CHAPTER 9

A Supervised Training Approach

There are 10 kinds of people in this world…

Those who understand binary and those who do not.

In this chapter, I'm going to show you a completely different way of training a neural network. Up to now, you've been using an *unsupervised* training technique. The alternative is to train your networks using a *supervised* technique. A supervised training approach can be used when you already have examples of data you can train the network with. I mentioned this in Chapter 7, "Neural Networks in Plain English," when I described how a network may be trained to recognize characters. It works like this: An input pattern is presented to the network and the output examined and compared to the target output. If the output differs from the target output, then all the weights are altered slightly so the next time the same pattern is presented, the output will be a little closer to the expected outcome. This is repeated many times with each pattern the network is required to learn until it performs correctly.

To show you how the weights are adjusted, I'm going to resort to using a simple mathematical function: the `XOR` function. But don't worry, after you've learned the principle behind the learning mechanism, I'll show you how to apply it to something much more exciting.

The XOR Function

For those of you unfamiliar with Boolean logic, the `XOR` (exclusive OR) function is best described with a table:

The `XOR` function has played a significant role in the history of neural networks. Marvin Minsky demonstrated in 1969 that a network consisting of just an input layer and an output layer could never solve this simple problem. This is because the XOR function is one of a large set of functions that are *linearly inseparable.* A function that is linearly inseparable is one, which when plotted on a 2D graph, cannot be separated with a straight line. Figure 9.1 shows graphs for the `XOR` function and the `AND` function. The `AND` function only outputs a 1 if both inputs are 1 and, as you can see, is a good example of a function that is linearly separable.

Table 9.1 The XOR Problem

A	B	A XOR B
1	1	0
0	0	0
1	0	1
0	1	1

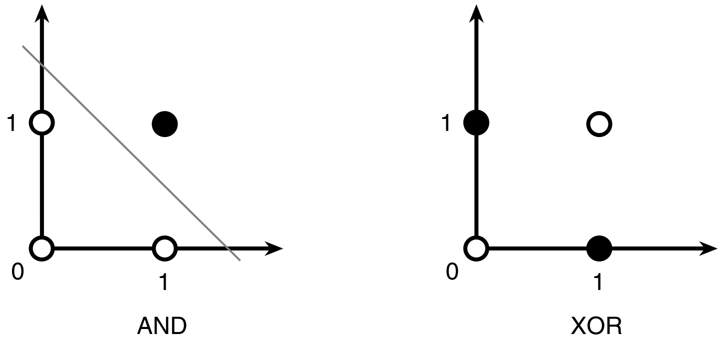

Figure 9.1

The XOR *and* AND *function. The gray line shows the linear separability of the* AND *function.*

Interesting Fact

Boolean algebra was invented by George Boole in the mid-nineteenth century. He was working with numbers that satisfy the equation $x^2 = x$ when he came up with his particular brand of logic. The only numbers that satisfy that equation are 1 and 0, or on and off, the two states of a modern digital computer. This is why you come across Boolean operators so often in conjunction with computers.

Although people were aware that adding layers between the input and output layers could, in theory, solve problems of this type, no one knew how a multilayer network like this could be trained. At this point, connectionism went out of fashion and the study of neural networks went into decline. Then in the mid seventies, a man named Werbos figured out a learning method for multilayer networks called the *backpropagation* learning method. Incredibly, this went more or less unnoticed until the early eighties when there was a great resurgence of interest in the field, and once again neural networks were the "in thing" to study among computer scientists.

How Does Backpropagation Work?

Backpropagation, or backprop for short, works like this: First create a network with one or more hidden layers and randomize all the weights—say to values between -1 and 1. Then present a pattern to the network and note its output. The difference between this value and the target output value is called the *error value*. This error value is then used to determine how the weights from the layer below the output layer are adjusted so if the same input pattern is presented again, the output will be a little closer to the correct answer. Once the weights for the current layer have been adjusted, the same thing is repeated for the previous layer and so on until the first hidden layer is reached and all the weights for every layer have been adjusted slightly. If done correctly, the next time the input pattern is presented, the output will be a little bit closer to the target output. This whole process is then repeated with all the different input patterns many times until the error value is within acceptable limits for the problem at hand. The network is then said to be *trained*.

To clarify, the training set required for an ANN to learn the XOR function would be a series of vectors like this:

This set of matched input/output patterns is used to train the network as follows:

1. Initialize weights to small random values.
2. For each pattern, repeat Steps a to e.
    a. Present to the network and evaluate the output, *o*.
    b. Calculate the error between *o* and the target output value (*t*).
    c. Adjust the weights in the output layer.
    For each hidden layer repeat d and e.

Table 9.2 The XOR Training Set

Input data	Output data (target)
(1, 1)	(0)
(1, 0)	(1)
(0, 1)	(1)
(0, 0)	(0)

   d. Calculate the error in the hidden layer.
   e. Adjust the weights in the hidden layer.

3. Repeat Step 2 until the sum of all the errors in Step b is within an acceptable limit.

This is how this learning method got its name; the error is propagated backward through the network. See Figure 9.2.

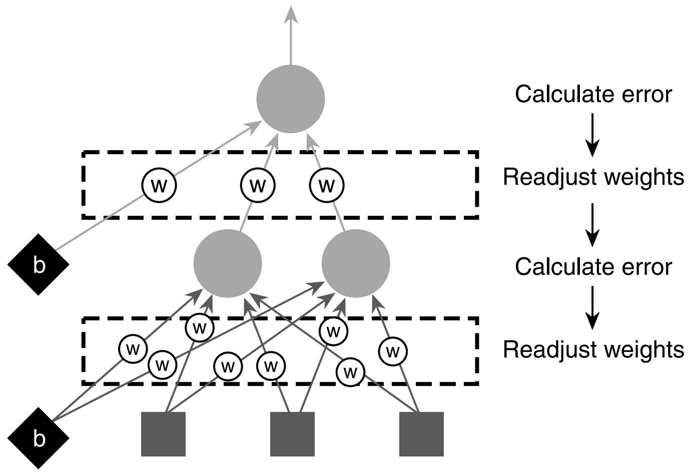

Figure 9.2

Backprop in action.

