

Restricted Boltzmann Machines Using C#

By [James McCaffrey](#)



A restricted Boltzmann machine (RBM) is a fascinating software component that has some similarities to a basic neural network. An RBM has two sets of nodes—visible and hidden. Each set of nodes can act as either inputs or outputs relative to the other set. Each node has a value of zero or one and these values are calculated using probability rather than deterministically.

Each visible-layer node is conceptually connected to each hidden-layer node with a numeric constant called a weight, typically a value between -10.0 and +10.0. Each node has an associated numeric constant called a bias. The best way to start to understand what an RBM is is by way of a diagram. Take a look at **Figure 1**.

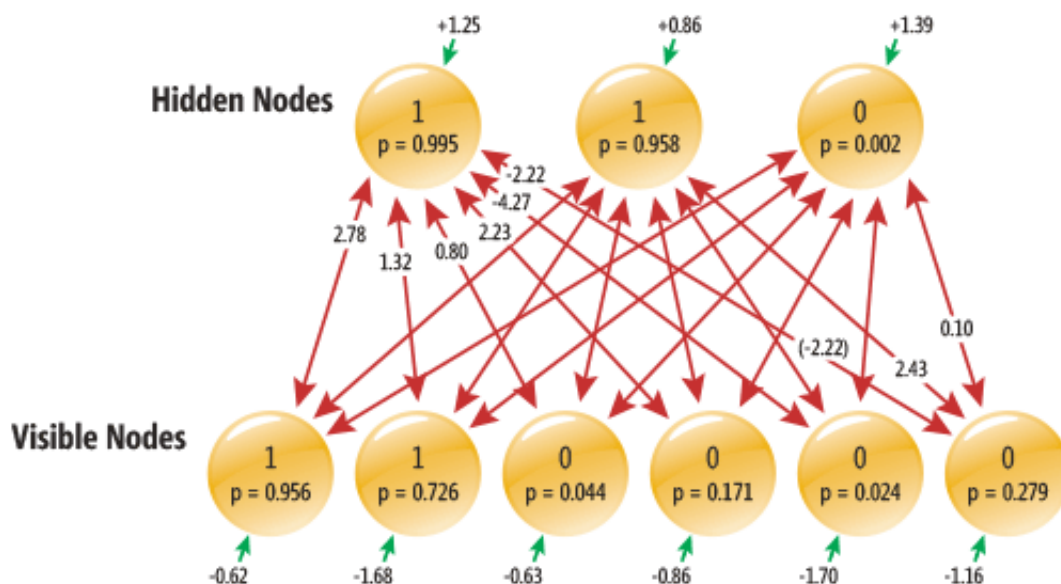


Figure 1 An Example of a Restricted Boltzmann Machine

In **Figure 1**, the visible nodes are acting as the inputs. There are six visible (input) nodes and three hidden (output) nodes. The values of the visible nodes are (1, 1, 0, 0, 0, 0) and the computed values of the hidden nodes are (1, 1, 0). There are $6 * 3 = 18$ weights connecting the nodes. Notice that there are no visible-to-visible or hidden-to-hidden weights. This restriction is why the word “restricted” is part of the RBM name.

Each of the red weight arrows in **Figure 1** points in both directions, indicating that information can flow in either direction. If nodes are zero-base indexed, then the weight from visible[0] to hidden[0] is 2.78, the weight from visible[5] to hidden[2] is 0.10 and so on. The bias values are the small green arrows pointing into each node so the bias for visible[0] is -0.62 and the bias for hidden[0] is +1.25 and so on. The p value inside each node is the probability that the node takes a value of one. So, hidden[0] has $p = 0.995$, which means that its calculated value will almost certainly be one, and in fact it is, but because RBMs are probabilistic, the value of hidden[0] could have been zero.

You probably have many questions right about now, such as where the weights and bias values come from, but bear with me—you'll see how all the parts of the puzzle fit together shortly. In the sections that follow, I'll describe the RBM input-output mechanism, explain where the weights and bias values come from, present a demo program that corresponds to **Figure 1**, and give an example of how RBMs can be used.

