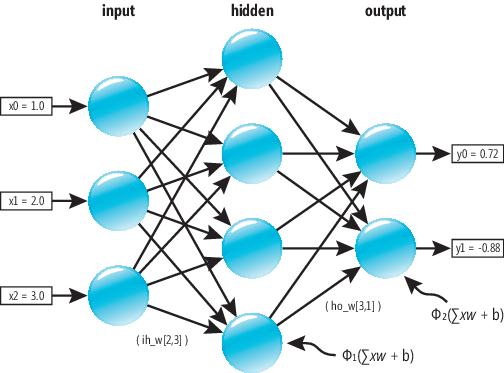
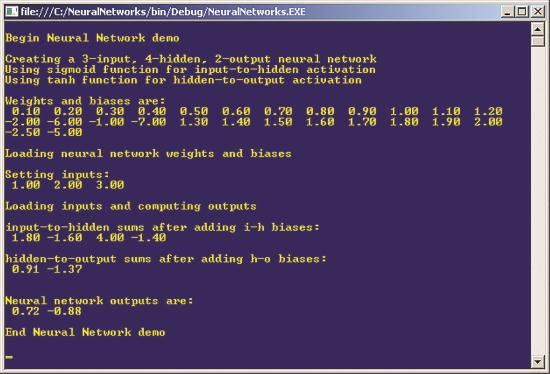
An artificial neural network (usually just called a neural network) is an abstraction loosely modeled on biological neurons and synapses. Although neural networks have been studied for decades, many neural network code implementations on the Internet are not, in my opinion, explained very well. In this month’s column, I’ll explain what artificial neural networks are and present C# code that implements a neural network.

The best way to see where I’m headed is to take a look at **Figure 1** and **Figure 2**. One way of thinking about neural networks is to consider them numerical input-output mechanisms. The neural network in **Figure 1** has three inputs labeled x0, x1 and x2, with values 1.0, 2.0 and 3.0, respectively. The neural network has two outputs labeled y0 and y1, with values 0.72 and -0.88, respectively. The neural network in **Figure 1** has one layer of so-called hidden neurons and can be described as a three-layer, fully connected, feedforward network with three inputs, two outputs and four hidden neurons. Unfortunately, neural network terminology varies quite a bit. In this article, I’ll generally—but not always—use the terminology described in the excellent neural network FAQ at [bit.ly/wfikTI](http://bit.ly/wfikTI).

  
**Figure 1 Neural Network Structure**



**Figure 2 Neural Network Demo Program**

**Figure 2** shows the output produced by the demo program presented in this article. The neural network uses both a sigmoid activation function and a tanh activation function. These functions are suggested by the two equations with the Greek letters phi in **Figure 1**. The outputs produced by a neural network depend on the values of a set of numeric weights and biases. In this example, there are a total of 26 weights and biases with values 0.10, 0.20 ... -5.00. After the weight and bias values are loaded into the neural network, the demo program loads the three input values (1.0, 2.0, 3.0) and then performs a series of computations as suggested by the messages about the input-to-hidden sums and the hidden-to-output sums. The demo program concludes by displaying the two output values (0.72, -0.88).

I’ll walk you through the program that produced the output shown in **Figure 2**. This column assumes you have intermediate programming skills but doesn’t assume you know anything about neural networks. The demo program is coded using the C# language but you should have no trouble refactoring the demo code to another language such as Visual Basic .NET or Python. The program presented in this article is essentially a tutorial and a platform for experimentation; it does not directly solve any practical problem, so I’ll explain how you can expand the code to solve meaningful problems. I think you’ll find the information quite interesting, and some of the programming techniques can be valuable additions to your coding skill set.

