAlgorithm().
                              isPreventRepeat()) {
                              offspring1.setGene(i,
                               getNotTaken(this, taken1));
                              offspring2.setGene(i,
                               getNotTaken(father, taken2));
                        } else {
                              offspring1.setGene(i,
                                    this.getGene(i));
                              offspring2.setGene(i,
                                    father.getGene(i));
                        }
                  }
            }
```

Copy the results for the two offspring.

```
            // copy results
            offspring1.calculateCost();
            offspring2.calculateCost();
```

A random number is used to see if the first offspring should be mutated.

```
if (Math.random() < this.geneticAlgorithm.getMutationPercent()) {
  offspring1.mutate();
}
```

Likewise, mutating the second offspring is considered.

```
if (Math.random() <
this.geneticAlgorithm.getMutationPercent()) {
  offspring2.mutate();
}
```

The exact process for mutation varies depending on the problem being solved. For a neural network application, some weight matrix value is scaled by a random percentage. For the traveling salesman problem, two random stops on the trip are swapped.

Multithreading Issues

Most new computers being purchased today are multicore. Multicore processors can execute programs considerably faster if they are written to be multithreaded. Neural network training, in particular, can benefit from multithreading. While backpropagation can be somewhat tricky to multithread, genetic algorithms are fairly easy. The **GeneticAlgorithm** class provided in this book is designed to use a thread pool, if one is provided. A complete discussion of Java's built-in thread-pooling capabilities is beyond the scope of this book; however, the basics will be covered.

To use Java's built-in thread pool, a class must be created that processes one unit of work. The real challenge in writing code that executes well on a multicore processor is breaking the work down into small packets that can be divided among the threads.

Classes that perform these small work packets must implement the **Callable** interface. For the generic genetic algorithm, the work packets are performed by the **MateWorker** class. **MateWorker** objects are created during each iteration of the training. These objects are created for each mating that must occur. Once all of the **MateWorker** objects are created, they are handed off to a thread pool. The **MateWorker** class is shown in Listing 6.1.

Listing 6.1: The MateWorker Class (MateWorker.java)

```
package com.heatonresearch.book.introneuralnet.neural.genetic;

import java.util.concurrent.Callable;

/**
 * MateWorker: This class is used in conjunction with a thread
 * pool.  This allows the genetic algorithm to offload all of
 * those calculations to a thread pool.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class MateWorker<CHROMOSME_TYPE extends Chromosome<?, ?>>
      implements Callable<Integer> {
```

```
	private final CHROMOSME_TYPE mother;
	private final CHROMOSME_TYPE father;
	private final CHROMOSME_TYPE child1;
	private final CHROMOSME_TYPE child2;

	public MateWorker(final CHROMOSME_TYPE mother,
		final CHROMOSME_TYPE father,
	final CHROMOSME_TYPE child1, final CHROMOSME_TYPE child2) {
		this.mother = mother;
		this.father = father;
		this.child1 = child1;
		this.child2 = child2;
	}

	@SuppressWarnings("unchecked")
	public Integer call() throws Exception {
		this.mother.mate((Chromosome)this.father,
				(Chromosome)this.child1,
				(Chromosome)this.child2);
		return null;
	}