This article assumes you have intermediate or better programming skills, but doesn't assume you know anything about RBMs. The demo program is coded using C#, but you should have no trouble refactoring the code to another language such as Python or JavaScript if you wish. The demo program is too long to present in its entirety, but the complete source code is available in the file download that accompanies this article. All error checking was removed to keep the main ideas as clear as possible

The RBM Input-Output Mechanism

The RBM input-output mechanism is (somewhat surprisingly) relatively simple and is best explained by an example. Suppose, as in **Figure 1**, the visible nodes act as inputs and have values (1, 1, 0, 0, 0, 0). The value of hidden node[0] is computed as follows: The six weights from the visible nodes into hidden[0] are (2.78, 1.32, 0.80, 2.23, -4.27, -2.22) and the bias value for hidden[0] is 1.25.

The p value for hidden[0] is the logistic sigmoid value of the sum of the products of input values multiplied by their associated weights, plus the target node bias. Put another way, multiply each input node value by the weight pointing from the node into the target node, add those products up, add in the target node bias value and then take the logistic sigmoid of that sum:

```
1 p[0] = logsig( (1 * 2.78) + (1 * 1.32) + (0 * 0.80) +  
2               (0 * 2.23) + (0 * -4.27) + (0 * -2.22) + 1.25 )  
3       = logsig( 2.78 + 1.32 + 1.25 )  
4       = logsig( 5.36 )  
5       = 0.995
```

The logistic sigmoid function, which appears in many machine learning algorithms, is defined as:

```
1 logsig(x) = 1.0 / (1.0 + exp(-x))
```

where the exp function is defined as:

$$1 \quad \exp(x) = e^x$$

where e (Euler's number) is approximately 2.71828.

So, at this point, the p value for `hidden[0]` has been calculated as 0.995. To calculate the final zero or one value for the `hidden[0]` node you'd use a random number generator to produce a pseudo-random value between 0.0 and 1.0. If the random number is less than 0.995 (which it will be 99.5 percent of the time), the node value is set to one; otherwise (0.05 percent of the time), it's set to zero.

The other hidden nodes would be computed in the same way. And if the hidden nodes were acting as inputs, the values of the visible nodes would be calculated as output values in the same way.

Determining the Weights and Bias Values

Determining a set of RBM output values for a given set of input values is easy, but from where do the weights and bias values come? Unlike neural networks, which require a set of training data with known input values and known, correct, output values, RBMs can essentially train themselves, so to speak, using only a set of values for the visible nodes. Interesting! Suppose you have a set of 12 data items, like so:

```
1 (1, 1, 0, 0, 0, 0) // A
2 (0, 0, 1, 1, 0, 0) // B
3 (0, 0, 0, 0, 1, 1) // C
4 (1, 1, 0, 0, 0, 1) // noisy A
5 (0, 0, 1, 1, 0, 0) // B
6 (0, 0, 0, 0, 1, 1) // C
7 (1, 0, 0, 0, 0, 0) // weak A
8 (0, 0, 1, 0, 0, 0) // weak B
9 (0, 0, 0, 0, 1, 0) // weak C
10 (1, 1, 0, 1, 0, 0) // noisy A
11 (1, 0, 1, 1, 0, 0) // noisy B
12 (0, 0, 1, 0, 1, 1) // noisy C
```

Because RBM visible node values are zero and one, you can think of them as individual binary features (such as “like” and “don’t like”) or as binary-encoded integers. Suppose each of the 12 data items represents a person’s like or don’t like opinion for six films: “Alien,” “Inception,” “Spy,” “EuroTrip,” “Gladiator,” “Spartacus.” The first two films are science fiction. The next two films are comedy (well, depending on your sense of humor) and the last two films are history (sort of).

The first person likes “Alien” and “Inception,” but doesn’t like the other four films. If you look at the data, you can imagine that there are three types of people. Type “A” people like only science fiction films. Type “B” like only comedy films and type “C” like only history films. Notice that there’s some noise in the data, and there are weak and noisy versions of each person type.

