Neural networks are biologically motivated computing structures that are conceptually modeled after the brain. The neural network is made up of a highly connected network of individual computing elements (mimicking neurons) that collectively can be used to solve interesting and difficult problems. Once trained, neural networks can generalize to solve different problems that have similar characteristics. This chapter will introduce the basics of neural networks and introduce a number of supervised learning algorithms. In Chapter 11, we’ll continue our exploration of neural networks and review some variants that can be used to solve different types of problems and in particular, those of unsupervised learning algorithms.

SHORT HISTORY OF NEURAL NETWORKS

The story of neural networks is interesting because, like AI itself, it's one of grand visions, eventual disappointment, and finally, silent adoption. In 1943, McCulloch and Pitts developed a neural network model based on their understanding of neurology, but the models were typically limited to formal logic simulations (simulating binary operations). In the early 1950s,
researchers in neural networks worked toward neural network models with the support of neuroscientists.

But it wasn’t until the late 1950s that promising models began to emerge. The Perceptron model, developed by Rosenblatt, was built for the purpose of understanding human memory and learning. The basic perceptron consisted of an input layer (for the stimulus) and an output layer (result) that was fully interconnected. Each connection was assigned a weight that was applied to the input stimulus (to which it was connected on the input layer). By adjusting the weights of the connections, a desired output could be formed for a given input. This allowed the perceptron to learn to recognize input patterns.

In the 1960s, another learning model emerged from Widrow and Hoff of Stanford University called ADALINE, or Adaptive Linear Element. This particular algorithm used least-mean-squares for adjusting the weights of the network, but this particular model could be implemented in the physical world using analog electronic components.

In 1969, the growing popularity of neural networks was brought to a halt. Marvin Minsky and Seymour Papert wrote a book entitled “Perceptrons” in which limitations of single-layer perceptrons were discussed, but then generalized to more powerful multi-layer models. The result was severe reductions in neural network research funding, and a corresponding reduction in the effort applied to the field. Luckily, several researchers continued to investigate neural network models, and successfully defined new models and methods for learning. In 1974, Paul Werbos developed the backpropagation algorithm, which permitted successful learning in multi-layer neural networks.

Since the 1970s, research and successful results in neural network design have attracted scientists back to the field. Many theoretical papers and practical treatments of neural networks have emerged, and neural networks can now be found outside of the lab and in real applications such as pattern recognition and classification. To support large neural networks, integrated circuits have been developed to speed the operation and training in production applications.

**BIOLOGICAL MOTIVATION**

In 1943, McCulloch and Pitts used their understanding of neurology to build a new information processing structure. The processing element of a neural network is modeled after a neuron, which is viewed as the fundamental processing element in our own brains.
The neuron is a simple processing device that has inputs (known as dendrites) and outputs (known as axons). The axon splits at its end into thousands of branches, each potentially influencing other neurons at a synapse (a small gap that separates axons and dendrites). When a neuron receives excitatory inputs that exceed its inhibitory inputs, a signal is transmitted down its axon to other neurons. This process continues in other neurons, creating a massively parallel network of neurons in an excited or inhibited state. Learning can then be defined as the altering of the synaptic junctions that change the manner in which one neuron is influenced by others.

While neural networks are modeled after our understanding of the way in which our brain works, surprisingly little is known about how our brains actually function. Through various types of inspection, we can see our brain in operation, but because of the massive number of neurons and interconnections between these neurons, how it works remains a mystery (though many theories exist).

**FUNDAMENTALS OF NEURAL NETWORKS**

Let’s begin with an exploration of neural networks applications, fundamental concepts behind neural networks, and then begin an investigation into a number of network models and learning algorithms.

You can find neural networks in a large variety of applications, from classification tasks (such as credit-risk assessment), data-processing tasks (adaptive signal processing), and approximation of arbitrary functions (time-series modeling and prediction). In this chapter, we’ll explore neural networks for classification (character recognition and data classification).
A neural network is made up of one or more neurons, which is the basic processing element. A neuron has one or more inputs (dendrites), each of which are individually weighted. A neuron has one or more outputs (axons) that are weighted when connecting to other neurons. The neuron itself includes a function that incorporates its inputs (via summation) and then normalizes its output via a transfer function (see Figure 8.2).

For each input of Figure 8.2, a weight is applied. These adjusted inputs are then summed and a transfer function is applied to determine the output. Eq 8.1 provides the equation for this simple neuron.

\[ O_0 = f\left(\sum_{j=0}^{n} (i_j w_j)\right) \]  
(Eq 8.1)

Single Layer Perceptrons (SLPs)

Single Layer Perceptrons (or SLPs) can be used to emulate logic functions such as NOT, NOR, OR, AND, and NAND, but cannot be used to emulate the XOR function (two layers of neurons are required for this function). We’ll explore this problem shortly.

Minsky and Papert documented the XOR limitation of single layer perceptrons, which ultimately resulted in vast reduction in neural network function during the 1970s.

A bias is also commonly applied to each neuron, which is added to the weighted sum of the inputs prior to passing through the transfer function. A weight is also commonly applied to the bias. The bias determines the level
of incoming activations (value of weighted inputs) that are required in order for the neuron to fire. The bias is commonly set to one, but a weight is also present for the bias which can be tuned by the learning algorithm.

An SLP should not be confused with a single neuron. Consider the network in Figure 8.3. This is also an SLP, because it consists of a single layer. For problems of higher dimensionality, we must use the MLPs, or Multi-Layer Perceptrons.

Representing SLPs is quite simple. As the inputs and weights have a one-to-one correspondence, it’s easy to compute the output. Consider the simple code in Listing 8.1.

Listing 8.1: Sample code illustrating SLP representation.

```c
#define NUM_INPUTS 3
/* Note: +1 here to account for the bias input */
double weights[ NUM_INPUTS+1 ];
double inputs[ NUM_INPUTS+1 ];
int step_function( double input )
{
    if (input > 0.0) return 1;
    else return -1;
}
int calc_output( void )
{
    int i;
    double sum = 0.0;
    /* Set the bias (could be done once at init) */
```
Artificial Intelligence

```c
inputs[NUM_INPUTS] = 1.0;
/* Compute the output (Equation 8.1) */
for (i = 0 ; i < NUM_INPUTS+1 ; i++) {
    sum += (weights[i] * inputs[i]);
}
/* Pass the output through the step (activation) function */
return step_function( sum );
```

**Multi-Layer Perceptrons (MLPs)**

As Minsky and Papert revealed in their book “Perceptrons,” single layer perceptrons have the disadvantage that they can only be used to classify linearly separable data. But what was found a short time later is that by stacking the single layer perceptrons into multiple layer perceptrons (see Figure 8.4), the ability to solve any classification problem theoretically could be realized. The MLP can model practically any function of arbitrary complexity, where the number of inputs and number of hidden layers determine the function complexity.

The neurons in an MLP have the same basic attributes of the SLP (bias, etc.). But with multiple layers, the output from one layer becomes the input to the next. The implementation for the MLP is a bit more complicated, but remains straightforward (see Listing 8.2). Note here the use of an activation function for both the hidden and output nodes. The sigmoid function can be used to squash the output of the neuron to 0.0 to 1.0.

![Example of a Multiple Layer Perceptron (MLP).](image)
Listing 8.2: Sample code illustrating MLP representation.

```c
#define NUM_INPUTS 4
#define NUM_HIDDEN_NEURONS 4
#define NUM_OUTPUT_NEURONS 3
typedef mlp_s {
    /* Inputs to the MLP (+1 for bias) */
    double inputs[NUM_INPUTS+1];
    /* Weights from Hidden to Input Layer (+1 for bias) */
    double w_h_i[NUM_HIDDEN_NEURONS+1][NUM_INPUTS+1];
    /* Hidden layer */
    double hidden[NUM_HIDDEN+1];
    /* Weights from Output to Hidden Layer (+1 for bias) */
    double w_o_h[NUM_OUTPUT_NEURONS][NUM_HIDDEN_NEURONS+1];
    /* Outputs of the MLP */
    double outputs[NUM_OUTPUT_NEURONS];
} mlp_t;

void feed_forward( mlp_t *mlp )
{
    int i, h, out;
    /* Feed the inputs to the hidden layer through the hidden
     * to input weights.
     */
    for ( h = 0 ; h < NUM_HIDDEN_NEURONS ; h++ ) {
        mlp->hidden[h] = 0.0;
        for ( i = 0 ; i < NUM_INPUT_NEURONS+1 ; i++ ) {
            mlp->hidden[h] += ( mlp->inputs[i] * mlp->w_h_i[h][i] );
        }
        mlp->hidden[h] = sigmoid( mlp->hidden[h] );
    }
    /* Feed the hidden layer activations to the output layer
    * through the output to hidden weights.
    */
    for( out = 0 ; out < NUM_OUTPUT_NEURONS ; out++ ) {
        mlp->output[out] = 0.0;
        for ( h = 0 ; h < NUM_HIDDEN_NEURONS ; h++ ) {
            mlp->outputs[out] += ( mlp->hidden[h] * mlp->w_o_h[out][h] );
        }
        mlp->outputs[out] = sigmoid( mlp->outputs[out] );
    }
}
```
Listing 8.2 implements the MLP neural network shown in Figure 8.5. This MLP has four input cells, four hidden cells, and three output cells. Bias cells are implemented as input and hidden cells, but these have a constant value of 1.0 (though the weights can be adjusted to modify their effect).

Note the flow in Listing 8.2. First, we calculate the output of the hidden cells (using the input cells and the weights between the hidden and input cells), and then calculate the next layer up, which in this case is the output cells. This process is commonly referred to as feeding the data forward, or more simply, *feedfoward*.

---

**NOTE**

We can think of neural networks as parallel computing systems. Each neuron is a processing element that takes one or more inputs and generates an output. The inputs and outputs can be thought of as messages. In MLP architectures, the outputs can be sent to multiple other process elements for further processing. The parallel nature of neural networks comes into play with multiple neurons in a layer. Each of these neurons can process their inputs at the same time, making neural networks with large numbers of neurons in a layer faster in multi-processor architectures.

---

![Graphic Network Implemented in Listing 8.2](image)
Supervised vs Unsupervised Learning Algorithms

There are two basic categories of learning algorithms for neural networks: supervised learning and unsupervised learning.

In the supervised learning paradigm, the neural network is trained with data that has known right and wrong answers. By calculating the output of the neural network and comparing this to the expected output for the given test data, we can identify the error and adjust the weights accordingly. Examples of supervised learning algorithms include the Perceptron learning algorithm, Least-Mean-Squares learning, and Backpropagation (each of which will be explored in this chapter).

Unsupervised learning algorithms are those in which there’s no answer given in the test data. What these algorithms do instead is analyze the data in order to understand their similarities and differences. In this way, relationships in the data can be found that may not have been apparent before. Examples of unsupervised learning algorithms include the k-Means clustering algorithm, Adaptive Resonance Theory (ART), and Kohonen Self-Organizing Maps (each of which, and more, will be discussed in Chapter 9).

Binary vs Continuous Inputs and Outputs

Neural networks can operate with a combination of input types. For example, we can use binary inputs (-1, 1) and binary outputs. We’ll explore this in our first two examples of SLPs. For other uses, such as audio or video applications, we’ll need continuous inputs (such as real-valued data). It’s also possible to use combinations, such as continuous inputs and binary outputs (for classification problems).

Now that we have a basic understanding of neural network topologies and learning algorithms, let’s start with an investigation of the perceptron and understand how it can be trained for simple pattern classification.

THE PERCEPTRON

A perceptron is a single neuron neural network that was first introduced by Frank Rosenblatt in the late 1950s. The perceptron is a simple model for neural networks that can be used for a certain class of simple problems called linear separable problems (otherwise called linear discriminants). These are often used to classify whether a pattern belongs to one of two classes (see Figure 8.6).
So given a set of inputs describing an object (sometimes called a feature vector), a perceptron has the ability to classify the data into two classes if the data is linearly separable. Given the set of possible inputs, the task then is to identify the weights that correctly classify the data (linearly separate) into two classes.

**TIP**

Another name for the perceptron is the Threshold Logic Unit, or TLU. The TLU is a linear discriminator that given a threshold (whether the feature sum is greater than the threshold, or less than the threshold).

The perceptron can accurately classify the standard boolean functions, such as AND, OR, NAND, and NOR. As shown in Figure 8.7, the AND and OR functions can be linearly separated by a line (in the case of two inputs, or a hyperplane for three inputs), but the XOR function is linearly inseparable. A bias component provides the offset of the line from the origin. If no bias existed, the line (or hyperplane) would be restricted to pass through the origin and the weights would only control the angle of the discriminant.
One result of Rosenblatt’s work on perceptrons was the Mark I perceptron at Cornell Aeronautical Laboratory. The Mark I was an analog computer based on the perceptron that had a 20 by 20 retina and learned to recognize letters.