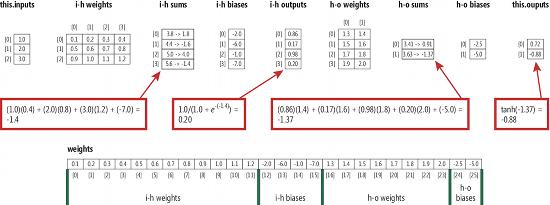
## Modeling a Neural Network

Conceptually, artificial neural networks are modeled on the behavior of real biological neural networks. In**Figure 1** the circles represent neurons where processing occurs and the arrows represent both information flow and numeric values called weights. In many situations, input values are copied directly into input neurons without any weighting and emitted directly without any processing, so the first real action occurs in the hidden layer neurons. Assume that input values 1.0, 2.0 and 3.0 are emitted from the input neurons. If you examine **Figure 1**, you can see an arrow representing a weight value between each of the three input neurons and each of the four hidden neurons. Suppose the three weight arrows shown pointing into the top hidden neuron are named w00, w10 and w20. In this notation the first index represents the index of the source input neuron and the second index represents the index of the destination hidden neuron. Neuron processing occurs in three steps. In the first step, a weighted sum is computed. Suppose w00 = 0.1, w10 = 0.5 and w20 = 0.9. The weighted sum for the top hidden neuron is (1.0)(0.1) + (2.0)(0.5) + (3.0)(0.9) = 3.8. The second processing step is to add a bias value. Suppose the bias value is -2.0; then the adjusted weighted sum becomes 3.8 + (-2.0) = 1.8. The third step is to apply an activation function to the adjusted weighted sum. Suppose the activation function is the sigmoid function defined by 1.0 / (1.0 \* Exp(-x)), where Exp represents the exponential function. The output from the hidden neuron becomes 1.0 / (1.0 \* Exp(-1.8)) = 0.86. This output then becomes part of the weighted sum input into each of the output layer neurons. In **Figure 1**, this three-step process is suggested by the equation with the Greek letter phi: weighted sums (xw) are computed, a bias (b) is added and an activation function (phi) is applied.

After all hidden neuron values have been computed, output layer neuron values are computed in the same way. The activation function used to compute output neuron values can be the same function used when computing the hidden neuron values, or a different activation function can be used. The demo program shown running in **Figure 2** uses the hyperbolic tangent function as the hidden-to-output activation function. After all output layer neuron values have been computed, in most situations these values are not weighted or processed but are simply emitted as the final output values of the neural network.

## Internal Structure

The key to understanding the neural network implementation presented here is to closely examine **Figure 3**, which, at first glance, might appear extremely complicated. But bear with me—the figure is not nearly as complex as it might first appear. **Figure 3** shows a total of eight arrays and two matrices. The first array is labeled this.inputs. This array holds the neural network input values, which are 1.0, 2.0 and 3.0 in this example. Next comes the set of weight values that are used to compute values in the so-called hidden layer. These weights are stored in a 3 x 4 matrix labeled i-h weights where the i-h stands for input-to-hidden. Notice in **Figure 1** that the demo neural network has four hidden neurons. The i-h weights matrix has a number of rows equal to the number of inputs and a number of columns equal to the number of hidden neurons.

[](http://msdn.microsoft.com/en-us/magazine/hh975375.McCaffrey_Figure3_hires_lrg(en-us,MSDN.10).png)  
**Figure 3 Neural Network Internal Structure**

The array labeled i-h sums is a scratch array used for computation. Note that the length of the i-h sums array will always be the same as the number of hidden neurons (four, in this example). Next comes an array labeled i-h biases. Neural network biases are additional weights used to compute hidden and output layer neurons. The length of the i-h biases array will be the same as the length of the i-h sums array, which in turn is the same as the number of hidden neurons.

The array labeled i-h outputs is an intermediate result and the values in this array are used as inputs to the next layer. The i-h sums array has length equal to the number of hidden neurons.

Next comes a matrix labeled h-o weights where the h-o stands for hidden-to-output. Here the h-o weights matrix has size 4 x 2 because there are four hidden neurons and two outputs. The h-o sums array, the h-o biases array and the this.outputs array all have lengths equal to the number of outputs (two, in this example).

