Chapter 3

Hopfield & Boltzmann Machines

- Hopfield Networks
- Energy Functions
- Hebbian Learning
- Associative Memory
- Optimization
- Boltzmann Machines

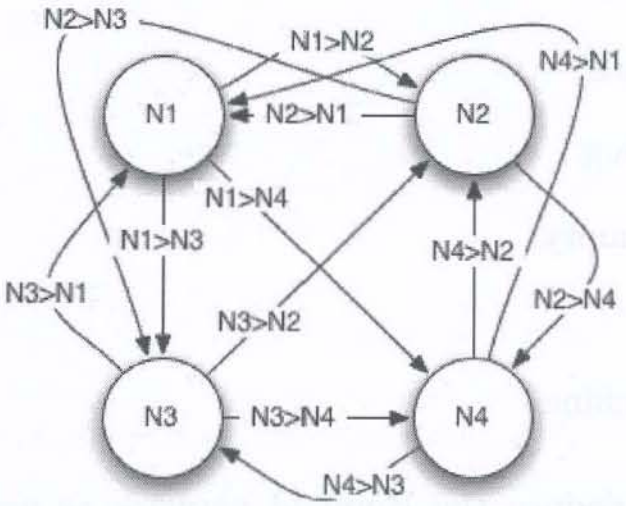
This chapter will introduce the Hopfield network as well as the Boltzmann machine. Though neither of these classic neural networks is used extensively in modern AI applications, both are foundational to more modern algorithms. The Boltzmann machine forms the foundation of the deep belief neural network (DBNN), which is one of the fundamental algorithms of deep learning. Hopfield networks are a very simple type of neural network that utilizes many of the same features that the more complex feedforward neural networks employ.

3.1 Hopfield Neural Networks

The Hopfield neural network (Hopfield, 1982) is perhaps the simplest type of neural network because it is a fully connected single layer, auto-associative network. In other words, it has a single layer in which each neuron is connected to every other neuron. Additionally, the term auto-associative means that the neural network will return the entire pattern if it recognizes a pattern. As a result, the network will fill in the gaps of incomplete or distorted patterns.

Figure 3.1 shows a Hopfield neural network with just four neurons. While a four-neuron network is handy because it is small enough to visualize, it can recognize a few patterns.

Figure 3.1: A Hopfield Neural Network with 12 Connections



Because every neuron in a Hopfield neural network is connected to every other neuron, you might assume that a four-neuron network would contain a four-by-four matrix, or 16 connections. However, 16 connections would require that every neuron be connected to itself as well as to every other neuron. In a Hopfield neural network, 16 connections do not occur; the actual number of connections is 12.

These connections are weighted and stored in a matrix. A four-by-four matrix would store the network pictured above. In fact, the diagonal of this matrix would contain 0's because there are no self-connections. All neural

network examples in this book will use some form of matrix to store their weights.

Each neuron in a Hopfield network has a state of either true (1) or false (-1). These states are initially the input to the Hopfield network and ultimately become the output of the network. To determine whether a Hopfield neuron's state is -1 or 1, use Equation 3.1:

$$s_i \leftarrow \begin{cases} +1 & \text{if } \sum_j w_{ij} s_j \geq \theta_i, \\ -1 & \text{otherwise.} \end{cases} \tag{3.1}$$

The above equation calculates the state (s) of neuron i. The state of a given neuron greatly depends on the states of the other neurons. The equation multiplies and sums the weight (w) and state (s) of the other neurons (j). Essentially, the state of the current neuron (i) is +1 if this sum is greater than the threshold (θ, theta). Otherwise it is -1. The threshold value is usually 0.

Because the state of a single neuron depends on the states of the remaining neurons, the order in which the equation calculates the neurons is very important. Programmers frequently employ the following two strategies to calculate the states for all neurons in a Hopfield network:

- **Asynchronous**: This strategy updates only one neuron at a time. It picks this neuron at random.
- **Synchronous**: It updates all neurons at the same time. This method is less realistic since biological organisms lack a global clock that synchronizes the neurons.

You should typically run a Hopfield network until the values of all neurons stabilize. Despite the fact that each neuron is dependent on the states of the others, the network will usually converge to a stable state.