The derivation of the equations for the backprop learning algorithm is difficult to understand without some knowledge of calculus, and it's not my intention to go into that aspect here. I'm just going to show you what the equations are and how to use them. If you find that you become interested in the more theoretical side of this algorithm, then you'll find plenty of references to good reading material in the bibliography.

First I'll show you the equations, then I'll run through the XOR problem putting in actual figures, and you'll get to see backprop in action.

There are basically two sets of equations: one to calculate the error and weight adjustment for the output layer and the other to calculate the error and weight adjustments for the hidden layers. To make things less complicated, from now on I'll be discussing the case of a network with only one hidden layer. You'll almost certainly find that one hidden layer is adequate for most of the problems you'll encounter, but if two or more layers are ever required then it's not too difficult to alter the code to accommodate this.

Adjusting the Weights for the Output Layer

First, let's look at the equation to adjust the weights leading into the output layer. The output from a neuron, *k*, will be given as o_k and target output from a neuron will be given as t_k. To begin with, the error value, E_k, for each neuron is calculated.

$$E_k = (t_k - o_k) \times o_k(1 - o_k)$$

To change the weight between a unit *j* in the hidden layer and an output unit *k*, use the following formula:

$$W_{jk} += L \times E_k \times o_j$$

in which L is a small positive value known as the *learning rate*. The bigger the learning rate, the more the weight is adjusted. This figure has to be adjusted by hand to give the best performance. I'll talk to you more about the learning rate in a moment.

Adjusting the Weights for the Hidden Layer/s

The equations for calculating the weight adjustments for a neuron, *j*, in a hidden layer go like this. As before, the error value is calculated first.

$$E_j = o_k(1 - 0_k) \times \sum_{k=l}^{k=n} E_k w_{jk}$$

in which *n* is the number of units in the output layer.

Knowing the error value, the weight adjustment from the hidden unit *j*, to the input units *i*, can be made:

$$w_{ij} += L \times E_j \times o_i$$

This entire process is repeated until the error value over all the training patterns has been reduced to an acceptable level.

An Example

Now that you know what the equations are, let's quickly run through an example for a network created to solve the XOR problem. The smallest network you can build that has the capability of solving this problem is shown in Figure 9.3. This network is a little unusual in that it is *fully connected*—the inputs are connected directly to the output as well as to the hidden neuron. Although this is not the type of network you will be building very often, I'm using it here because its size enables me to concisely show you the calculations required for backprop.

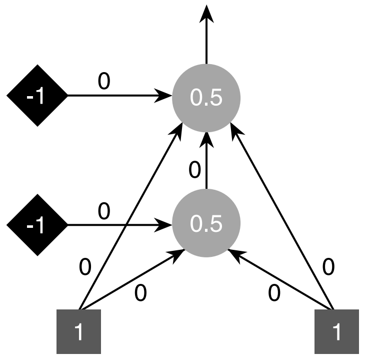

Figure 9.3

Training an XOR network.

Assume the network has been initialized with all the weights set to zero (normally, they would be set to small random values). For the sake of this demonstration, I'll just be running through the calculations using one pattern from the training set, (1, 1), so the expected target output is therefore a 0. The numbers in the neurons show the activation from that neuron. Don't forget, the sigmoid activation function gives a result of 0.5 for a zero input.

As we have discussed, the training will follow these steps:

1. Calculate the error values at the output neurons.
2. Adjust the weights using the result from Step 1 and the learning rate L.
3. Calculate the error values at the hidden neurons.
4. Adjust the weights using the result from Step 3 and the learning rate L.
5. Repeat until the error value is within acceptable limits.

Now to plug in some numbers.

Step one. 0 is the target output t_k and 0.5 is the network output o_k, so using the equation:

$$E_k = (t_k - o_k) \times o_k (1 - o_k)$$

$$\text{error} = (0 - 0.5) \times 0.5 \times (1 - 0.5) = -0.125$$

Step two. Adjust the weights going into the output layer using the equation:

$$W_{jk} += L \times E_k \times o_j$$

Calculating the new weights into the output layer going from left to right and using a learning rate of 0.1.

$$\text{New weight(bias)} = 0 + 0.1 \times -0.125 \times -1 = 0.0125$$

$$\text{New weight1} = 0 + 0.1 \times -0.125 \times 1 = -0.0125$$

New weight2 = 0 + 0.1 × -0.125 × 0.5 = -0.00625

New weight3 = 0 + 0.1 × -0.125 × 1 = -0.0125

Step three. Calculate the error value for the neuron in the hidden layer using the equation:

$$E_j = o_k(1 - 0_k) \times \sum_{k=1}^{k=n} E_k w_{jk}$$

error = 0.5 × (1 – 0.5) × -0.125 × 0.00625 = -0.000195

Notice how I've used the *updated* weight computed in Step two for w_{jk} to calculate the error. This is not essential. It's perfectly reasonable to step through the network calculating all the errors first and then adjust the weights. Either way works. I do it this way because it feels more intuitive when I write the code.

Step four. Adjust the weights going into the hidden layer. Again, going from left to right and using the equation:

$$w_{ij} += L \times E_j \times o_i$$

New weight(bias) = 0 + 0.1 × 0.000195 × -1 = 0.0000195

New weight1 = 0 + 0.1 × -0.000195 × 1 = -0.0000195

New weight2 = 0 + 0.1 × -0.000195 × 1 = -0.0000195

After this single iteration of the learning method, the network looks like Figure 9.4.

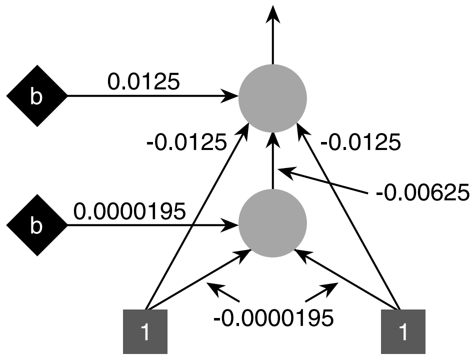

Figure 9.4

The XOR network after one iteration of backprop.

The output this updated network gives is 0.496094. Just a little bit closer to the target output of 0 than the original output of 0.5.