**Perceptron Learning Algorithm**

Perceptron learning is a supervised learning algorithm and is a simple procedure where the weights are adjusted to classify the training set. Each sample from the training set is applied to the perceptron, and the error (expected result minus the actual result) is used to adjust the weights. A learning rate is also applied (small number between 0 and 1) to minimize the changes that are applied at each step.

We’ll use the perceptron shown in Figure 8.8 to illustrate the perceptron learning algorithm. The weights of the perceptron will initially be set to zero. There are two inputs with two corresponding weights and also a bias with a weight. The bias will be set to one, but the weight for the bias will be adjusted to alter its affect. Calculating the output of the perceptron can then be defined (Eq 8.2).

\[ R = \text{step}(i_0 w_0 + i_1 w_1 + w_b) \]  

(Eq 8.2)

The step function simply pushes the result to 1.0 if it exceeds a threshold; otherwise, the result is -1.0.

Given a training set for our perceptron, we apply each of the elements of the training set to the perceptron and for each sample, adjust the weights based on the error. The error is defined as the expected result minus the actual result. Each is adjusted using Eq 8.3 (called the Perceptron Rule).

\[ w_i = w_i + a T_i \]  

(Eq 8.3)
In this equation, \( \alpha \) is the learning rate (small number less than one), \( T \) is the target (or expected) result, and \( i \) is the input value for the current weight \( w_i \). Eq 8.3 very simply adjusts the weight by incrementing or decrementing the weight based on the direction of the error and the corresponding input (which is identified by multiplying the expected result by the training input for this connection).

The perceptron learning algorithm is an example of a supervised learning algorithm. We present a training set to our perceptron and penalize it when it arrives at the wrong answer (an incorrect classification).

The application of the learning algorithm continues until no changes are made to the weights because all tests are properly classified.

**Perceptron Implementation**

The implementation of the perceptron learning algorithm is very simple (see Listing 8.3). In this implementation, the perceptron is trained with a training set (a boolean function), and after calculating the error (of desired vs actual result), the weights are adjusted per Eq 8.3. Calculating the output of the perceptron is shown in the compute function. This function implements the Eq 8.2. The training process continues for a maximum number of iterations, and when complete, the truth table is emitted for the boolean function.

**Listing 8.3: Perceptron learning implementation.**

```c
#define MAX_TESTS  4
training_data_t training_set[MAX_TESTS] = {
    {-1.0, -1.0, -1.0},
    {-1.0,  1.0,  1.0},
    { 1.0, -1.0,  1.0},
    { 1.0,  1.0,  1.0}  
};
double compute( int test )
{
    double result;
    /* Equation 8.2 */
    result = ((training_set[test].a * weights[0]) +
              (training_set[test].b * weights[1]) +
              (1.0 * weights[2]) );
    /* Clip the result */
    if (result > 0.0) result = 1.0;
    else result = -1.0;
```
return result;
}
int main()
{
    int i, test;
    double output;
    int change;
    /* Initialize the weights for the perceptron */
    for ( i = 0; i < NUM_WEIGHTS; i++ ) weights[i] = 0.0;
    /* Train the perceptron with the training set */
    change = 1;
    while (change) {
        change = 0;
        for ( test = 0; test < MAX_TESTS; test++ ) {
            /* Test on the perceptron */
            output = compute(test);
            /* Perceptron Supervised Learning Algorithm */
            if ( (int)training_set[test].expected != (int)output ) {
                /* Use Equation 8.3 */
                weights[0] += ALPHA * training_set[test].expected *
                              training_set[test].a;
                weights[1] += ALPHA * training_set[test].expected *
                              training_set[test].b;
                weights[2] += ALPHA * training_set[test].expected;
                change = 1;
            }
        }
    }
    /* Check the status of the Perceptron */
    for (i = 0; i < MAX_TESTS; i++) {
        printf("%g OR %g = %g\n", training_set[i].a, training_set[i].b, compute(i));
    }
    return 0;
}

The perceptron learning implementation can be found on the CD-ROM at ./software/ch8/perceptron.c.
LEAST-MEAN-SQUARE (LMS) LEARNING

The LMS algorithm goes by a number of names, including the Widrow-Hoff rule and also the Delta rule (LMS was originally introduced by Widrow and Hoff in 1959). LMS is a fast algorithm that minimizes the Mean-Square Error (MSE). Recall from perceptron learning that the algorithm operates until it correctly classifies the entire training set. At this point, a solution has been found. But just how good is the solution? Is the data linearly separated by an optimal discriminator (as shown in Figure 8.6)?

Another approach is to train the perceptron using another termination criterion. So instead of training the perceptron until a solution is found, another criterion is to continue training while the MSE is greater than a certain value. This is the basis for the LMS algorithm.

LMS learning is based on gradient descent, where the local minimum of the error is achieved by adjusting the weights proportional to the negative of the gradient. Additionally, the weights are adjusted with a learning rate ($\rho$) to allow it to settle into a solution and avoid oscillating around the MSE.

First, let’s explore the MSE. The MSE is simply the average of the weighted sum of the error for N training samples (see Eq 8.4).

$$MSE = \frac{1}{N} \sum_{j=1}^{N} (R - C_j)^2$$  \hspace{1cm} (Eq 8.4)

In Eq 8.4, R is the output of the perceptron given the current set of weights multiplied by the current test inputs ($C_j$).

LMS Learning Algorithm

To train the perceptron using LMS, we iterate through the test set, taking a set of inputs, computing the output, and then using the error to adjust the weights. This process is done either randomly for the test set, or for each test of the set in succession.

The learning rule (see Eq 8.5) adjusts the weight based on the error (R-C, or expected minus actual output). Once the error is calculated, the weights are adjusted by a small amount ($\rho$) in the direction of the input ($E$). This has the effect of adjusting the weights to reduce the output error.
One of the major differences between LMS and perceptron learning is that LMS can operate with real-values. Recall that perceptron learning operates solely with binary inputs.

**NOTE**

LMS is a standard tool for adaptive signal processing and can solve problems such as echo cancellation, noise cancellation, and equalization.

**LMS Implementation**

Like the perceptron algorithm, LMS is also very simple (see Listing 8.4). Initially, the weights vector is initialized with small random weights. The main loop then randomly selects a test, calculates the output of the neuron, and then calculates the error (expected result minus the actual result). Using the error, Eq 8.5 is applied to each weight in the vector (note that weight[2] is the bias, and its input is always 1.0). The loop then continues, where we check the MSE to see if it has reached an acceptable value, and if so, we exit and emit the computed truth table for the neuron.

Recall that single neuron models can only classify training data into two sets. In this case, the AND function is separable, so the neuron can be successfully trained. Nonseparable training sets will result in the algorithm never converging on a solution.

**Listing 8.4: LMS learning algorithm.**

```cpp
double weights[NUM_WEIGHTS];
#define MAX_TESTS 4
const training_data_t training_set[MAX_TESTS]={
  /* a   b   expected */
  {-1.0, -1.0, -1.0},
  {-1.0,  1.0, -1.0},
  { 1.0, -1.0, -1.0},
  { 1.0,  1.0,  1.0}
};
double compute_output( test )
{
  double result;
  result = ((training_set[test].a * weights[0]) +
             (training_set[test].b * weights[1]) +
             weights[2])
   * (1.0 - result);
  return result;
}
```

\[ w_{t+1} = w + \rho (R - C)E \]  \hspace{1cm} (Eq 8.5)
(1.0 * weights[2]));
return (result);
}
int classify(int test)
{
double result;
result = compute_output(test);
if (result > 0.0) return 1;
else return -1;
}
double MSE(void)
{
int test;
double sum = 0.0;
/* Equation 8.4 */
for (test = 0; test < MAX_TESTS; test++) {
    sum += sqr(training_set[test].expected - compute_output(test));
}
return (sum / (double)MAX_TESTS);
}
int main()
{
int i, test;
double result, error;
RANDINIT();
/* Pick random weights for the perceptron */
for (i = 0; i < NUM_WEIGHTS; i++) {
    weights[i] = RAND_WEIGHT;
}
/* Train the perceptron with the training set */
while (MSE() > 0.26) {
    test = RANDMAX(MAX_TESTS);
    /* Compute the output (weighted sum) */
    result = compute_output(test);
    /* Calculate the error */
    error = training_set[test].expected - result;
    /* Delta Rule Learning Algorithm (Equation 8.5) */
    weights[0] += (RHO * error * training_set[test].a);
    weights[1] += (RHO * error * training_set[test].b);
    weights[2] += (RHO * error);
LEARNING WITH BACKPROPAGATION

Let’s now investigate what can be considered the most popular of the MLP learning algorithms, backpropagation. The backpropagation algorithm can be succinctly defined as follows. For a test set, propagate one test through the MLP in order to calculate the output (or outputs). Compute the error,
which will be the difference of the expected value and the actual value. Finally, backpropagate this error through the network by adjusting all of the weights; starting from the weights to the output layer and ending at the weights to the input layer (see Figure 8.9).

Like LMS learning, backpropagation adjusts the weights in an amount proportional to the error for the given unit (hidden or output) multiplied by the weight and its input. The training process continues until some termination criterion, such as a predefined mean-squared error, or a maximum number of iterations.

Backpropagation is one of the most popular learning algorithms, and is used to train neural networks for a variety of applications. We’ll first look at the details of the algorithm, and then explore a neural network that can recognize numbers from a standard 5 by 7 character bitmap format.

**FIGURE 8.10:** A simplified flow of backpropagation.
Backpropagation Algorithm

The backpropagation algorithm is a typical supervised learning algorithm, where inputs are provided and propagated forward to generate one or more outputs. Given the output, the error is calculated using the expected output. The error is then used to adjust the weights (see Figure 8.10). Propagating the inputs forward was previously explored in Listing 8.2.

It's important to note that there are two types of error functions for backpropagation. The first error function (Eq 8.6) is used for output cells, and the second is used only for hidden cells (Eq 8.7).

\[ E_o = (y_i - u_i)g'(u_i) \quad \text{(Eq 8.6)} \]

\[ E_h = \left( \sum_{i=1}^{i=n} (w_{hi}E_o) \right) g(u_h) \quad \text{(Eq 8.7)} \]

Note that in both equations, \( u \) is the output of the given cell, otherwise known as its activation. \( Y \) is the expected or correct result. Finally, \( w \) represents all of the weights (from 1 to \( n \)) connecting the hidden cell to all inputs cells (in a fully connected network).

The activation, or transfer, function \( g \) to be used will be the standard sigmoid squashing function (see Figure 8.11). While \( g \) represents the sigmoid, \( g' \) represents the first derivative of the sigmoid, as shown in Eq 8.8.

\[ g'(u) = u(1-u) \quad \text{(Eq 8.8)} \]

**FIGURE 8.11:** The sigmoid squashing function.
At this point, given our test input and expected result, we have the error calculated for each output and hidden node. The next step is to use this error to adjust the corresponding weights for the node. We’ll use Eq 8.9 for this purpose, which utilizes the error previously calculated for the node (whether hidden or output).

\[ w_{ij} = w_{ij} + \rho E u_i \]  

(Eq 8.9)

For the given error \( E \) and activation (or cell output, \( u_i \)), we multiply by a learning rate \( \rho \) and add this to the current weight. The result is a minimization of the error at this cell, while moving the output cell activation closer to the expected output.

**Backpropagation Implementation**

Neural networks are a great tool for classifying a set of inputs to a set of outputs. Let’s look at a very visual example of neural networks from the domain of pattern recognition. Consider the bitmap character images in Figure 8.12. We’ll train a neural network to take the cells of this image as the input (35 independent cells) and activate one of ten output cells representing the recognized pattern. While any of the output cells could be activated, we’ll take the largest activation as the cell to use in a style called winner-takes-all.

Since we could very simply implement a comparison classifier to recognize the pattern (by looking for the specific pattern at the input), we’ll introduce noise to the pattern when we test the neural network. This will make the classification problem more difficult, and test the generalization features of the neural network.

**TIP**

Generalization is one of the greatest characteristics of neural networks. This means that after training a neural network with a set of training data, it can generalize its training to correctly classify data that it has not seen before. Generalization can be trained out of a neural network by training

![Sample bitmaps for training the number recognition neural network.](image)

*FIGURE 8.12: Sample bitmaps for training the number recognition neural network.*
the network for too long with a data set. When this happens, the network overfits the data and is not able to generalize for new unseen data.

The neural network that we’ll use is called a *winner-takes-all* network in which we have a number of output nodes, and we’ll select the one that has the largest activation. The largest activation indicates the number that was recognized. Figure 8.13 shows the neural network that will be used for the pattern recognition problem. The input layer consists of 35 input cells (for each pixel in the image input), with 10 cells in the hidden layer. The output layer consists of 10 cells, one for each potential classification. The network is fully interconnected, with 350 connections between the input and hidden layer, and another 350 connections between the hidden layer and output layer (for a total of 700 weights).