It is important to have some indication of how close the network is to converging to a stable state. You can calculate an energy value for Hopfield networks. This value decreases as the Hopfield network moves to a more stable state. To evaluate the stability of the network, you can use the energy function. Equation 3.2 shows the energy calculation function:

$$E = -\left(\sum_{i<j} w_{ij}\, s_i\, s_j + \sum_i \theta_i\, s_i\right) \tag{3.2}$$

Boltzmann machines, discussed later in the chapter, also utilize this energy function. Boltzmann machines share many similarities with Hopfield neural networks. When the threshold is 0, the second term of Equation 3.2 drops out. Listing 3.1 contains the code to implement Equation 3.1:

Listing 3.1: Hopfield Energy

```
def energy(weights,state,threshold):
  # First term
  a = 0
  for i in range(neuron_count):
    for j in range(neuron_count):
      a = a + weight[i][j] * state[i] * state[j]

  a = a * -0.5
  # Second term
  b = 0
  for i in range(neuron_count):
    b = b + state[i] * threshold[i]

  # Result
  return a + b
```

3.1.1 Training a Hopfield Network

You can train Hopfield networks to arrange their weights in a way that allows the network to converge to desired patterns, also known as the training set.

These desired training patterns are a list of patterns with a Boolean value for each of the neurons that comprise the Boltzmann machine. The following data might represent a four-pattern training set for a Hopfield network with eight neurons:

```
1 1 0 0 0 0 0 0
0 0 0 0 1 1 0 0
1 0 0 0 0 0 0 1
0 0 0 1 1 0 0 0
```

The above data are completely arbitrary; however, they do represent actual patterns to train the Hopfield network. Once trained, a pattern similar to the one listed below should find equilibrium with a pattern close to the training set:

```
1 1 1 0 0 0 0 0
```

Therefore, the state of the Hopfield machine should change to the following pattern:

```
1 1 0 0 0 0 0 0
```

You can train Hopfield networks with either Hebbian (Hopfield, 1982) or Storkey (Storkey, 1999) learning. The Hebbian process for learning is biologically plausible, and it is often expressed as, "cells that fire together, wire together." In other words, two neurons will become connected if they frequently react to the same input stimulus. Equation 3.3 summarizes this behavior mathematically:

$$w_{ij} = \frac{1}{n} \sum_{\mu=1}^{n} \epsilon_i^\mu \epsilon_j^\mu \tag{3.3}$$

The constant n represents the number of training set elements (ϵ, epsilon). The weight matrix will be square and will contain rows and columns equal to the number of neurons. The diagonal will always be 0 because a neuron is not connected to itself. The other locations in the matrix will contain values specifying how often two values in the training pattern are either +1 or -1. Listing 3.2 contains the code to implement Equation 3.3:

Listing 3.2: Hopfield Hebbian Training

```
def add_pattern(weights,pattern,n):
  for i in range(neuron_count):
    for j in range(neuron_count):
      if i==j:
        weights[i][j] = 0
      else:
        weights[i][j] = weights[i][j]
          + ((pattern[i] * pattern[j])/n)
```

We apply the **add_pattern** method to add each of the training elements. The parameter **weights** specifies the weight matrix, and the parameter **pattern** specifies each individual training element. The variable **n** designates the number of elements in the training set.

It is possible that the equation and the code are not sufficient to show how the weights are generated from input patterns. To help you visualize this process, we provide an online Javascript application at the following URL:

http://www.heatonresearch.com/aifh/vol3/hopfield.html

Consider the following data to train a Hopfield network:

```
[1 ,0 ,0 ,1]
[0 ,1 ,1 ,0]
```

The previous data should produce a weight matrix like Figure 3.2:

Figure 3.2: Hopfield Matrix

	0	1	2	3
0	0	0	0	0.5
1	0	0	0.5	0
2	0	0.5	0	0
3	0.5	0	0	0

To calculate the above matrix, divide 1 by the number of training set elements. The result is 1/2, or 0.5. The value 0.5 is placed into every row and column that has a 1 in the training set. For example, the first training element has a 1 in neurons #0 and #3, resulting in a 0.5 being added to row 0, column 3 and row 3, column 0. The same process continues for the other training set element.

Another common training technique for Hopfield neural networks is the Storkey training algorithm. Hopfield neural networks trained with Storkey

have a greater capacity of patterns than the Hebbian method just described. The Storkey algorithm is more complex than the Hebbian algorithm.

