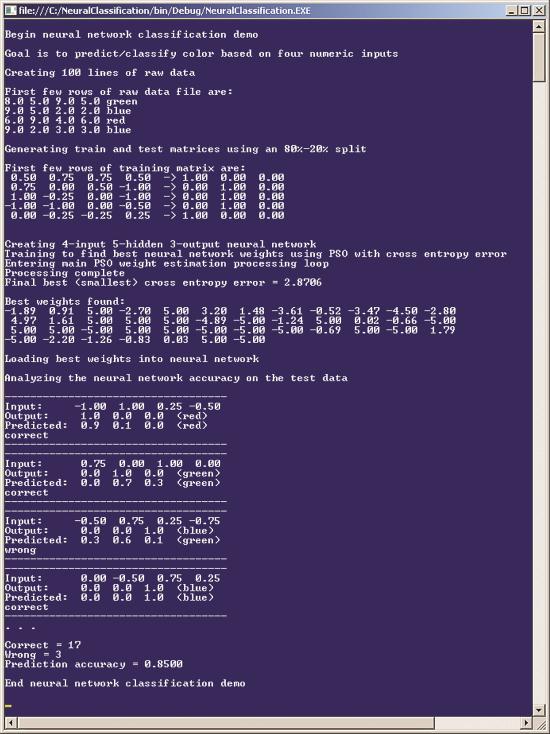
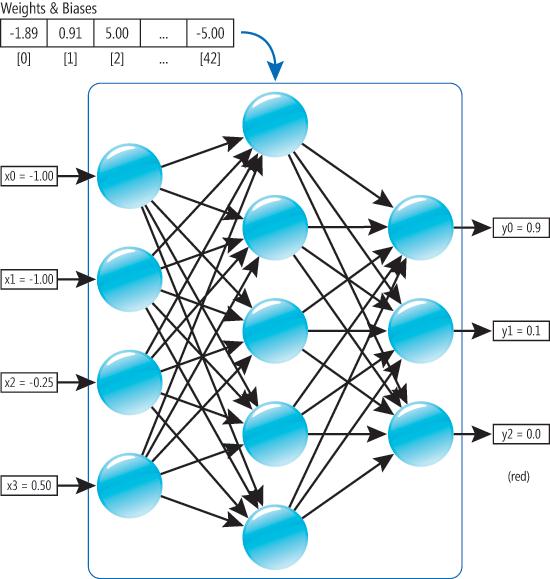
Each line of space-delimited data has five fields. The first four numeric fields are sepal (the green bud covering) length, sepal width, petal (the colored part of the flower) length and petal width. The fifth field is the species: Setosa, Versicolor or Virginica. The goal of classification is to determine an equation or set of rules that predicts which species or class an iris belongs to. Then the rule set can be used to predict the class of a new iris based on its values for sepal and petal length and width. This iris plant data is a classic example first used by R. A. Fisher in 1936. It may not excite you, but classification is extremely important. Examples include classifying an applicant’s credit rating based on variables such as income and monthly expenses (or equivalently, predicting their credit worthiness), and classifying whether a hospital patient has cancer based on the values from a blood test.

There are many approaches for classifying data, including the use of neural networks. One way to think about neural networks is that they are virtual input-output devices that accept any number of numeric inputs and produce any number of numeric outputs. The best way for you to see where I’m headed is to examine the screenshot in **Figure 1** and the image in **Figure 2**. **Figure 1** shows neural network classification in action. To keep the concepts of classification using neural networks clear, I didn’t use real-life data. Instead I used artificial data where the input x-values are four arbitrary numeric values with no particular meaning. The output y-variable to classify is color, and it can take one of three categorical values: red, green or blue. The program shown in **Figure 1** begins by generating a text file with 100 lines of artificial data (for example, “8.0 5.0 9.0 5.0 green”), and then displays the first four lines of that data. Next, the program reads the raw data file into memory as a training matrix with 80 rows of data and a testing matrix with 20 rows. Notice that two transformations are applied to the raw data. The raw numeric input data is normalized so that all values are between -1.00 and +1.00, and the raw output data (such as “red”) is encoded into a vector with three values (“1.0 0.0 0.0”).

  
**Figure 1 Neural Network Classification in Action**

  
**Figure 2 Neural Network Structure**

After creating the training and testing matrices, the demo program creates a fully connected feed-forward neural network with three input neurons, five hidden neurons for computation and three output neurons. It turns out that a 4-5-3 fully connected neural network requires 43 weights and biases. Next, the classification program analyzes the training data to find the best 43 weights and biases (those that minimize the total classification error). The program uses particle swarm optimization along with cross-entropy error to estimate the best values of the weights and biases.