For our implementation, let’s first discuss the neural network representation (see Listing 8.5). We’ll maintain three vectors containing the input values, current activations of the hidden layer and current activations of the output layer. Note that we’ll also maintain an extra cell at the input and hidden layers which will represent the bias (set to a constant 1.0). The weights will be represented by two, two-dimensional arrays representing the hidden layer weights and the output layer weights.
The full source implementation for backpropagation learning can be found on the CD-ROM at ./software/ch8/backprop.c. The following discussion provides only the relevant functions to illustrate backpropagation learning.

Listing 8.5: Neural network representation (inputs, activations, and weights).

```c
#define INPUT_NEURONS 35
#define HIDDEN_NEURONS 10
#define OUTPUT_NEURONS 10
double inputs[INPUT_NEURONS+1];
double hidden[HIDDEN_NEURONS+1];
double outputs[OUTPUT_NEURONS];
double w_h_i[HIDDEN_NEURONS][INPUT_NEURONS+1];
double w_o_h[OUTPUT_NEURONS][HIDDEN_NEURONS+1];
```

Computing the activations of the output cells is very straightforward (see Listing 8.6). Note the use of the sigmoid function to squash the activations into the range 0 to 1.

Listing 8.6: Calculating the output activations with the `feed_forward` function.

```c
void feed_forward( void )
{
    int i, j;
    /* Calculate outputs of the hidden layer */
    for (i = 0 ; i < HIDDEN_NEURONS ; i++) {
        hidden[i] = 0.0;
        for (j = 0 ; j < INPUT_NEURONS+1 ; j++) {
            hidden[i] += (w_h_i[i][j] * inputs[j]);
        }
        hidden[i] = sigmoid( hidden[i] );
    }
    /* Calculate outputs for the output layer */
    for (i = 0 ; i < OUTPUT_NEURONS ; i++) {
        outputs[i] = 0.0;
        for (j = 0 ; j < HIDDEN_NEURONS+1 ; j++) {
            outputs[i] += (w_o_h[i][j] * hidden[j]);
        }
        outputs[i] = sigmoid( outputs[i] );
    }
}
```
The backpropagation algorithm (shown in Listing 8.7) is just slightly more complicated than feeding forward. Using Eq 8.6 and Eq 8.7, we calculate the error for the output and hidden nodes. Finally, the weights are updated given the hidden and output errors, input value, and a small learning rate.

**Listing 8.7: Updating the weights given the backpropagation algorithm.**

```c
void backpropagate_error( int test )
{
    int out, hid, inp;
    double err_out[OUTPUT_NEURONS];
    double err_hid[HIDDEN_NEURONS];
    /* Compute the error for the output nodes (Equation 8.6) */
    for (out = 0 ; out < OUTPUT_NEURONS ; out++) {
        err_out[out] = ((double)tests[test].output[out] - outputs[out]) * sigmoid_d(outputs[out]);
    }
    /* Compute the error for the hidden nodes (Equation 8.7) */
    for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {
        err_hid[hid] = 0.0;
        /* Include error contribution for all output nodes */
        for (out = 0 ; out < OUTPUT_NEURONS ; out++) {
            err_hid[hid] += err_out[out] * w_o_h[out][hid];
        }
        err_hid[hid] *= sigmoid_d(hidden[hid]);
    }
    /* Adjust the weights from the hidden to output layer (Equation 8.9) */
    for (out = 0 ; out < OUTPUT_NEURONS ; out++) {
        for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {
            w_o_h[out][hid] += RHO * err_out[out] * hidden[hid];
        }
    }
    /* Adjust the weights from the input to hidden layer (Equation 8.9) */
    for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {
        for (inp = 0 ; inp < INPUT_NEURONS+1 ; inp++) {
            w_h_i[hid][inp] += RHO * err_hid[hid] * inputs[inp];
        }
    }
}```
The main function (shown in Listing 8.8) performs the neural network training as well as the test of the trained network. The first step is initializing the network by setting each weight to a small random value (via a call to init_network). We then enter the training loop where a test is selected at random, the inputs loaded from the test into the inputs vector (set_network_inputs), and the output activation calculated (backpropagate_error). Finally, the MSE is calculated and tested against our termination criteria.

Listing 8.8: The training and test loop (main function).

int main( void )
{
    double mse, noise_prob;
    int    test, i, j;
    RANDINIT();
    init_network();
    /* Training Loop */
    do {
        /* Pick a test at random */
        test = RANDMAX(MAX_TESTS);
        /* Grab input image (with no noise) */
        set_network_inputs( test, 0.0 );
        /* Feed this data set forward */
        feed_forward();
        /* Backpropagate the error */
        backpropagate_error( test );
        /* Calculate the current MSE */
        mse = calculate_mse( test );
    } while (mse > 0.001);
    /* Now, let’s test the network with increasing amounts of noise */
    test = RANDMAX(MAX_TESTS);
    /* Start with 5% noise probability, end with 25% (per pixel) */
    noise_prob = 0.05;
    for (i = 0 ; i < 5 ; i++) {
        set_network_inputs( test, noise_prob );
        feed_forward();
    }
    return;
}
for (j = 0 ; j < INPUT_NEURONS ; j++) {
    if ((j % 5) == 0) printf("\n");
    printf("%d ", (int)inputs[j]);
}
printf("\nclassified as %d\n\n", classifier());
noise_prob += 0.05;
return 0;
}

The final step in the main function (Listing 8.8) is the neural network test. This test verifies the generalization capabilities of the neural network by inducing noise into the input image. We start by selecting one of the tests (a number to recognize) and then add increasing amounts of noise to the image. Once the noise is added (as part of the call to set_network_inputs), the output activations are computed (feed_forward) and then the classification emitted (through a call to classifier). This classifier function inspects each of the output activations, and chooses the largest one in the winner-takes-all fashion.

Figure 8.14 graphically illustrates the generalization capabilities of the network trained using error backpropagation. In both cases, once the error rate reaches 20%, the image is no longer recognizable.

What’s shown in main is a common pattern for neural network training and use. Once a neural network has been trained, the weights can be saved off and used in the given application.

\textbf{TIP}\n
This approach for pattern recognition is considered brittle if you consider the rotation of the image by a small degree. A production character recognizer would instead use features of the image space, rather than requiring that pixels be roughly in the correct place.

\textbf{FIGURE 8.14:} The pattern recognition capabilities of the multi-Layer neural network.
Tuning Backpropagation

Backpropagation is a great technique to train a neural network for a classification task. But there are a number of things that can be done to tune the algorithm either for better generalization, or for faster training. We’ll explore some of these techniques here.

Training Variants

One of the problems that can be created in training is what’s called “over-learning,” or “over-fitting.” The desire for the neural network is to provide a correct mapping of the test data, but maintain the ability to generalize for yet to be seen data. This is difficult to quantify, but the results of over-learning can be very apparent in the test stage.

There are a number of things that can be done to avoid over-learning. One of the simplest is called early-stopping. As the name suggests, we train for a limited amount of time. Practically, we can train until the training set is correctly classified, and ignore the MSE. In this way, we’ve not optimized the neural network to the data set, and its generalization capabilities should remain intact.

Another method to avoid over-learning is to incorporate noise into the training. Rather than simply provide the test set verbatim to the neural network, small amounts of noise can be induced to maintain some amount of flexibility in the network’s generalization capability. The addition of noise keeps the network from being focused solely on the test data.

The network could also be trained with a subset of the available training data. In this way, the network is initially trained with the subset, and then tested with the remainder of the test data to ensure that it generalizes properly. The availability of lots of training data can also help in generalization and avoid overfitting.

Finally, one could maintain generalization capabilities by minimizing the changes made to the network as time progresses. Minimizing the changes can be achieved by reducing the learning rate. By reducing the changes over time, we reduce the possibility that the network will become focused on the training set.

Ultimately, there’s no silver bullet. What can be done is to experiment with network topologies, number of hidden layers, number of hidden nodes per layer, learning rate, etc., to find the best combination that works for the problem at hand.

Weight Adjustment Variants

With backpropagation, there are a number of other weight adjustment strategies that can be applied that can speed learning or avoid becoming
trapped in local minima. The first involves a momentum term where a portion of the last weight change is applied to the current weight adjustment round. Eq 8.1 shows an updated weight adjustment variant based on Eq 8.9. The difference is that a portion of the last weight change (identified as $\Delta w_{ij}$) is accumulated using a small momentum multiplier ($m$), as shown in Eq 8.10. The last weight change is stored using Eq 8.11.

$$w_{ij} = w_{ij} + \rho E_i + m \Delta w_{ij} \quad \text{(Eq 8.10)}$$

$$\Delta w_{ij} = \rho E_i \quad \text{(Eq 8.11)}$$

Another useful weight adjustment technique is called weight decay. Weight decay can improve generalization in neural networks by suppressing irrelevant components of the weight vector by choosing the smallest vector that solves the particular problem. [Krogh 1995] Weight decay works by slowly decaying the weights of a neural network during training. Eq 8.12 is one method by which this can be done.

$$w_{ij} = \lambda w_{ij} \quad \text{(Eq 8.12)}$$

In this equation, $\gamma$ is a large number (such as 0.95) so that 5% of the weight is decayed for each iteration. In this way, large weights are penalized by larger amounts than small weights. This is a similar method to weight elimination (or pruning), whose goal is to remove connections between layers that do not contribute to the network’s classification capabilities.

**PROBABILISTIC NEURAL NETWORKS (PNN)**

A useful neural network architecture with fundamental differences from backpropagation is called a Probabilistic Neural Network (or PNN). This architecture is similar to backpropagation in that it is feedforward in nature, but differs very much in the way that learning occurs. Both backpropagation and PNN are supervised learning algorithms, but PNN includes no weights in its hidden layer. Instead each hidden node represents an example vector, with the example acting as the weights to that hidden node. These are not adjusted at all. Figure 8.15 illustrates a sample PNN.

As shown in Figure 8.15, the PNN consists of an input layer, which represents the input pattern (or feature vector). The input layer is fully interconnected with the hidden layer, which consists of the example vectors.
Artificial Intelligence

The actual example vector serves as the weights as applied to the input layer. Finally, an output layer represents each of the possible classes for which the input data can be classified. Note here that the hidden layer is not fully interconnected to the output layer. The example nodes for a given class connect only to that class’s output node and none other. In the example PNN, our network has four classes for which there are two training examples each.

One other important element of the PNN is the output layer and the determination of the class for which the input layer fits. This is done through a winner-takes-all approach. The output class node with the largest activation represents the winning class. While the class nodes are connected only to the example hidden nodes for their class, the input feature vector connects to all examples, and therefore influences their activations. It’s therefore the sum of the example vector activations that determines the class of the input feature vector.

**PNN Algorithm**

Calculating the class-node activations is therefore a simple process. For each class node, the example vector activations are summed, which are the sum of the products of the example vector and the input vector. The hidden node activation, shown in Eq 8.13, is simply the product of the two vectors ($E$ is the example vector, and $F$ is the input feature vector).

$$h_i = E_i F \quad \text{(Eq. 8.13)}$$

The class output activations are then defined in Eq 8.14.
Neural Networks I

\[ c_j = \frac{\sum_{i=1}^{N} (h_i \cdot N)}{N} \]  \hspace{1cm} \text{(Eq. 8.14)}

where \( N \) is the total number of example vectors for this class, \( h_i \) is the hidden-node activation, and \( \nu \) is a smoothing factor. The smoothing factor is chosen through experimentation. If the smoothing factor is too large, details can be lost, but if the smoothing factor is too small, the classifier may not generalize well.

What's interesting about the PNN is that there's no real training that occurs since the example vectors serve as the weights to the hidden layer of the network. Given an unknown input vector, the hidden node activations are computed and then summed at the output (class nodes) layer. The class node with the largest activation determines the class to which the input feature vector belongs.

As no training is required, classifying an input vector is fast, depending on the number of classes and example vectors that are present. It's also very easy to add new examples to the network. We simply add the new hidden node, and its output is used by the particular class node. This can be done dynamically as new classified examples are found. The PNN also generalizes very well, even in the context of noisy data.

Let's now look at the implementation of a PNN with a graphical example that will illustrate its classification capabilities.

Probabilistic neural networks are a form of normalized Radial-Basis Form (RBF) neural networks, where each hidden node is a "kernel" implementing a probability density function.

PNN Implementation

Building a PNN is quite simple once the structures are in place to represent the training dataset (which fundamentally defines the weights between the hidden layer and the input layer). Let's begin with a short discussion of the dataset.

For this example, we'll implement a two-dimensional classifier so that we can visualize it graphically. The dataset represents points on a two-dimensional map that have been pre-classified into three classes. Each point is a feature vector that is represented by the example_t type.

