Chapter 5

ARTIFICIAL NEURAL NETWORK

Inspired by the sophisticated functionality of human brains where hundreds of billions of interconnected neurons process information in parallel, researchers have successfully tried demonstrating certain levels of intelligence on silicon. Examples include language translation and pattern recognition software. While simulation of human consciousness and emotion is still in the realm of science fiction, we, in this chapter, consider artificial neural networks as universal function approximators. Especially, we introduce neural networks which are suited for time series forecasts.

5.1 Introduction

An artificial neural network (or simply neural network) consists of an input layer of neurons (or nodes, units), one or two (or even three) hidden layers of neurons, and a final layer of output neurons. Figure 5.1 shows a typical architecture, where lines connecting neurons are also shown. Each connection is associated with a numeric number called *weight*. The output, h_i, of neuron i in the hidden layer is,

$$h_i = \sigma\Big(\sum_{j=1}^{N} V_{ij}x_j + T_i^{hid}\Big), \tag{5.1}$$

where $\sigma()$ is called activation (or transfer) function, N the number of input neurons, V_{ij} the weights, x_j inputs to the input neurons, and T_i^{hid} the threshold terms of the hidden neurons. The purpose of the activation function is, besides introducing nonlinearity into the neural network, to bound the value of the neuron so that the neural network is not paralyzed by divergent neurons. A common example of the activation function is the sigmoid (or logistic) function

S.-C. Wang, *Interdisciplinary Computing in Java Programming*

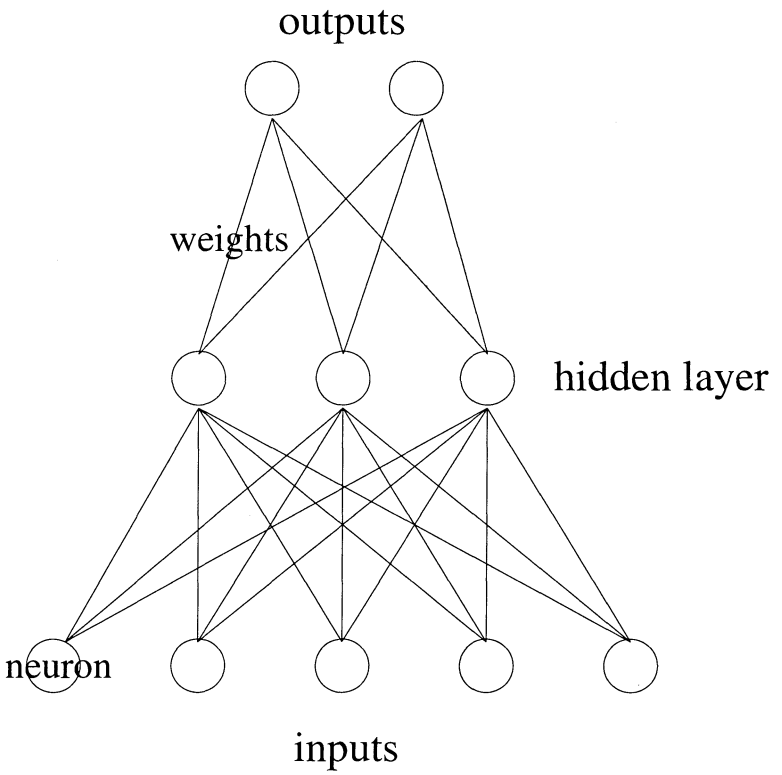

Figure 5.1. Architecture of a neural network

defined as (Figure 5.2),

$$\sigma(u) = \frac{1}{1 + \exp(-u)}. \tag{5.2}$$

Other possible activation functions are arc tangent and hyperbolic tangent. They have similar response to the inputs as the sigmoid function, but differ in the output ranges.

It has been shown that a neural network constructed the way above can approximate any computable function to an arbitrary precision. Numbers given to the input neurons are independent variables and those returned from the output neurons are dependent variables to the function being approximated by the neural network. Inputs to and outputs from a neural network can be binary (such as yes or no) or even symbols (green, red, ...) when data are appropriately encoded. This feature confers a wide range of applicability to neural networks.

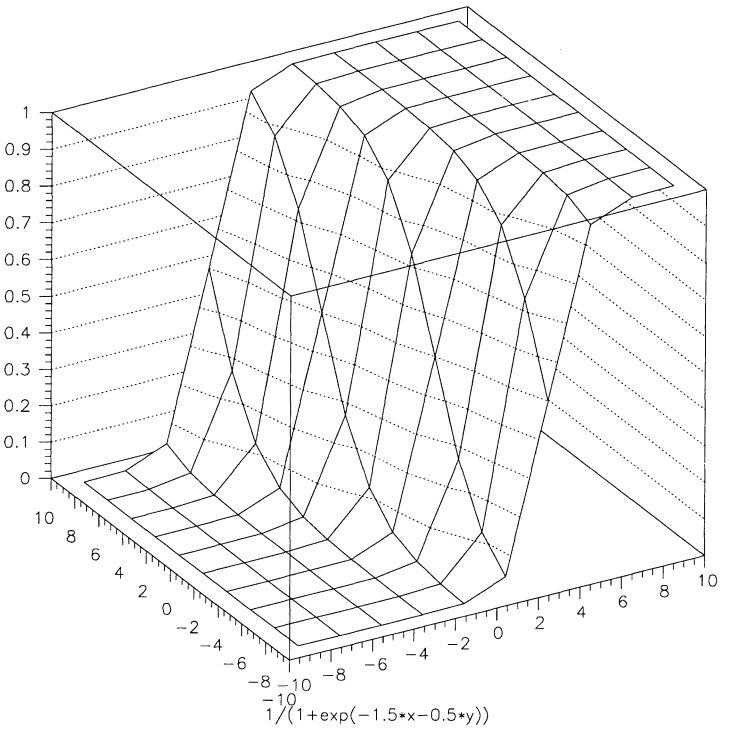

Figure 5.2. An example of sigmoid function with two neurons

After the architecture is described, we introduce the other essential ingredient of a neural network application, namely, *training*. Similar to human learning by examples, a neural network is trained by presenting to it a set of input data called *training set*. The desired outputs of the training data are known so that the aim of the training is to minimize, by adjusting the weights between the connected neurons, an *error function* which is normally the sum of the squared differences between the neural network outputs and the desired outputs.