The classification program then loads the best weights and biases into the neural network and evaluates the predictive accuracy of the model on the 20 rows of data in the test matrix. Notice the output of the neural network has been designed so that the three output values sum to 1.0. In this example, the model correctly predicts 17 of the 20 test vectors. The image in **Figure 2** illustrates the neural network accepting input of (-1.00, 1.00, 0.25, -0.50) and generating a predicted output of (0.9, 0.1, 0.0), which corresponds to red.

The example program points out that there are five main decisions to make when using neural networks for classification where the input data is numeric and the output data is categorical:

1. How to normalize numeric input data
2. How to encode categorical output data
3. How to generate neural output in the range [0.0, 1.0]
4. How to measure error when training
5. How to measure accuracy when testing

In the sections that follow I’ll explain that the choices made in the example program are:

1. Perform a linear transformation on numeric input data
2. Use 1-of-N encoding for categorical output data
3. Use a Softmax activation function for the output layer
4. Use cross-entropy error to determine the best weights
5. Use a winner-takes-all approach to determine accuracy

The program code that generated the screenshot in **Figure 1** is a bit too long to present in this article, so I focus on the algorithms used instead. The complete program source is available from the MSDN code download site at [archive.msdn.microsoft.com/mag201207TestRun](http://archive.msdn.microsoft.com/mag201207TestRun). This article assumes you have advanced programming skills and a basic understanding of neural networks. I explain neural network basics in the May 2012 issue of MSDN Magazine ([msdn.microsoft.com/magazine/hh975375](http://msdn.microsoft.com/magazine/hh975375)).

## Overall Program Structure

**Figure 3** lists the program structure of the example shown running in **Figure 1**. I used Visual Studio 2010 to create a single C# console application named NeuralClassification. In the Solution Explorer window, I renamed file Program.cs to the more descriptive NeuralClassificationProgram.cs, which automatically renamed the class containing Main. I removed unneeded using statements that were generated by the Visual Studio template and added a reference to the System.IO namespace.

**Figure 3 Neural Classification Program Structure**

1. **using** System;
2. **using** System.IO;
3. **namespace** NeuralClassification
4. {
5. **class** NeuralClassificationProgram
6. {
7. **static** Random rnd = **null**;
8. **static** **void** Main(**string**[] args)
9. {
10. **try**
11. {
12. Console.WriteLine("\nBegin Neural network classification\n");
13. rnd = **new** Random(159); // 159 makes a nice example
14. **string** dataFile = "..\\..\\colors.txt";
15. MakeData(dataFile, 100);
16. **double**[][] trainMatrix = **null**;
17. **double**[][] testMatrix = **null**;
18. MakeTrainAndTest(dataFile, **out** trainMatrix, **out** testMatrix);
19. NeuralNetwork nn = **new** NeuralNetwork(4, 5, 3);
20. **double**[] bestWeights = nn.Train(trainMatrix);
21. nn.SetWeights(bestWeights);
22. **double** accuracy = nn.Test(testMatrix);
23. Console.WriteLine("\nEnd neural network classification\n");
24. }
25. **catch** (Exception ex)
26. {
27. Console.WriteLine("Fatal: " + ex.Message);
28. }
29. } // Main()
30. **static** **void** MakeData(**string** dataFile, **int** numLines) { ... }
31. **static** **void** MakeTrainAndTest(**string** file, **out** **double**[][] trainMatrix,
32. **out** **double**[][] testMatrix) { ... }
33. }
34. **class** NeuralNetwork
35. {
36. // Class member fields here
37. **public** NeuralNetwork(**int** numInput, **int** numHidden,
38. **int** numOutput) { ... }
39. **public** **void** SetWeights(**double**[] weights) { ... }
40. **public** **double**[] ComputeOutputs(**double**[] currInputs) { ... }
41. **private** **static** **double** SigmoidFunction(**double** x) { ... }
42. **private** **static** **double**[] Softmax(**double**[] hoSums) { ... }
43. **public** **double**[] Train(**double**[][] trainMatrix) { ... }
44. **private** **double** CrossEntropy(**double**[][] trainData,
45. **double**[] weights) { ... }
46. **public** **double** Test(**double**[][] testMatrix) { ... }
47. }
48. **public** **class** Helpers
49. {
50. **static** Random rnd = **new** Random(0);
51. **public** **static** **double**[][] MakeMatrix(**int** rows, **int** cols) { ... }
52. **public** **static** **void** ShuffleRows(**double**[][] matrix) { ... }
53. **public** **static** **int** IndexOfLargest(**double**[] vector) { ... }
54. **public** **static** **void** ShowVector(**double**[] vector, **int** decimals,
55. **bool** newLine) { ... }
56. **public** **static** **void** ShowMatrix(**double**[][] matrix, **int** numRows) { ... }
57. **public** **static** **void** ShowTextFile(**string** textFile, **int** numLines) { ... }
58. }
59. **public** **class** Particle
60. {
61. // Class member fields here
62. **public** Particle(**double**[] position, **double** fitness,
63. **double**[] velocity, **double**[] bestPosition,
64. **double** bestFitness) { ... }
65. **public** **override** **string** ToString() { ... }
66. }
67. } // ns