```c
#define DIMENSIONALITY 2
typedef structure example_s {
```

A dataset is then defined as a collection of examples. We’ll have a dataset per class, which is easily defined as:

```c
#define EXAMPLES 10
typedef struct data_set_s {
    example_t example[EXAMPLES];
} data_set_t;
#define CLASSES 3
data_set_t dataset[CLASSES] = {
    /* Class 0 */
    { {{13,  1}},
        {{11,  2}},
        ... 
        {{13, 10}} },
    /* Class 1 */
    { {{36,  4}},
        {{34,  5}},
        ... 
        {{37, 11}} },
    /* Class 2 */
    { {{24, 27}},
        {{22, 29}},
        ... 
        {{22, 38}} }
};
```

We now have a collection of data points that are split into examples for our three classes. Let’s now look at how these are used by the PNN algorithm to classify an unseen data point.

Listing 8.9 provides the pnn_classifier function. The sole purpose of this function is to take a feature vector (example), and identify the class to which it belongs. The function begins with an outer loop that iterates through each of the classes in the dataset. For each class, each of the examples is iterated, calculating the sum of the feature vector and example vector products.

Finally, the output array (class vector) is passed to a function called winner_takes_all, which returns the class with the largest activation. This is the class which the PNN classified as the example feature vector.
Listing 8.9: Implementation of the simple PNN classifier function.

```c
int pnn_classifier( void )
{
    int c, e, d;
    double product;
    double output[CLASSES];
    /* Calculate the class sum of the example multiplied by each of
     * the feature vectors of the class.
     */
    for ( c = 0 ; c < CLASSES ; c++ ) {
        output[c] = 0.0;
        for ( e = 0 ; e < EXAMPLES ; e++ ) {
            product = 0.0;
            /* Equation 8.13 */
            for ( d = 0 ; d < DIMENSIONALITY ; d++ ) {
                product += (SCALE(example[d]) * SCALE(dataset[c].example[e].feature[d]));
            }
            /* Equation 8.14 -- part 1 */
            output[c] += exp( (product-1.0) / sqr(SIGMA) );
        }
        /* Equation 8.14 -- part 2 */
        output[c] = output[c] / (double)EXAMPLES;
    }
    return winner_takes_all( output );
}
```

The full source implementation for the probabilistic neural network classifier can be found on the CD-ROM at ./software/ch8/pnn.c. The previous source discussion provides only the PNN function to illustrate the classifier.

Let’s now use the PNN classifier to classify the entire two-dimensional space in which our example feature vectors exist. In this example, the two-dimensional space is iterated to identify the class to which each point belongs. This classification is emitted, with space in place for the example feature vectors (see Listing 8.10).
Listing 8.10: Output of the PNN classifier over a two-dimensional space of three classes.
The example shown in Listing 8.10 illustrates the clustering capabilities of the PNN as applied to this simple problem. PNNs have been applied to complex problems such as speaker-independent speech recognition and many other applications.

OTHER NEURAL NETWORK ARCHITECTURES

In addition to single and multi-layer perceptrons, there are variations of neural network architectures that can support different types of problems. Let’s look at two neural network architectures that support time-series (or signal) processing and feedback systems (or those with memory).

Time-Series Processing Architecture

Consider the time series shown in Figure 8.16. This signal could be a portion of speech, a fetal heartbeat, or the stock price of a company. The signal can be sampled at a given frequency and used as input to a neural network to predict a future value. The neural network could also be used to filter the input signal for noise cancellation.

In addition to sampling points from the time series, the neural network can operate over a window of the time series through the use of a sliding window. Using a sliding window allows the neural network to operate on multiple data points, making it a better predictor of future time-series behavior.

\[ \text{FIGURE 8.16: Example of a time series with a sliding window.} \]
Figure 8.17 illustrates a four-layer, multi-layer perceptron for time-series processing. As input samples are acquired, the contents of the input nodes are shifted right (losing the right-most value) and then inserting the new value in the left-most node. The network is then fed-forward, resulting in an output activation (in this model, predicting the next value of the time series).

This particular network could be trained with backpropagation on a sample time series using the known value and predicted value to determine the error that is to be backpropagated for weight adjustment.

Neural networks, which operate over a window of time-series data, are often called tapped-delay-line networks, as there is an implicit delay between input samples as they shift down the line of inputs.
Recurrent Neural Network

A recurrent neural network is one that includes feedback connections, where higher layer nodes in the network feed lower layer nodes. Consider the recurrent neural network as shown in Figure 8.18. In this network model, the output feeds back as input to the next iteration.

Recurrent neural networks are very biologically plausible, as our own brains are recurrent. Neural networks with feedback are able to implement almost any arbitrary sequential behavior and are therefore useful in control systems, robotics, and many other applications.

Recurrent neural networks are also an interesting take on long-and-short term memory. The feedback loop provides the means for short-term memory (recalling the last output, in the example, and using it as an input). A tapped-delay-line could be used to keep the previous state longer (as in the tapped-delay-line model). The tapped-delay-line could also be modified to keep the last, 10th previous, and 100th previous output to give the network a longer history.

TIPS FOR BUILDING NEURAL NETWORKS

Neural networks can benefit from careful design of inputs, outputs, and internal structure. Let’s explore a few techniques that can help in building efficient neural network architectures.

Defining the Inputs

How the inputs are defined to a neural network can help speed learning, or help to minimize the overall size of the neural network. Consider a set of inputs that describe the weather as one of four states {sunny, cloudy, raining, snowing}. A simple representation is to define an input node for each of the inputs, as shown in Figure 8.20.

![FIGURE 8.20: Using a single input per category.](image)
An alternative for a distinct set of inputs is to encode as a binary set of inputs (increasing the input information density, while reducing the number of input nodes that are necessary). Figure 8.21 illustrates this binary encoding.

Note that this works only for mutually exclusive inputs. Using real-valued inputs, we could set sunny and cloudy each to 0.5 (and raining and snowing to 0.0) to represent partly cloudy. Using real-valued inputs in this way provides a means of fuzziness to the input, and introduces the concept of probability or input confidence.

Defining the Outputs

The winner-takes-all model is great for determining a classification (mapping the inputs to a particular output node representing a class). Real-valued outputs are ideal for real-valued outputs representing signal values or time-series predictions.

Note that output values are not restricted to activations in the range 0 to 1.0 (as can be forced through the sigmoid activation function). Activations can take any real-value, including negative values, if the particular output activation function supports this.

It’s also useful to use target output values of the range 0.1 to 0.9 (for a valid classification). This avoids saturating the weights in the network to force the activation toward the extremes.

Choice of Activation Functions

There are a number of activation functions that can be used, but it’s important to note that the power of multi-layer neural networks is that they are non-linear. Therefore, using a linear activation function (for example) results in a linear activation.

For single-layer networks, linear activations and also step functions can be used. For multi-layer networks, the sigmoid, Gaussian, and hyperbolic
tangent (tanh) can be used. The only requirement for backpropagation is that the activation function must be differentiable.

**Number of Hidden Layers**

A single hidden layer can model any continuous function, and is easily trained with backpropagation. With two hidden layers, any arbitrary function can be computed, with a complexity defined by the number of nodes. For this reason, neural networks with two hidden layers are universal computing devices. But neural networks with multiple hidden layers can be more difficult to train, and therefore, models with a single hidden layer should be used if the target problem supports this.

**CHAPTER SUMMARY**

Neural networks are an ideal solution to problems that can’t be formulated easily to an algorithm, and for which lots of examples exist from which the network can be trained. Additionally, neural networks are highly parallel and distributed structures with the desirable property that they generalize well to unseen data. This chapter presented an introduction to neural networks using supervised learning algorithms, including perceptron learning, the delta-rule, backpropagation, and probabilistic neural networks.

**REFERENCES**


**EXERCISES**

1. Describe the difference between supervised and unsupervised learning.
2. What are the major components of a neural network?
4. What is the primary limitation of single-layer perceptrons?
5. Multi-layer perceptrons include non-linear activation functions at the hidden and output layers - why is this important?

6. What is the purpose of the bias component?

7. Describe the fundamental process behind perceptron learning.

8. What is a principle advantage of the Delta rule over perceptron learning?

9. Describe the basic process behind learning with the Delta rule.

10. Consider the perceptron shown in Figure 8.22. Find the error and then calculate the new weights given the training example. After the new weights are available, recalculate the error for the given sample and verify that the error has been reduced given the prior weight adjustment.
11. With the example neural network shown in Figure 8.23, calculate the weight adjustments using backpropagation learning. Recalculate the output to verify that the error has been reduced given the prior weight adjustment.


13. Using the test data shown in Figure 8.24, identify to which class the unseen feature belongs.

14. Describe a neural network architecture for time-series data prediction.

15. How can long-term memory be implemented in a neural network architecture?
In Chapter 8, we explored a number of neural network architectures and learning algorithms that were able to train with a set of example data, and then generalize for unseen data. This is called supervised learning, as the network learns with the aid of a teacher (definition of output error). This chapter will present a different model for learning, called unsupervised learning. In this model, the network is given a set of raw data, for which it must analyze with no prior knowledge of its structure, and then classify it based on the inherent similarities.

**UNSUPERVISED LEARNING**

Unsupervised learning is a valuable method for data classification as it requires nothing more than raw data, and does not require an understanding of the underlying structure of the data. Unsupervised learning algorithms can in fact identify the underlying structure by segregating the data into distinct groups for later analysis. It does this by finding similarities in the raw data, and then grouping those pieces of data together. What unsupervised learning algorithms cannot do is name the clusters of data; it has no knowledge of why training data are grouped together, and what the grouping represents (see Figure 9.1). It simply identifies the similarities and clusters appropriately.
Reinforcement learning also uses a form of unsupervised learning.

While conceptually simple, unsupervised learning is extremely valuable and can be used in a variety of problem domains. In addition to being used to identify the underlying structure of a set of raw data, it can also be used to compress or transform the data. We'll explore a number of unsupervised learning algorithms in this chapter, along with their relative strengths and weaknesses.

**HEBBIAN LEARNING**

The work of Donald Hebb represents the earliest development of a learning rule that is both biologically plausible and can be implemented as an algorithm.

This has been summarized concisely in what's called Hebb's postulate of learning:

“When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.” [Hebb 1949]

In simpler terms, if two neuron cells are firing simultaneously, then any weight connecting between them should be increased. In this way, connected neurons reinforce each other to support learning. Note here that Hebb introduced the concept of increasing weights, but not decreasing...
weights. Nevertheless, Hebb’s work served as the basis for further research in connectionist models, such as Frank Rosenblatt’s work on the perceptron learning algorithm (see Chapter 8).

In the period that Hebb introduced his work in neuron modeling, Norbert Weiner introduced the concept of Cybernetics, or the “control and communication in the animal and the machine.” Cybernetics was a multi-disciplinary field that included such established fields as electrical engineering, mathematics, biology, neurophysiology, anthropology, and psychology.

Hebb’s Rule

The basic idea behind Hebb’s rule is that whenever there are two cells with similar activations, the weights that connect them should be strengthened. From a pattern recognition perspective, this provides the means to strengthen weights when inputs are similar, and to weaken them when they’re dissimilar. This results in the very simple Eq 9.1, which implements the learning algorithm for Hebb’s rule.

\[ \Delta w_{ij} = \varepsilon x_i y_j \]  

(Eq 9.1)

where \( \varepsilon \) is a learning rate (0<\( \varepsilon \)<=1), \( x \) is the input, \( y \) is the output, and \( w_{ij} \) is the weight that connects them.

The neural network that uses the Hebb Rule is a very simple network that utilizes some number of input cells, and an identical set of output cells (see Figure 9.2). The network is fully interconnected, with weights connecting between each output cell and each input cell (so that every input has some influence over the output).

**FIGURE 9.2:** Simple pattern recall network using Hebb’s rule.
To train the network, we provide the network with the input pattern (and also duplicate the value of the input cells at the output, so that the Hebb rule builds the association). Once the weights have been trained using Hebb’s learning rule, the network can recall output patterns from those presented to the input (the weights encode the memory of the patterns) as shown in Figure 9.3.

The major disadvantage with Hebb’s Rule is that it can only create a map over orthogonal patterns of inputs. This is due to the lack of a hidden layer within the network.

**Hebb Rule Implementation**

Implementing the Hebb rule is quite simple, and can be done with a minimal amount of code. We’ll represent our network as we have with other single-layer neural networks (see Chapter 10).

We’ll show three functions from this implementation: locking in the input and expected output pattern, computing the activations, and the function implementing the Hebb rule.

*The implementation of pattern recall using the Hebb rule can be found on the CD-ROM at ./software/ch9/hebb.c.*

In the first function, we set the input pattern and also lock this same pattern into the output (define_pattern). This will be subsequently used by the Hebb rule to train the weights for the pattern (see Listing 9.1). The input and output vectors are also shown. Note that while the inputs are defined as integers, the outputs are double, since we used a real-valued learning rate. We’ll use a linear threshold to clip the values, anything greater than 0 is a ‘1,’ and anything equal to, or less than 0 is a ‘-1.’
Listing 9.1: Setting the input and output patterns for Hebb learning.