Step five. Go to step one.

The idea is to keep iterating through the learning process until the error value drops below an acceptable value. The number of iterations can be very large and is proportional to the size of the learning rate: the smaller the learning rate, the more iterations backprop requires. However, when increasing the learning rate, you run the risk of the algorithm falling into a local minima.

The example shown only ran one input pattern (1,1) through the algorithm. In practice, each pattern would be run through the algorithm every iteration. The entire process goes like this:

1. Create the network.
2. Initialize the weights to small random values with a mean of 0.
3. For each training pattern:

    calculate the error value for the neurons in the output layer

    adjust the weights of the output layer

    calculate the error value for neurons in the hidden layer

    adjust the weights of the hidden layer
4. Repeat Step 3 until the error is below an acceptable value.

Changes to the CNeuralNet Code

To implement backpropagation, the `CNeuralNet` class and related structures have to be altered slightly to accommodate the new training method. The first change is to the `SNeuron` structure so that a record of each neuron's error value and activation can be made. These values are accessed frequently by the algorithm.

```
struct SNeuron
{
  //the number of inputs into the neuron
  int             m_iNumInputs;

  //the weights for each input
  vector<double>  m_vecWeight;

  //the activation of this neuron
  double         m_dActivation;

  //the error value
  double        m_dError;

  //ctor
  SNeuron(int NumInputs);
};
```

The `CNeuralNet` class has also changed to accommodate the new learning algorithm. Here is the header for the new version with comments against the changes. I've removed any extraneous methods (used in the last two chapters) for clarity.

```
//define a type for an input or output vector (used in
//the training method)
typedef vector<double> iovector;
```

A training set consists of a series of `std::vector`s of doubles. This `typedef` just helps to make the code more readable.

```
class CNeuralNet
{

private:

  int         m_iNumInputs;

  int         m_iNumOutputs;

  int         m_iNumHiddenLayers;

  int         m_iNeuronsPerHiddenLyr;

  //we must specify a learning rate for backprop
  double      m_dLearningRate;

  //cumulative error for the network (sum (outputs - expected))
  double      m_dErrorSum;

  //true if the network has been trained
  bool        m_bTrained;

  //epoch counter
  int         m_iNumEpochs;

  //storage for each layer of neurons including the output layer
  vector<SNeuronLayer>  m_vecLayers;

  //given a training set this method performs one iteration of the
  //backpropagation algorithm. The training sets comprise of series
  //of vector inputs and a series of expected vector outputs. Returns
  //false if there is a problem.
  bool            NetworkTrainingEpoch(vector<iovector> &SetIn,
                                       vector<iovector> &SetOut);

  void            CreateNet();

  //sets all the weights to small random values
  void            InitializeNetwork();

  //sigmoid response curve
  inline double   Sigmoid(double activation, double response);


public:



  CNeuralNet::CNeuralNet(int    NumInputs,
                         int    NumOutputs,
                         int    HiddenNeurons,
                         double LearningRate);



  //calculates the outputs from a set of inputs
  vector<double>  Update(vector<double> inputs);

  //trains the network given a training set. Returns false if
  //there is an error with the data sets
  bool            Train(CData* data, HWND hwnd);

  //accessor methods
  bool            Trained()const{return m_bTrained;}
  double          Error()const  {return m_dErrorSum;}
  int             Epoch()const  {return m_iNumEpochs;}
};
```

Before we move on to the first code project, let me list the actual code implementation of the backprop algorithm. This method takes a training set (which is a series of `std::vectors` of doubles representing each input vector and its matching output vector) and runs the set through one iteration of the backprop algorithm. A record of the cumulative error for the training set is kept in `m_dErrorSum`. This error is calculated as the sum of the squares of each output minus its target output. In the literature, this method of calculating the error is usually abbreviated to SSE (Sum of the Squared Errors).

The `CNeuralNet::Train` method calls `NetworkTrainingEpoch` repeatedly until the SSE is below a predefined limit. At this point, the network is considered to be trained.

```
bool CNeuralNet::NetworkTrainingEpoch(vector<iovector> &SetIn,
                                      vector<iovector> &SetOut)
{
  //create some iterators
  vector<double>::iterator  curWeight;
  vector<SNeuron>::iterator curNrnOut, curNrnHid;

  //this will hold the cumulative error value for the training set
  m_dErrorSum = 0;

  //run each input pattern through the network, calculate the errors and update
  //the weights accordingly
  for (int vec=0; vec<SetIn.size(); ++vec)
  {
    //first run this input vector through the network and retrieve the outputs
    vector<double> outputs = Update(SetIn[vec]);

    //return if error has occurred
    if (outputs.size() == 0)
    {
      return false;
    }

    //for each output neuron calculate the error and adjust weights
    //accordingly
```

```
 for (int op=0; op<m_iNumOutputs; ++op)
 {
   //first calculate the error value
   double err = (SetOut[vec][op] - outputs[op]) * outputs[op]
                * (1 - outputs[op]);

   //update the error total. (when this value becomes lower than a
   //preset threshold we know the training is successful)
   m_dErrorSum += (SetOut[vec][op] - outputs[op]) *
                  (SetOut[vec][op] - outputs[op]);

   //keep a record of the error value
   m_vecLayers[1].m_vecNeurons[op].m_dError = err;

   curWeight = m_vecLayers[1].m_vecNeurons[op].m_vecWeight.begin();
   curNrnHid = m_vecLayers[0].m_vecNeurons.begin();

   //for each weight up to but not including the bias
   while(curWeight != m_vecLayers[1].m_vecNeurons[op].m_vecWeight.end()-1)
   {
     //calculate the new weight based on the backprop rules
     *curWeight += err * m_dLearningRate * curNrnHid->m_dActivation;

     ++curWeight; ++curNrnHid;
   }

   //and the bias for this neuron
   *curWeight += err * m_dLearningRate * BIAS;
 }

//**moving backwards to the hidden layer**
 curNrnHid = m_vecLayers[0].m_vecNeurons.begin();

 int n = 0;

 //for each neuron in the hidden layer calculate the error signal
 //and then adjust the weights accordingly
 while(curNrnHid != m_vecLayers[0].m_vecNeurons.end())
 {
      double err = 0;

      curNrnOut = m_vecLayers[1].m_vecNeurons.begin();

      //to calculate the error for this neuron we need to iterate through
      //all the neurons in the output layer it is connected to and sum
      //the error * weights
      while(curNrnOut != m_vecLayers[1].m_vecNeurons.end())
      {
        err += curNrnOut->m_dError * curNrnOut->m_vecWeight[n];

        ++curNrnOut;
      }

      //now we can calculate the error
      err *= curNrnHid->m_dActivation * (1 - curNrnHid->m_dActivation);

      //for each weight in this neuron calculate the new weight based
      //on the error signal and the learning rate
      for (int w=0; w<m_iNumInputs; ++w)
      {
        //calculate the new weight based on the backprop rules
        curNrnHid->m_vecWeight[w] += err * m_dLearningRate * SetIn[vec][w];
      }

      //and the bias
      curNrnHid->m_vecWeight[m_iNumInputs] += err * m_dLearningRate * BIAS;

      ++curNrnHid;
      ++n;
    }

  }//next input vector
  return true;
}
```