The number of visible nodes in an RBM is determined by the number of dimensions of the input data—six in this example. The number of hidden nodes is a free parameter that you must choose. Suppose you set the number of hidden nodes to three. Because each RBM node value can be zero or one, with three hidden nodes there are a total of eight people types that can be detected: (0, 0, 0), (0, 0, 1), . . . (1, 1, 1).

There are several ways to train an RBM. The most common algorithm is called CD-1, which stands for contrastive divergence, single-step. The algorithm is very clever and isn't at all obvious. The CD-1 training algorithm is presented in high-level pseudo-code in **Figure 2**.

Figure 2 The CD-1 Training Algorithm:

```
1  (v represents the visible nodes)
2  (h represents the hidden nodes)
3  (lr is a small learning rate value)
4  loop n times
5      for each data item
6          compute h from v
7          let posGrad = outer product(v, h)
8          compute v' from h
9          compute h' from v'
10         let negGrad = outer product(v', h')
11         let delta W = lr * (posGrad - negGrad)
12         let delta vb = lr * (v - v')
13         let delta hb = lr * (h - h')
14     end for
15 end loop
```

The goal of training is to find a set of weights and bias values so that when the RBM is fed a set of input values in the visible nodes and generates a set of output nodes in the hidden nodes, then when the hidden nodes are used as inputs, the original visible nodes values will (usually) be regenerated. The only way I was able to understand CD-1 was by walking through a few concrete examples.

Suppose the learning rate is set to 0.01 and that at some point the current training input item is (0, 0, 1, 1, 0, 0) and the 18 weights are:

1	0.01	0.02	0.03
2	0.04	0.05	0.06
3	0.07	0.08	0.09
4	0.10	0.11	0.12
5	0.13	0.14	0.15
6	0.16	0.17	0.18

The row index, 0 to 5, represents the index of a visible node and the column index, 0 to 2, represents the index of a hidden node. So the weight from visible[0] to hidden[2] is 0.03.

The first step in CD-1 is to compute the h values from the v values using the probabilistic mechanism described in the previous section. Suppose h turns out to be $(0, 1, 0)$. Next, the positive gradient is computed as the outer product of v and h :

1	0	0	0
2	0	0	0
3	0	1	0
4	0	1	0
5	0	0	0
6	0	0	0

The outer product isn't very common in machine learning algorithms (or any other algorithms for that matter), so it's quite possible you haven't seen it before. The Wikipedia article on the topic gives a pretty good explanation. Notice that the shape of the positive gradient matrix will be the same as the shape of the weight matrix.

Next, the h values are used as inputs, along with the current weights, to produce new output values v' . Suppose v' turns out to be $(0, 1, 1, 1, 0, 0)$. Next, v' is used as the input to compute a new h' . Suppose h' is $(0, 0, 1)$.

The negative gradient is the outer product of v' and h' and so is:

1	0	0	0
2	0	0	1
3	0	0	1
4	0	0	1
5	0	0	0
6	0	0	0

The result of $\text{posGrad} - \text{negGrad}$ is:

1	0	0	0
2	0	0	-1
3	0	+1	-1
4	0	+1	-1
5	0	0	0
6	0	0	0

If you review the algorithm carefully, you'll see that cell values in the delta gradient matrix can only be one of three values: 0, +1 or -1. Delta gradient values of +1 correspond to weights that should be increased slightly. Values of -1 correspond to weights that should be decreased slightly. Clever! The amount of increase or decrease is set by a learning rate value. So the weight from $\text{visible}[1]$ to $\text{hidden}[2]$ would be decreased by 0.01 and the weight from $\text{visible}[2]$ to $\text{hidden}[1]$ would be increased by 0.01. A small learning rate value makes training take longer, but a large learning rate can skip over good weight values.

So, how many iterations of training should be performed? In general, setting the number of training iterations and choosing a value of the learning rate are matters of trial and error. In the demo program that accompanies this article, I used a learning rate of 0.01 and a maximum number of iterations set to 1,000. After training, I got the weights and bias values shown in **Figure 1**.