If we are experimenting architectures of neural networks, an independent data set called *validation set* can be applied to the trained neural networks. The one which performs best is then picked up as the one of choice. After validation, yet another independent data set called *test set* is used to determine the performance level of the neural network which tells how confident we are when using the neural network. It should be understood that a neural network can never learn what is not present in the training set. The size of the training

set has therefore to be large enough for the neural network to memorize the features/trends embedded in the training set. On the other hand, if too much unimportant details are contained in the training set, the neural network might waste its resources (weights) fitting the noise. A judicious selection and/or representation of the data are therefore critical to successful implementations of neural networks.

Note that the definitions of validation and test set are reversed among authors of different fields. We have here followed the definition of B.D. Ripley (1996).

This section serves as a general introduction to neural networks. The following sections describe a step-by-step procedure to design a neural network for time series prediction.

5.2 Structural vs. Temporal Pattern Recognition

Conventional neural networks such as the one in Figure 5.1 have proven to be a promising alternative to traditional techniques for structural pattern recognition. In such applications, attributes of the sample data are presented to the neural network at the same time during training. After successful training, the neural network is supposed to be able to categorize the features buried in the training data set.

By contrast, in temporal pattern recognition, features evolve over time. A neural network for temporal pattern recognition is required to both remember and recognize patterns. This additional requirement poses a challenge to not only the design of the neural network architecture but the training procedure and data representation since all these tasks are inter-related; they can in fact be viewed as different aspects of the same underlying problem.

In the next section, we introduce a neural network architecture which has built-in memories and is therefore most suited for tasks of time series prediction.

5.3 Recurrent Neural Network

The network architecture of Figure 5.1 is called feedforward neural network because the direction of information flow is from the input, through the hidden layers, to the output. It has been indicated that for time series prediction recurrent neural networks are more competent. A recurrent neural network is a type of network that has feedbacks similar to the feedback circuitry in electronics. Figure 5.3 shows a simple yet powerful recurrent neural network introduced by J.L. Elman. Because of the delayed feedback, the network now has 'memory'.

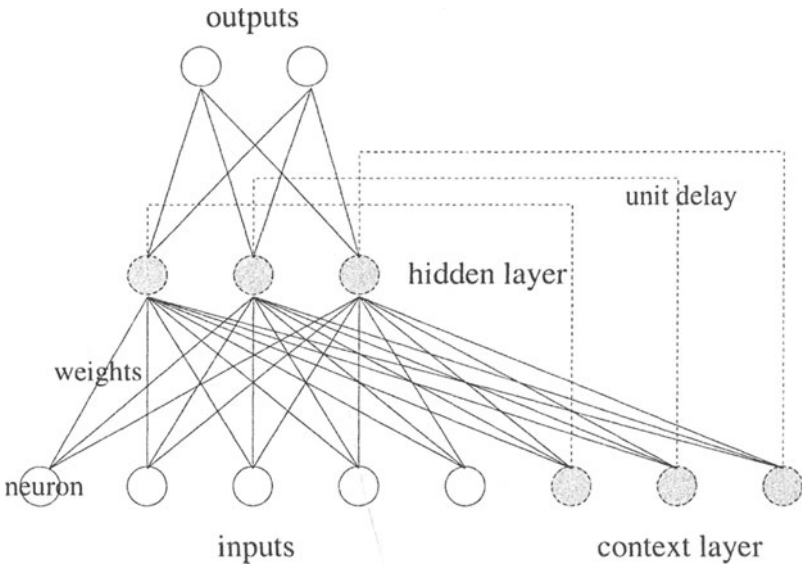

Figure 5.3. Architecture of a recurrent neural network

The output of the neuron i in the hidden layer becomes, instead of Eq. (5.1),

$$h_i(t) = \sigma\Big(\sum_{j=1}^{N} V_{ij}x_j(t) + \sum_{k=1}^{H} W_{ik}h_k(t-1) + T_i^{hid}\Big). \tag{5.3}$$

Time is explicitly indexed above to highlight the delayed feedback. When W_{ik} are all zero, the network reduces to the feedforward neural network of Figure 5.1. The output neurons, $o_i(t)$, at time t are updated as,

$$o_i(t) = \sigma\Big(\sum_{j=1}^{H} Q_{ij}h_j(t) + T_i^{out}\Big), \tag{5.4}$$

where Q_{ij} are the weights between the hidden neurons and the output neurons, and T_i^{out} are the thresholds of the output neurons. For one-unit-time-ahead predictions, we define, in the sense of least squares, the following error function,

$$\chi^2 = \sum_{t=1}^{\tau}\sum_{i=1}^{N}\big(o_i(t) - x_i(t+1)\big)^2, \tag{5.5}$$

where time is discrete and τ is the horizon of the data. For simplicity, we have assumed that the dimensionality of the output vector is the same as that of the input. A properly defined measure of error is one of the few key factors for successful neural network applications. An example will be shown below.