The first step in the Storkey algorithm is to calculate a value called the local field. Equation 3.4 calculates this value:

$$h_{ij} = \sum_{k=1,k\neq i,j} w_{ik}\epsilon_k \tag{3.4}$$

We calculate the local field value (h) for each weight element (i & j). Just as before, we use the weights (w) and training set elements (ϵ, epsilon). Listing 3.3 provides the code to calculate the local field:

Listing 3.3: Calculate Storkey Local Field

```
def calculate_local_field(weights, i, j, pattern):
  sum = 0
  for k in range(len(pattern)):
    if k != i:
      sum = sum + weights[i][k] * pattern[k]
  return sum
```

Equation 3.5 has the local field value that calculates the needed change (ΔW):

$$\Delta w_{ij} = \frac{1}{n}\epsilon_i\epsilon_j - \frac{1}{n}\epsilon_i h_{ji} - \frac{1}{n}\epsilon_j h_{ij} \tag{3.5}$$

Listing 3.4 calculates the values of the weight deltas:

Listing 3.4: Storkey Learning

```
def add_pattern(weights, pattern):
  sum_matrix = matrix(len(pattern),len(pattern))
  n = len(pattern)
  for i in range(n):
    for j in range(n):
      t1 = (pattern[i] * pattern[j])/n
      t2 = (pattern[i] *
         calculate_local_field(weights,j,i,pattern))/n
      t3 = (pattern[j] *
         calculate_local_field(weights,i,j,pattern))/n
      d = t1-t2-t3;
      sum_matrix[i][j] = sum_matrix[i][j] + d
  return sum_matrix
```

Once you calculate the weight deltas, you can add them to the existing weight matrix. If there is no existing weight matrix, simply allow the delta weight matrix to become the weight matrix.

3.2 Hopfield-Tank Networks

In the last section, you learned that Hopfield networks can recall patterns. They can also optimize problems such as the traveling salesman problem (TSP). Hopfield and Tank (1984) introduced a special variant, the Hopfield-Tank network, to find solutions to optimization problems.

The structure of a Hopfield-Tank network is somewhat different than a standard Hopfield network. The neurons in a regular Hopfield neural network can hold only the two discrete values of 0 or 1. However, a Hopfield-Tank neuron can have any number in the range 0 to 1. Standard Hopfield networks possess discrete values; Hopfield-Tank networks keep continuous values over a range. Another important difference is that Hopfield-Tank networks use sigmoid activation functions.

To utilize a Hopfield-Tank network, you must create a specialized energy function to express the parameters of each problem to solve. However, producing such an energy function can be a time-consuming task. Hopfield & Tank (2008) demonstrated how to construct an energy function for the traveling salesman problem (TSP). Other optimization functions, such as simulated annealing and Nelder-Mead, do not require the creation of a complex energy function. These general-purpose optimization algorithms typically perform better than the older Hopfield-Tank optimization algorithms.

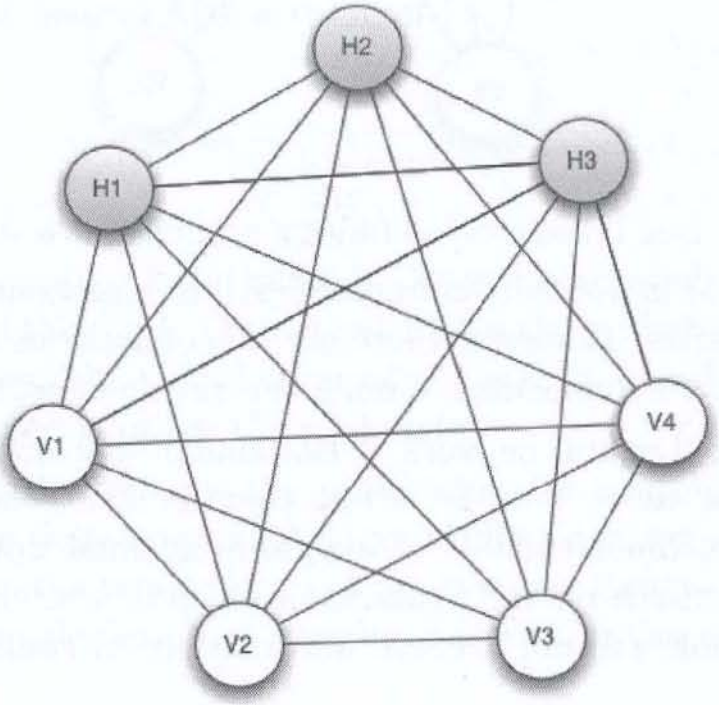
Because other algorithms are typically better choices for optimizations, this book does not cover the optimization Hopfield-Tank network. Nelder-Mead and simulated annealing were demonstrated in *Artificial Intelligence for Humans, Volume 1: Fundamental Algorithms*. Chapter 6, "Backpropagation Training," will have a review of stochastic gradient descent (SGD), which is one of the best training algorithms for feedforward neural networks.