In addition to the class that contains the Main method, the program has three other classes. Class NeuralNetwork encapsulates a fully connected feed-forward neural network. All of the core program logic is contained in this class. Class Helpers contains six utility routines. Class Particle defines a particle object that’s used by the particle swarm optimization algorithm in the Train method of the NeuralNetwork class. One characteristic of classification programs is that there are many possible program structures; the organization presented here is just one possibility.

## Generating the Raw Data File

In most classification scenarios you will already have a set of raw data, but for this article I created dummy raw data using method MakeData. Here’s the process in pseudo-code:

create 43 arbitrary weights between -2.0 and +2.0

create a 4-5-3 neural network

load weights into the neural network

open a result file for writing

loop 100 times

  generate four random inputs x0, x1, x2, x3 between 1.0 and 9.0

  compute the y0, y1, y2 neural outputs for the input values

  determine largest of y0, y1, y2

  if y0 is largest write x0, x1, x2, x3, red

  else if y1 is largest write x0, x1, x2, x3, green

  else if y2 is largest write x0, x1, x2, x3, blue

end loop

close result file

The goal here is to get some data that is definitely classifiable with 100 percent accuracy, rather than random data where it’s not clear how effective any classification approach would be. In other words, I use a neural network to create raw data and then I start over and use a neural network to attempt to classify that data.

## Creating the Training and Testing Matrices

When performing classification analysis with a set of existing data, one common approach, called holdout validation, is to split the data into a larger data set (often 80 percent) for training the neural network and a smaller data set (20 percent) for testing the model. Training means finding the neural network weights and biases that minimize some error value. Testing means evaluating the neural network with the best weights found during training, using some measure of accuracy. Here, method MakeTrainAndTest creates training and testing matrices and also normalizes the numeric input data and encodes the categorical output data. In pseudo-code, the method works like this:

determine how many rows to split data into

create a matrix to hold all data

loop

  read a line of data

  parse each field

  foreach numeric input

    normalize input to between -1.0 and +1.0

  end foreach

  encode categorical output using 1-of-N

  place normalized and encoded data in matrix

end loop

shuffle matrix

create train and test matrices

transfer first 80% rows of matrix to train, remaining rows to test

The method signature is:

1. **static** **void** MakeTrainAndTest(**string** file, **out** **double**[][] trainMatrix,
2. **out** **double**[][] testMatrix)

The parameter called file is the name of the raw data file to create. Parameters trainMatrix and testMatrix are output parameters where the results are placed. The method begins with:

1. **int** numLines = 0;
2. FileStream ifs = **new** FileStream(file, FileMode.Open);
3. StreamReader sr = **new** StreamReader(ifs);
4. **while** (sr.ReadLine() != **null**)
5. ++numLines;
6. sr.Close(); ifs.Close();
7. **int** numTrain = (**int**)(0.80 \* numLines);
8. **int** numTest = numLines - numTrain;

This code counts the number of lines in the raw data file and then computes how many lines constitute 80 percent and 20 percent of the data. Here, the percentages are hard-coded; you may want to parameterize them. Next, a matrix that will hold all data is allocated:

1. **double**[][] allData = **new** **double**[numLines][];
2. **for** (**int** i = 0; i < allData.Length; ++i)
3. allData[i] = **new** **double**[7];

There are seven columns: one column for each of the four numeric inputs and three columns for the 1-of-N encoded value of the categorical color variable. Recall that for this example, the goal is to predict color, which can be one of three categorical values: red, green or blue. Encoding using the 1-of-N technique in this situation means encoding red as (1.0, 0.0, 0.0), green as (0.0, 1.0, 0.0) and blue as (0.0, 0.0, 1.0). Categorical data must be encoded numerically because neural networks deal directly only with numeric values. It turns out that encoding color using a simple approach such as 1 for red, 2 for green and 3 for blue is a bad idea. The explanation of why this is bad is a bit lengthy and outside the scope of this article.