```c
int    inputs[MAX_CELLS];
double weights[MAX_CELLS][WEIGHTS_PER_ACTIVATION];
double outputs[MAX_CELLS];
void define_pattern( int *inp )
{
    int i;
    for (i = 0 ; i < MAX_CELLS ; i++) {
        inputs[i] = inp[i];
        outputs[i] = (double)inp[i];
    }
    return;
}
```

Computing the activations using Hebb’s rule is shown in Listing 9.2 (compute_activations). For each output cell, we iterate through each input and accumulate it by multiplying the associated weight by the input. Once the output activations have been computed, a step function is used to force the outputs to either a value of 1 or -1.

Listing 9.2: Computing the output activations using Hebb’s Rule.

```c
void compute_activations( int adjust_weights )
{
    int out, weight;
    /* Compute the activations */
    for (out = 0 ; out < MAX_CELLS ; out++) {
        outputs[out] = 0.0;
        for (weight = 0 ; weight < WEIGHTS_PER_ACTIVATION ; weight++) {
            outputs[out] += (weights[out][weight] * (double)inputs[weight]);
        }
        /* Clip the outputs */
        if (outputs[out] > 0.0) outputs[out] = 1.0;
        else outputs[out] = -1.0;
    }
    return;
}
```

Finally, adjusting the weights with Hebb’s rule is the simple process of accumulating weight changes using Eq 9.1. For each output cell, each
weight connecting to the input is adjusted by multiplying the input by the 
output and a small learning rate (see Listing 9.3). This adjusts the weights 
in an effort to map the input vector to the output vector.

**Listing 9.3: Weight adjustment with Hebb’s rule.**

```c
void adjust_weights( void )
{
    int out, weight;
    /* Hebbian Learning Rule */
    for (out = 0 ; out < MAX_CELLS ; out++) {
        for (weight = 0 ; weight < WEIGHTS_PER_ACTIVATION ; weight++) {
            /* Equation 9.1 */
            weights[out][weight] += ETA * (outputs[out] *
                (double)inputs[weight]);
        }
    }
    return;
}
```

To learn a new pattern, we simply specify a new image and load it with 
the define_pattern function. A call to adjust_weights can then be 
performed to create a mapping for the new pattern, for example:

```c
int pattern1[MAX_CELLS] = {-1,  1,  1, -1,
    1, -1, -1,  1,
    1, -1, -1,  1,
    -1,  1,  1,-1};
define_pattern( pattern1 );
/* Train for Pattern 1 */
adjust_weights();
```

We can then test for this pattern by computing the output activations 
with a call to compute_activations. Note that in this example, we’ve 
presented a partial input pattern in an attempt to test its recall capabilities:

```c
/* Test for Pattern 1 */
{
```
int patternA[MAX CELLS] = {-1, -1, -1, -1,
   -1, 1, 1, -1,
   1, 1, 1, 1,
   1, 1, 1, 1};
define_pattern( patternA );
}
show_activations();

Note that show_activations simply emits the input and output vector to illustrate the mapping.

Now let’s look at the algorithm in action. We’ll train our network for two distinct training patterns in a 4 by 4 matrix. Pattern one is a simple box, and pattern two is a representation of an ‘X’ pattern (see Figure 9.4).

FIGURE 9.4: Training patterns for the Hebb rule neural network.

FIGURE 9.5: Result of pattern recall using the Hebbian network.
To test the recall capabilities of the network, we’ll present the network with elements of the input pattern, but not the entire pattern (see Figure 9.5). As is shown in Figure 9.5, Hebbian learning does a reasonable job of recalling patterns given pieces of the original patterns. The Hebb rule is very simple, and can be very effective when there’s a strong correlation between the presented input pattern, and desired recall pattern.

**SIMPLE COMPETITIVE LEARNING**

In competitive learning, the output nodes of the network (which represent the distinct classes available for classification), compete with one another to fire and therefore represent the class of the feature vector at the input nodes.

*NOTE*

Clustering is a very interesting technique with very practical applications. Clustering a raw data set allows a better understanding of the structure of the data. Using clustering, relationships in data can be found that could otherwise not be seen. One very useful example of clustering is in recommendation systems, where customer data is clustered. Customers in a given cluster are then unique classes of customers and their differences can be used to recommend purchases to those in the cluster who lack that purchase.

The competitive learning network is made up of two layers. The first layer is the input layer and represents the feature vector, and the second layer is the output layer representing the class (see Figure 9.6).

*FIGURE 9.6: Sample competitive learning network.*
The weights between the input and output layer encode the ability to classify a particular input feature vector to an output class node. In some competitive learning architectures, inhibitory connections between the output layer are included. These connections allow the output nodes to influence each other in a true competitive sense.

**Vector Quantization**

One example of competitive learning is called *vector quantization*. Vector quantization is a surprisingly simple network architecture and learning algorithm. The input layer has a number of nodes defined by the length of the feature vector, and the output layer has a number of nodes defined by the number of anticipated classes. Each output node connects to every input node by a trainable weight.

As each prototype vector (sample to classify) is applied to the network, a simple feedforward algorithm computes the activation of each output node, as defined by Eq 9.2. For each output node $M_i$, the sum of the products of the prototype vector ($x_i$) and weights determines the activation.

$$y_M = \sum w_{M,i} x_i \quad \text{(Eq 9.2)}$$

A winner-takes-all approach then decides which particular output activation is chosen as the correct classification. But instead of choosing the largest activation (as we've seen with other approaches, such as Probabilistic Neural Networks), we choose the lowest activation, as it represents the class with the closest Euclidean distance to the input prototype vector. In this sense, the weights of the neural network define the centroid of all prototype vectors assigned to this class.

*NOTE*

*Vector quantization effectively divides the prototype vector input space into a Voronoi tessellation. A Voronoi tessellation (named after Georgy Voronoi) is a decomposition of a space to a specific set of discrete objects in the space (such as centroids).*

After a prototype vector is found to be closest to a given class (output node), only the weights associated with this output node are updated. The weights associated with the winning output node ($M$) are updated for input $x$, and learning rate $\rho$, as shown in Eq 9.3.

$$w_{M,i} = w_{M,i} + \rho (x_i - w_{M,i}) \quad \text{(Eq 9.3)}$$
The process of training continues until we reach some termination criteria. This could be after some number of runs, or when the network has reached equilibrium (no changes are made, such as prototype vectors changing classification).

**Vector Quantization Implementation**

Vector quantization, as illustrated by the introduction, is very easily implemented. We’ll explore a few functions that make up the implementation, the remaining source can be found on the CD-ROM.

The implementation of object classification using vector quantization can be found on the CD-ROM at ./software/ch9/vq.c.

Let’s begin with a discussion of the neural network representation for vector quantization (see Listing 9.4). In this approach, the prototype feature vectors are represented with the feature_vector_t type. This contains the current class to which the prototype belongs, and the feature vector itself (features).

The neural network is implemented very simply. Since there are no hidden layers, we need only maintain the value of the output nodes (the outputs array). The weights between the output layer and input layer are represented by a two-dimensional array (first dimension is the output node, second are the individual weights for the output node).

**Listing 9.4: Major types and variables.**

```c
#define MAX_FEATURE_VECTORS 40
#define MAX_FEATURES 3
typedef struct {
    int class;
    int features[MAX_FEATURES];
} feature_vector_t;
feature_vector_t fvec[MAX_FEATURE_VECTORS];
/*
 * Neural Network Representation for Vector Quantization
 */
#define OUTPUT_NODES 3         /* Unique classes
*/
#define CLASSES             OUTPUT_NODES
double outputs[OUTPUT_NODES];
```
double weights[OUTPUT NODES][MAX FEATURES];
#define LEARNING_RATE ((double)0.01)

Note that we’ll normalize the feature vector so that the values scale to the range 0 to 1, but we’ll not use the value zero and one (the extremes) to avoid saturating the weights of the network (see Figure 9.7).

Next, let’s continue the implementation discussion with the main loop (see Listing 9.5). The main function provides a fundamental algorithm for vector quantization, including the termination criteria. We begin by first initializing the prototype feature vectors with a call to initialize_vectors. This function creates a random set of feature vectors given the feature length and available individual features (see the source on the CD-ROM for the details of this function). Next, we initialize the neural network with a call to initialize_network. This function simply takes the first N feature prototype vectors (N being the number of classes available for classification), and associates the output node with the feature vector. This is done through a call to updateNetwork, which initializes the weights for the output node given the prototype feature vector (we’ll review this function shortly). Note that this can be problematic, especially if there are strong similarities between the first N feature prototype vectors, but it is simple, and works well given a good random distribution.

Next, we enter the main loop. This loop continues to operate while changes occur (and by change, we mean a feature prototype vector changing classes, based on the class feature centroids changing themselves). Each of the sample feature prototype vectors are applied in succession to the network, using the feed_forward function. This function returns the class to which the prototype was assigned. This information is then applied to the updateNetwork function, which updates the particular output node’s (class’s) weights given the current prototype feature vector. This process then continues for the next sample, until for each iteration through the sample set, no changes occur.

```
<table>
<thead>
<tr>
<th>Feature Vector</th>
<th>Color</th>
<th>Texture</th>
<th>Shape</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>Smooth</td>
<td>Box</td>
<td>0.3</td>
<td></td>
</tr>
<tr>
<td>Blue</td>
<td>Rough</td>
<td>Sphere</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>Black</td>
<td>Dimple</td>
<td>Cylinder</td>
<td>0.7</td>
<td></td>
</tr>
</tbody>
</table>
```

**FIGURE 9.7:** Representing the prototype feature vector.
Listing 9.5: Vector quantization main loop (portion).

```c
int main()
{
    int i, j;
    int sample = CLASSES;
    int class;
    changes = 0;
    /* Initialize the feature vectors */
    initialize_vectors();
    /* Initialize the neural network */
    initialize_network();
    /* Continue to run while feature vectors change classes */
    while (changes > 0) {
        if (sample >= MAX_FEATURE_VECTORS) {
            /* Reset the sample index */
            sample = 0;
            /* If no changes occurred in the last iteration, the exit,
             * otherwise, reset the changes counter to zero and continue.
             */
            if (changes == 0) break;
            else changes = 0;
        }
        /* Feed the sample prototype vector through the network. */
        class = feed_forward(sample);
        /* Update the weights for the winning output node (class). */
        updateNetwork(sample, class);
        /* Next sample */
        sample++;
    }
    return 0;
}
```

The `feed_forward` function implements the output node activations based on Eq 9.2 (see Listing 9.6). Given the input prototype feature vector index, we calculate the activations for each output node. This, using Eq 9.2, is the sum of the products of the weights for the particular class, and the values of the input feature vector. In a winner-takes-all fashion, we identify the output node with the lowest value, which represents the class to which the input feature vector belongs. Finally, we check to see if the feature vector
has changed classes, and if so, we increment the changes variable to let the main loop that a class change occurred.

**Listing 9.6:** Computing the output node activations with `feed_forward`.