Well, now that you've seen the theory, let's start another fun project to illustrate how to implement it.

RecognizeIt—Mouse Gesture Recognition

Imagine you're playing a real-time strategy game and instead of having to memorize a zillion shortcut keys for troop attack and defense patterns, all you have to do is make a gesture with your mouse and your soldiers comply by rearranging themselves into the appropriate formation. Make a "V" gesture and your soldiers obediently shuffle into a "V" formation. A few minutes later they become threatened so you make a box-like gesture and they shuffle together with their shields and pikes facing outward. One more sweep of your hand and they divide into two groups. This can be achieved by training a neural network to recognize any gestures the user makes with the mouse, thereby eliminating the usual "click fest" that these sort of games usually require to get anything done. Also, the user need not be tied down to using just the built-in gestures; it's pretty easy to let the users define their own custom gestures too. Cool, huh? Let me tell you how it's done…

NOTE

If you are impatient and want to try the demo program before you read any further, you can find an executable in the Chapter9/Executables/RecognizeIt V1.0 folder on the CD.

First, you must wait until the network is trained. Then, to make a gesture, press the right mouse button and while it is still depressed make the gesture. Then release the mouse button.

All the pre-defined gestures are shown in Figure 9.7. If the network recognizes your gesture, the name of the gesture will appear in blue in the upper left-hand corner. If the network is unsure, it will have a guess.

The other versions of the RecognizeIt program you can see on the CD utilize improvements and/or alternative methods described later in this chapter.

To solve this problem, we have to:

1. Find a way of representing gestures in such a way that they may be input into a neural network.
2. Train the neural network with some predefined gestures using the method of representation from 1.
3. Figure out a way of knowing when the user is making a gesture and how to record it.
4. Figure out a way of converting the raw recorded mouse data into a format the neural network can recognize.
5. Enable the user to add his own gestures.

Representing a Gesture with Vectors

The first task is to work out how the mouse gesture data can be presented to the ANN. There are a few ways you can do this, but the method I've chosen is to represent the mouse path as a series of 12 vectors. Figure 9.5 shows how the mouse gesture for **Right Arrow** can be represented as a series of vectors.

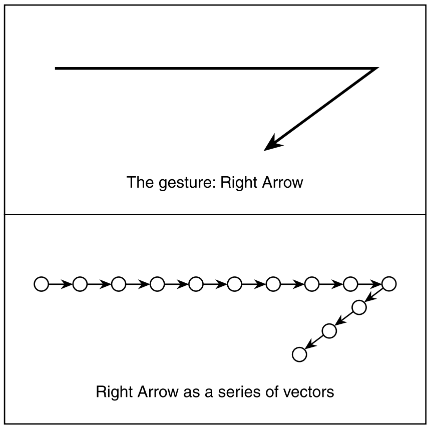

Figure 9.5

Gestures as vectors.

To aid training, these vectors are then normalized before becoming part of the training set. Therefore, all the inputs into the network have, as in previous examples, been standardized. This also gives the added advantage, when we come to process the gestures made by the user, of evenly distributing the vectors through the gesture pattern, which will aid the ANN in the recognition process.

The neural network will have the same number of outputs as there are patterns to recognize. If, for example, there are only four predefined gestures the network is required to learn: **Right, Left, Down**, and **Up** as shown in Figure 9.6, the network would have 24 inputs (to represent the 12 vectors) and four outputs.

The training set for these patterns is shown in Table 9.3.

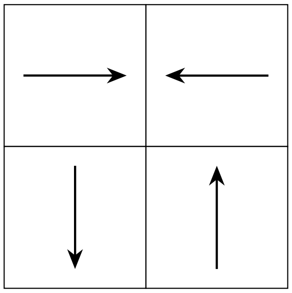

Figure 9.6

*The gestures **Right, Left, Down,** and **Up**.*

Table 9.3 Training Set to Learn the Gestures: Right, Left, Down, and Up

Gesture	Input data	Output data
Right	(1,0, 1,0, 1,0, 1,0, 1,0, 1,0, 1,0, 1,0, 1,0, 1,0, 1,0, 1,0)	(1,0,0,0)
Left	(-1,0, -1,0, -1,0, -1,0, -1,0, -1,0, -1,0, -1,0, -1,0, -1,0, -1,0, -1,0)	(0,1,0,0)
Down	(0,1, 0,1, 0,1, 0,1, 0,1, 0,1, 0,1, 0,1, 0,1, 0,1, 0,1, 0,1)	(0,0,1,0)
Up	(0,-1, 0,-1, 0,-1, 0,-1, 0,-1, 0,-1, 0,-1, 0,-1, 0,-1, 0,-1, 0,-1, 0,-1)	(0,0,0,1)

As you can see, if the user makes the gesture for **Right,** the neural net should output a 1 from the first output neuron and zero from the others. If the gesture is **Down,** the network should output a 1 from the third output neuron and zero from the others. In practice though, these types of "clean" outputs are rarely achieved because the data from the user will be slightly different every time. Even when repeatedly making a simple gesture like the gesture for **Right,** it's almost impossible for a human to draw a perfect straight line every time! Therefore, to determine what pattern the network *thinks* is being presented to it, all the outputs are scanned and the one with the highest output is the most likely candidate. If that neuron is the highest, but only outputting a figure like 0.8, then most likely the gesture is not one that the network recognizes. If the output is above 0.96 (this is the default `#defined` in the code project as `MATCH_TOLERANCE`), there is a very good chance that the network recognizes the gesture.