}
```

As you can see from the above listing, the class is provided with everything that it needs to mate two chromosomes and produce two offspring. The actual work is done inside the **run** method. The work consists of nothing more than calling the **mate** method of one of the parents.

The Traveling Salesman Problem

In this section you will be introduced to the traveling salesman problem (TSP). Genetic algorithms are commonly used to solve the traveling salesman problem, because the TSP is an NP-hard problem that generally cannot be solved by traditional iterative algorithms.

Understanding the Traveling Salesman Problem

The traveling salesman problem involves a “traveling salesman” who must visit a certain number of cities. The task is to identify the shortest route for the salesman to travel between the cities. The salesman is allowed to begin and end at any city, but must visit each city once. The salesman may not visit a city more than once.

This may seem like an easy task for a normal iterative program, however, consider the speed with which the number of possible combinations grows as the number of cities increases. If there are one or two cities, only one step is required. Three increases the possible routes to six. Table 6.2 shows how quickly these combinations grow.

Table 6.2: Number of Steps to Solve TSP with a Conventional Program

Number of Cities	Number of Steps
1	1
2	1
3	6
4	24
5	120
6	720
7	5,040
8	40,320
9	362,880
10	3,628,800
11	39,916,800
12	479,001,600
13	6,227,020,800
...	...
50	3.041 * 10^64

The formula behind the above table is the factorial. The number of cities, n, is calculated using the factorial operator (!). The factorial of some arbitrary value n is given by n * (n – 1) * (n – 2) * ... * 3 * 2 * 1. As you can see from the above table, these values become incredibly large when a program must do a "brute force" search. The sample program that we will examine in the next section finds a solution to a 50-city problem in a matter of minutes. This is done by using a genetic algorithm, rather than a normal brute-force approach.

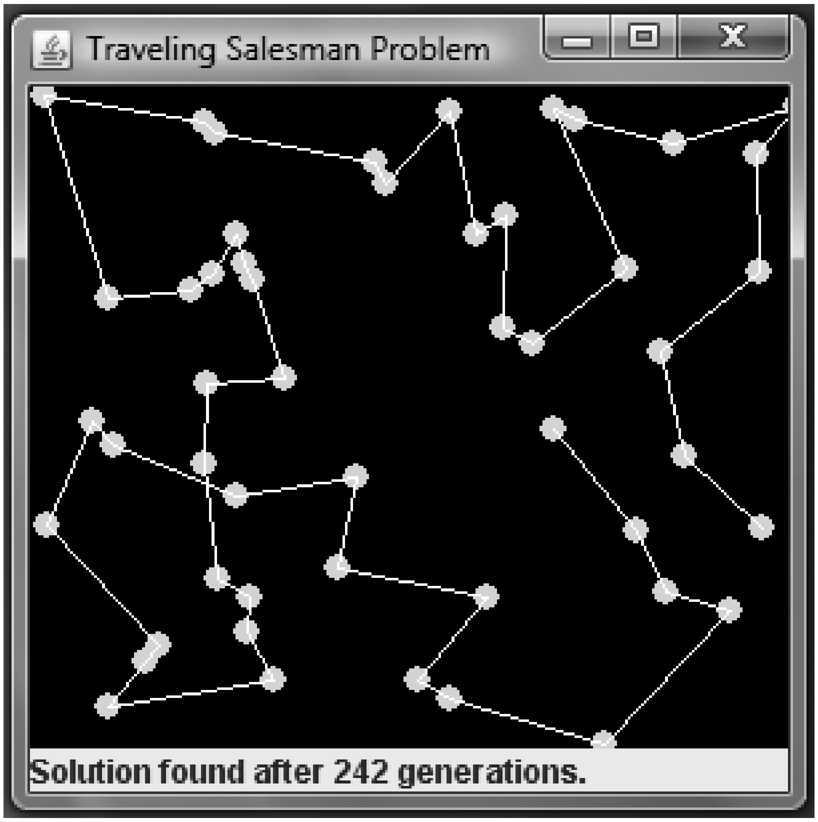
Implementing the Traveling Salesman Problem

So far, we have discussed the basic principles of genetic algorithms and how they are used. Now it is time to examine a Java example. In this section, you will be shown a complete application that is capable of finding solutions to the TSP. As this program is examined, you will be shown how the user interface is constructed, and also how the genetic algorithm itself is implemented.

Using the Traveling Salesman Program

The traveling salesman program itself is very easy to use. This program displays the cities, shown as dots, and the current best solution. The number of generations and the mutation percentage are also shown. As the program runs, these values are updated. The final output from the program is shown in Figure 6.2.

Figure 6.2: The traveling salesman program.



Solution found after 242 generations.

As the program is running, you will see white lines change between the green cities. Eventually, a path will begin to emerge. The path currently being displayed is close to the shortest path in the entire population.

When the program is nearly finished, you will notice that new patterns are not introduced; the program seems to stabilize. Yet, you will also notice that additional generations are still being calculated. This is an important part of the genetic algorithm—knowing when it is done! It is not as straightforward as it might seem. You do not know how many steps are required, nor do you know the shortest distance.

Termination criteria must be specified, so the program will know when to stop. This particular program stops when the optimal solution does not change for 100 generations. Once this has happened, the program indicates that it has found a solution after the number of generations indicated, which includes the 99 generations that did not change the solution. Now that you have seen how this GA program works, we will examine how it was constructed. We will begin by examining the user interface.

Overall Structure

The traveling salesman program uses five Java classes. It is important to understand the relationship between the individual classes that make up the traveling salesman program. These classes, and their functions, are summarized in Table 6.3.

Table 6.3: Classes Used for the GA Version of the Traveling Salesman

Class	Purpose
City	This class stores individual city coordinates. It also contains methods that are used to calculate the distance between cities.
GeneticTravelingSalesman	This class implements the user interface and performs general initialization.
TSPChromosome	This class implements the chromosome. It is the most complex class of the program, as it implements most of the functionality of the genetic algorithm.
TSPGeneticAlgorithm	This class implements the genetic algorithm. It is used to perform the training and process the chromosomes.
WorldMap	This class assists the GeneticTravelingSalesman class by drawing the map of cities.

Most of the work is done by the **`TSPChromosome`** class. This class is covered in the next section.

Traveling Salesman Chromosomes

When implementing a genetic algorithm using the classes provided in this book, you must generally create your own cost calculation method, named **`calculateCost`**, as well as your own mutation function, named **`mutate`**. The signature for the traveling salesman problem's **`calculateCost`** method is shown here:

```
public void calculateCost() throws NeuralNetworkError
```

Calculating the cost for the traveling salesman problem is relatively easy; the distance between each of the cities is summed. The program begins by initializing a running cost variable to zero and looping through the entire list of cities.

```
double cost = 0.0;
for (int i = 0; i < this.cities.length - 1; i++) {
```

For each city, the distance between this city and the next is calculated.

```
  final double dist = this.cities[getGene(i)]
      .proximity(this.cities[getGene(i + 1)]);
```

The distance is added to the total cost.

```
  cost += dist;
}
```

Finally, the cost is saved to an instance variable.

```
setCost(cost);
```

A **`mutate`** method is also provided. The signature for the **`mutate`** method is shown here:

```
public void mutate()
```

First, the length is obtained and two random cities are chosen to be swapped.

```
          final int length = this.getGenes().length;
          final int iswap1 = (int) (Math.random() * length);
          final int iswap2 = (int) (Math.random() * length);
```

The two cities are then swapped.

```
final Integer temp = getGene(iswap1);
setGene(iswap1, getGene(iswap2));
setGene(iswap2, temp);
```

The preceding code shows you how the generic genetic algorithm provided in this book was extended to solve a general problem like the traveling salesman. In the next section, classes will be provided that will allow you to use the generic algorithm to train a neural network using training sets in place of backpropagation.

XOR Operator

In the last chapter, backpropagation was used to solve the XOR operator problem. In this section, you will see how a genetic algorithm, the **`TrainingSetNeuralGeneticAlgorithm`** class, can be used to train for the XOR operator. This version of the XOR solver can be seen in Listing 6.2.

Listing 6.2: XOR with a Genetic Algorithm (GeneticXOR.java)

```
package com.heatonresearch.book.introneuralnet.ch6.xor;

import com.heatonresearch.book.introneuralnet.neural.feedforward.
FeedforwardLayer;
import com.heatonresearch.book.introneuralnet.neural.feedforward.
FeedforwardNetwork;
import com.heatonresearch.book.introneuralnet.neural.feedforward.
train.genetic.TrainingSetNeuralGeneticAlgorithm;
```