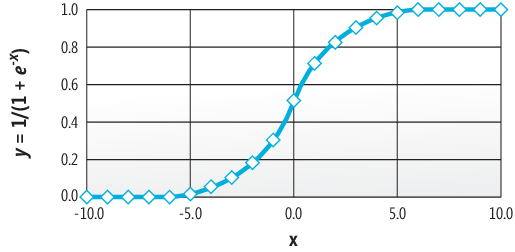
The array labeled weights at the bottom of **Figure 3** holds all the input-to-hidden and hidden-to-output weights and biases. In this example, the length of the weights array is (3 \* 4) + 4 + (4 \* 2) + 2 = 26. In general, if Ni is the number of input values, Nh is the number of hidden neurons and No is the number of outputs, then the length of the weights array will be Nw = (Ni \* Nh) + Nh + (Nh \* No) + No.

## Computing the Outputs

After the eight arrays and two matrices described in the previous section have been created, a neural network can compute its output based on its inputs, weights and biases. The first step is to copy input values into the this.inputs array. The next step is to assign values to the weights array. For the purposes of a demonstration you can use any weight values you like. Next, values in the weights array are copied to the i-h weights matrix, the i-h biases array, the h-o weights matrix and the h-o biases array. **Figure 3**should make this relationship clear.

The values in the i-h sums array are computed in two steps. The first step is to compute the weighted sums by multiplying the values in the inputs array by the values in the appropriate column of the i-h weights matrix. For example, the weighted sum for hidden neuron [3] (where I’m using zero-based indexing) uses each input value and the values in column [3] of the i-h weights matrix: (1.0)(0.4) + (2.0)(0.8) + (3.0)(1.2) = 5.6. The second step when computing i-h sum values is to add each bias value to the current i-h sum value. For example, because i-h biases [3] has value -7.0, the value of i-h sums [3] becomes 5.6 + (-7.0) = -1.4.

After all the values in the i-h sums array have been calculated, the input-to-hidden activation function is applied to those sums to produce the input-to-hidden output values. There are many possible activation functions. The simplest activation function is called the step function, which simply returns 1.0 for any input value greater than zero and returns 0.0 for any input value less than or equal to zero. Another common activation function, and the one used in this article, is the sigmoid function, which is defined as f(x) = 1.0 / (1.0 \* Exp(-x)). The graph of the sigmoid function is shown in **Figure 4**.

  
**Figure 4 The Sigmoid Function**

Notice the sigmoid function returns a value in the range strictly greater than zero and strictly less than one. In this example, if the value for i-h sums [3] after the bias value has been added is -1.4, then the value of i-h outputs [3] becomes 1.0 / (1.0 \* Exp(-(-1.4))) = 0.20.

After all the input-to-hidden output neuron values have been computed, those values serve as the inputs for the hidden-to-output layer neuron computations. These computations work in the same way as the input-to-hidden computations: preliminary weighted sums are calculated, biases are added and then an activation function is applied. In this example I use the hyperbolic tangent function, abbreviated as tanh, for the hidden-to-output activation function. The tanh function is closely related to the sigmoid function. The graph of the tanh function has an S-shaped curve similar to the sigmoid function, but tanh returns a value in the range (-1,1) instead of in the range (0,1).

## Combining Weights and Biases

All of the neural network implementations I’ve seen on the Internet don’t maintain separate weight and bias arrays, but instead combine weights and biases into the weights matrix. How is this possible? Recall that the computation of the value of input-to-hidden neuron [3] resembled (i0 \* w03) + (i1 \* w13) + (i2 \* w23) + b3, where i0 is input value [0], w03 is the weight for input [0] and neuron [3], and b3 is the bias value for hidden neuron [3]. If you create an additional, fake input [4] that has a dummy value of 1.0, and an additional row of weights that hold the bias values, then the previously described computation becomes: (i0 \* w03) + (i1 \* w13) + (i2 \* w23) + (i3 \* w33), where i3 is the dummy 1.0 input value and w33 is the bias. The argument is that this approach simplifies the neural network model. I disagree. In my opinion, combining weights and biases makes a neural network model more difficult to understand and more error-prone to implement. However, apparently I’m the only author who seems to have this opinion, so you should make your own design decision.