All the training data for the program is encapsulated in a class called `CData`. This class creates a training set from the predefined patterns (defined as constants at the beginning of CData.cpp) and also handles any alterations to the training set when, for example, a user defined gesture is added. I'm not going to list the source for `CData` here but please take a look at the source on the CD if you require further clarification of how this class creates a training set. You can find all the source code for this first attempt at gesture recognition in the Chapter9/RecognizeIt v1.0 folder.

Training the Network

Now that you know how to represent a gesture as a series of vectors and have created a training set, it's a piece of cake to train the network. The training set is passed to the `CNeuralNet::Train` method, which calls the backprop algorithm repeatedly with the training data until the SSE (Sum of the Squared Errors) is below the value `#defined` as `ERROR_THRESHOLD` (default is 0.003). Here's what the code looks like:

```
bool CNeuralNet::Train(CData* data, HWND hwnd)
{
  vector<vector<double> > SetIn  = data->GetInputSet();
  vector<vector<double> > SetOut = data->GetOutputSet();

   //first make sure the training set is valid
   if ((SetIn.size()     != SetOut.size())  ||
       (SetIn[0].size()  != m_iNumInputs)   ||
       (SetOut[0].size() != m_iNumOutputs))
   {
     MessageBox(NULL, "Inputs != Outputs", "Error", NULL);

     return false;
   }

   //initialize all the weights to small random values
   InitializeNetwork();

   //train using backprop until the SSE is below the user defined
   //threshold
   while( m_dErrorSum > ERROR_THRESHOLD )
   {
     //return false if there are any problems
     if (!NetworkTrainingEpoch(SetIn, SetOut))
     {
       return false;
     }

     //call the render routine to display the error sum
     InvalidateRect(hwnd, NULL, TRUE);
     UpdateWindow(hwnd);
   }

   m_bTrained = true;

   return true;
}
```

When you load up the source into your own compiler, you should play with the settings for the learning rate. The default value is 0.5. As you'll discover, lower values slow the learning process but are almost always guaranteed to converge. Larger values speed up the process but may get the network trapped in a local minimum. Or, even worse, the network may not converge at all. So, like a lot of the other parameters you've encountered so far in this book, it's worth spending the time tweaking this value to get the right balance.

Figure 9.7 shows all the predefined gestures the network learns when you run the program.

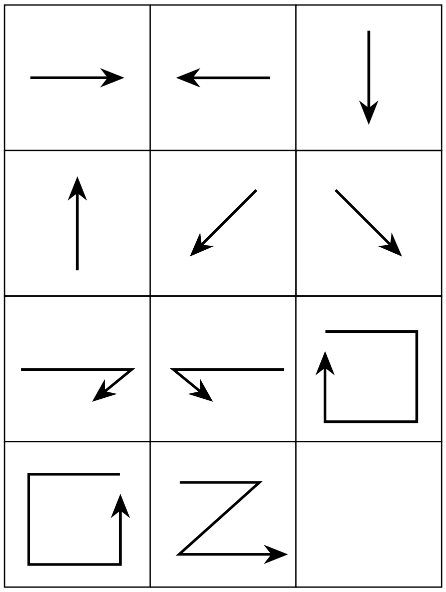

Figure 9.7

Predefined gestures.

Recording and Transforming the Mouse Data

To make a gesture, the user depresses the right mouse button and draws a pattern. The gesture is finished when the user releases the right mouse button. The gesture is simply recorded as a series of `POINTS` in a `std::vector`. The `POINTS` structure is defined in windef.h as:

```
typedef struct tagPOINTS {
    SHORT x;
    SHORT y;
} POINTS;
```

Unfortunately, this vector can be any size at all, depending entirely on how long the user keeps the mouse button depressed. This is a problem because the number of inputs into a neural network is fixed. We, therefore, need to find a way of reducing the number of points in the path to a fixed predetermined size. While we are at it, it would also be useful to "smooth" the mouse path data somewhat to take out any small kinks the user may have made in making the gesture. This will help the user to make more consistent gestures.

As discussed earlier, the example program uses an ANN with 24 inputs representing 12 vectors. To make 12 vectors, you need 13 points (see Figure 9.5), so the raw mouse data has to be transformed in some way to reduce it to those 13 points. The method I've coded does this by iterating through all the points, finding the smallest span between the points and then inserting a new point in the middle of this shortest span. The two end points of the span are then deleted. This procedure reduces the number of points by one. The process is repeated until only the required number of points remains.

The code to do this can be found in the `CController` class and looks like this:

```
bool CController::Smooth()
{
  //make sure it contains enough points for us to work with
  if (m_vecPath.size() < m_iNumSmoothPoints)
  {
    //return
    return false;
  }

  //copy the raw mouse data
  m_vecSmoothPath = m_vecPath;

  //while there are excess points iterate through the points
  //finding the shortest spans, creating a new point in its place
  //and deleting the adjacent points.
  while (m_vecSmoothPath.size() > m_iNumSmoothPoints)
  {
    double ShortestSoFar = 99999999;

    int PointMarker = 0;

    //calculate the shortest span
    for (int SpanFront=2; SpanFront<m_vecSmoothPath.size()-1; ++SpanFront)
    {
      //calculate the distance between these points
      double length =
      sqrt( (m_vecSmoothPath[SpanFront-1].x - m_vecSmoothPath[SpanFront].x) *
            (m_vecSmoothPath[SpanFront-1].x - m_vecSmoothPath[SpanFront].x) +

            (m_vecSmoothPath[SpanFront-1].y - m_vecSmoothPath[SpanFront].y)*
            (m_vecSmoothPath[SpanFront-1].y - m_vecSmoothPath[SpanFront].y));

      if (length < ShortestSoFar)
      {
        ShortestSoFar = length;

        PointMarker = SpanFront;
      }
    }

    //now the shortest span has been found calculate a new point in the
    //middle of the span and delete the two end points of the span
    POINTS newPoint;

    newPoint.x = (m_vecSmoothPath[PointMarker-1].x +
                  m_vecSmoothPath[PointMarker].x)/2;

    newPoint.y = (m_vecSmoothPath[PointMarker-1].y +
                  m_vecSmoothPath[PointMarker].y)/2;

    m_vecSmoothPath[PointMarker-1] = newPoint;

    m_vecSmoothPath.erase(m_vecSmoothPath.begin() + PointMarker);
  }

  return true;
}
```