```
/**
 * Chapter 6: Training using a Genetic Algorithm
 *
 * XOR: Learn the XOR pattern with a feedforward neural network
 * that uses a genetic algorithm.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class GeneticXOR {
      public static double XOR_INPUT[][] = { { 0.0, 0.0 },
            { 1.0, 0.0 },
                  { 0.0, 1.0 }, { 1.0, 1.0 } };

      public static double XOR_IDEAL[][] = { { 0.0 }, { 1.0 },
      { 1.0 }, { 0.0 } };

      public static void main(final String args[]) {
            FeedforwardNetwork network = new FeedforwardNetwork();
            network.addLayer(new FeedforwardLayer(2));
            network.addLayer(new FeedforwardLayer(3));
            network.addLayer(new FeedforwardLayer(1));
            network.reset();

            // train the neural network
            final TrainingSetNeuralGeneticAlgorithm train =
                  new TrainingSetNeuralGeneticAlgorithm(
                        network, true, XOR_INPUT, XOR_IDEAL,
                        5000, 0.1, 0.25);

            int epoch = 1;

            do {
                  train.iteration();
                  System.out
                              .println("Epoch #" + epoch +
                        " Error:" + train.getError());
                  epoch++;
            } while ((epoch < 5000) && (train.getError()
                  > 0.001));

            network = train.getNetwork();

            // test the neural network
            System.out.println("Neural Network Results:");
            for (int i = 0; i < XOR_IDEAL.length; i++) {
                  final double actual[] = network.
computeOutputs(XOR_INPUT[i]);
                  System.out.println(XOR_INPUT[i][0] + ","
                        + XOR_INPUT[i][1]
                              + ", actual=" + actual[0] + ",
                              ideal=" + XOR_IDEAL[i][0]);
            }
      }
}
```

The above listing is very similar to the XOR solver from the last chapter. The main difference is in the following lines:

```
final TrainingSetNeuralGeneticAlgorithm train =
      new TrainingSetNeuralGeneticAlgorithm(
      network, true, XOR_INPUT, XOR_IDEAL, 5000, 0.1, 0.25);
```

As you can see, several parameters are passed to the constructor of the training class. First, the network to be trained is passed. The value of **true** specifies that all of the initial life forms should have their weight matrixes randomly initialized. The input and ideal arrays are also provided, just as they are with the backpropagation algorithm. The value of 5000 specifies the size of the population. The value 0.1 indicates that 10% of the population will be chosen to mate. The 10% will be able to mate with any life form in the top 25%.

Calculate Cost

A specialized method is then used to calculate the cost for the XOR pattern recognition. The signature for the cost calculation method is shown here:

```
public void calculateCost() throws NeuralNetworkError {
```

First, the contents of the chromosome are copied back into the neural network.

```
this.updateNetwork();
```

The root mean square error of the neural network can be used as a cost. The RMS error will be calculated just as it was in previous chapters. The **input** and **ideal** arrays are copied to local variables.

```
final double input[][] = this.getGeneticAlgorithm().getInput();

final double ideal[][] = this.getGeneticAlgorithm().getIdeal();
```

The neural network's **calculateError** function is used to calculate the error of the neural network given the **input** and **ideal** arrays.

```
setCost(getNetwork().calculateError(input, ideal));
```

Although the XOR program uses a specialized cost calculation method, the program uses the same mating method as previously discussed.

Mutate

Mutation is handled by a specialized **mutate** function. The signature for this mutate function is shown here:

```
public void mutate()
```

The mutation method begins by obtaining the number of genes in a chromosome. Each gene will be scaled using a random percentage.

```
final int length = getGenes().length;
```

The function loops through all of the genes.

```
for (int i = 0; i < length; i++) {
```

It obtains a gene's value and multiplies it by a random ratio within a specified range.