3.3 Boltzmann Machines

Hinton & Sejnowski (1985) first introduced Boltzmann machines, but this neural network type has not enjoyed widespread use until recently. A special type of Boltzmann machine, the restricted Boltzmann machine (RBM), is one of the foundational technologies of deep learning and the deep belief neural network (DBNN). In this chapter, we will introduce classic Boltzmann machines. Chapter 9, "Deep Learning," will include deep learning and the restricted Boltzmann machine.

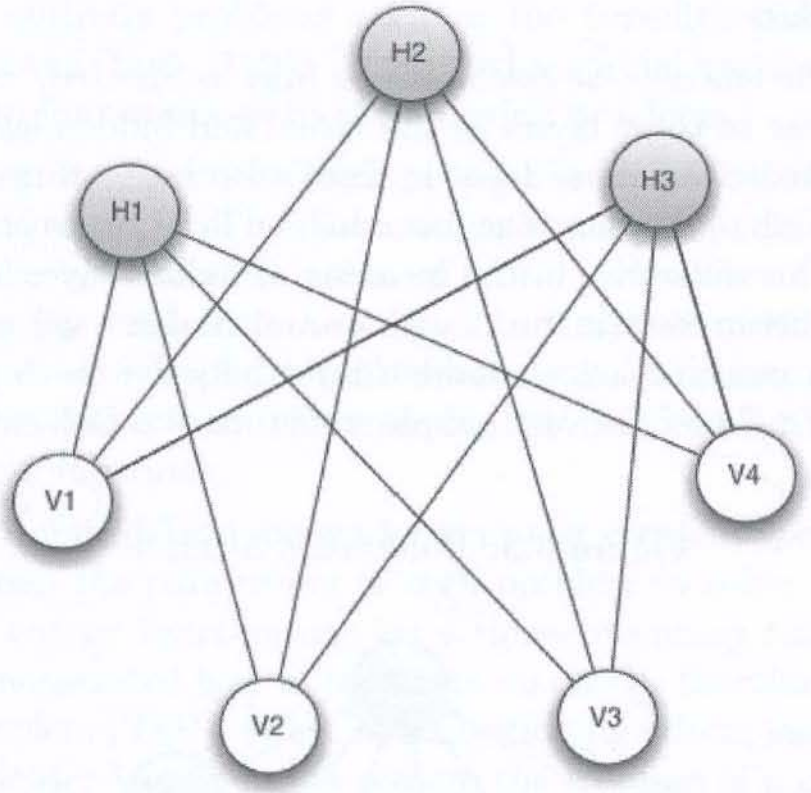
A Boltzmann machine is essentially a fully connected, two-layer neural network. We refer to these layers as the visual and hidden layers. The visual layer is analogous to the input layer in feedforward neural networks. Despite the fact that a Boltzmann machine has a hidden layer, it functions more as an output layer. This difference in the meaning of hidden layer is often a source of confusion between Boltzmann machines and feedforward neural networks. The Boltzmann machine has no hidden layer between the input and output layers. Figure 3.3 shows the very simple structure of a Boltzmann machine:

Figure 3.3: Boltzmann Machine



The above Boltzmann machine has three hidden neurons and four visible neurons. A Boltzmann machine is fully connected because every neuron has a connection to every other neuron. However, no neuron is connected to itself. This connectivity is what differentiates a Boltzmann machine from a restricted Boltzmann machine (RBM), as seen in Figure 3.4:

Figure 3.4: Restricted Boltzmann Machine (RBM)



The above RBM is not fully connected. All hidden neurons are connected to each visible neuron. However, there are no connections among the hidden neurons nor are there connections among the visible neurons.