This method of reducing the number of points is not perfect because it doesn't account for features in a shape, such as corners. Therefore, you'll notice that when you draw a gesture like **Clockwise Square,** the smoothed mouse path will tend to have rounded corners. However, this algorithm is fast, and because the ANN is trained using smoothed data, enough information is retained for the neural network to recognize the patterns successfully.

Adding New Gestures

The program also lets the user define his own gestures. This is simple to do provided the gestures are all sufficiently unique, but I wanted to write a paragraph or two about it because it addresses an important point about adding data to a training set. If you have a trained neural network and you need to add an additional pattern for that network to learn, it's usually a bad idea to try and run the backprop algorithm again for just that additional pattern. When you need to add data, first add it to the existing training set and start afresh. Wipe any existing network you have and completely retrain it with the new training set.

A user may add a new gesture by pressing the L key and then making a gesture as normal. The program will then ask the user if he or she is happy with the entered gesture. If the user is satisfied, the program smoothes the gesture data, adds it to the current training set, and retrains the network from scratch.

The CController Class

Before I move on to some of the improvements you can make to the program, let me show you the header file for the `CController` class. As usual, the `CController` class is the class that ties all the other classes together. All the methods for handling, transforming, and testing the mouse data can be found here.

```
class CController
{

private:

  //the neural network
  CNeuralNet*     m_pNet;

  //this class holds all the training data
  CData*          m_pData;

  //the user mouse gesture paths - raw and smoothed
  vector<POINTS> m_vecPath;
```

```
vector<POINTS> m_vecSmoothPath;

//the smoothed path transformed into vectors
vector<double> m_vecVectors;

//true if user is gesturing
bool    m_bDrawing;

//the highest output the net produces. This is the most
//likely candidate for a matched gesture.
double  m_dHighestOutput;

//the best match for a gesture based on m_dHighestOutput
int     m_iBestMatch;

//if the network has found a pattern this is the match
int     m_iMatch;

//the raw mouse data is smoothed to this number of points
int     m_iNumSmoothPoints;

//the number of patterns in the database;
int     m_iNumValidPatterns;

//the current state of the program
mode    m_Mode;
```

The program can be in one of four states: TRAINING when a training epoch is underway, ACTIVE when the network is trained and the program is ready to recognize gestures, UNREADY when the network is untrained, and finally LEARNING when the user is entering a custom-defined gesture.

```
//local copy of the application handle
HWND    m_hwnd;

//clears the mouse data vectors
void    Clear();

//given a series of points this method creates a path of
```

```
  //normalized vectors
  void    CreateVectors();

  //preprocesses the mouse data into a fixed number of points
  bool    Smooth();

  //tests for a match with a pre-learnt gesture by querying the
  //neural network
  bool    TestForMatch();

  //dialog box procedure. A dialog box is spawned when the user
  //enters a new gesture.
  static BOOL CALLBACK DialogProc(HWND   hwnd,
                                  UINT   msg,
                                  WPARAM wParam,
                                  LPARAM lParam);

  //this temporarily holds any newly created pattern names
  static string m_sPatternName;


public:

  CController(HWND hwnd);

  ~CController();

  //call this to train the network using backprop with the current data
  //set
  bool TrainNetwork();

  //renders the mouse gestures and relevant data such as the number
  //of training epochs and training error
  void Render(HDC &surface, int cxClient, int cyClient);

  //returns whether or not the mouse is currently drawing
  bool Drawing()const{return m_bDrawing;}

  //this is called whenever the user depresses or releases the right
  //mouse button.
  //If val is true then the right mouse button has been depressed so all
  //mouse data is cleared ready for the next gesture. If val is false a
  //gesture has just been completed. The gesture is then either added to
  //the current data set or it is tested to see if it matches an existing
  //pattern.
  //The hInstance is required so a dialog box can be created as a child
  //window of the main app instance. The dialog box is used to grab the
  //name of any user defined gesture
  bool Drawing(bool val, HINSTANCE hInstance);

  //clears the screen and puts the app into learning mode, ready to accept
  //a user defined gesture
  void LearningMode();

  //call this to add a point to the mouse path
  void AddPoint(POINTS p)
  {
    m_vecPath.push_back(p);
  }
};
```

Some Useful Tips and Techniques

There are many tips and tricks that enable your network to learn quicker or to help it generalize better, and I'm going to spend the next few pages covering some of the more popular ones.

Adding Momentum

As you've seen, the backprop algorithm attempts to reduce the error of the neural network a little each epoch. You can imagine the network having an error landscape, similar to the fitness landscapes of genetic algorithms. Each iteration, backprop determines the gradient of the error at the current point in the landscape and attempts to move the error value toward a global minimum. See Figure 9.8.

Unfortunately, most error landscapes are not nearly so smooth and are more likely to represent the curve shown in Figure 9.9. Therefore, if you are not careful, your algorithm can easily get stuck in a local minima.

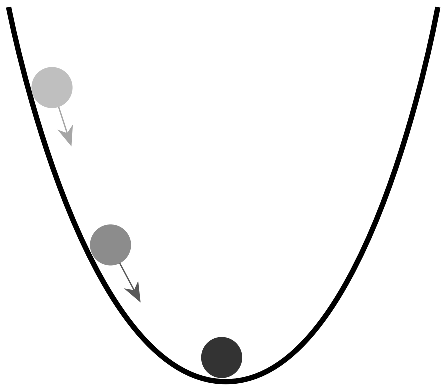

Figure 9.8

Finding the global minimum of the error landscape.