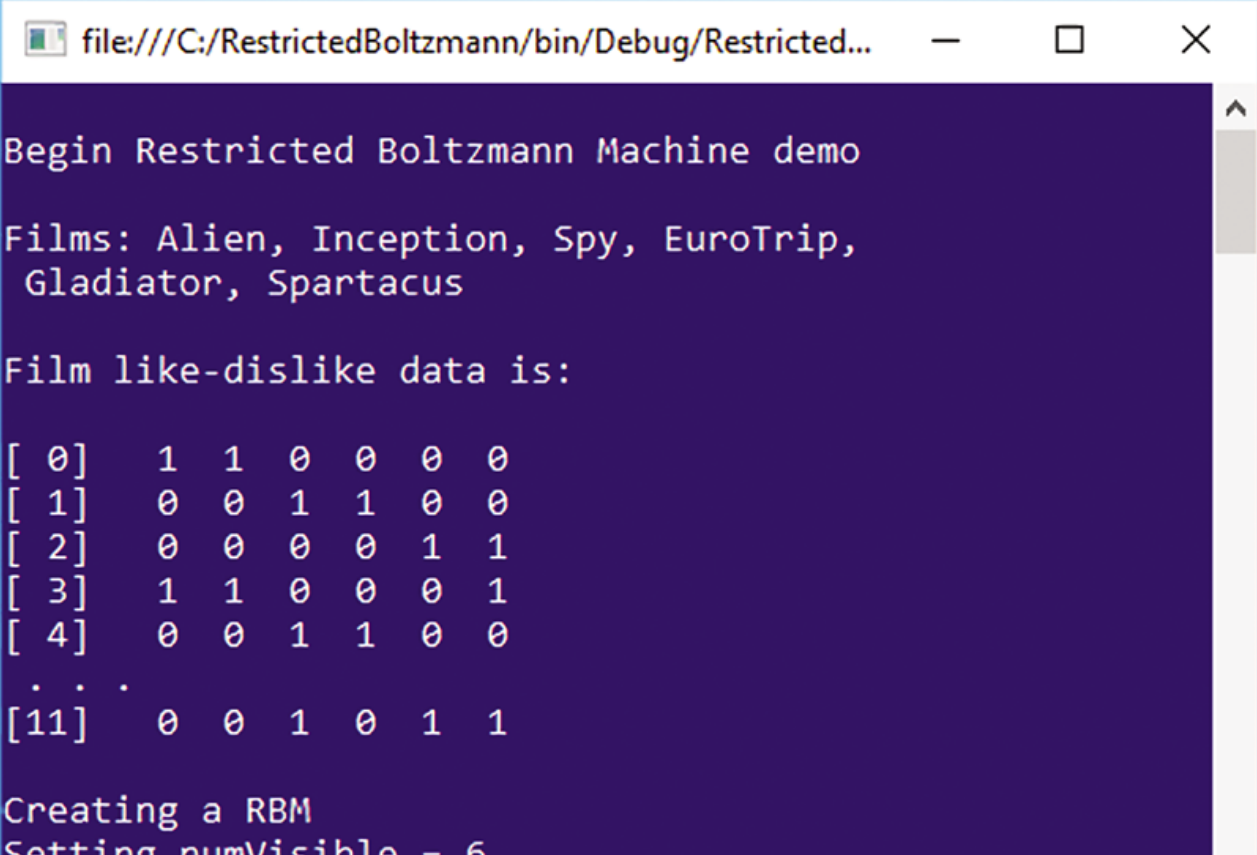
Interpreting a Restricted Boltzmann Machine

OK, so it's possible to take a set of data where each value is zero or one, then set a number of hidden nodes, and get some weights and bias values. What's the point?

One way to think of an RBM is as a kind of compression machine. For the example film preference data, if you feed a type A person as input (1, 1, 0, 0, 0, 0), you'll usually get (1, 1, 0) as output. If you feed (1, 1, 0) as input to the hidden nodes, you almost always get (1, 1, 0, 0, 0, 0) as output in the visible nodes. In other words, (1, 1, 0, 0, 0, 0) and slight variations are mapped to (1, 1, 0). This behavior is closely related to, but not quite the same as, factor analysis in classical statistics.