```
  double d = getGene(i);
  final double ratio = (int) ((RANGE * Math.random()) - RANGE);

  d*=ratio;
  setGene(i,d);
}
```

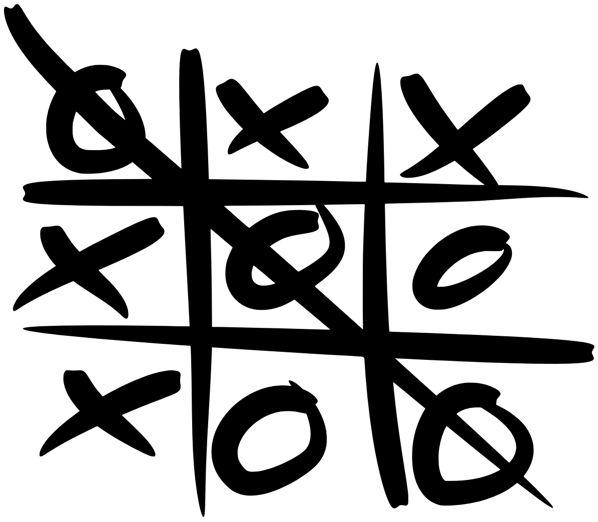
The result is that each weight matrix element is randomly changed.

Tic-Tac-Toe

The game of tic-tac-toe, also called naughts and crosses in many parts of the world, can be an interesting application of neural networks trained by genetic algorithms. Tic-tac-toe has very simple rules. Most human players quickly grow bored with the game, since it is so easy to learn. Once two people have mastered tic-tac-toe, games almost always result in a tie.

The simple rules for tic-tac-toe are as follows: One player plays X and another plays O. They take turns placing these characters on a 3x3 grid until one player gets three in a row. If the grid is filled before one of them gets three in a row, then the game is a draw. Figure 6.3 shows a game of tic-tac-toe in progress.

Figure 6.3: The game of tic-tac-toe.



There are many implementations of tic-tac-toe in Java. This book makes use of one by Thomas David Baker, which was released as open source software. It provides several players that can be matched for games:

- Boring – Just picks the next open spot.
- Human – Allows a human to play.
- Logical – Uses logic to play a near perfect game.
- MinMax – Uses the min-max algorithm to play a perfect game.
- Random – Moves to random locations.

Each player can play any other player. A class simply has to implement the `Player` interface and it can play against the others. Some of the players are more advanced. This gives the neural network several levels of players to play against.

The most advanced player is the min-max player. This player uses a min-max algorithm. A min-max algorithm uses a tree to plot out every possible move. This technique can be used in a game as simple as tic-tac-toe; however, it would not be effective for a game with an extremely large number of combinations, such as chess or go.

The example for this chapter creates a class named `PlayerNeural`. This class uses a neural network to play against the other players. The neural player will not play a perfect game. Yet, it will play reasonably well against some of the provided players.

The goal is not to produce a perfect player using only a neural network, but to demonstrate the use of neural networks. Many games use a hybrid approach for optimal results. The min-max algorithm, together with a neural network to prune entire branches of the search tree can be a very effective combination. We will, however, implement a pure neural-network approach.

The tic-tac-toe example illustrates a very important concept. Thus far, we have only trained neural networks using training sets. The tic-tac-toe example will not use training sets. Rather, the neural network will be matched against any of the other players and a genetic algorithm will be used to train it. Since a genetic algorithm uses "natural selection" it is particularly well suited to learning to play a game.

Using the Sample Tic-Tac-Toe Implementation

There are many options available with this version of tic-tac-toe. There are several command line options that allow you to specify which two players will play against each other. The general format of a command is as follows:

```
NeuralTicTacToe [Command] [Player 1] [Player 2]
```

The command also specifies the mode in which the program should run. The following modes are available:

- game – Play a single game.
- match – Play 100 games.
- train – Train (and save) a neural network.

You must choose the two players. The following options are available:

- Boring
- Human
- Logic
- MinMax
- NeuralBlank
- NeuralLoad
- Random

For example, to play a match between the MinMax algorithm and the Random player, use the following command:

```
NeuralTicTacToe Match MinMax Random
```

If you would like to train a new blank neural network against the random player, you would use the following command:

```
NeuralTicTacToe Train NeuralBlank Random
```

This will train a blank neural network; but be aware it can take a considerable amount of time. The genetic algorithm will use a thread pool, so a multicore computer will help. Once the training is complete, the neural network will be saved to disk. It will be saved with the name "tictactoe.net". The download for this book contains an example "tictactoe.net" file that is already trained for tic-tac-toe. This neural network took nearly 20 hours to train on my computer. Appendix A explains how to download the examples for this book.

To play a trained neural network, use the following command:

```
NeuralTicTacToe Play NeuralLoad Human
```

The sample neural network will always load and save a neural network named "tictactoe.net".

Saving and Loading Neural Networks

You may be wondering how to load and save the neural networks in this book. These networks can be loaded and saved using regular Java serialization techniques. For example, the following command saves a neural network:

```
SerializeObject.save("filename.net", neuralNetwork);
```

Loading a neural network is almost as easy. The following command loads a neural network from disk.

```
FeedforwardNetwork result =
(FeedforwardNetwork)SerializeObject.load("neuralnet.net");
```

Some of the neural networks in this book take a considerable amount of time to train. Thus, it is valuable to save the neural network so it can be quickly reloaded later.

Structure of the Tic-Tac-Toe Neural Network

One of the most important questions that a neural network programmer must always consider is how to structure the neural network. Perhaps the most obvious way to structure a neural network for tic-tac-toe is with nine inputs and nine outputs. The nine inputs will specify the current board configuration. The nine outputs will allow the neural network to specify where it wants to move.

The first version of the example used this configuration. It did not work particularly well. One problem is that the neural network had to spend considerable time just learning the valid and invalid moves. Furthermore, this configuration did not play very well against the other players.

The final version of the neural network has a configuration with nine inputs and a single output. Rather than asking the neural network where to move, this structure asks the neural network if a proposed board position is favorable. Whenever it is the neural player's turn, every possible move is determined. There can be at most nine possible moves. A temporary board is constructed that indicates what the game board would look like after each of these possible moves. Each board is then presented to the neural network. The board position that receives the highest value from the output neuron will be the next move.

Neural Player

The **`Player`** interface requires that the **`NeuralPlayer`** class include a **`getMove`** method. This method determines the next move the neural player will make. The signature for the **`getMove`** method is shown here:

```
public Move getMove(final byte[][] board, final Move prev, final byte
color)
```

First, two local variables are established. The **`bestMove`** variable will hold the best move found so far. The **`bestScore`** variable will hold the score of the **`bestMove`** variable.

```
Move bestMove = null;
double bestScore = Double.MIN_VALUE;
```

Next, all of the potential moves on the board's grid are examined.

```
for (int x = 0; x < board.length; x++) {
  for (int y = 0; y < board.length; y++) {
```

A sample board is constructed to represent this potential move.

```
    final Move move = new Move((byte) x, (byte) y, color);
```

The potential move is examined to determine if it is valid.

```
    if (Board.isEmpty(board, move)) {
```

If the potential move is valid, then the **`tryMove`** method is called to see what score its board position would provide.

```
      final double d = tryMove(board, move);
```

If the score for that board position beats the current best score, then this move is saved as the best move encountered thus far.

```
      if ((d > bestScore) || (bestMove == null)) {
        bestScore = d;
        bestMove = move;
      }
    }
  }
}
```

Finally, the best move is returned.

```
return bestMove;
```

The **getMove** method makes use of the **tryMove** method to build a temporary board for each possible move. The signature for the **tryMove** method is shown here:

```
private double tryMove(final byte[][] board, final Move move) {
```

First, an **input** array is constructed for the neural network. A local variable **index** is kept that will remember the current position within the **input** array.

```
final double input[] = new double[9];
int index = 0;
```

Next, every position of the board grid is considered.

```
for (int x = 0; x < board.length; x++) {
  for (int y = 0; y < board.length; y++) {
```

Each square on the grid is checked and the input array is set to reflect the status of that square.

