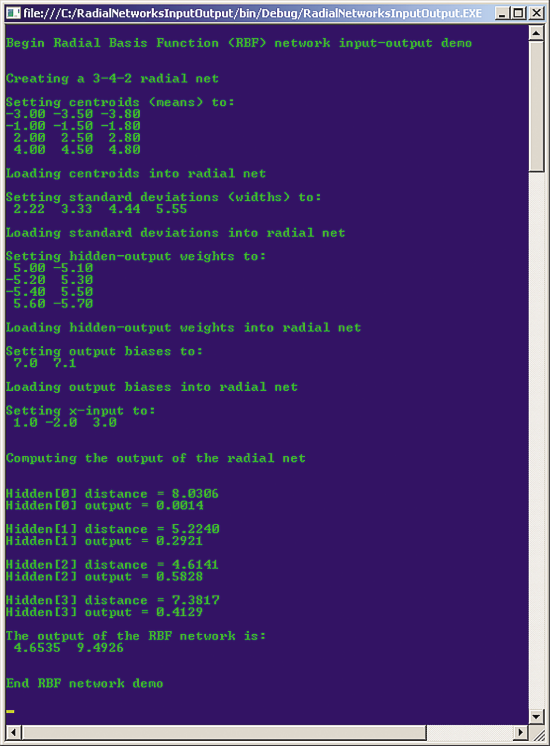
Radial basis function (RBF) networks are software systems that have certain similarities to neural networks. An RBF network accepts one or more numeric input values, such as (1.0, -2.0, 3.0), and generates one or more numeric output values, such as (4.6535, 9.4926). RBF networks (sometimes called radial nets) can be used to classify data and make predictions. For example, an RBF network could be used to predict the scores of two football teams that are scheduled to play each other, based on historical data such as each team’s current winning percentage, home field advantage (-1.0 or +1.0) and so on. Or an RBF network could be used to classify a hospital patient’s risk of cancer (low = 0, high = 1) based on the values of medical test results and other factors such as age and sex.

In my opinion, RBF networks are among the most fascinating of all machine-learning techniques. But even though there are many research papers that explain the complex mathematics behind these networks, there are very few resources that explain them from a programmer’s point of view. This article will describe exactly what RBF networks are, explain how they compute their outputs and illustrate with a complete RBF network input-output demo program.

The best way for you to see where this article is headed is to take a look at the demo program in **Figure 1**. The demo creates an RBF network, configures it, sends an input vector with values of (1.0, -2.0, 3.0), and displays the output vector of (4.6535, 9.4926). The demo begins by creating a 3-4-2 RBF network. The 3 indicates an input to the radial net has three numeric values. The 2 indicates there are two numeric outputs. The 4 indicates the radial net has four hidden, internal processing nodes.

  
**Figure 1 Radial Basis Function Network Input-Output Demo**

The radial net requires four sets of configuration infor­mation, usually referred to as the parameters of the system. The first parameter is a set of so-called centroids. Centroids are sometimes called means. In the demo, the centroids are (-3.0, -3.5, -3.8), (-1.0, -1.5, 1.8), (2.0, 2.5, 2.8) and (4.0, 4.5, 4.8). Notice there’s one centroid for every hidden node and that each centroid has the same number of values as an input vector. Because the purpose of the demo is only to explain how RBF networks compute their output, rather than to solve a realistic problem, the number of hidden nodes (four) is artificially small, and the values of the four centroids are arbitrary.

The second set of parameters for the demo radial net is four standard deviations. Standard deviations are sometimes called RBF widths. The values in the demo are 2.22, 3.33, 4.44 and 5.55. There’s one standard deviation for every hidden-node processing unit. Again, the values used in the demo are dummy values and don’t correspond to a real problem.

The third set of RBF network parameters is the weights. Notice in **Figure 1** that there are eight weights. If an RBF net has j hidden nodes and k output nodes, there will be j \* k weights. Here, the 4 \* 2 = 8 weights are 5.0, -5.1, -5.2, 5.3, -5.4, 5.5, 5.6 and -5.7. Like the other sets of parameters, these values are purely arbitrary and are for demonstration purposes only.