## Implementation

I implemented the neural network shown in **Figures 1**, **2** and **3** using Visual Studio 2010. I created a C# console application named NeuralNetworks. In the Solution Explorer window I right-clicked on file Program.cs and renamed it to NeuralNetworksProgram.cs, which also changed the template-generated class name to NeuralNetworksProgram. The overall program structure, with most WriteLine statements removed, is shown in **Figure 5**.

**Figure 5 Neural Network Program Structure**

1. **using** System;
2. **namespace** NeuralNetworks
3. {
4. **class** NeuralNetworksProgram
5. {
6. **static** **void** Main(**string**[] args)
7. {
8. **try**
9. {
10. Console.WriteLine("\nBegin Neural Network demo\n");
11. NeuralNetwork nn = **new** NeuralNetwork(3, 4, 2);
12. **double**[] weights = **new** **double**[] {
13. 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2,
14. -2.0, -6.0, -1.0, -7.0,
15. 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0,
16. -2.5, -5.0 };
17. nn.SetWeights(weights);
18. **double**[] xValues = **new** **double**[] { 1.0, 2.0, 3.0 };
19. **double**[] yValues = nn.ComputeOutputs(xValues);
20. Helpers.ShowVector(yValues);
21. Console.WriteLine("End Neural Network demo\n");
22. }
23. **catch** (Exception ex)
24. {
25. Console.WriteLine("Fatal: " + ex.Message);
26. }
27. }
28. }
29. **class** NeuralNetwork
30. {
31. // Class members here
32. **public** NeuralNetwork(**int** numInput, **int** numHidden, **int** numOutput) { ... }
33. **public** **void** SetWeights(**double**[] weights) { ... }
34. **public** **double**[] ComputeOutputs(**double**[] xValues) { ... }
35. **private** **static** **double** SigmoidFunction(**double** x) { ... }
36. **private** **static** **double** HyperTanFunction(**double** x) { ... }
37. }
38. **public** **class** Helpers
39. {
40. **public** **static** **double**[][] MakeMatrix(**int** rows, **int** cols) { ... }
41. **public** **static** **void** ShowVector(**double**[] vector) { ... }
42. **public** **static** **void** ShowMatrix(**double**[][] matrix, **int** numRows) { ... }
43. }
44. } // ns

I deleted all the template-generated using statements except for the one referencing the System namespace. In the Main function, after displaying a begin message, I instantiate a NeuralNetwork object named nn with three inputs, four hidden neurons and two outputs. Next, I assign 26 arbitrary weights and biases to an array named weights. I load the weights into the neural network object using a method named SetWeights. I assign values 1.0, 2.0 and 3.0 to an array named xValues. I use method ComputeOutputs to load the input values into the neural network and determine the resulting outputs, which I fetch into an array named yValues. The demo concludes by displaying the output values.

## The NeuralNetwork Class

The NeuralNetwork class definition starts:

1. **class** NeuralNetwork
2. {
3. **private** **int** numInput;
4. **private** **int** numHidden;
5. **private** **int** numOutput;
6. ...

As explained in the previous sections, the structure of a neural network is determined by the number of input values, the number of hidden layer neurons and the number of output values. The class definition continues as:

1. **private** **double**[] inputs;
2. **private** **double**[][] ihWeights; // input-to-hidden
3. **private** **double**[] ihSums;
4. **private** **double**[] ihBiases;
5. **private** **double**[] ihOutputs;
6. **private** **double**[][] hoWeights;  // hidden-to-output
7. **private** **double**[] hoSums;
8. **private** **double**[] hoBiases;
9. **private** **double**[] outputs;
10. ...

These seven arrays and two matrices correspond to the ones shown in **Figure 3**. I use an ih prefix for input-to-hidden data and an ho prefix for hidden-to-output data. Recall that the values in the ihOutputs array serve as the inputs for the output layer computations, so naming this array precisely is a bit troublesome.