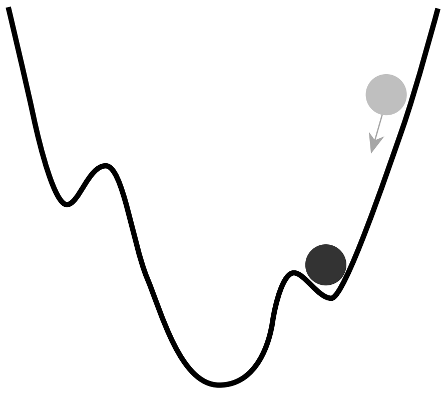

Figure 9.9

Stuck in a local minima!

One way of preventing this is by adding *momentum* to the weight update. To do this, you simply add a fraction of the previous time-step's weight update to the current weight update. This will help the algorithm zip past any small fluctuations in the error landscape, thereby giving a much better chance of finding the global minimum. Using momentum also has the added bonus of reducing the number of epochs it takes to train a network. In this example, momentum reduces the number of epochs from around 24,000 to around 15,000.

The equation shown earlier for the weight update:

$$w_{ij} += L \times E_j \times o_i$$

with momentum added becomes:

$$w_{ij} += L \times E_j \times o_i + m \times \Delta w_{ij}$$

in which the [CapDelta]w_{ij} is the previous time-step's weight update, and *m* represents the fraction to be added. *m* is typically set to 0.9.

Momentum is pretty easy to implement. The `SNeuron` structure has to be changed to accommodate another `std::vector` of doubles, in which the previous time-step's weight updates are stored. Then additional code is required for the backprop training itself. You can find the source code in the folder Chapter9/RecognizeIt v2.0 (with momentum) on the CD.

Overfitting

As you may have realized by now, a neural network is basically a function approximator. Given a training set, the ANN attempts to find the function that will fit the input data to the output data. One of the problems with neural networks is that they can learn to do this too well and lose the ability to generalize. To show you what I mean, imagine a network that is learning to approximate the function that fits the data shown in graph A in Figure 9.10.

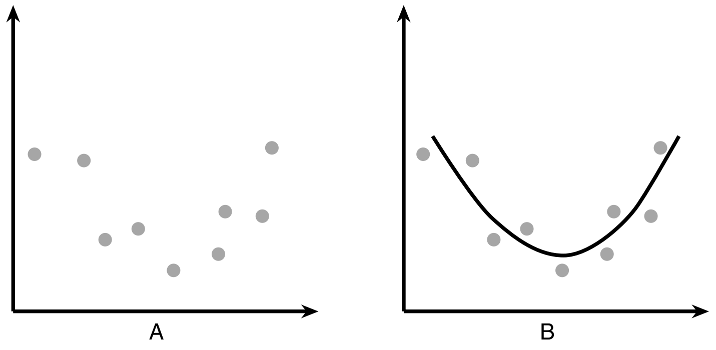

Figure 9.10

Finding the best fit for a data set.

The simplest curve that fits the data is shown in graph B, and this is the curve you ideally want the network to learn. However, if the network is designed incorrectly, you may end up with it *overfitting* the data set. If this is the case, you may end up with it learning a function that describes the curve shown in Figure 9.11.

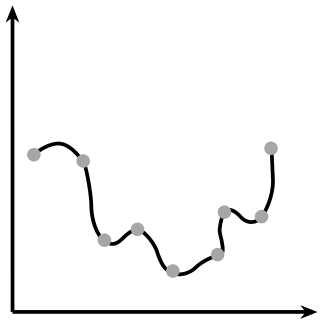

Figure 9.11

Overfitting a data set.

With a network like this, although it's done a great job of fitting the data it has been trained with, it will have great difficulty predicting exactly where any new data presented to it may fit. So what can you do to prevent overfitting? Here are a few techniques you can try:

Minimizing the number of neurons. The first thing you should do is reduce the number of hidden neurons in your network to a minimum. As previously mentioned, there is no rule of thumb for judging the amount; as usual, you'll need to determine it by good old trial and error. I've been using just six hidden units for the RecognizeIt app and I got there by starting off with 12 and reducing the number until the performance started to degrade.

Adding jitter. In this context, *jitter* is not some long-forgotten dance from the '50s, but a way of helping the network to generalize by adding noise (random fluctuations around a mean of zero) to the training data. This prevents the network from fitting any specific data point too closely and therefore, in some situations can help prevent overfitting. The example found in Chapter 9/RecognizeIt v3.0 (with jitter) has a few additional lines of code in the `CNeuralNet::Update` method that adds noise to the input data. The maximum amount of noise that can be added is `#defined` as `MAX_NOISE_TO_ADD`. However, adding jitter to the mouse gesture application only makes a very small amount of difference. You will find you will get better results with jitter when using large training sets.

Early stopping. Early stopping is another simple technique and is a great one to use when you have a large amount of training data. You split the training data into two sets: a training set and a *validation* set. Then, using a small learning rate and a network with plenty of hidden neurons—there's no need to worry about having too many in this case—train the network using the training set, but this time make periodic tests against the validation set. The idea is to stop the training when the *error from testing against the validation set* begins to increase as opposed to reducing the SSE below a predefined value. This method works well when you have a large enough data set to enable splitting and can be very fast.

The Softmax Activation Function

Some problems, like the mouse gesture application, are *classification* problems. That is to say, given some data, the network's job is to place it into one of several categories. In the example of the RecognizeIt program, the neural network has to decide which category of pattern the user's mouse gestures fall into. So far we've just been choosing the output with the highest value as the one representing the best match. This is fine, but sometimes it's more convenient if the outputs represent a *probability* of the data falling into the corresponding category. To represent a probability, all the outputs must add up to one. To achieve this, a completely different activation function must be used for the output neurons: the *softmax* activation function. It works like this:

For a network with n output units, the activation of output neuron o_i is given by

$$\text{output} = \frac{\exp(w_i x_j)}{\sum_{0}^{n} \exp(w_n x_n)}$$

in which $w_i x_i$ is the sum of all the inputs × weights going into that neuron.