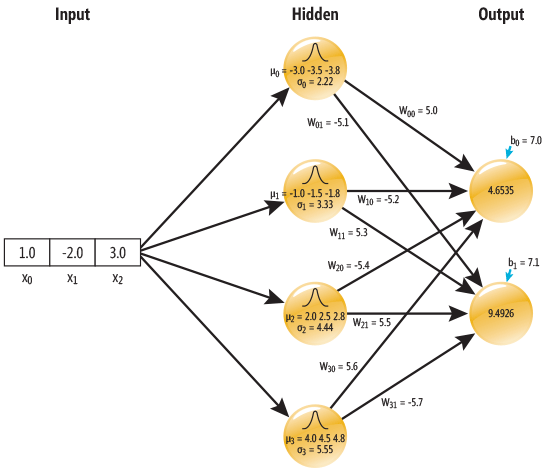
The fourth set of radial net parameters in the demo is two bias values. The number of bias values in an RBF network is equal to the number of output values, two in this case. The two dummy bias values are 7.0 and 7.1.

After loading the four sets of parameter values into the RBF network, an input of (1.0, -2.0, 3.0) is fed to the network. Using the parameter values, an output of (4.6535, 9.4926) is computed. The demo program displays some intermediate values of the computation—a distance and output for each hidden node—but these intermediate values are shown only to help you understand the RBF input-output mechanism and aren’t normally displayed.

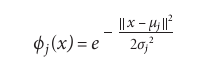
This article assumes you have at least intermediate-level programming skills with a C-family language, but doesn’t assume you know anything about radial basis function networks. The demo program is coded using C#, but you should have no trouble refactoring my code to another language such as Python or Windows PowerShell. I’ve removed most normal error checking to keep the main ideas clear. All the key code for the demo program is presented in this article, but I’ve omitted some of the helper display methods. The complete source code for the demo is available at[archive.msdn.microsoft.com/mag201310TestRun](http://archive.msdn.microsoft.com/mag201310TestRun).

## The RBF Input-Output Mechanism

In order to understand the RBF input-output mechanism, take a look at the diagram in **Figure 2**. The values in the diagram corre­spond to the ones in the demo program. Processing occurs in two major steps. First, an input vector is sent to each hidden node and each hidden node independently computes an intermediate output value. Second, the intermediate hidden-node output values are combined to produce the final system output values.

  
**Figure 2 Radial Basis Function Network Architecture**

The diagram in **Figure 2** uses zero-based indexing for all objects. Expressed mathematically, the output of a hidden node j is:



This equation is an example of what’s called the Gaussian function and when graphed has a characteristic bell-shaped curve. The bell-shaped icon at the top of each hidden node indicates that the hidden-node outputs are computed using a Gaussian function.

The equation is simpler than it may initially appear. The Greek letter phi represents the output of a hidden node, j in this case. The x represents the input. Here x is a vector (1.0, -2.0, 3.0) rather than a single scalar value. Lowercase e is Euler’s constant (2.71828 ...) and e raised to some value corresponds to the Exp function available in most programming languages. The Greek letter mu is the jth centroid vector. The pair of double-bar symbols, when applied to the difference between two vectors, is equivalent to a shorthand notation for the Euclidean geometry distance function, which is the square root of the sum of the squared differences between the components of the two vectors. The Greek letter sigma represents the standard deviation of the jth hidden node.

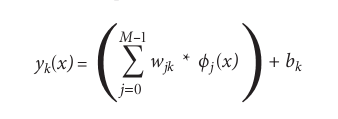
I’ll demonstrate how the intermediate output of the bottommost hidden node [3] is computed using the values of the demo program. The input x is (1.0, -2.0, 3.0). The centroid, mu, is (4.0, 4.5, 4.8). The standard deviation, sigma, is 5.55. Recall that these values are purely arbitrary and for demonstration purposes, and do not correspond to a real problem.

The distance between x and mu is Sqrt( (1.0 - 4.0)^2 + (-2.0 - 4.5)^2 + (3.0 - 4.8)^2 ) = Sqrt(9.00 + 42.25 + 3.24) = 7.3817 as shown in **Figure 1**. The distance squared is 54.49. Notice that the squaring operation undoes the square root operation of the distance formula, which means that the equation could’ve been simplified a bit.

Putting all the values together, the output for the hidden node 3 is Exp(-54.49 / (2 \* (5.55)^2)) = Exp(-0.88451) = 0.4129, which is the value shown in **Figure 1**.