5.4 Steps in Designing a Forecasting Neural Network

We take as an illustration market prediction which is familiar to most readers irrespective of the professional background. The first step in neural network design is the selection of variables. To predict the price of some publicly traded commodity, say Japanese yen, at a future time $t+1$, you would first of all try writing down some equation like,

$$\begin{aligned} \text{yen}(t+1) = \text{yen}(t) + \\ f\big(&\text{dollar index}(t),\ \text{Euro index}(t),\ \text{Sterling index}(t), \\ &10-\text{year bond index}(t),\ \text{Nikkei}-225(t),\ \text{DJIA}(t), \\ &\text{oil price}(t),\ \cdots\big). \end{aligned} \tag{5.6}$$

Variables of the function $f()$ are the indicators of the market forces at present time t that you believe will affect the price of yen in the currency exchange market. The function $f()$ in Eq. (5.6) is presumably highly nonlinear. And particularly there exit no theories to model the behavior of yen over time. It is this difficulty that leads us to the method of neural networks. The choice of the frequency of the data, namely hourly, daily, or monthly exchange rates, depends on the objective of the researcher. The horizon of the historical data for training is another issue since trends change over time. You have also to make sure that values of the variables have been evaluated in a consistent way over time.

The second step is data representation. Raw data are rarely fed into neural networks without preprocessing/transformation. One of the the common transformations is to take natural logarithm of the change in the variable value. The transformed values form a distribution whose mean and standard deviation can be readily obtained. A scaling operation is then performed so that the range of the data is bounded between 0 and 1 or -1 and 1. A typical scaling is to assign to 1 (0) all values beyond (below) 3 standard deviations away from the mean. Values within 3 standard deviations are then linearly scaled between 0 and 1. Again which of the two ranges to use depends on the activation function. It was however pointed out that sigmoid activation functions were better for neural networks learning average behavior, while hyperbolic tangent worked better if learning involves deviations from the average.

Step 3 is training, validation, and testing. We move forward in time and divide the whole historical data set into training, validation, and test set at a proportion of, say, 7:2:1. The advantage of having the validation set follow the training set is that these data contain the most up-to-date market trends. On the other hand, care has to be taken to make sure that the trained neural network does not favor a subgenre of market trends.

5.5 How Many Hidden Neurons/Layers ?

We have known that recurrent neural networks of Figure 5.3 are suitable for time series prediction. However, to be specific, the next step is to determine the number of hidden layers and the number of neurons in a hidden layer.

There are as many answers to the question of the optimal number of hidden neurons/layers as there are many in-house proprietary neural network softwares in the world.

It was found that neural networks with a single hidden layer can do most of the job. It is therefore suggested that you start with a single hidden layer. Neural networks hardly have more than two hidden layers. We hereafter refer to neural networks of only one hidden layer.

There is no theory on the number of hidden neurons. Researchers have thus relied on experimentation and offered a handful of rules of thumb, which can again still be contradicting to one another. Nevertheless, we summarize some of the rules for you to kick off the game. For a neural network with N input neurons and M output neurons, T. Masters suggested a number of $\sqrt{NM}$ hidden neurons. The actual optimal number can still vary between one half and two times the geometrical mean value. D. Baily and D.M. Thompson (J.O. Katz) suggested the number of hidden neurons be 75% (50-300%) of the number of input neurons. C.C. Klimasauskas explicitly linked the number of training data with the number of neurons, suggesting that the number of training facts be at least 5 times that of the weights. The rules can turn out to limit the number of input neurons, which was discussed in step one. We see the interdependence in neural network designs.

The next step concerns output neurons and the error function.

5.6 Error Function

A common error function to be minimized is the squared errors defined in Eq. (5.5). Suppose, for example, the neural network is trained to recommend the user whether to buy or sell yen in the currency exchange market. The output of the neural network can have only one neuron and is therefore a scaler. If the output of the neuron gives a value greater than 0.9, it signals a sell; if it is less than -0.9 then it is a buy. Values between -0.9 and 0.9 render no decision. For the neural network to yield profits, it must be able to predict turning points in the evolving curve of the yen index. Before training, we examine the historical curve of the yen index and assign a 'buy' when the index is at the trough and a 'sell' when it is at the peak. At times between troughs and peaks, the desired (target) neural network outputs are assigned 'no decision'.

During training, weights of the neural network are adjusted in the hope that its outputs (buy, sell, or no decision) over time match the desired ones. The

error function is then defined to be the squared differences between the neural network outputs and the desired outputs.

However, we might remove all the 'no decision' target patterns in the error function so that the neural network does not waste weights remembering unimportant fluctuations. All three patterns are however needed in inputs as they make up the continuous history. If the error function is so defined, the neural network is forced to output either buy or sell. The strategy of the user is then changed to buy (sell) yens when the neural network outputs, say, 5 consecutive buys (sells). In this way, the neural network might have detected a falling in the yen price and is predicting a turning over at the fifth buy signal. Patterns less than 5 consecutive buys might simply identify minor troughs which need no attention since they make no profits considering the transaction fees. The actual strategy has to be experimented and depends on the user's portfolio.

The last step in the design is to tune the numerical values of the weights. The error function is a function of the neuron connecting weights whose number is often huge. Furthermore, the function can have lots of local minima in its weight space. A powerful function minimization method capable of finding the global minimum is simulated annealing introduced in Chapter 4 (cf. Section 4.7). Once the neural network is deployed, frequent retraining is beneficial and sometimes mandatory because the important temporal patterns might have changed since the last training.

5.7 Kohonen Self-Organizing Map

Problems are most often solved with greater ease in one way than the other. A properly presented/addressed problem simplifies both the input and output layer of the neural network and hence the architecture. In the rest of the chapter, we introduce and implement a variant of neural networks which is good at clustering multi-dimensional data. Categorized data are often served as inputs to neural networks for pattern recognition.