Like the Hopfield neural network, a Boltzmann machine's neurons acquire only binary states, either 0 or 1. While there is some research on continuous Boltzmann machines capable of assigning decimal numbers to the neurons, nearly all research on the Boltzmann machine centers on binary units. Therefore, this book will not include information on continuous Boltzmann machines.

Boltzmann machines are also called a generative model. In other words, a Boltzmann machine does not generate constant output. The values presented to the visible neurons of a Boltzmann machine, when considered with the weights, specify a probability that the hidden neurons will assume a value of 1, as opposed to 0.

Although a Boltzmann machine and Hopfield neural networks have some characteristics in common, there are several important differences:

- Hopfield networks suffer from recognizing certain false patterns.
- Boltzmann machines can store a greater capacity of patterns than Hopfield networks.
- Hopfield networks require the input patterns to be uncorrelated.
- Boltzmann machines can be stacked to form layers.

3.3.1 Boltzmann Machine Probability

When the program queries the value of 1 of the Boltzmann machine's hidden neurons, it will randomly produce a 0 or 1. Equation 3.6 obtains the calculated probability for that neuron with a value of 1:

$$p_{\text{i=on}} = \frac{1}{1 + \exp(-\frac{\Delta E_i}{T})} \tag{3.6}$$

The above equation will calculate a number between 0 and 1 that represents a probability. For example, if the value 0.75 were generated, the neuron would return a 1 in 75% of the cases. Once it calculates the probability, it can produce the output by generating a random number between 0 and 1 and returning 1 if the random number is below the probability.

The above equation returns the probability for neuron i being on and is calculated with the delta energy (ΔE) at i. The equation also uses the value T, which represents the temperature of the system. Equation 3.2, from earlier in the chapter, can calculate T. The value θ (theta) is the neuron's bias value.

The change in energy is calculated using Equation 3.7:

$$\Delta E_i = \sum_j w_{ij} s_j + \theta_i \tag{3.7}$$

This value is the energy difference between 1 (on) and 0 (off) for neuron i. It is calculated using the θ (theta), which represents the bias.

Although the values of the individual neurons are stochastic (random), they will typically fall into equilibrium. To reach this equilibrium, you can repeatedly calculate the network. Each time, a unit is chosen while Equation 3.6 sets its state. After running for an adequate period of time at a certain temperature, the probability of a global state of the network will depend only upon that global state's energy.

In other words, the log probabilities of global states become linear in their energies. This relationship is true when the machine is at thermal equilibrium, which means that the probability distribution of global states has converged. If we start running the network from a high temperature and gradually decrease it until we reach a thermal equilibrium at a low temperature, then we may converge to a distribution where the energy level fluctuates around the global minimum. We call this process simulated annealing.

3.4 Applying the Boltzmann Machine

Most research around Boltzmann machines has moved to the restricted Boltzmann machine (RBM) that we will explain in Chapter 9, "Deep Learning." In this section, we will focus on the older, unrestricted form of the Boltzmann, which has been applied to both optimization and recognition problems. We will demonstrate an example of each type, beginning with an optimization problem.

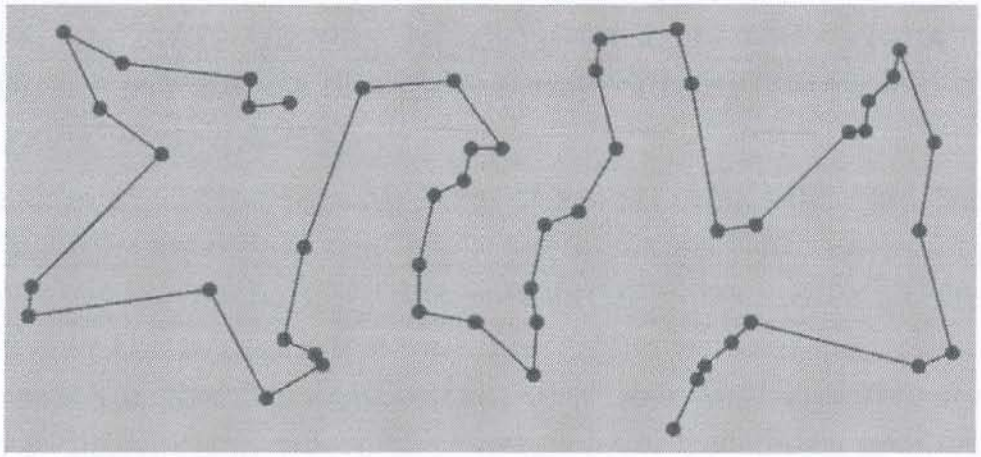
3.4.1 Traveling Salesman Problem

The traveling salesman problem (TSP) is a classic computer science problem that is difficult to solve with traditional programming techniques. Artificial intelligence can be applied to find potential solutions to the TSP. The program