The output values for hidden nodes 0, 1 and 2 are calculated in the same way and are 0.0014, 0.2921 and 0.5828, respectively.

Now I’ll show how the hidden-node outputs are combined to yield the final RBF network outputs. An RBF network’s output is the weighted sum of products of all hidden-node outputs times an associated weight, plus a final bias value. Expressed as an equation, the value of output node k is:



Here, k is the index of an output node (0, 1 in the demo); j is the index of a hidden node (0, 1, 2, 3); and M is the number of hidden nodes. The term wjk is the weight from hidden node j to output node k. The term bk is the bias value associated with output k.

For example, if h0, h1, h2, and h3 (using the easier-to-type letter h instead of Greek letter phi) are the outputs of the hidden nodes 0 through 3, then final output 0 is computed as (w00 \* h0) + (w10 \* h1) + (w20 \* h2) + (w30 \* h3) + b0 = (5.0 \* 0.0014) + (-5.2 \* 0.2921) + (-5.4 \* 0.5828) + (5.6 \* 0.4129) + 7.0 = 0.0070 + -1.5189 + -3.1471 + 2.3122 + 7.0 = 4.6535 (rounded), as shown in **Figure 1**.

## Demo Program Structure

The overall structure of the demo program, with most WriteLine statements removed and some minor edits, is presented in **Figure 3**. The RBF network functionality is contained in class RadialNet. Class Helpers contains three display methods.

**Figure 3 RBF Network Demo Program Structure**

1. **using** System;
2. **namespace** RadialNetworksInputOutput
3. {
4. **class** RadialNetIOProgram
5. {
6. **static** **void** Main(**string**[] args)
7. {
8. Console.WriteLine("\nBegin Radial Basis Function (RBF) network demo\n");
9. **int** numInput = 3;
10. **int** numHidden = 4;
11. **int** numOutput = 2;
12. Console.WriteLine("\nCreating a 3-4-2 radial net");
13. RadialNet rn = **new** RadialNet(numInput, numHidden, numOutput);
14. **double**[][] centroids = **new** **double**[4][];
15. centroids[0] = **new** **double**[] { -3.0, -3.5, -3.8 };
16. centroids[1] = **new** **double**[] { -1.0, -1.5, -1.8 };
17. centroids[2] = **new** **double**[] { 2.0, 2.5, 2.8 };
18. centroids[3] = **new** **double**[] { 4.0, 4.5, 4.8 };
19. rn.SetCentroids(centroids);
20. **double**[] stdDevs = **new** **double**[4] { 2.22, 3.33, 4.44, 5.55 };
21. rn.SetStdDevs(stdDevs);
22. **double**[][] hoWeights = **new** **double**[4][];
23. hoWeights[0] = **new** **double**[2] { 5.0, -5.1 };
24. hoWeights[1] = **new** **double**[2] { -5.2, 5.3 };
25. hoWeights[2] = **new** **double**[2] { -5.4, 5.5 };
26. hoWeights[3] = **new** **double**[2] { 5.6, -5.7 };
27. rn.SetWeights(hoWeights);
28. **double**[] oBiases = **new** **double**[2] { 7.0, 7.1 };
29. rn.SetBiases(oBiases);
30. **double**[] xValues = **new** **double**[3] { 1.0, -2.0, 3.0 };
31. Console.WriteLine("\nSetting x-input to:");
32. Helpers.ShowVector(xValues, 1, 4, **true**);
33. Console.WriteLine("\nComputing the output of the radial net\n");
34. **double**[] yValues = rn.ComputeOutputs(xValues);
35. Console.WriteLine("\nThe output of the RBF network is:");
36. Helpers.ShowVector(yValues, 4, 4, **true**);
37. Console.WriteLine("\nEnd RBF network demo\n");
38. Console.ReadLine();
39. } // Main
40. } // Program
41. **public** **class** RadialNet { ... }
42. **public** **class** Helpers { ... }
43. } // ns

To create the demo program, I launched Visual Studio 2012. The demo has no significant .NET dependencies, so any version of Visual Studio should work. I created a new C# console appli­cation project named RadialNetworksInputOutput. After the template code loaded, I renamed file Program.cs to Radial NetIOProgram.cs and Visual Studio automatically renamed class Program accordingly. I deleted all unnecessary using statements at the top of the source code.