A Kohonen self-organizing map is a neural network with an input layer and a normally 2-dimensional output layer as shown in Figure 5.4. The unique property of a Kohonen neural network is that it is designed to preserve, during mapping, the topology of the input vectors, which are usually of very high dimensions (namely, have many components), so that, after successful learning, clusters emerge in the (2-dimensional) output layer. Each cluster corresponds to the vectors which are close to one another in the input vector space. We can say that the number of clusters that are formed represents, if the number of neurons in the output layer is large enough, the number of categories there are among the input vectors. We then label the clusters. Later on, a new input vector, when presented to the learned neural network, falls on one of the clus-

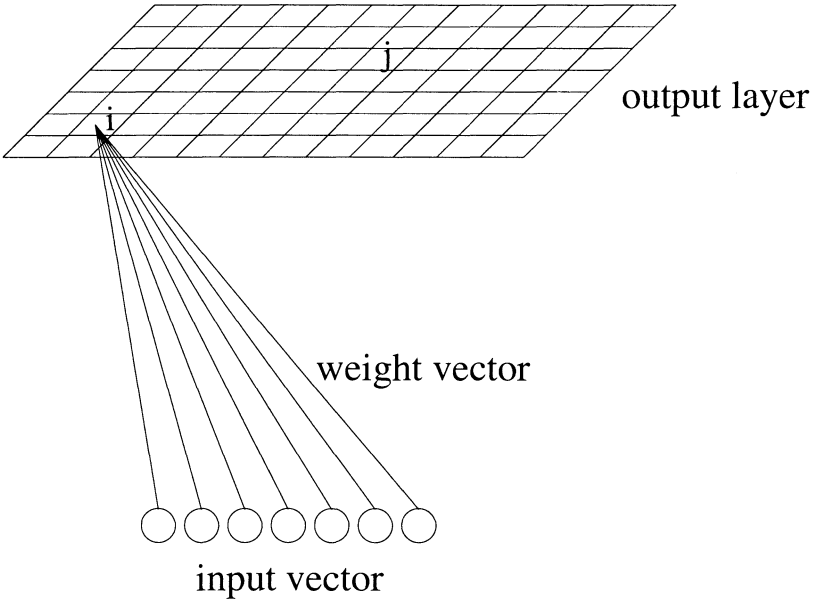

Figure 5.4. Architecture of a Kohonen neural network

ters (labels). A Kohonen neural network therefore works as a classifier. The neural network can also be thought of as performing feature extraction and visualization.

Application of Kohonen's feature map can be found in a broad spectrum of fields. For example, the method was used to classify folk song collections based on the distributions of melodic pitches, intervals, and durations.

In the next section, we describe the learning of the Kohonen neural network which altogether explains the term self-organizing.

5.8 Unsupervised Learning

Learning (or training depending on whose perspective) is an integral part of a neural network, much like a kid learns toward maturing. The learning method described early in this chapter is supervised learning in the sense that desired outputs are given and the neural network's predictions are forced to come close to the desired ones. This goal is accomplished when the values of the weights are so tuned that the error function is minimized.

In contrast, in Kohonen neural networks, given an input vector, the Euclidean distances between the input vector and any of the weight vectors, which

has the same dimensions as the input vector, are calculated. The one having the shortest distance is declared the winner and the weight vectors which are in the winner's neighborhood are updated according to the following rule,

$$w_j(k+1) = w_j(k) + \beta(k)C_{ij}(k)(\mathbf{x} - w_j(k)), \tag{5.7}$$

where w_j is the weight vector into neuron j on the output layer, k is the iteration (or time) index. $\beta(k)$ is called learning rate and usually defined as,

$$\beta(k) = \beta_{\text{initial}}\left(\frac{\beta_{\text{final}}}{\beta_{\text{initial}}}\right)^{k/k_{\max}}, \tag{5.8}$$

where $k_{\max}$ is the maximum number of iterations. It is seen that if $\beta_{\text{initial}} = 1.0$ and $\beta_{\text{final}} = 0.05$, the value of β starts from 1.0 and drops as k increases until it becomes 0.05 at $k = k_{\max}$. This is to say the neural network learns less and less harder as time goes on, just like a human does when she grows. $C_{ij}(k)$ defines the neighborhood and in many cases has the following form,

$$C_{ij} = \exp\left[-\frac{1}{2}\left(\frac{|i-j|}{\sigma(k)}\right)^2\right], \tag{5.9}$$

where $|i - j|$ is the Euclidean distance between neuron i and neuron j in the output layer and $\sigma(k)$ is a linear decreasing function of k. C_{ij} tells the weight vectors closer to the winning weight to move more toward the winner, while those farther away from the winner move less. The effect of the weight updating by Eqs. (5.7), (5.8), and (5.9) is that, after all the input vectors are fed to the neural network, clusters form by themselves in the end of the iteration at $k = k_{\max}$. Kohonen neural networks are thus self-organizing and require no supervision.