```c
int feed_forward( int feature_vector )
{
    int output, weight, best;
    double best_value;
    /* Compute each output node activation for the current
     * prototype vector.
     */
    for (output = 0 ; output < CLASSES ; output++) {
        outputs[output] = 0.0;
        /* Equation 9.2 */
        for (weight = 0 ; weight < MAX_FEATURES ; weight++) {
            outputs[output] +=
                weights[output][weight] *
                SCALE(fvec[feature_vector].features[weight]);
        }
    }
    /* Set the current best to class 0 */
    best = 0;
    best_value = outputs[0];
    /* Iterate through the remaining classes to identify which was best. */
    for (output = 1 ; output < CLASSES ; output++) {
        if (outputs[output] < best_value) {
            best = output;
            best_value = outputs[output];
        }
    }
    /* Keep track of when a prototype vector changes classes, for use as
     * a termination criteria.
     */
    if (best != fvec[feature_vector].class) {
        changes++;
        fvec[feature_vector].class = best;
    }
    return best;
}
```
Finally, during training, once we identify that a prototype feature vector belongs to a particular class, we update the weights for the class. This is done through a call to `updateNetwork` (see Listing 9.7). To the function, we provide the prototype vector index and the class to which it belongs. Each of the weights associated with the particular output node (class) is then updated per Eq 9.3. Note that we apply a learning rate to the weight update to provide a gradual change to the weights.

**Listing 9.7: Updating the weights for a particular class with `updateNetwork`**

```c
void updateNetwork( int feature_vector, int class )
{
    int weight;
    for (weight = 0 ; weight < MAX_FEATURES ; weight++) {
        /* Equation 9.3 */
        weights[class][weight] += LEARNING_RATE * 
            ( SCALE(fvec[feature_vector].features[weight]) - 
                weights[class][weight] );
    }
    return;
}
```

Let’s now see vector quantization in practice (Listing 9.8). Per the sample implementation, a set of random data is created that represents a variety of objects that differ in shape, size, and color. These are encoded into a feature vector (shape, texture, color) and applied to the network in order to train the weights. When training is complete (equilibrium is reached in the network), the classification is emitted.

For this set of randomized data, we see a very clear classification emerge from the data. From the feature vectors, we can see that class 0 contains all objects that have the box shape. Class 2 includes all objects that are smooth (except for boxes). Finally, Class 3 includes all objects not previously classified. This includes any object that isn’t a box and isn’t smooth. That’s one possible classification, but given the number of box shapes that exist, it’s not surprising that this particular categorization was found.

**Listing 9.8: Sample classification of random object data.**

```
$ ./vq.exe
Class 0 contains:
```
Class 1 contains:

- 0 [BLACK SMOOTH SPHERE]
- 12 [BLACK SMOOTH SPHERE]
- 14 [RED SMOOTH SPHERE]
- 15 [BLUE SMOOTH SPHERE]
- 17 [RED SMOOTH SPHERE]
- 24 [RED SMOOTH CYLINDER]
- 26 [RED SMOOTH CYLINDER]
- 33 [BLUE SMOOTH CYLINDER]
- 34 [RED SMOOTH CYLINDER]
- 37 [RED SMOOTH CYLINDER]

Class 2 contains:

- 4 [BLUE DIMPLED CYLINDER]
- 5 [BLACK DIMPLED SPHERE]
- 6 [BLACK ROUGH SPHERE]
- 9 [BLACK ROUGH SPHERE]
- 10 [BLACK DIMPLED CYLINDER]
- 11 [BLACK ROUGH SPHERE]
- 19 [BLACK ROUGH SPHERE]
- 20 [BLUE ROUGH SPHERE]
- 22 [BLACK ROUGH SPHERE]
- 28 [BLUE DIMPLED CYLINDER]
- 30 [RED ROUGH CYLINDER]
Note that this method was unsupervised. The algorithm had no knowledge of the data, but did a reasonable job of segregating the data based on characteristics of the data. In one cluster, the data is segmented based on the shape, but in another, it's segmented based on a texture.

The simplicity of this algorithm makes it a great choice for clustering. One primary disadvantage of the algorithm is that the number of output classes must be defined up front. This is a significant disadvantage because it assumes that we have some general knowledge of the data and how it should be classified.

In addition to its clustering capabilities, you’ll find vector quantization in other applications such as image and audio compression, and even speaker recognition. Identifying a speaker is an interesting problem and fundamentally comes down to a classification problem. The incoming voice audio is reduced to a feature vector, which is applied to a vector quantizer to identify the class (or speaker) that fits best.

**K-MEANS CLUSTERING**

A very popular algorithm for unsupervised clustering of feature vector data is called *k-Means* (where there are k clusters, and the average of the cluster contents determine the cluster centroid). This algorithm is popular primarily because it works relatively well and is extremely simple both to understand and to implement (see Figure 9.8).

![Basic flow of the k-Means clustering algorithm.](image-url)
The fundamental approach to k-Means clustering is based on centroids representing the average of the current set of feature vectors contained within the cluster. The centroid is the average of all feature vectors, and is recalculated whenever an object moves into or out of a cluster. Choosing a cluster for a given feature vector is based on the Euclidean distance of the feature vector and the available cluster centroids (see Figure 9.9). The closer the Euclidean distance, the higher the probability the feature vector belongs to the cluster.

Note that each time a cluster changes, its centroid also changes (as it’s recalculated after every change). This means that over the duration of the algorithm, as the cluster’s membership changes, the centroids change, which can mean that additional feature vectors can change cluster membership. For this reason, a typical termination criteria is an iteration through all feature vectors with none changing cluster membership.

**k-Means Algorithm**

The k-Means algorithm is one of many clustering algorithms, but remains popular due to its simplicity. It’s also relatively efficient, but does have some disadvantages (such as inefficient methods for defining the clusters at initialization time). Let’s first explore the details of the k-Means algorithm, and then we’ll discuss the advantages and disadvantages of this method.

*As with vector quantization, we’ll use a very simple feature vector to discuss and explore k-Means in an implementation. Feature vectors will describe objects in terms of a number of dimensions (color, texture, and shape).*

We begin with a set of prototype feature vectors that are to be clustered based on their similarities and differences (see the flow shown in Figure 9.10). The number of clusters \( k \) must also be defined *a priori*. Each cluster has an associated centroid feature vector which represents the average of the prototype feature vectors contained within that cluster.
A centroid is defined as the center of the mass. In this context, the centroid is the center of a set of multi-dimensional prototype feature vectors.

We then take $k$ of the prototype feature vectors and assign one each to the available clusters. These $k$ prototype feature vectors can be picked at random, but ideally, we would choose vectors that are dissimilar. The simplest approach is to choose the first $k$ feature vectors and assign them to the clusters. With the clusters now containing one prototype feature vector, we initialize the centroid. Since the cluster contains only one prototype feature vector, the centroid is equal to the prototype feature vector.

Next, we iterate through each of the available prototype feature vectors, and compute the Euclidean distance of the centroid to the prototype feature vector (see Eq 9.4, $i$ is the feature index, $x$ is the input feature vector, and $c$ is the centroid). We pick the nearest centroid (defined as the smallest Euclidean distance) and assign the prototype vector to that centroid. We then recalculate the centroids to take into consideration their new members. Note that this can be done after each evaluation or in a batch mode where the feature vectors are assigned, and then the centroids updated at the end.

$$d = \sqrt{\sum_{i}^{\text{Dim}} (x_{i} - c_{i})^2} \quad \text{(Eq 9.4)}$$
If at the end of evaluating all prototype feature vectors, none have changed clusters, we can assume that we’ve reached equilibrium and the clustering is complete. If a centroid did change (because of gaining and/or losing a prototype feature vector), we re-evaluate each prototype vector again. Recalculating the centroid simply involves summing the prototype feature vectors and dividing the resulting vector by the total number of prototype vectors in the cluster.

The primary disadvantage of k-Means is that the number of clusters must be defined \textit{a priori}. This can require some experimentation for the problem at hand, to identify how many clusters should be present to properly classify the data. As defined in the introduction, initialization of the clusters can also be problematic. Therefore, multiple runs of the algorithm may be required to find a proper classification.

That’s the basic algorithm for k-Means clustering. Let’s now explore a sample implementation that clusters objects (based on Figure 9.7).

\textbf{k-Means Implementation}

To demonstrate the k-Means algorithm, we’ll reuse the data infrastructure from the vector quantization example (object classification based on shape, color, and texture). The prototype vectors contain both a feature vector and the current class to which the prototype vector belongs. The centroids are represented as a floating-point vector (see Listing 9.9). The centroids differ from the prototype feature vectors because the centroids will represent the average of the member prototype vectors, and therefore require floating-point precision.

\textbf{Listing 9.9: k-Means types and symbolic constants.}

```c
#define MAX_FEATURE_VECTORS 40
#define MAX_FEATURES 3
typedef struct {
  int class;
  int features[MAX_FEATURES];
} feature_vector_t;
/* Prototype Feature Vectors */
feature_vector_t fvec[MAX_FEATURE_VECTORS];
/* Number of clusters */
#define K 3
/* Cluster Centroids */
double centroids[K][MAX_FEATURES];
```
We’ll present a few of the functions that implement the k-Means algorithm: the entire implementation can be found on the CD-ROM. For the purposes of explanation, we’ll explore the k-Means main loop, identifying the nearest centroid, computing the geometric distance, and finally recalculating a centroid given the current prototype feature vectors.

The implementation of object classification using k-Means clustering can be found on the CD-ROM at ./software/ch9/kmeans.c.

The k-Means main loop implements the high-level algorithm for k-Means clustering (see Listing 9.10). It begins by initializing the random set of prototype feature vectors, and then assigning some number of them (K) to clusters. Once we have a set of initialized clusters (with one random item each), we calculate the centroid values for the clusters for later use by the algorithm.

The main loop for k-Means then begins, with the termination criteria that all clusters are satisfactorily classified (no prototype vectors moved in the last iteration). Note that we iterate through the prototype vectors backwards because the first K has been assigned to the clusters as part of the initialize_membership function. As part of the iteration, we first call the partition_feature_vector to identify to which cluster the prototype feature vector belongs. If the vector changes classes, we recalculate the centroid for the cluster that lost the vector as well as the cluster that gained it.

Listing 9.10: The k-Means algorithm main loop.

```c
void k_means_clustering( void )
{
    int done = 0, i;
    int old, new;
    int proto_vector;
    /* Create the random set of prototype feature vectors */
    initialize_prototype_vectors();
    /* Set K vectors to clusters (to initialize the centroids) */
    initialize_membership();
    /* Compute the centroids for each cluster */
    for (i = 0 ; i < K ; i++) {
        compute_centroids( i );
    }
    while (!done) {
        done = 1;
        for (i = 0 ; i < K ; i++) {
            old = K
            /* Recompute the centroids for each cluster */
            compute_centroids( i );
            new = K
            if (new == old) done = 0;
        }
    }
}
```
for (proto_vector = MAX_FEATURE_VECTORS-1 ; proto_vector >= 0 ;
    proto_vector--) {
    /* Find the cluster to which this prototype vector belongs */
    new = partition_feature_vector( proto_vector );
    /* Did the prototype feature vector change classes */
    if (new != fvec[proto_vector].class) {
        old = fvec[proto_vector].class;
        fvec[proto_vector].class = new;
        /* Recompute the affected centroids (-1 = not yet clustered) */
        if (old != -1) {
            compute_centroids( old );
        }
        compute_centroids( new );
        done = 0;
    }
}
}

The function partition_feature_vector is used to identify the cluster
with the centroid nearest the prototype feature vector under review. The
algorithm iterates through each of the available clusters, and calculates the
Euclidean distance from the prototype feature vector to the cluster's centroid
with a call to geometricDistance. As each distance is calculated, the cluster
representing the lowest distance is saved and returned as the cluster to which
this prototype vector should be moved.

Listing 9.11: Finding the nearest cluster.

int partition_feature_vector( int proto_vector )
{
    int cluster, best = 0;
    double gdCur, gdBest = 999.99;
    /* Find the centroid that best matches the prototype feature vector */
    for (cluster = 0 ; cluster < K ; cluster++) {
        gdCur = geometricDistance( proto_vector, cluster );
        /* Keep track of the closest cluster centroid */
        if (gdCur < gdBest) {
            best = cluster;
Calculating the Euclidean distance between the prototype feature vector and the cluster centroid is done with a call to geometricDistance. Using Eq 9.4, we step through each element of the vector, summing the squares of the difference. After each element of the vector has been iterated, the square root of the result is returned.

Listing 9.12: Computing the geometric distance between the feature vector and the centroid.

```c
double geometricDistance( int proto_vector, int centroid )
{
  int feature;
  double gd = 0.0;
  /* Equation 9.4 */
  for (feature = 0 ; feature < MAX_FEATURES ; feature++) {
    gd += ( ((double)fvec[proto_vector].features[feature] -
             centroids[centroid][feature]) *
            ((double)fvec[proto_vector].features[feature] -
             centroids[centroid][feature]) );
  }
  return( sqrt(gd) );
}
```

Finally, the `compute_centroids` function is used to recalculate the centroid for the defined cluster. This function very simply sums the fields of the prototype feature vectors that are contained within the current cluster, and then divides each field by the total number. The resulting centroid represents the multi-dimensional center of the cluster’s prototype feature vectors.

Listing 9.13: Recalculating the cluster centroids.