An exception to the 1-of-N encoding guideline for categorical output data is that when there are only two possible values, such as “male” or “female,” you can use 1-of-(N-1) encoding where you have a single numeric output value so that, for example, 0.0 means male and 1.0 means female.

The encoding is performed by this code:

1. tokens = line.Split(' ');
2. allData[row][0] = **double**.Parse(tokens[0]);
3. allData[row][1] = **double**.Parse(tokens[1]);
4. allData[row][2] = **double**.Parse(tokens[2]);
5. allData[row][3] = **double**.Parse(tokens[3]);
6. **for** (**int** i = 0; i < 4; ++i)
7. allData[row][i] = 0.25 \* allData[row][i] - 1.25;
8. **if** (tokens[4] == "red") {
9. allData[row][4] = 1.0;
10. allData[row][5] = 0.0;
11. allData[row][6] = 0.0; }
12. **else** **if** (tokens[4] == "green") {
13. allData[row][4] = 0.0;
14. allData[row][5] = 1.0;
15. allData[row][6] = 0.0; }
16. **else** **if** (tokens[4] == "blue") {
17. allData[row][4] = 0.0;
18. allData[row][5] = 0.0;
19. allData[row][6] = 1.0; }

Recall that a line of raw data looks like this:

8.0 5.0 9.0 5.0 green

The five fields are parsed using String.Split. Experience has shown that in most situations, better results are obtained when numeric input is scaled to values between -1.0 and +1.0. Each of the first four numeric inputs is converted by multiplying by 0.25 and subtracting 1.25. Recall that the numeric inputs in the dummy data file are all between 1.0 and 9.0. In a real classification problem, you would have to scan the raw data and determine the min and max values. We want -1.0 to correspond to 1.0 and +1.0 to correspond to 9.0. Doing a linear transformation means finding the slope (0.25 here) and intercept (-1.25). These values can be computed as:

slope = 2 / (max value - min value) = 2 / (9.0 - 1.0) = 0.25

intercept = 1.0 - (slope \* max value) = 1 - (0.25 \* 9.0) = -1.25

There are many alternatives to performing a linear transform on numeric input values, but the approach presented here is simple and a good starting point in most situations.

After the four numeric input values have been transformed, the color value from the raw data file is encoded using 1-of-N encoding. When all values in the raw data file have been computed and placed in the allData matrix, that matrix has its rows rearranged randomly using the ShuffleRows utility method in the Helpers class. After the order of the rows in allData has been shuffled, space for matrices trainMatrix and testMatrix is allocated, then the first numTrain rows of allData are copied into trainMatrix and the remaining numTest rows of allData are copied into testMatrix.

An important design alternative to the train-test approach is to split raw data into three sets: train, validate and test. The idea is to use the train data to determine the best set of neural network weights in conjunction with the validate data, which is used to know when to stop training. There are other approaches as well, collectively called cross-validation techniques.

## The Softmax Activation Function

When performing classification using a neural network where the output variable is categorical, there’s a rather tricky relationship among the neural network output activation function, computing error during training and computing the predictive accuracy of the neural network. When categorical output data (such as color with values red, green or blue) is encoded using 1-of-N encoding—for example, (1.0 0.0 0.0) for red—you want the neural network to emit three numeric values so you can determine an error when training the network. You don’t, however, want three arbitrary numeric values to be emitted, because then it’s not entirely clear how to compute an error term. But suppose the neural network emits three numeric values that are all between 0.0 and 1.0 and that sum to 1.0. Then the emitted values can be interpreted as probabilities, which, as it turns out, makes it easy to compute an error term when training and to compute accuracy when testing. The Softmax activation function emits output values in this form. The Softmax function accepts neural hidden-to-output sums and returns the final neural output values; it can be implemented like this:

1. **private** **static** **double**[] Softmax(**double**[] hoSums)
2. {
3. **double** max = hoSums[0];
4. **for** (**int** i = 0; i < hoSums.Length; ++i)
5. **if** (hoSums[i] > max) max = hoSums[i];
6. **double** scale = 0.0;
7. **for** (**int** i = 0; i < hoSums.Length; ++i)
8. scale += Math.Exp(hoSums[i] - max);
9. **double**[] result = **new** **double**[hoSums.Length];
10. **for** (**int** i = 0; i < hoSums.Length; ++i)
11. result[i] = Math.Exp(hoSums[i] - max) / scale;
12. **return** result;
13. }

In principle, the Softmax function computes a scaling factor by taking Exp of each hidden-to-output sum, summing them and then dividing the Exp of each value by the scaling factor. For example, suppose three hidden-to-output sums are (2.0, -1.0, 4.0). The scaling factor would be Exp(2.0) + Exp(-1.0) + Exp(4.0) = 7.39 + 0.37 + 54.60 = 62.36. Then the Softmax output values would be Exp(2.0)/62.36, Exp(-1.0)/62.36, Exp(4.0)/62.36) = (0.12, 0.01, 0.87). Notice that the final outputs are all between 0.0 and 1.0 and do in fact sum to 1.0, and further that the largest hidden-to-output sum (4.0) has the largest output/probability (0.87), and the relationship is similar for the second- and third-largest values.

Unfortunately, a naive implementation of Softmax can often fail because the Exp function becomes very large very quickly and can produce arithmetic overflow. The preceding implementation uses the fact that Exp(a - b) = Exp(a) / Exp(b) to compute output in a way that avoids overflow. If you trace the execution of the implementation using (2.0, -1.0, 4.0) as inputs, you’ll get the same (0.12, 0.01, 0.87) outputs as explained in the preceding section.

## Cross-Entropy Error

The essence of training a neural network is to find the set of weights that produces the smallest error for the data in the training set. For example, suppose a row of normalized, encoded training data is (0.75 -0.25 0.25 -0.50 0.00 1.00 0.00). Recall that the first four values are the normalized inputs and the last three values represent green in 1-of-N encoding. Now suppose the inputs are fed through the neural network, which has been loaded with some set of weights, and the Softmax output is (0.20 0.70 0.10). A traditional approach to computing error for this test vector is to use sum of squared differences, which in this case would be (0.00 - 0.20)^2 + (1.00 - 0.70)^2 + (0.00 - 0.10)^2 = 0.14. But suppose the Softmax output was (0.30 0.70 0.00). Both this vector and the previous vector predict that the output is green because the probability of green is 0.70. However, the sum of squared deviations for this second vector is 0.18, which is different from the first error term. Although sum of squared deviations can be used to compute training error, some research results suggest that using an alternative measure called cross-­entropy error is preferable.

In principle, the cross-entropy error for a given neural network output vector v and a test output vector t is computed by determining the negative sum of the product of each v vector component and the corresponding t vector component. As usual, this is best understood by example. If a test vector is (0.75 -0.25 0.25 -0.50 0.00 1.00 0.00) and the corresponding Softmax neural network output is (0.20 0.70 0.10), then the cross-entropy error is -1 \* (0.00 \* Log(0.20)) + (1.00 \* Log(0.70)) + (0.00 \* Log(0.10)) = -1 \* (0 + -0.15 + 0) = 0.15. Notice that all but one of the terms in the sum will always be zero when 1-of-N encoding is used. The cross-entropy error for the entire training set can be computed as the sum of the cross-entropy of all test vectors, or the average cross-entropy of each test vector. An implementation of cross-entropy error is listed in **Figure 4**.

**Figure 4 Cross-Entropy Error**

1. **private** **double** CrossEntropy(**double**[][] trainData,
2. **double**[] weights)
3. {
4. **this**.SetWeights(weights);
5. **double** sce = 0.0; // sum of cross entropies
6. **for** (**int** i = 0; i < trainData.Length; ++i)
7. {
8. **double**[] currInputs = **new** **double**[4];
9. currInputs[0] = trainData[i][0];
10. currInputs[1] = trainData[i][1];
11. currInputs[2] = trainData[i][2];
12. currInputs[3] = trainData[i][3];
13. **double**[] currExpected = **new** **double**[3];
14. currExpected[0] = trainData[i][4];
15. currExpected[1] = trainData[i][5];
16. currExpected[2] = trainData[i][6];
17. **double**[] currOutputs = **this**.ComputeOutputs(currInputs);
18. **double** currSum = 0.0;
19. **for** (**int** j = 0; j < currOutputs.Length; ++j)
20. {
21. **if** (currExpected[j] != 0.0)
22. currSum += currExpected[j] \* Math.Log(currOutputs[j]);
23. }
24. sce += currSum; // accumulate
25. }
26. **return** -sce;
27. }