5.9 A Clustering Example

Coding of Eqs. (5.7), (5.8), and (5.9) is really simple. To help easily visualize the emergence of clusters on a 2-dimensional plane, we use various instances of the `Color` class in Java programming language to serve as our input data vectors. The `Color` class is based on the 3-component RGB model where `new Color(1.0f,0.0f,0.0f)`, `new Color(0.0f,1.0f,0.0f)`, and `new Color(0.0f,0.0f,1.0f)` generate respectively the 3 primary colors red, green, and blue. Any other colors are obtained by component values between `0.0f` and `1.0f`. For example, `(1.0f,1.0f,0.0f)` gives yellow, `(0.0f,0.0f, 0.0f)` black, `(1.0f,1.0f,1.0f)` white, and so on. Listing 5.1 gives the class responsible for initializing the input data, which are stored in an array of `Neuron` objects defined in Listing 5.2.

```
/*      Sun-Chong Wang
```

```
        TRIUMF
        4004 Wesbrook Mall
        Vancouver, V6T 2A3
        Canada
        e-mail: wangsc@triumf.ca

        DataBase.java initializes the input vectors
        for the Kohonen's self-organizing map */

import java.util.*;

class DataBase extends Observable {
 Neural parent;
 Neuron[] data;  // array of Neuron objects
 final int num_samples = 100;

 public DataBase(Neural parent) {
        super();
        this.parent = parent;
        data = new Neuron[num_samples];
        initializeData();
 }      // end of constructor

 public void DrawMap(Neuron[][] weight, double[][] similarity) {
        parent.plotting.weight = weight;
        parent.plotting.similarity = similarity;
        setChanged();
        notifyObservers(parent.plotting);
 }      // end of method

 private void initializeData() {
        Random rand = new Random();
        rand.setSeed(1L);

        // 100 different colors
        for (int i=0; i<num_samples; i++) {
            data[i] = new Neuron();
            data[i].x = rand.nextDouble();
            data[i].y = rand.nextDouble();
            data[i].z = rand.nextDouble();
        }
        // six (or eight) different colors
        for (int i=0; i<num_samples/2; i++) {
            data[i] = new Neuron();
            data[i].x = 1.0;            // red
            data[i].y = 0.0;
            data[i].z = 0.0;
        }
        for (int i=50; i<60; i++) {
            data[i] = new Neuron();
            data[i].x = 0.0;            // green
            data[i].y = 1.0;
            data[i].z = 0.0;
        }
        for (int i=60; i<70; i++) {
            data[i] = new Neuron();
            data[i].x = 0.0;            // blue
            data[i].y = 0.0;
            data[i].z = 1.0;
        }
        for (int i=70; i<80; i++) {
            data[i] = new Neuron();
            data[i].x = 1.0;            // yellow
            data[i].y = 1.0;
            data[i].z = 0.0;
        }
        for (int i=80; i<85; i++) {
```

```
            data[i] = new Neuron();
            data[i].x = 0.0;            // cyan
            data[i].y = 1.0;
            data[i].z = 1.0;
        }
        for (int i=85; i<90; i++) {
            data[i] = new Neuron();
            data[i].x = 1.0;            // magenta
            data[i].y = 0.0;
            data[i].z = 1.0;
        }
        for (int i=90; i<95; i++) {
            data[i] = new Neuron();
            data[i].x = 0.0;            // black
            data[i].y = 0.0;
            data[i].z = 0.0;
        }
        for (int i=95; i<100; i++) {
            data[i] = new Neuron();
            data[i].x = 1.0;            // white
            data[i].y = 1.0;
            data[i].z = 1.0;
        }
    }       // end of initializeData
}       // end of class DataBase
```

Listing 5.1 DataBase.java

```
/*      Sun-Chong Wang
        TRIUMF
        4004 Wesbrook Mall
        Vancouver, V6T 2A3
        Canada
        e-mail: wangsc@triumf.ca

        Neuron.java defines the class to hold the input
        vector as well as the weights */

public class Neuron {
 public double x,y,z;

 public Neuron() {
        x = 0.0;
        y = 0.0;
        z = 0.0;
 }      // end of constructor
}       // end of class Neuron
```

Listing 5.2 Neuron.java

In the first illustration of clustering, we prepare for six input colors. The weights are assigned random colors. In the beginning of the learning, you thus see colorful dots randomly distributed across the canvas. As time goes on, similar colors aggregate and finally in the end (after 10,000 iterations) six blocks of distinct colors are really formed as shown in Figure 5.5. The network is performing clustering !

Note that when you run the learning again (with different initial random weights by different seeds to the random number generator), six clusters still form but their locations on the map may change. This is because of the random initial weights. The implication is that the similarity between adjacent clusters is not necessarily higher than that of disjoint clusters. Segregation patterns depend on initialized values of the weights. We therefore write, in `SOM.java`, the

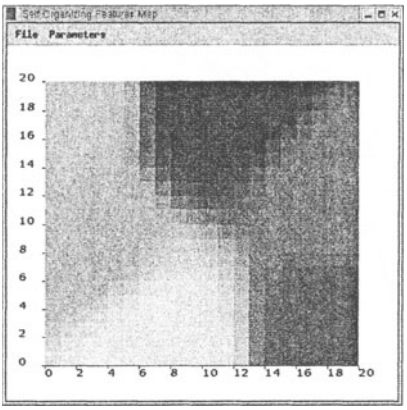

Figure 5.5. Six clusters resulting from six input colors

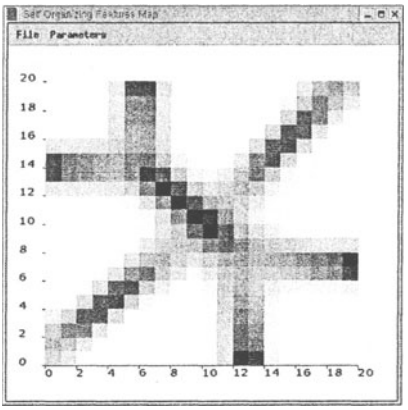

Figure 5.6. Similarity plot of the clusters in Figure 5.5

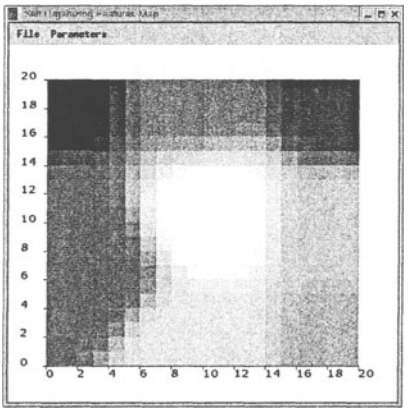

Figure 5.7. Clustering with eight input colors

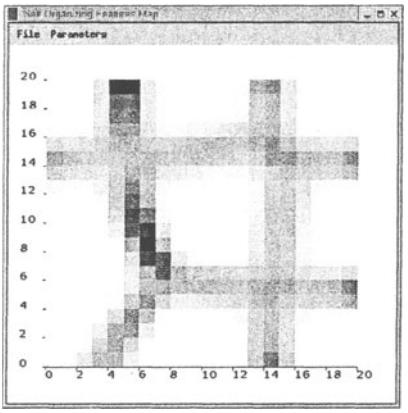

Figure 5.8. Similarity plot of Figure 5.7

`Similarity()` method which calculates the average distance between neighboring weights. Darker colors are assigned to larger average distances in the plotting class `Plotter.java`. The similarity plot associated with Figure 5.5 is shown in Figure 5.6. It is seen there that clusters are isolated by dark ridges.