You should be able to understand the statements in the Main method without difficulty. The statement that instantiates the RBF network object is:

1. RadialNet rn = **new** RadialNet(numInput, numHidden, numOutput);

The four statements that load RBF network parameter values are:

1. rn.SetCentroids(centroids);
2. rn.SetStdDevs(stdDevs);
3. rn.SetWeights(hoWeights);
4. rn.SetBiases(oBiases);

The statement that loads the input and computes and returns the RBF network output is:

1. **double**[] yValues = rn.ComputeOutputs(xValues);

## The RadialNet Class

The definition of the RBF network class begins with:

1. **public** **class** RadialNet
2. {
3. **private** **int** numInput;
4. **private** **int** numHidden;
5. **private** **int** numOutput;
6. **private** **double**[] inputs;
7. **private** **double**[][] centroids; // Aka means
8. **private** **double**[] stdDevs; // Aka widths
9. **private** **double**[][] hoWeights;
10. **private** **double**[] oBiases;
11. **private** **double**[] outputs;
12. ...

The purpose of each of these class members should be clear, based on the explanation of how an RBF network computes its output. The weights are stored as an array-of-arrays-style matrix where the first index indicates the hidden node and the second index indicates the output node. The C# language, unlike most languages, supports a true matrix data type, and you may wish to use it instead of an array of arrays for the weights matrix.

The RadialNet constructor is defined as:

1. **public** RadialNet(**int** numInput, **int** numHidden, **int** numOutput)
2. {
3. **this**.numInput = numInput;
4. **this**.numHidden = numHidden;
5. **this**.numOutput = numOutput;
6. **this**.inputs = **new** **double**[numInput];
7. **this**.centroids = MakeMatrix(numHidden, numInput);
8. **this**.stdDevs = **new** **double**[numHidden];
9. **this**.hoWeights = MakeMatrix(numHidden, numOutput);
10. **this**.oBiases = **new** **double**[numOutput];
11. **this**.outputs = **new** **double**[numOutput];
12. }

If you refer to the diagram in **Figure 2**, you’ll see how the size of each class array and matrix is related to numInput, numHidden and numOutput. A static utility method, MakeMatrix, is called by the constructor just to keep the code a bit cleaner. Method MakeMatrix is defined as:

1. **private** **static** **double**[][] MakeMatrix(**int** rows, **int** cols)
2. {
3. **double**[][] result = **new** **double**[rows][];
4. **for** (**int** r = 0; r < rows; ++r)
5. result[r] = **new** **double**[cols];
6. **return** result;
7. }

Class RadialNet has four methods to set the values of the centroids, standard deviations, weights and biases. An alternative design is to overload the constructor to accept all RBF parameter values, but I prefer separate set methods in most situations. Method SetCentroids is:

1. **public** **void** SetCentroids(**double**[][] centroids)
2. {
3. **if** (centroids.Length != numHidden)
4. **throw** **new** Exception("Bad number of centroids");
5. **if** (centroids[0].Length != numInput)
6. **throw** **new** Exception("Bad centroid size");
7. **for** (**int** i = 0; i < numHidden; ++i)
8. **for** (**int** j = 0; j < numInput; ++j)
9. **this**.centroids[i][j] = centroids[i][j];
10. }

Method SetStdDevs is:

1. **public** **void** SetStdDevs(**double**[] stdDevs)
2. {
3. **if** (stdDevs.Length != numHidden)
4. **throw** **new** Exception("Bad number of stdDevs");
5. Array.Copy(stdDevs, **this**.stdDevs, stdDevs.Length);
6. }

Method SetWeights is:

1. **public** **void** SetWeights(**double**[][] hoWeights)
2. {
3. **if** (hoWeights.Length != numHidden)
4. **throw** **new** Exception("Bad number of weights");
5. **if** (hoWeights[0].Length != numOutput)
6. **throw** **new** Exception("Bad number of weights");
7. **for** (**int** i = 0; i < numHidden; ++i)
8. **for** (**int** j = 0; j < numOutput; ++j)
9. **this**.hoWeights[i][j] = hoWeights[i][j];
10. }

Method SetBiases is:

1. **public** **void** SetBiases(**double**[] oBiases)
2. {
3. **if** (oBiases.Length != numOutput)
4. **throw** **new** Exception("Bad number of hoBiases");
5. Array.Copy(oBiases, **this**.oBiases, oBiases.Length);
6. }

## The ComputeOutputs Method

The heart of the RadialNet class is method ComputeOutputs. The method’s definition begins:

1. **public** **double**[] ComputeOutputs(**double**[] xValues)
2. {
3. Array.Copy(xValues, inputs, xValues.Length);
4. **double**[] hOutputs = **new** **double**[numHidden];
5. ...