**Figure 6** shows how the NeuralNetwork class constructor is defined.

**Figure 6 The NeuralNetwork Class Constructor**

1. **public** NeuralNetwork(**int** numInput, **int** numHidden, **int** numOutput)
2. {
3. **this**.numInput = numInput;
4. **this**.numHidden = numHidden;
5. **this**.numOutput = numOutput;
6. inputs = **new** **double**[numInput];
7. ihWeights = Helpers.MakeMatrix(numInput, numHidden);
8. ihSums = **new** **double**[numHidden];
9. ihBiases = **new** **double**[numHidden];
10. ihOutputs = **new** **double**[numHidden];
11. hoWeights = Helpers.MakeMatrix(numHidden, numOutput);
12. hoSums = **new** **double**[numOutput];
13. hoBiases = **new** **double**[numOutput];
14. outputs = **new** **double**[numOutput];
15. }

After copying the input parameter values numInput, numHidden and numOutput into their respective class fields, each of the nine member arrays and matrices are allocated with the sizes I explained earlier. I implement matrices as arrays of arrays rather than using the C# multidimensional array type so that you can more easily refactor my code to a language that doesn’t support multidimensional array types. Because each row of my matrices must be allocated, it’s convenient to use a helper method such as MakeMatrix.

The SetWeights method accepts an array of weights and bias values and populates ihWeights, ihBiases, hoWeights and hoBiases. The method begins like this:

1. **public** **void** SetWeights(**double**[] weights)
2. {
3. **int** numWeights = (numInput \* numHidden) +
4. (numHidden \* numOutput) + numHidden + numOutput;
5. **if** (weights.Length != numWeights)
6. **throw** **new** Exception("xxxxxx");
7. **int** k = 0;
8. ...

As explained earlier, the total number of weights and biases, Nw, in a fully connected feedforward neural network is (Ni \* Nh) + (Nh \* No) + Nh + No. I do a simple check to see if the weights array parameter has the correct length. Here, “xxxxxx” is a stand-in for a descriptive error message. Next, I initialize an index variable k to the beginning of the weights array parameter. Method SetWeights concludes:

1. **for** (**int** i = 0; i < numInput; ++i)
2. **for** (**int** j = 0; j < numHidden; ++j)
3. ihWeights[i][j] = weights[k++];
4. **for** (**int** i = 0; i < numHidden; ++i)
5. ihBiases[i] = weights[k++];
6. **for** (**int** i = 0; i < numHidden; ++i)
7. **for** (**int** j = 0; j < numOutput; ++j)
8. hoWeights[i][j] = weights[k++];
9. **for** (**int** i = 0; i < numOutput; ++i)
10. hoBiases[i] = weights[k++]
11. }

Each value in the weights array parameter is copied sequentially into ihWeights, ihBiases, hoWeights and hoBiases. Notice no values are copied into ihSums or hoSums because those two scratch arrays are used for computation.

## Computing the Outputs

The heart of the NeuralNetwork class is method ComputeOutputs. The method is surprisingly short and simple and begins:

1. **public** **double**[] ComputeOutputs(**double**[] xValues)
2. {
3. **if** (xValues.Length != numInput)
4. **throw** **new** Exception("xxxxxx");
5. **for** (**int** i = 0; i < numHidden; ++i)
6. ihSums[i] = 0.0;
7. **for** (**int** i = 0; i < numOutput; ++i)
8. hoSums[i] = 0.0;
9. ...

First I check to see if the length of the input x-values array is the correct size for the NeuralNetwork object. Then I zero out the ihSums and hoSums arrays. If ComputeOutputs is called only once, then this explicit initialization is not necessary, but if ComputeOutputs is called more than once—because ihSums and hoSums are accumulated values—the explicit initialization is absolutely necessary. An alternative design approach is to not declare and allocate ihSums and hoSums as class members, but instead make them local to the ComputeOutputs method. Method ComputeOutputs continues:

1. **for** (**int** i = 0; i < xValues.Length; ++i)
2. **this**.inputs[i] = xValues[i];
3. **for** (**int** j = 0; j < numHidden; ++j)
4. **for** (**int** i = 0; i < numInput; ++i)
5. ihSums[j] += **this**.inputs[i] \* ihWeights[i][j];
6. ...