In the second illustration, we input eight colors to the same program. Results of the feature map and similarity map are shown in Figures 5.7 and 5.8.

The source code for the learning is given in Listing 5.3. Now let's increase the number of different input colors to 100. The resulting maps are shown in

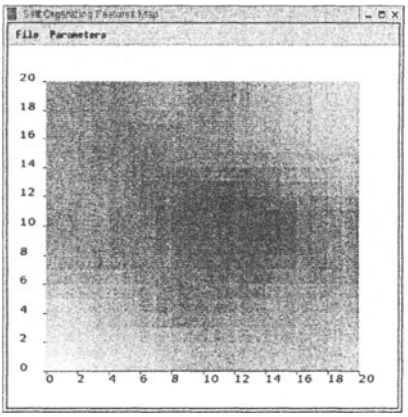

Figure 5.9. Clustering 100 different input colors

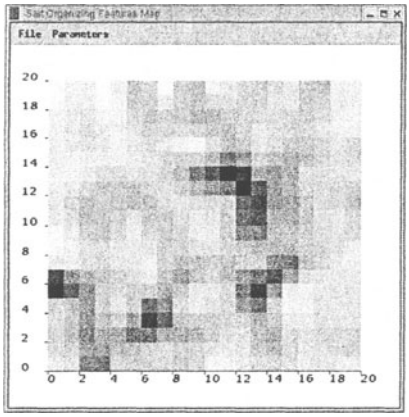

Figure 5.10. Similarity plot of Figure 5.9

Figures 5.9 and 5.10. It is noticed that a grid of 20 by 20 output neurons might be insufficient as evidenced by the blurring boundaries in the similarity map of Figure 5.10. We then increase the number of output layer neurons to 40 by 40 (Figures 5.11 and 5.12) through the user interface dialog box of Figure 5.13.

```
/*      Sun-Chong Wang
        TRIUMF
        4004 Wesbrook Mall
        Vancouver, V6T 2A3
        Canada
        e-mail: wangsc@triumf.ca

        SOM.java codes the unsupervised learning algorithm of
        the self-organizing map: Eqs. (5.7), (5.8), (5.9) */

import java.lang.*;
import java.util.*;
import java.util.Random;

public class SOM implements Runnable { // a thread
 Random rand;

 int XSize, YSize;
 int iTime, ifreq;
 int num_samples;
 // initial/final beta/sigma
 double beta_i, beta_f, sigma_i, sigma_f;
 double[][] distance,similarity;
 Neuron[][] weight;
 Neuron[] data;

 SOMDialog parent;

 public SOM(SOMDialog parent) {
        this.parent = parent;

        rand     = new Random();
        System.out.println("random number = " + rand.nextFloat());

        XSize = 0;
```

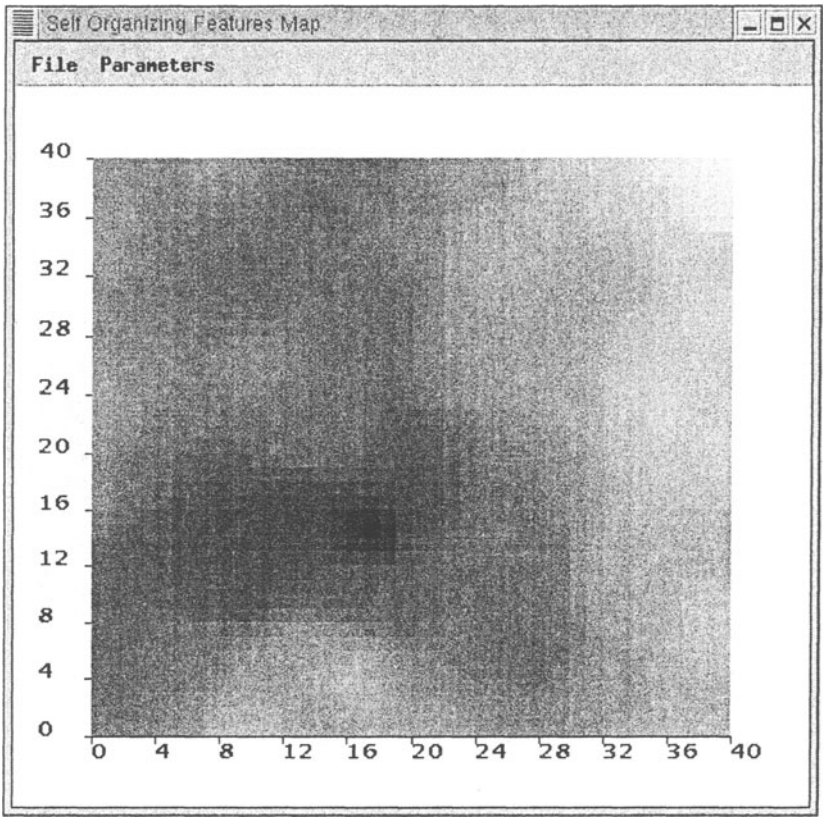

Figure 5.11. 100 input colors and 4 times of output neurons

```
        YSize = 0;
        iTime = 10000;
        ifreq = 100;
        beta_i = 1.0;
        beta_f = 0.01;
        sigma_i = (XSize+YSize)/2.0/2.0;
        sigma_f = 1.0;
}       // end of SOM class constructor

public void run() { // executed by a thread
        for (int i=0; i<iTime; i++) {
            Learning(i);
            if (i%ifreq == 0) {
               Similarity();
               parent.parent.db.DrawMap(weight,similarity);
            }  // db is an instance of the DataBase class
            parent.jbar.setValue(Math.round(((float)i)/
                                 ((float)iTime)*100.0f));
        }
}

public void Setup(int XSize, int YSize) {
```