```
    if (board[x][y] == TicTacToe.NOUGHTS) {
      input[index] = -1;
    } else if (board[x][y] == TicTacToe.CROSSES) {
      input[index] = 1;
    } else if (board[x][y] == TicTacToe.EMPTY) {
      input[index] = 0;
    }
```

If the square contains an “X” (cross), then a value of –1 is inserted into the input array. If the square contains an “O” (nought), then a value of 1 is placed in the array.

If the square contains the current move, then the input is set to –1, or a value of “X,” which is what the neural player is playing.

```
    if ((x == move.x) && (y == move.y)) {
      input[index] = -1;
    }
```

The next element of the input array is then examined.

```
  index++;
  }
}
```

Finally, the output for this input array is computed.

```
final double output[] = this.network.computeOutputs(input);
return output[0];
```

The output is returned to the caller. The higher this value, the more favorable the board position.

Chromosomes

The tic-tac-toe neural network is trained using a genetic training algorithm. Each chromosome is tested by playing 100 games against its opponent. This causes a score to be generated that allows each chromosome's effectiveness to be evaluated. Each chromosome has a **`calculateCost`** method that performs this operation. The signature for **`calculateCost`** is shown here:

```
public void calculateCost() {
```

First, the neural network is updated using the gene array.

```
try {
  this.updateNetwork();
```

Next, a **`PlayerNeural`** player is constructed to play against the chosen component. To keep this example simple, the neural network player is always player one, and thus always moves first.

```
  final PlayerNeural player1 = new PlayerNeural(getNetwork());
```

The second player is created, and a match is played.

```
  Player player2;

  player2 = (Player) this.getGeneticAlgorithm().getOpponent()
                            .newInstance();
  final ScorePlayer score = new ScorePlayer(player1, player2, 
false);
```

The match is managed by the **`ScorePlayer`** class. This class allows two players to play 100 games and a score is generated.

```
  setCost(score.score());
```

The score then becomes the cost for this chromosome.

Chapter Summary

In this chapter, you were introduced to genetic algorithms. Genetic algorithms provide one approach for finding potential solutions to complex NP-hard problems. An NP-hard problem is a problem for which the number of steps required to solve the problem increases at a very high rate as the number of units in the program increases.

An example of an NP-hard problem, which was examined in this chapter, is the traveling salesman problem. The traveling salesman problem attempts to identify the shortest path for a salesman traveling to a certain number of cities. The number of possible paths that a program has to search increases factorially as the number of cities increases.

To solve such a problem, a genetic algorithm is used. The genetic algorithm creates a population of chromosomes. Each of the chromosomes is one path through the cities. Each leg in that journey is a gene. The best chromosomes are determined and they are allowed to "mate." The mating process combines the genes of two parents. The chromosomes that have longer, less desirable, paths are not allowed to mate. Because the population has a fixed size, the less desirable chromosomes are purged from memory. As the program continues, natural selection causes the better-suited chromosomes to mate and produce better and better solutions.

The actual process of mating occurs by splitting the parent chromosomes into three splices. These splices are then used to build new chromosomes. The result of all of this will be two offspring chromosomes. Unfortunately, the mating process does not introduce new genetic material. New genetic material is introduced through mutation.

Mutation randomly changes the genes of some of the newly created offspring. This introduces new traits. Many of these mutations will not be well suited for the particular problem and will be purged from memory. However, others can be used to advance an otherwise stagnated population. Mutation is also introduced to help find an optimal solution.

Thus far in this book you have been shown how to train neural networks with backpropagation and genetic algorithms. Neural networks can also be trained by a technique that simulates the way a molten metal cools. This process is called simulated annealing. Simulated annealing will be covered in the next chapter.

Vocabulary

Chromosome

Evolution

Gene

Mate

Min-Max Algorithm

Multicore

Mutate

Population

Thread Pool

Questions for Review

1. What is the role of a chromosome in a genetic algorithm? What is the role of a gene?

2. How can a neural network be expressed as a chromosome and genes?

3. What is the role of mutation in a genetic algorithm?

4. How can the cost be calculated for a potential traveling salesman solution? For a neural network?

5. For what types of problems are genetic algorithms best suited? For what types of problems are they not well suited?