Here, x-input values are simply copied into member array inputs. Because method ComputeOutputs uses but doesn’t change the x-input values, member array inputs isn’t really needed. However, in some cases you might want to perform some preliminary processing of the x-input. And I feel that using an explicit class inputs array is a cleaner design and worth the extra work of copying in values. The local hOutputs array holds the outputs of each hidden node. An alternative design is to define hOutputs as a class member.

Next, ComputeOutputs computes the outputs for each hidden node:

1. **for** (**int** j = 0; j < numHidden; ++j)
2. {
3. **double** d = Distance(inputs, centroids[j]);
4. // Display distance here if you wish
5. **double** r = -1.0 \* (d \* d) / (2 \* stdDevs[j] \* stdDevs[j]);
6. **double** g = Math.Exp(r);
7. // Display hidden output here if you wish
8. hOutputs[j] = g;
9. }
10. ...

The distance is computed by a static utility method, Distance, which is defined:

1. **public** **static** **double** Distance(**double**[] x, **double**[] c)
2. {
3. **double** sum = 0.0;
4. **for** (**int** i = 0; i < x.Length; ++i)
5. sum += (x[i] - c[i]) \* (x[i] - c[i]);
6. **return** Math.Sqrt(sum);
7. }

Notice that Distance performs a square root operation and that this value is stored into variable d, which is then squared. An alternative is to define a method DistanceSquared, which is the same as Distance except for the call to the square root function, and then not square the value of d. Although this approach is more efficient, it makes the code out of sync with the standard math definitions used in RBF literature.

The code in ComputeOutputs that calculates the final outputs is:

1. **for** (**int** k = 0; k < numOutput; ++k)
2. outputs[k] = 0.0;
3. **for** (**int** k = 0; k < numOutput; ++k)
4. **for** (**int** j = 0; j < numHidden; ++j)
5. outputs[k] += (hOutputs[j] \* hoWeights[j][k]);
6. **for** (**int** k = 0; k < numOutput; ++k)
7. outputs[k] += oBiases[k];
8. ...

Method ComputeOutputs finishes up by copying the values in the member outputs array to a return value:

1. ...
2. **double**[] result = **new** **double**[numOutput];
3. Array.Copy(outputs, result, outputs.Length);
4. **return** result;
5. }

An alternative design is to define ComputeOutputs using a void return type, and define a separate GetOutputs method.

## Wrapping Up

The code and explanation presented in this article should get you started if you want to explore radial basis function networks. There are many different types of RBF networks. The RBF net in this article uses a Gaussian function to compute hidden-node output. RBF networks can use many other functions, with names such as multi-quadratic and thin-plate spline. The particular function used by an RBF network is called the kernel of the network.

You may be wondering exactly where the rather exotic architecture of RBF networks comes from. RBF networks were a result of academic research into the theory of function approximation. It can be proved that, with a few assumptions and loosely speaking, RBF networks can replicate any mathematical function. This means in theory at least, RBF networks can be used to make predictions on data that follows any underlying model. Neural networks share this universal-function characteristic. In my opinion, it’s not clear whether RBF networks are more or less effective than neural networks, or roughly the same, when it comes to machine-learning classification and prediction. However, there’s good evidence that RBF networks can be trained much more quickly than neural networks, and unlike neural networks, which typically require very large amounts of training data, RBF networks can work well even with small amounts of training data.

In order to use an RBF network to make predictions for a realistic problem, the network must be trained. This means using existing historical data and finding the best set of centroids, standard deviations, weights and bias values—that is, the set of parameter values that generates RBF network outputs that most closely match the historical data. Training an RBF network also involves finding the optimal number of hidden nodes. Training RBF networks is an interesting problem that will be explained in a future article