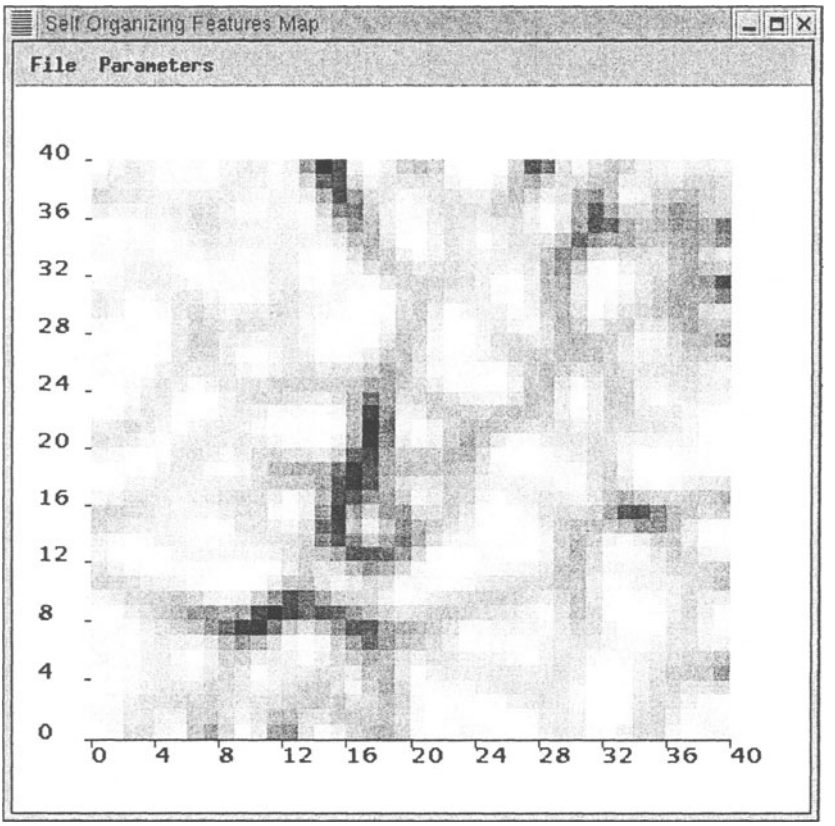

Figure 5.12. Similarity plot of Figure 5.11

```
if (XSize != this.XSize || YSize != this.YSize) {
   this.XSize = XSize;
   this.YSize = YSize;
   distance = new double[XSize][YSize];
   weight = new Neuron[XSize][YSize];  // array of objects
   for (int i=0; i<XSize; i++) {
       for (int j=0; j<YSize; j++) {
           weight[i][j] = new Neuron();// call the constructor
       }
   }
   similarity = new double[XSize][YSize];
   // average radius
   sigma_i = (XSize+YSize)/2.0/2.0;
}  // end if

rand.setSeed(2L);
for (int i=0; i<XSize; i++) {
    for (int j=0; j<YSize; j++) {
        weight[i][j].x = rand.nextDouble();
        weight[i][j].y = rand.nextDouble();
        weight[i][j].z = rand.nextDouble();
```