This can be a fairly confusing equation, so let me run through it again to make sure you understand what's going on. To get the output from any particular output neuron, you must first sum all the weights × inputs for every output neuron in the neural network. Let's call that total *A*. Once you've done that, to calculate the output, you iterate through each output neuron in turn and divide the exponential of that neuron's *A* with the sum of the exponentials of all the output neurons' *A*s. And just to make doubly sure, here's what it looks like in code:

```
double expTot = 0;

//first calculate the exp for the sum of the outputs
for (int o=0; o<outputs.size(); ++o)
{
   expTot += exp(outputs[o]);
}

//now adjust each output accordingly
for (o=0; o<outputs.size(); ++o)
{
  outputs[o] = exp(outputs[o])/expTot;
}
```

Got it? Great. If you check the `CNeuralNet::Update` method in version 4.0 of the RecognizeIt source found in the Chapter9/RecognizeIt v4.0 (softmax) folder on the CD, you'll see how I've altered it to accommodate the softmax activation function.

Although you can use the sum squared error function (SSE), as used previously, a better error function to use when utilizing softmax is the *cross-entropy* error function. You don't have to worry where this equation comes from, just be assured that this is the better error function to apply when your network is designed to produce probabilities. It looks like this:

$$E = \sum_{j=0}^{n} t_j \log y_j$$

Where *n* is the number of output neurons, *t* is the target value, and *y* is the actual output.

Applications of Supervised Learning

As you have learned, supervised techniques are useful whenever you have a series of input patterns that need to be mapped to matching output patterns. Therefore, you can use this technique for anything from Pong to beat'emups and racing games.

As an example, let's say you are working on a racing game and you want your neural network to drive the cars as well as that spotty-chinned games tester your company employs who does nothing but race your cars and discuss Star Trek all day long. You get the guy to drive the car, and this time while he's zipping full blast around the course, you create a training set by recording any relevant data. Each frame (or every N frames), for the input training set, you would record information like:

- Distance to left curb
- Distance to right curb
- Current speed
- Curvature of current track segment
- Curvature of next track segment
- Vector to best driving line

And for the output training set, you would record the driver's responses:

- Amount of steering left or right
- Amount of throttle
- Amount of brake
- Gear change

After a few laps and over a few different courses, you will have amassed enough data to train a neural network to behave in a similar fashion. Given enough data and the correct training, the neural network should be able to generalize what it has learned and handle tracks it has never seen before. Cool, huh?

A Modern Fable

Before I finish this chapter, I'd like to leave you with a little story. Apparently, the story is a true one but I haven't been able to get that confirmed. However, please keep the story in mind when you are training your own neural networks because I'm sure you wouldn't want to make the same mistake as the military <smile>.

Once upon a time, a few Wise Men thought it would be a terrific idea to mount a camera on the side of a tank and continually scan the environment for possible threats, like… well, another bigger tank hiding behind a tree. They thought this would be a great idea because they knew computers were exceptional at doing repetitive tasks. Computers never grow tired or complacent. They never grow bored and they never need a break. Unfortunately, computers are terrible at recognizing things. The Wise Men knew this also, but they also knew about neural networks. They'd heard good things about this newfangled technology and were prepared to spend some serious money on it. And they made it so.

The following day, the Wise Men decreed that two sets of images be made. One set of images were of tanks partially hidden among trees and the other set were of trees alone, standing tall and proud. The Wise Men examined the images and saw that they were good. Half of the images from each set were put away for safe keeping in a darkened room with the door firmly locked and the windows barred—for the Wise Men were big on security.

On the third day, a state-of-the-art mainframe computer was purchased and its towering bulk was lowered by crane into a specially constructed room. One of the Wise Men's underlings flicked a switch and the gigantic machine whirred into life, along with five tons of air conditioning equipment and a state-of-the-art shiny steel coffee machine. A team of incredibly intelligent programmers were hired at great cost and flown in from all the corners of the world. The programmers observed the machine with its many flickering lights, whirring magnetic tapes, and glowing terminals, and saw that it was good.

On the fourth day, the programmers brought forth an artificial neural network according to their kind. After many hours of testing to make sure it was working properly and without bugs, they started to feed the network the images of the tanks and the trees. Each time an image was shown to the network, the machine had to guess if there was a tank among the trees or not. At first, the machine did poorly but the clever programmers punished the machine for its mistakes, and in no time at all it was improving in leaps and bounds. By the end of the fourth day, the network was getting every single answer correct. Life was good.

Although, the coffee, by now had grown thick and rank.

On the morning of the fifth day, the clever programmers double-checked the results and called the Wise Men forth. The Wise Men watched the machine accurately recognizing the tanks and saw that it was good. Then they commanded that the doors of the darkened room be flung open and the remaining images be brought forth. The clever programmers were apprehensive because although they had known this moment would come, they did not know what to expect. Would the neural network perform well or not? It was impossible to say because although they had designed the network, they didn't really have a clue what was happening inside. And so, with trembling hands, the clever programmers fed the machine the new images one by one.

Verily, they were all much relieved and happy to see that every answer was good and much joy was felt in their hearts.

The sixth day dawned and the Wise Men were concerned, for they knew that things never go this well in the world of mortals. And so they decreed that a new set of images be taken and be brought forth with all speed. The new images were presented to the machine, but to their horror the answers were completely random. *"Oh no!"* cried the clever programmers. *"Verily we hath truly made a mighty screw up!"*

One Wise Man pointed his bony index finger at the all-of-a-sudden-not-so-clever-programmers, who promptly vanished in a puff of smoke.

On the seventh and eighth day, and for many more days thereafter, the Wise Men and a newly hired team of clever programmers wondered how it had all gone so wrong. No one could guess until one day an observant programmer noticed that the images with tanks in the initial set of photos were all taken on a cloudy day, whilst the images without the tanks were all taken on a sunny day. The machine had simply learned to distinguish between a sunny day and an overcast one!

Stuff to Try

1. Try adjusting the learning rate and other parameters to see what effect they have on the network training. While you are doing this, make sure you try altering the activation response of the sigmoid function to see how changing the response curve affects the learning.

2. Train a neural network to play Pong. First, figure out a way of training it using a supervised approach. Once you've cracked that, write some code to evolve networks to play Pong as per the last couple of chapters.
3. Train a network to play tic-tac-toe as above.

This page intentionally left blank