```c
void compute_centroids( int cluster )
{
  int proto_vector, feature;
```
```c
int total = 0;
/* Clear the centroid vector */
for (feature = 0; feature < MAX_FEATURES; feature++) {
    centroids[cluster][feature] = 0.0;
}
/* Calculate the centroid vector for the current cluster */
for (proto_vector = 0; proto_vector < MAX_FEATURE_VECTORS; proto_vector++) {
    if (fvec[proto_vector].class == cluster) {
        for (feature = 0; feature < MAX_FEATURES; feature++) {
            centroids[cluster][feature] += (double)fvec[proto_vector].features[feature];
        }
        total++;
    }
}
/* Compute the average for the centroid */
for (feature = 0; feature < MAX_FEATURES; feature++) {
    centroids[cluster][feature] /= (double)total;
}
return;
}
```

Let’s now look at a sample result of the $k$-Means algorithm (see Listing 9.14). Class 0 is clearly a cluster of smooth objects, and Class 2 contains red objects that are not smooth. Class 1 then becomes all objects that are neither red, nor smooth.

**Listing 9.14: Sample $k$-Means classification of random object data.**

```
$ ./kmeans.exe
Class 0 contains:
  5  [RED SMOOTH BOX   ]
  6  [RED SMOOTH BOX   ]
  7  [RED SMOOTH BOX   ]
  8  [RED SMOOTH SPHERE ]
 11 [BLACK SMOOTH SPHERE ]
 12 [BLACK SMOOTH BOX   ]
 14 [BLUE SMOOTH CYLINDER ]
 16 [BLACK SMOOTH BOX   ]
```
Recall that from initialization, the first k prototype feature vectors are assigned to their same numbered cluster (vector 0 to cluster 0, vector 1 to
cluster 1, etc.). Note in this example that while prototype vector 2 remained in cluster 2, prototype vector 0 moved to cluster 1 (since cluster 0 was used to classify smooth objects).

The $k$-Means algorithm is a useful algorithm because it’s simple and works well. It’s not without its issues (such as a priori definition of $k$, and sensitivity to initialization), but these can be combated through multiple runs of the algorithm on the data set.

*It’s good to note that there’s no theoretical solution for understanding the optimal number of classes for any dataset. An experimental solution is to execute the $k$-Means algorithm on the dataset multiple times and review the results. Fewer clusters mean better generalization in the results, where more clusters tends to end in clusters with very specific attributes and a risk of over-fitting.*

For $k$-Means and vector quantization, the primary disadvantage is that the number of clusters must be predefined. Let’s now explore an algorithm that can dynamically create new clusters when the dataset warrants them.

**ADAPTIVE RESONANCE THEORY (ART)**

Adaptive Resonance Theory is a collection of models for unsupervised learning. In this section, we’ll focus solely on ART-1, which is applicable to binary input vectors. For continuous variables, the ART-2 algorithm can be used.

ART-1 in particular was designed to resolve the stability-plasticity dilemma. This refers to a conflict in the ability to maintain old learned information while still being adaptive to learn new information. An algorithm is defined as plastic if it can adapt to new information. Additionally, an algorithm is stable if it can retain previously learned knowledge. The goal then is to create an algorithm that can retain previously learned knowledge while at the same time integrating newly discovered knowledge. In this way, the algorithm is both stable and plastic. Many clustering algorithms are one or the other, but not necessarily both.

As we discussed in k-Means clustering, an interesting advantage to ART-1 is in its ability to create a new cluster if the underlying data warrants. It accomplishes this with a vigilance parameter that helps to determine when to cluster a feature vector to a “close” cluster, or when to simply create a new cluster into which this feature vector is inserted.
ART-1 Algorithm

The ART-1 algorithm is simple, not quite as simple as the k-Means algorithm, but straightforward, nonetheless. Let’s begin with a quick overview of ART-1 and then we’ll dig deeper into the algorithm to understand how it works.

We begin the algorithm with a set of unclustered feature vectors and some number of clusters. We take each feature vector, in turn, and attempt to find the cluster to which it’s closest. If no clusters are found that are near the feature vector, we create a new cluster and assign the feature vector to it. Otherwise, with a cluster that’s near the feature vector, we test how close the feature vector is to the cluster. If the feature vector is near the cluster, then we assign the feature vector to the cluster, and update the cluster. Otherwise, we continue to test the feature vector to all available clusters. If all available clusters have been exhausted, then we simply assign the feature vector to a new cluster.

We then start the process again with a new feature vector. If we work through all available feature vectors, and none change their cluster, all samples have been satisfactorily clustered and the process is complete. That’s ART-1 in a nutshell. Now let’s explore a little further to understand how the algorithm maintains both stability and plasticity.

First, each feature vector is a binary string of a given width \(n\). Each cluster is represented by a vector that is the boolean AND operator of all feature vectors contained within the cluster (see Figure 9.11).

Note from Figure 9.11 that the width of the feature vector (and the cluster, which must be the same) is defined as \(n\) (or 8). We’ll use the symbol \(p\) to represent the current cluster (typically indexed, as in \(p_i\)). The symbol \(I\) represents the current feature vector. Finally, symbols \(\beta\) and \(\rho\) are constant values, and we’ll discuss these shortly.

Given a feature vector, and a list of available clusters (initially, all will be empty), the first step is to test the similarity of a feature vector to the cluster’s vector. This is done using Eq 9.5.

\[
\text{FIGURE 9.11: Boolean AND relationship of feature vectors to the cluster vector for ART-1.}
\]
Note that in Eq 9.5, the double vertical bars simply represent the number of 1s that are set in the particular vector. For example, the vector representing cluster A in Figure 9.11 would be 3. The inverted ‘U’ represents the boolean AND operation between the cluster vector and the current feature vector.

The similarity test in Eq 9.5 calculates how near the feature vector is to the cluster vector. The higher the value, the closer the vector is to the cluster. Therefore, if Eq 9.5 is satisfied, then the feature vector can be defined as sufficiently close to the cluster. If this equation fails for all cluster vectors, then we simply create a new cluster for this feature vector and continue to the next feature vector. Otherwise, if it is sufficiently close, we test for vigilance acceptability. The $\beta$ parameter is used as a “tie-breaker” to give deference to clusters with more 1s in the case that the feature vector and cluster are similar. This parameter is typically a small number (greater than zero and less than $n$).

The vigilance test is the final determiner for whether the feature vector should be added to the particular cluster (see Eq 9.6).

$$\frac{|p_i \cap I|}{\|p_i\|} \geq \rho \quad (\text{Eq 9.6})$$

This equation simply identifies if the feature vector is sufficiently close to the cluster (as a ratio of the matching 1s between the feature vector and the cluster). This means that if vigilance ($\rho$) is high (such as 0.9) more clusters will tend to be created, and if vigilance is low, then fewer clusters are created. If $\rho$ is set to 1.0, then the feature vector must match the cluster exactly in order to join it.

If the feature vector fails the vigilance test, and there are no further clusters to test, then a new cluster is created for this feature vector (as there are no similar clusters available).

Note that while clusters are created, feature vectors may drop out of a cluster and into another based on new feature vectors being added, and adjusting the cluster vector. When no cluster changes are made for an entire iteration through the available feature vectors, the algorithm is complete.

When a feature vector joins a cluster, the cluster’s vector must be updated to incorporate the features of the new addition. If the feature vector
added is the only member of the cluster, then the cluster vector is identical to the feature vector. Otherwise, each of the cluster’s feature vectors ($I_i$) is assimilated using Eq 9.7.

\[ p = p \cap I_i \]  
(Eq 9.7)

ART-1 is stable (retains knowledge), but is also plastic in that it can indefinitely incorporate new knowledge given a sufficient number of clusters. Let’s now explore the implementation of ART-1 for an object classification task.

**ART-Implementation**

Let’s start with a discussion of the clustering problem at hand and then review the representation for the feature vector and cluster vector. To demonstrate the ART-1 algorithm, we’ll use a feature vector representing attributes of a number of animals (see Figure 9.12). These characteristics include whether the animal gives live birth, or lays eggs, or whether the animal has fur, hair, or naked-skin, etc.

![Figure 9.12: Representing an animal's characteristics for the ART-1 algorithm.](image)

The implementation of animal clustering using ART-1 can be found on the CD-ROM at ./software/ch9/art1.c. The major functions that make up ART-1 are explored here, but the entire implementation is provided on the CD-ROM.

Both feature vectors and clusters are represented with the vector_t type. This contains not only the binary feature vector, but a union which represents the current class (in the case of a feature vector, for fvec) or the member count (in the case of a cluster, pvec). The feature vectors are statically initialized, while the clusters (pvec) are initialized with a call to initialize_prototype_vectors.
Listing 9.15: Feature vector and cluster representation for ART-1.

```c
/* Number of clusters */
#define MAX_CLUSTERS 5

/* Number of prototype feature vectors */
#define MAX_FEATURE_VECTORS 19

/* Size (width) of feature vector */
#define MAX_FEATURES 13

typedef struct {
    union {
        int class;    /* For Feature Vectors */
        int count;    /* For Clusters */
    } u;
    int features[MAX_FEATURES];
} vector_t;

/* Prototype Feature Vectors */
vector_t fvec[MAX_FEATURE_VECTORS] = {
    /* Robin */ { {-1}, { 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0 } },
    /* Spider */ { {-1}, { 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0 } },
    /* Cat */ { {-1}, { 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0 } },
    /* Salmon */ { {-1}, { 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1 } },
    /* Mouse */ { {-1}, { 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0 } },
    /* Moose */ { {-1}, { 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 } },
    /* Bat */ { {-1}, { 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0 } },
    /* Dog */ { {-1}, { 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0 } },
    /* Snake */ { {-1}, { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 } },
    /* Lion */ { {-1}, { 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0 } },
    /* Iguana */ { {-1}, { 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0 } },
    /* Dolphin */ { {-1}, { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 } },
    /* Zebra */ { {-1}, { 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0 } },
    /* Ostrich */ { {-1}, { 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0 } },
    /* Penguin */ { {-1}, { 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0 } },
    /* Tiger */ { {-1}, { 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0 } },
    /* Platypus */ { {-1}, { 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0 } },
    /* Octopus */ { {-1}, { 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0 } }
};
```
/* Clusters */
vector_t pvec[MAX_CLUSTERS];

/* Algorithm constants */
#define BETA   ((double)8.0)
#define VIGILANCE   ((double)0.2)

Let's now have a look at the support functions for ART-1, which include creating a new cluster, adding a feature vector to a cluster, and recomputing the vector for a cluster.

Creating a new cluster involves finding a cluster that's empty, and then adding the defined feature vector to it. The feature vector's class is then adjusted for the cluster and the cluster's count set (see Listing 9.16). The routine begins by searching for a cluster that has no members. If one is not found, the feature vector remains unclustered (class set to -1) and the function returns. Otherwise, the cluster vector is copied from the feature vector, and the feature vector class and cluster count are initialized.

Listing 9.16: Creating a new cluster with create_new_cluster.

```c
void create_new_cluster( int vector ) {
    int cluster, i;
    /* Find an empty cluster */
    for (cluster = 0 ; cluster < MAX_CLUSTERS ; cluster++) {
        if (pvec[cluster].u.count == 0) break;
    }
    /* No cluster available -- unclassified */
    if (cluster == MAX_CLUSTERS) {
        fvec[vector].u.class = -1;
        return;
    }
    /* Set the feature vector's class to this new cluster */
    fvec[vector].u.class = cluster;
    /* Copy the feature vector to the cluster */
    for (i = 0 ; i < MAX_FEATURES ; i++) {
        pvec[cluster].features[i] = fvec[vector].features[i];
    }
    pvec[cluster].u.count = 1;
    return;
}
```
Adding a feature vector to an existing cluster (one with existing members) is achieved with the function `add_to_cluster`. The first thing this function must do is determine if the feature vector is being removed from another cluster (if the class is not -1). If another cluster is losing this feature vector, then we must reset the class for the feature vector and then make a call to `recompute_cluster` to redefine the cluster's vector. Then, we add the feature vector to the intended cluster and recompute its vector (with another call to `recompute_cluster`).

Listing 9.17: Adding a feature vector to a cluster with `add_to_cluster`.

```c
void add_to_cluster( int cluster, int vector )
{
    int old_cluster;
    /* If feature vector had previously been clustered */
    if (fvec[vector].u.class != -1) {
        old_cluster = fvec[vector].u.class;
        fvec[vector].u.class = -1;
        pvec[old_cluster].u.count--;
        recompute_cluster( old_cluster );
    }
    /* Add the feature vector to the new cluster */
    fvec[vector].u.class = cluster;
    recompute_cluster( cluster );
    pvec[cluster].u.count++;
    return;
}
```

Now let's bring it all together with the ART-1 main loop. This function, `art1`, implements the fundamental ART-1 algorithm, using the previously discussed vector support functions. We iterate through each of the feature vectors, looking for the nearest cluster. This begins with the similarity test (per Eq 9.5), which is followed by the vigilance test (Eq 9.6). If the feature vector is near the cluster and passes the vigilance test, then the feature vector is added to the cluster (through a call to `add_to_cluster`). If the feature vector is not clustered after checking all of the available clusters, then a new cluster is created for it using `create_new_cluster`. This process continues until no changes to clusters are made.
Listing 9.18: The ART-1 main loop (art1).