The values in the xValues array parameter are copied to the class inputs array member. In some neural network scenarios, input parameter values are normalized, for example by performing a linear transform so that all inputs are scaled between -1.0 and +1.0, but here no normalization is performed. Next, a nested loop computes the weighted sums as shown in **Figures 1** and **3**. Notice that in order to index ihWeights in standard form where index i is the row index and index j is the column index, it’s necessary to have j in the outer loop. Method ComputeOutputs continues:

1. **for** (**int** i = 0; i < numHidden; ++i)
2. ihSums[i] += ihBiases[i];
3. **for** (**int** i = 0; i < numHidden; ++i)
4. ihOutputs[i] = SigmoidFunction(ihSums[i]);
5. ...

Each weighted sum is modified by adding the appropriate bias value. At this point, to produce the output shown in **Figure 2**, I used method Helpers.ShowVector to display the current values in the ihSums array. Next, I apply the sigmoid function to each of the values in ihSums and assign the results to array ihOutputs. I’ll present the code for method SigmoidFunction shortly. Method ComputeOutputs continues:

1. **for** (**int** j = 0; j < numOutput; ++j)
2. **for** (**int** i = 0; i < numHidden; ++i)
3. hoSums[j] += ihOutputs[i] \* hoWeights[i][j];
4. **for** (**int** i = 0; i < numOutput; ++i)
5. hoSums[i] += hoBiases[i];
6. ...

I use the just-computed values in ihOutputs and the weights in hoWeights to compute values into hoSums, then I add the appropriate hidden-to-output bias values. Again, to produce the output shown in **Figure 2**, I called Helpers.ShowVector. Method ComputeOutputs finishes:

1. **for** (**int** i = 0; i < numOutput; ++i)
2. **this**.outputs[i] = HyperTanFunction(hoSums[i]);
3. **double**[] result = **new** **double**[numOutput];
4. **this**.outputs.CopyTo(result, 0);
5. **return** result;
6. }

I apply method HyperTanFunction to the hoSums to generate the final outputs into class array private member outputs. I copy those outputs to a local result array and use that array as a return value. An alternative design choice would be to implement ComputeOutputs without a return value, but implement a public method GetOutputs so that the outputs of the neural network object could be retrieved.

## The Activation Functions and Helper Methods

Here’s the code for the sigmoid function used to compute the input-to-hidden outputs:

1. **private** **static** **double** SigmoidFunction(**double** x)
2. {
3. **if** (x < -45.0) **return** 0.0;
4. **else** **if** (x > 45.0) **return** 1.0;
5. **else** **return** 1.0 / (1.0 + Math.Exp(-x));
6. }

Because some implementations of the Math.Exp function can produce arithmetic overflow, checking the value of the input parameter is usually performed. The code for the tanh function used to compute the hidden-to-output results is:

1. **private** **static** **double** HyperTanFunction(**double** x)
2. {
3. **if** (x < -10.0) **return** -1.0;
4. **else** **if** (x > 10.0) **return** 1.0;
5. **else** **return** Math.Tanh(x);
6. }

The hyperbolic tangent function returns values between -1 and +1, so arithmetic overflow is not a problem. Here the input value is checked merely to improve performance.

The static utility methods in class Helpers are just coding conveniences. The MakeMatrix method used to allocate matrices in the NeuralNetwork constructor allocates each row of a matrix implemented as an array of arrays:

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

Methods ShowVector and ShowMatrix display the values in an array or matrix to the console. You can see the code for these two methods in the code download that accompanies this article (available at[archive.msdn.microsoft.com/mag201205TestRun](http://archive.msdn.microsoft.com/mag201205TestRun)).

## Next Steps

The code presented here should give you a solid basis for understanding and experimenting with neural networks. You might want to examine the effects of using different activation functions and varying the number of inputs, outputs and hidden layer neurons. You can modify the neural network by making it partially connected, where some neurons are not logically connected to neurons in the next layer. The neural network presented in this article has one hidden layer. It’s possible to create more complex neural networks that have two or even more hidden layers, and you might want to extend the code presented here to implement such a neural network.

Neural networks can be used to solve a variety of practical problems, including classification problems. In order to solve such problems there are several challenges. For example, you must know how to encode non-numeric data and how to train a neural network to find the best set of weights and biases. I will present an example of using neural networks for classification in a future article.

s