must determine the order of a fixed set of cities that minimizes the total distance covered. The traveling salesman is called a combinational problem. If you are already familiar with TSP or you have read about it in a previous volume in this series, you can skip this section.

TSP involves determining the shortest route for a traveling salesman who must visit a certain number of cities. Although he can begin and end in any city, he may visit each city only once. The TSP has several variants, some of which allow multiple visits to cities or assign different values to cities. The TSP in this chapter simply seeks the shortest possible route to visit each city one time. Figure 3.5 shows the TSP problem used here, as well as a potential shortest route:

Figure 3.5: The Traveling Salesman



Finding the shortest route may seem like an easy task for a normal iterative program. However, as the number of cities increases, the number of possible combinations increases drastically. If the problem has one or two cities, only one or two routes are possible. If it includes three cities, the possible routes increase to six. The following list shows how quickly the number of paths grows:

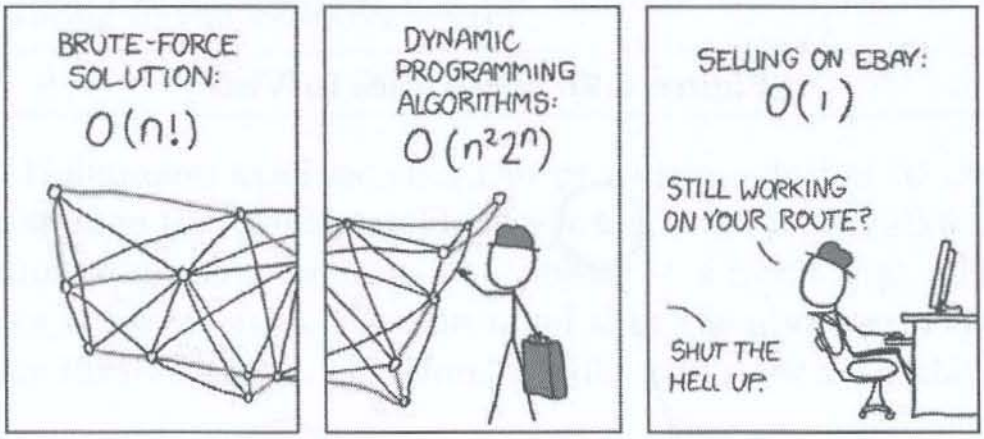
```
1 city has 1 path
2 cities have 2 paths
3 cities have 6 paths
4 cities have 24 paths
5 cities have 120 paths
6 cities have 720 paths
7 cities have 5,040 paths
8 cities have 40,320 paths
9 cities have 362,880 paths
10 cities have 3,628,800 paths
11 cities have 39,916,800 paths
12 cities have 479,001,600 paths
```

```
13 cities have 6,227,020,800 paths
...
50 cities have 3.041 * 10^64 paths
```

In the above table, the formula to calculate total paths is the factorial. The number of cities, n, is calculated using the factorial operator ($!$). The factorial of some arbitrary value n is given by $n * (n - 1) * (n - 2) * ... * 3 * 2 * 1$. These values become incredibly large when a program must do a brute-force search. The traveling salesman problem is an example of a non-deterministic polynomial time (NP) hard problem. Informally, NP-hard is defined as any problem that lacks an efficient way to verify a correct solution. The TSP fits this definition for more than 10 cities. You can find a formal definition of NP-hard in *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Garey, 1979).

Dynamic programming is another common approach to the traveling salesman problem, as seen in xkcd.com comic in Figure 3.6:

Figure 3.6: The Traveling Salesman (from xkcd.com)



Although this book does not include a full discussion of dynamic programming, understanding its essential function is valuable. Dynamic programming breaks a large problem, such as the TSP, into smaller problems. You can reuse work for many of the smaller programs, thereby decreasing the amount of iterations required by a brute-force solution.