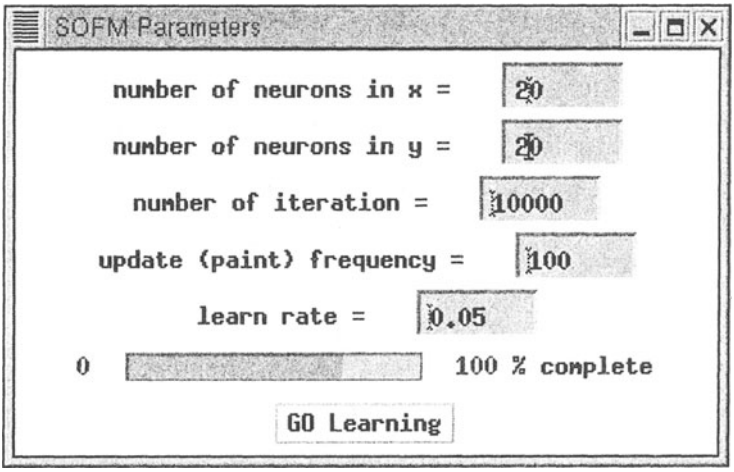

Figure 5.13. User interface for the SOM application

```
            }
        }
}       // end of Setup

private void Learning(int itime) {
        int i_win=0,j_win=0;
        double exponent,beta,sigma,adjust,distance_ij,dmin;

        // randomly picks up an input vector
        int k = rand.nextInt(num_samples);

        dmin = Double.MAX_VALUE;
        // calculates the distance between the input vector and
        // the weight vector.  i,j are the coordinates of the weight
        // on the 2-dimensional output layer
        for (int i=0; i<XSize; i++) {
            for (int j=0; j<YSize; j++) {
                distance[i][j] = (data[k].x - weight[i][j].x)*
                                 (data[k].x - weight[i][j].x)+
                                 (data[k].y - weight[i][j].y)*
                                 (data[k].y - weight[i][j].y)+
                                 (data[k].z - weight[i][j].z)*
                                 (data[k].z - weight[i][j].z);
                if (distance[i][j] < dmin) {
                   dmin = distance[i][j];
                   // keeps the coordinates of the winner
                   i_win = i;
                   j_win = j;
                }
            }
        }

        exponent = itime/((double)(iTime-1.0));
        beta = beta_i*Math.pow(beta_f/beta_i,exponent); // Eq. (5.8)
        sigma = sigma_i*Math.pow(sigma_f/sigma_i,exponent);

        for (int i=0; i<XSize; i++) {
            for (int j=0; j<YSize; j++) {
```

```
                // distance to the winning weight
                distance_ij = (i-i_win)*(i-i_win)+(j-j_win)*(j-j_win);
                // Eq. (5.9)
                adjust = beta*Math.exp(-distance_ij/sigma/sigma/2.0);
                // the update rule: Eq. (5.7)
                weight[i][j].x += adjust*(data[k].x-weight[i][j].x);
                weight[i][j].y += adjust*(data[k].y-weight[i][j].y);
                weight[i][j].z += adjust*(data[k].z-weight[i][j].z);
            }
        }
}       // end of Learning

// average distance between the weight vectors
// to serve as a measure of similarity
private void Similarity() {
        double max;

        for (int i=1; i<XSize-1; i++) {
            for (int j=1; j<YSize-1; j++) {
                similarity[i][j] = (w_distance(i,j,i-1,j)+
                                    w_distance(i,j,i+1,j)+
                                    w_distance(i,j,i,j-1)+
                                    w_distance(i,j,i,j+1))/4.0;
            }
        }

        // boundary cells
        for (int j=1; j<YSize-1; j++) {
            similarity[0][j] = (w_distance(0,j,0,j+1)+
                                w_distance(0,j,0,j-1)+
                                w_distance(0,j,1,j))/3.0;
            similarity[XSize-1][j] =
                       (w_distance(XSize-1,j,XSize-1,j+1)+
                       w_distance(XSize-1,j,XSize-1,j-1)+
                       w_distance(XSize-1,j,XSize-2,j))/3.0;
        }
        for (int i=1; i<XSize-1; i++) {
            similarity[i][0] = (w_distance(i,0,i+1,0)+
                                w_distance(i,0,i-1,0)+
                                w_distance(i,0,i,1))/3.0;
            similarity[i][YSize-1] =
                       (w_distance(i,YSize-1,i+1,YSize-1)+
                       w_distance(i,YSize-1,i-1,YSize-1)+
                       w_distance(i,YSize-1,i,YSize-2))/3.0;
        }

        // corner cells
        similarity[0][0] =
           (w_distance(0,0,0,1)+w_distance(0,0,1,0))/2.0;
        similarity[0][YSize-1] =(w_distance(0,YSize-1,0,YSize-2)+
                                 w_distance(0,YSize-1,1,YSize-1))/2.0;
        similarity[XSize-1][YSize-1] = (w_distance(XSize-1,YSize-1,
                                        XSize-2,YSize-1)+
                                        w_distance(XSize-1,YSize-1,
                                        XSize-1,YSize-2))/2.0;
        similarity[XSize-1][0] = (w_distance(XSize-1,0,XSize-2,0)+
                                 w_distance(XSize-1,0,XSize-1,1))/2.0;

        max = 0.0;
        for (int i=0; i<XSize; i++) {
            for (int j=0; j<YSize; j++) {
                if (similarity[i][j] > max) max = similarity[i][j];
            }
        }
```

```
        // normalizing (for black and white plotting)
        for (int i=0; i<XSize; i++) {
            for (int j=0; j<YSize; j++) {
                similarity[i][j] /= max;
            }
        }
    }       // end of Similarity

    private double w_distance(int i, int j, int m, int n) {
        double tmp;
        tmp = (weight[i][j].x-weight[m][n].x)*
              (weight[i][j].x-weight[m][n].x)+
              (weight[i][j].y-weight[m][n].y)*
              (weight[i][j].y-weight[m][n].y)+
              (weight[i][j].z-weight[m][n].z)*
              (weight[i][j].z-weight[m][n].z);
        return tmp;
    }       // end of w_distance
}       // end of SOM class
```

Listing 5.3 SOM.java

5.10 Summary

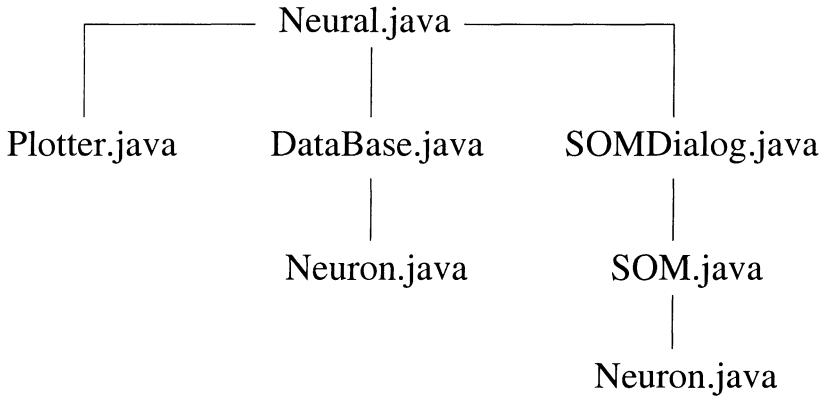

Figure 5.14. Source files for the Kohonen self-organizing map in this chapter

`Neural.java`, containing the `main()` method, is easily written using `Spin.java` of Chapter 4 as a template. `Plotter.java`, extending `Canvas` and implementing `Observer`, is similar to the one in the appendix. `SOMDialog.java`, a dialog box for user interaction, can be obtained by modifying the dialog box class, `SADialog.java`, in Chapter 4.

We implemented a Kohonen self-organizing map. Vectors of colors were input to the Kohonen neural network, and grouped into clusters on its own on the 2-dimensional output layer in the end of the learning.

We laid out steps when designing a neural network. A recurrent neural network architecture which has 'memory' neurons was introduced. Economic time series prediction was used as an example throughout the designing steps.

A neural network, after training, is able of generalizing the patterns embedded in the training data set. However, it is not expected to detect patterns that do not exist in the training data. A neural network can become more powerful when its predicting/recognizing capability is combined with adaptivity, which is the subject of the next chapter.

5.11 References and Further Reading

An introduction to self-organizing map is, T. Kohonen, "Self-organizing Maps", 2nd ed. Springer-Verlag, Berlin (1997)

Two textbooks on neural networks are, C.M. Bishop, "Neural Networks for Pattern Recognition", Oxford University Press, Oxford (1995), and B.D.Ripley, "Pattern Recognition and Neural Networks", Cambridge University Press, Cambridge (1996)

Numerous resources on neural networks can be found in the on-line FAQ located at, `ftp://ftp.sas.com/pub/neural/FAQ.html`