## Training the Neural Network

There are many ways to train a neural network classifier to find the set of weights that best matches the training data (or, equivalently, yields the smallest cross-entropy error). At a high level of abstraction, training a neural network looks like this:

create an empty neural network

loop

  generate a candidate set of weights

  load weights into neural network

  foreach training vector

    compute the neural output

    compute cross-entropy error

    accumulate total error

  end foreach

  if current weights are best found so far

    save current weights

  end if

until some stopping condition

return best weights found

By far, the most common technique used to train neural networks is called back-propagation. This technique is the subject of a large number of research articles—so many, in fact, that if you’re new to the field of neural network classification, you could easily be led to believe that back-propagation is the only technique used for training. Estimating the best set of weights for a neural network is a numeric minimization problem. Two common alternatives to using back-propagation are using a real-valued genetic algorithm (also called an evolutionary optimization algorithm), and using particle swarm optimization. Each estimation technique has strengths and weaknesses. The program shown in **Figure 1**uses particle swarm optimization. I describe evolutionary optimization algorithms in the June 2012 issue of MSDN Magazine ([msdn.microsoft.com/magazine/jj133825](http://msdn.microsoft.com/magazine/jj133825)) and particle swarm optimization in the August 2011 issue ([msdn.microsoft.com/magazine/hh335067](http://msdn.microsoft.com/magazine/hh335067)).

There are many techniques to determine when to stop training a neural network. Although you could simply let a training algorithm run until the cross-entropy error is very close to zero, indicating a near-perfect fit, the danger is that the resulting weights will over-fit the training data and the weights will create a neural network that classifies poorly for any data that isn’t in the training set. Also, training until there is no change in the cross-entropy error can easily lead to model over-fitting. A simple approach, and one used by the example program, is to limit training to a fixed number of iterations. In most cases, a much better tactic to avoid over-fitting is to split the source data set into train-validate-test sets. Typically these three data sets use 60 percent, 20 percent and 20 percent of the source data, respectively. The technique computes error on the training set as previously described, but after each iteration in the main loop the technique computes the cross-entropy error on the validation data set. When the cross-entropy error on the validation set starts to show a consistent increase in error, there’s a good chance the training algorithm has begun to over-fit the data and the training should halt. There are many other stopping techniques possible.

## Evaluating the Neural Network Classifier

After the neural network classifier has been trained and has produced a set of best weights and biases, the next step is to determine how accurate the resulting model (where model means the neural network with the set of best weights) is on the test data. Although measures such as sum of squared deviations or cross-entropy error can be used, a reasonable measure of accuracy is simply the percentage of correct predictions made by the model. Once again, there are several approaches for measuring accuracy, but a simple technique is to use what’s called the winner-takes-all approach. And, as usual, the technique is best explained by example. Suppose a test vector is (-1.00 1.00 0.25 -0.50 1.00 0.00 0.00), as shown in the first set of prediction data in **Figure 1**. With the set of best weights, the neural network generates a predicted Softmax output of (0.9 0.1 0.0). If each output is interpreted as a probability, then the highest probability is 0.9 and the predicted output can be thought of as (1 0 0), so the model makes the correct prediction. Put another way, the winner-takes-all technique determines the neural network output component with the largest value, makes that component a 1 and all other components 0, and compares that result with the actual data in the training vector. In the third set of accuracy analysis data in **Figure 1**, the test vector is (-0.50 0.75 0.25 -0.75 0.0 0.0 1.0). The actual output data is (0.0 0.0 1.0), which corresponds to blue. The predicted output is (0.3 0.6 0.1). The largest component is the 0.6, so the model prediction is (0 1 0), which corresponds to green. The model made an incorrect prediction.

## Wrapping Up

Classification with neural networks is an important, fascinating and complex topic. The example presented here should give you a solid basis for experimenting with neural network classification. However, this article describes only one very specific neural network classification scenario—numeric input variables with a categorical output variable—and is just a starting point. Other scenarios require slightly different techniques. For example, if the input data contains a categorical variable, you might expect that it would be encoded using 1-of-N encoding just like a categorical output variable. In this case, however, the input data should be encoded using the 1-of-(N-1) technique. If you want to learn more about classification using neural networks, I recommend the set of seven FAQs on neural networks available at faqs.org. The links to these FAQs tend to move around but you should be able to find them easily with an Internet search.