```c
void art1()
{
    int done = 0, cluster, i, clustered;
    vector_t result;
    double max, sim, res_magnitude, fvec_magnitude, pvec_magnitude;
    while (!done) {
        done = 1;
        /* Iterate through each of the prototype feature vectors */
        for (i = 0 ; i < MAX_FEATURE_VECTORS ; i++) {
            clustered = 0;
            /* Iterate through each of the active clusters */
            for (cluster = 0 ; cluster < MAX_CLUSTERS ; cluster++) {
                /* Skip empty clusters */
                if (pvec[cluster].u.count == 0) continue;
                /* Mask the feature vector with the cluster vector */
                vectorAnd( &result, &fvec[i], &pvec[cluster] );
                res_magnitude = (double)vectorMagnitude( &result );
                fvec_magnitude = (double)vectorMagnitude( &fvec[i] );
                pvec_magnitude = (double)vectorMagnitude( &pvec[cluster] );
                max = res_magnitude / ( BETA + fvec_magnitude );
                sim = pvec_magnitude / ( BETA + (double)MAX_FEATURE_VECTORS );
            }
            /* Test similarity of feature vector to cluster */
            /* Equation 9.5 */
            if (max > sim) {
                /* Feature vector is sufficiently similar to cluster. Next, test
                 * for vigilance acceptability (Equation 9.6).
                 */
                if ( (res_magnitude / pvec_magnitude) >= VIGILANCE ) {
                    if (fvec[i].u.class != cluster) {
                        add_to_cluster( cluster, i );
                        done = 0;
                    }
                    clustered = 1;
                    break;
                }
            }
        } /* clusters loop */
    }
}
```
if (!clustered) {
    /* Add to an empty cluster */
    create_new_cluster( i );
    done = 0;
}
} /* vectors loop */
return;
}

Let's now have a look at ART-1 in action. Listing 9.19 shows the result of clustering the animal feature vector data shown in Listing 9.15 (with the feature columns shown in Figure 9.12). What's shown is a reasonable clustering of the data into five classes. Class 0 represents animals that fly while Class 1 contains all animals that are carnivores. Class 2 includes all four-legged animals and Class 3 has a single member (salmon), representing fish. Finally, Class 4 represents non-flying birds, but have clustered based on their having feathers and laying eggs.

**Listing 9.19:** Sample output of ART-1 clustering.

```
$ ./art1.exe
Class 0 contains:
  0 [1 1 0 0 0 0 1 0 0 1 1 0 0 ] Robin
  6 [0 1 1 0 0 0 1 0 0 1 0 1 0 ] Bat
Class 1 contains:
  1 [0 0 0 1 1 0 0 1 0 0 1 0 0 ] Spider
  8 [0 0 0 0 0 0 1 0 0 1 0 1 ] Snake
  9 [0 0 1 0 0 1 0 1 0 0 0 1 0 ] Lion
 11 [0 0 0 0 0 0 1 0 0 1 1 ] Dolphin
 16 [0 0 1 0 0 1 0 1 0 0 0 1 0 ] Tiger
 17 [0 0 1 0 0 1 0 1 0 1 0 0 ] Platypus
 18 [0 0 0 1 0 0 1 0 0 1 0 1 ] Octopus
Class 2 contains:
  2 [0 0 1 0 0 1 0 0 1 0 1 0 ] Cat
  4 [0 0 1 0 0 1 0 0 1 0 1 0 ] Mouse
  5 [0 0 0 1 0 1 0 0 1 0 0 1 ] Moose
  7 [0 0 1 0 0 1 0 0 1 0 1 0 ] Dog
 10 [0 0 0 0 1 0 0 1 1 0 1 ] Iguana
 12 [0 0 1 0 0 1 0 0 1 0 0 1 ] Zebra
```
13 [0 0 1 0 0 1 0 0 1 0 1 0] Horse
Class 3 contains:
   3 [0 0 0 0 0 0 0 0 1 1 0 1] Salmon
Class 4 contains:
   14 [1 0 0 0 0 1 0 1 0 1 0 0] Ostrich
   15 [1 0 0 0 0 1 0 0 1 0 0] Penguin

Note that these clusters were based on a $\beta$ of 8.0 and a $\rho$ of 0.2. By decreasing the vigilance parameter, we could classify the feature data into fewer clusters.

The ART-1 algorithm is a useful clustering algorithm with the obvious advantage over k-Means in that new clusters can be created if the feature data requires it. ART-1 can also be tuned using the B ("tie-breaker") and $\rho$ (vigilance) parameters. Regardless of these settings, ART-1 is stable in that once the clusters have formed, performing additional iterations of the algorithm on the same data will not change the clusters.

**HOPFIELD AUTO-ASSOCIATIVE MODEL**

As our final example of unsupervised learning, let's explore the Hopfield auto-associative model for pattern recall. Recall the discussion of a simple auto-associator early in this chapter for Hebbian learning.

An auto-associative network has the ability to store the set of training examples so that they can be recalled later. Additionally, if incomplete or noisy input patterns are provided, the auto-associative model can recall the original pattern (or memory), making them operate as a Content-Addressable Memory (or CAM).

![Figure 9.13: The Hopfield auto-associative network uses a recurrent weight structure.](image-url)
We’ll focus on the Hopfield model here, which is a recurrent neural network with discrete inputs and activations (in the domain \{-1, 1\}). Every input connects to every other input, but self-connections are not permitted. Additionally, the inputs and outputs are the same cells (see Figure 9.13).

Early learning algorithms for auto-associators were commonly called one-shot learning algorithms. Using this algorithm, the training examples are used once to generate the weights for the network, rather than tuning the weights by iterating through the examples multiple times. One-shot learning is both simple, and also very fast.

**Hopfield Auto-Associator Algorithm**

Building a simple Hopfield auto-associator is quite simple, and the algorithm is very straightforward. The first step is the creation of the weights for the recurrent neural network. This is done by summing the outer products of each training example that is to be “memorized” by the network (see Eq 9.8).

\[
W = \sum E_i E_i \quad \text{(Eq 9.8)}
\]

In our example implementation, we’ll use a one-dimensional vector to represent the examples. The result will be a two-dimensional matrix of weights (but the diagonal will be zero, as self-connections are not permitted). The weight matrix is summed over the examples that are to be trained.

We now have a weight matrix that can be used for recall of training examples. To validate the recall features of the weight matrix, we can apply the weight matrix to the example vector to produce the activation. It’s important to note that depending on the size of the vector, and number of training examples, not all examples will be stored in their entirety. Therefore, some examples may not be fully recalled. It is shown that for \(N\) cells in the Hopfield network (for the discrete case), \(0.15N\) training examples can be memorized. [Gallant 1994]

When considering recall, there are two fundamental modes by which this can operate. In the synchronous mode, all cells are updated at the same time, and therefore, each cell is able to use its inputs statically (as none of the other cells are changing while the cell is updated). In the asynchronous mode, the cells of the network fire independently and asynchronously of one another. This means that the recall is dependent on the firing order, and therefore multiple cycles may be required in order reach a state of equilibrium.
During recall, each cell of the recall vector is a sum of the products of the current input and associated weights (see Eq 9.9). Further, the output is bound to the discrete range of \([-1, 1]\) using the sign function (1 if \(S_i \geq 0\), -1 if \(S_i < 0\)).

\[
S_i(t) = \sum_j w_{ij} x_j(t) \quad \text{(Eq 9.9)}
\]

The pattern recall \(Y\) can therefore be defined as a simple matrix product of the test (example) vector \(E\), and the weight matrix \(W\) (see Eq 9.10).

\[
Y = E \cdot W \quad \text{(Eq 9.10)}
\]

Let’s now explore a sample implementation of the discrete Hopfield algorithm that demonstrates pattern recall, even in the presence of noise.

**Hopfield Implementation**

Let’s begin with a discussion of the network representation and the essential types that are used to implement the Hopfield algorithm (see Listing 9.20). We’ll use a 9 by 9 matrix for patterns in the Hopfield implementation, but represent them as a one-dimensional vector of size \(N\). As we’re implementing the discrete model of Hopfield, a two-dimensional int array is used to represent the weights. The weights array can be viewed as the first dimension representing the source vector index, and the second dimension representing the weight index to the alternate cells.

The type `example_t` is used to represent the examples, inputs vector, and outputs vector. Recall that we’re implementing symmetric updates, so we’ll maintain separate input and output vectors. A single example is shown in Listing 9.20, which demonstrates the representation of the training vector.

<table>
<thead>
<tr>
<th>Listing 9.20: Fundamental types and symbolics for Hopfield implementation.</th>
</tr>
</thead>
</table>
| ```
#define M 9
#define N (M*M)
#define MAX_EXAMPLES 4
typedef int example_t[N];
int weights[N][N];
example_t inputs;
example_t outputs;
#define SGN(x)  ((x) >= 0 ? 1 : -1)
example_t examples[MAX_EXAMPLES]={
/*Plus */ { 1, 1, 1, 1, -1, 1, 1, 1, 1,
``` |
The first step in the Hopfield algorithm is the training of the weights using the example training data (Listing 9.21). We iterate through each of the training examples, summing their outer products to produce a new matrix of connection weights. Note that self connections are not permitted, so a zero diagonal will be present in the weight matrix.

Listing 9.21: Generating the weights array using the example vectors.

```c
void generate_weights_from_examples( void )
{
    int e, r, c;
    /* First, clear the weights */
    for (r = 0 ; r < N ; r++) {
        for (c = 0 ; c < N ; c++) {
            weights[r][c] = 0;
        }
    }
    /* Equation 9.8 */
    for (e = 0 ; e < MAX_EXAMPLES ; e++) {
        for (r = 0 ; r < N ; r++) {
            for (c = 0 ; c < N ; c++) {
                /* Don't permit self-connections */
                if (r == c) continue;
                weights[r][c] += examples[e][r] * examples[e][c];
            }
        }
    }
    return;
}
```
Listing 9.22 shows the function for computing the network activations. This very simply is a matrix multiplication of the input matrix (1 by 81) by the connection weight matrix (81 by 81) resulting in the output matrix (81 by 1). We then scale the activations (output matrix) using the SGN function to bound it to the discrete output values (-1 and 1).

Listing 9.22  Computing the output activations for a given input vector.

```c
void compute_activations( void )
{
    int r,c;
    int temp[N][N];
    bzero( (void *)temp, sizeof(temp) );
    for (r = 0 ; r < N ; r++) {
        for (c = 0 ; c < N ; c++) {
            if (r == c) continue;
            temp[r][c] += inputs[r] * weights[r][c];
        }
    }
    for (c = 0 ; c < N ; c++) {
        outputs[c] = 0;
        for (r = 0 ; r < N ; r++) {
            outputs[c] += temp[r][c];
        }
        outputs[c] = SGN(outputs[c]);
    }
    return;
}
```

Using the Hopfield network model for training and recall (using the functions discussed here) is then a simple linear process (see Listing 9.23). It begins with the generation of the connection weight matrix (generate_weights_from_examples). Next, we take a training example and copy it to the inputs matrix, with some amount of noise (using set_inputs_to_example). Finally, we compute the output activations using compute_activations and then emit the resulting output matrix with emit_result.

Listing 9.23  Using the Hopfield network model.

```c
generate_weights_from_examples();
set_inputs_to_example( e, noise );
```
compute_activations();
emit_result( outputs );

Let’s now look at the Hopfield network model in action. We’ll use a 9x9 matrix to represent the input and output matrices. The implementation supports four different input patterns with increasing amounts of noise to test the recall capabilities of the network. Figure 9.14 demonstrates one instance of pattern recall for each of the four patterns. On the left side are the inputs patterns (sample pattern with up to 20% noise) and on the right is the activation of the Hopfield network (the original input pattern).

Note the similarities between Hopfield’s model and the Hebbian learning model. Weight connections are strengthened when the example cells are similar in sign, and are weakened when the signs of the cells are different.

CHAPTER SUMMARY

This chapter explored a variety of unsupervised neural network architectures and learning algorithms. Unsupervised learning algorithms are useful to discover the underlying structure of a data set. We began with a biologically plausible learning method for patterns called Hebbian Learning and demonstrated its pattern storage and recall capabilities. Next, we explored a number of clustering algorithms that were capable of segregating data based on their similarities and differences. Algorithms explored included vector quantization, k-Means clustering, and Adaptive Resonance Theory (ART-1). Finally, we ended the
chapter with a return to pattern storage and recall with a discussion of the Hopfield auto-associator (another biologically plausible model).

REFERENCES


EXERCISES

1. Explain some of the uses of unsupervised learning algorithms.
2. Define the fundamental idea behind Hebbian learning.
3. Using the Hebb rule implementation on the CD-ROM, experiment with two different datasets. In the first, use patterns that are similar, and in the second use patterns that are very different. How does recall differ in the presence of noise or incomplete input patterns?
4. In a typical winner-takes-all network, the largest activation is used as the proper classification. What is the relevance of using the smallest activation in vector quantization?
5. Define the type of neural network used for vector quantization, and explain the learning algorithm.
6. What is a primary disadvantage of vector quantization?
7. Describe the fundamental ideas behind k-Means clustering and the use of centroids for cluster representation.
8. Define the primary advantage and disadvantage of the k-Means clustering algorithm.
9. What are the issues for k-Means cluster initialization, and in what ways could it be improved?
10. What is meant by the terms plastic and stable for a clustering algorithm?
11. Describe the basic process of the ART-1 algorithm.
12. What is the purpose of the vigilance test in the ART-1 algorithm?
13. Describe the purpose of the Beta and Rho parameters for ART-1.
14. The Hopfield auto-associative model is useful for biologically plausible memory storage and recall. Describe its architecture and learning algorithm.
15. What is one-shot learning and what are its advantages?
16. Both Hebbian learning and Hopfield can be used for pattern storage and recall. Describe the fundamental differences between these two approaches.