Unlike brute-force solutions and dynamic programming, a genetic algorithm is not guaranteed to find the best solution. Although it will find a good

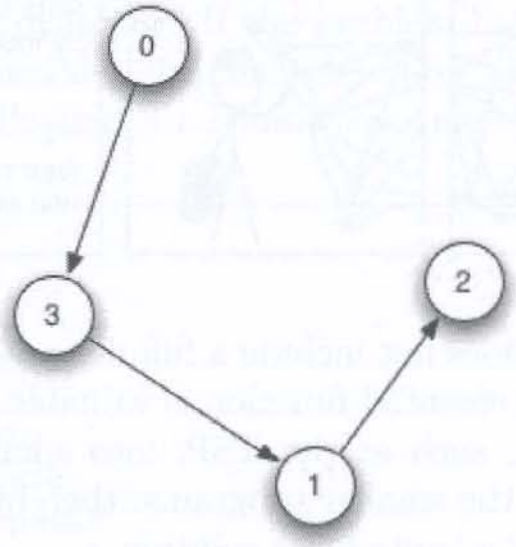
solution, the score might not be the best. The sample program examined in the next section shows how a genetic algorithm produced an acceptable solution for the 50-city problem in a matter of minutes.

3.4.2 Optimization Problems

To use the Boltzmann machine for an optimization problem, it is necessary to represent a TSP solution in such a way that it fits onto the binary neurons of the Boltzmann machine. Hopfield (1984) devised an encoding for the TSP that both Boltzmann and Hopfield neural networks commonly use to represent this combinational problem.

The algorithm arranges the neurons of the Hopfield or Boltzmann machine on a square grid with the number of rows and columns equal to the number of cities. Each column represents a city, and each row corresponds to a segment in the journey. The number of segments in the journey is equal to the number of cities, resulting in a square grid. Each row in the matrix should have exactly one column with a value of 1. This value designates the destination city for each of the trip segments. Consider the city path shown in Figure 3.7:

Figure 3.7: Four Cities to Visit



Because the problem includes four cities, the solution requires a four-by-four grid. The first city visited is City #0. Therefore, the program marks 1 in the first column of the first row. Likewise, visiting City #3 second produces a 1 in the final column of the second row. Figure 3.8 shows the complete path:

Figure 3.8: Encoding of Four Cities

	City #0	City #1	City #2	City #3
Stop #0	1	0	0	0
Stop #1	0	0	0	1
Stop #2	0	1	0	0
Stop #3	0	0	1	0

Of course, the Boltzmann machines do not arrange neurons in a grid. To represent the above path as a vector of values for the neuron, the rows are simply placed sequentially. That is, the matrix is flattened in a row-wise manner, resulting in the following vector:

```
[1,0,0,0,  0,0,0,1,  0,1,0,0,  0,0,1,0]
```

To create a Boltzmann machine that can provide a solution to the TSP, the program must align the weights and biases in such a way that allows the states of the Boltzmann machine neurons to stabilize at a point that minimizes the total distance between cities. Keep in mind that the above grid can also find itself in many invalid states. Therefore, a valid grid must have the following:

- A single 1 value per row.
- A single 1 value per column.

As a result, the program needs to construct the weights so that the Boltzmann machine will not reach equilibrium in an invalid state. Listing 3.5 shows the pseudocode that will generate this weight matrix:

Listing 3.5: Boltzmann Weights for TSP

```
gamma = 7
# Source
for source_tour in range(NUM_CITIES):
  for source_city in range(NUM_CITIES):
    source_index = source_tour * NUM_CITIES + source_city
# Target
    for targetTour in range(NUM_CITIES):
      for (int target_city in range(NUM_CITIES):
        target_index = target_tour * NUM_CITIES + target_city
# Calculate the weight
        weight = 0
# Diagonal weight is 0
        if source_index != target_index:
# Determine the next and previous element in the tour.
# Wrap between 0 and last element.
          prev_target_tour = wrapped next target tour
          next_target_tour = wrapped previous target tour
# If same tour element or city, then -gama
          if (source_tour == target_tour)
            or (source_city == target_city):
            weight = -gamma
# If next or previous city, -gamma
          elif ((source_tour == prev_target_tour)
            or (source_tour == next_target_tour))
            weight = -distance(source_city,target_city)
# Otherwise 0
        set_weight(source_index, target_index, weight)
# All biases are -gamma/2
    set_bias(source_index, -gamma / 2)
```

Figure 3.9 displays part of the created weight matrix for four cities:

Figure 3.9: Boltzmann Machine Weights for TSP (4 cities)

	Tour:0					Tour:1					Tour:2				
	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4
0(0,0)	\	-g	-g	-g	-g	-g	-d(0,1)	-d(0,2)	-d(0,3)	-d(0,4)	-g	0	0	0	0
1(0,1)	-g	\	-g	-g	-g	-d(1,0)	-g	-d(1,2)	-d(1,3)	-d(1,4)	0	-g	0	0	0
2(0,2)	-g	-g	\	-g	-g	-d(2,0)	-d(2,1)	-g	-d(2,3)	-d(2,4)	0	0	-g	0	0
3(0,3)	-g	-g	-g	\	-g	-d(3,0)	-d(3,1)	-d(3,2)	-g	-d(3,4)	0	0	0	-g	0
4(0,4)	-g	-g	-g	-g	\	-d(4,0)	-d(4,1)	-d(4,2)	-d(4,3)	-g	0	0	0	0	-g
5(1,0)	-g	-d(0,1)	-d(0,2)	-d(0,3)	-d(0,4)	\	-g	-g	-g	-g	-g	-d(0,1)	-d(0,2)	-d(0,3)	-d(0,4)
6(1,1)	-d(1,0)	-g	-d(1,2)	-d(1,3)	-d(1,4)	-g	\	-g	-g	-g	-d(1,0)	-g	-d(1,2)	-d(1,3)	-d(1,4)

Depending on your viewing device, you might have difficulty reading the above grid. Therefore, you can generate it for any number of cities with the Javascript utility at the following URL:

http://www.heatonresearch.com/aifh/vol3/boltzmann_tsp_grid.html

Essentially, the weights have the following specifications:

- Matrix diagonal is assigned to 0. Shown as "\" in Figure 3.9.
- Same source and target position, set to -γ (gamma). Shown as -g in Figure 3.9.
- Same source and target city, set to -γ (gamma). Shown as -g in Figure 3.9.
- Source and target next/previous cities, set to -*distance*. Shown as $d(x,y)$ in Figure 3.9.
- Otherwise, set to 0.

The matrix is symmetrical between the rows and columns.

3.4.3 Boltzmann Machine Training

The previous section showed the use of hard-coded weights to construct a Boltzmann machine that was capable of finding solutions to the TSP. The program constructed these weights through its knowledge of the problem. Manually setting the weights is a necessary and difficult step for applying Boltzmann machines to optimization problems. However, this book will not include information about constructing weight matrices for general optimization problems because Nelder-Mead and simulated annealing are more often used for general-purpose algorithms.

3.5 Chapter Summary

In this chapter, we explained several classic neural network types. Since Pitts (1943) introduced the neural network, many different neural network types have been invented. We have focused primarily on the classic neural network types that still have relevance and that establish the foundation for other architectures that we will cover in later chapters of the book.

The self-organizing map (SOM) is an unsupervised neural network type that can cluster data. The SOM has an input neuron count equal to the number of attributes for the data to be clustered. An output neuron count specifies the number of groups into which the data should be clustered.

The Hopfield neural network is a simple neural network type that can recognize patterns and optimize problems. You must create a special energy function for each type of optimization problem that requires the Hopfield neural network. Because of this quality, programmers choose algorithms like Nelder-Mead or simulated annealing instead of the optimized version of the Hopfield neural network.

The Boltzmann machine is a neural network architecture that shares many characteristics with the Hopfield neural network. However, unlike the Hopfield network, you can stack the deep belief neural network (DBNN). This stacking ability allows the Boltzmann machine to play a central role in the implementation of the deep belief neural network (DBNN), the basis of deep learning.

In the next chapter, we will examine the feedforward neural network, which remains one of the most popular neural network types. This chapter will focus on classic feedforward neural networks that use sigmoid and hyperbolic tangent activation functions. New training algorithms, layer types, activation functions and other innovations allow the classic feedforward neural network to be used with deep learning.