Take a look at the demo program in **Figure 3**. The demo corresponds to the film like-dislike example. The demo creates a 6-3 RBM and trains it using the 12 data items presented in the previous section. The hardcoded data is set up like so:

```
1  int[][] trainData = new int[12][];
2  trainData[0] = new int[] { 1, 1, 0, 0, 0, 0 };
3  trainData[1] = new int[] { 0, 0, 1, 1, 0, 0 };
4  ...
5  trainData[11] = new int[] { 0, 0, 1, 0, 1, 1 };
```



```
file:///C:/RestrictedBoltzmann/bin/Debug/Restricted...

Begin Restricted Boltzmann Machine demo

Films: Alien, Inception, Spy, EuroTrip,
Gladiator, Spartacus

Film like-dislike data is:

[ 0]  1  1  0  0  0  0
[ 1]  0  0  1  1  0  0
[ 2]  0  0  0  0  1  1
[ 3]  1  1  0  0  0  1
[ 4]  0  0  1  1  0  0
. . .
[11]  0  0  1  0  1  1

Creating a RBM
Setting numVisible = 6
```

```
Setting numVisible = 6
Setting numHidden = 3

Training RBM using CD1 algorithm
Setting learnRate = 0.010
Setting maxEpochs = 1000
Training complete

Trained machine's weights and biases are:

2.7810    0.9239   -3.8556
1.3265    1.3348   -3.5729
0.8048   -3.2463    2.8220
2.2390   -2.9565   -0.3361
-4.2739    2.2556    2.5457
-2.2257    2.4399    0.1063

visible bias [0] value = -0.6283
visible bias [1] value = -1.6883
visible bias [2] value = -0.6365
visible bias [3] value = -0.8616
visible bias [4] value = -1.7065
visible bias [5] value = -1.1639

hidden bias [0] value = 1.2500
hidden bias [1] value = 0.8610
hidden bias [2] value = 1.3972

Using trained machine . . .

visible = 1 1 0 0 0 0  -> 1 1 0
hidden = 1 1 0  -> 1 1 0 0 0 0

End RBM demo
```

Figure 3 Demo of a Restricted Boltzmann Machine

In most situations you'd read data from a text file using a helper method. The demo RBM is created and trained like this:


```

1  int numVisible = 6;
2  int numHidden = 3;
3  Machine rbm = new Machine(numVisible, numHidden);
4  double learnRate = 0.01;
5  int maxEpochs = 1000;
6  rbm.Train(trainData, learnRate, maxEpochs);

```

The choice of setting the number of hidden nodes to three was arbitrary and the values for learnRate and maxEpochs were determined by trial and error. After training, the RBM is exercised like this:

```

1  int[] visibles = new int[] { 1, 1, 0, 0, 0, 0 };
2  int[] computedHidden = rbm.HiddenFromVis(visibles);
3  Console.WriteLine("visible = ");
4  ShowVector(visibles, false);
5  Console.WriteLine("  -> ");
6  ShowVector(computedHidden, true);

```

If you experiment with the code, you'll notice that the computed hidden values are almost always one of three patterns. Person type A (or weak or noisy version) almost always generates (1, 1, 0). Type B generates (1, 0, 1). And type C generates (0, 1, 1). And if you feed the three patterns as inputs, you'll almost always get the three primary input patterns back. The point is the RBM has deduced that the data can be placed into one of three buckets. The specific bit patterns aren't important.

Yet another interpretation of this behavior is that an RBM acts as an auto-encoder. And it's also possible to chain several RBMs together to create a prediction system called a deep belief network (DBN). In fact, this is arguably the most common use of RBMs.

Implementing a Restricted Boltzmann Machine

Once you understand how RBMs work, they're actually quite simple. But coding a demo program is a bit more complex than you might expect. There are many design possibilities for an RBM. Take a look at the demo run in **Figure 3**.

The demo illustrates the film preference example from the previous sections of this article so there are six visible nodes. The demo program defines a Machine object with 10 member fields. The first six fields in the class definition are:


```

1 public class Machine
2 {
3     public Random rnd;
4     public int numVisible;
5     public int numHidden;
6     public int[] visValues;
7     public double[] visProbs;
8     public double[] visBiases;
9     ...

```

All fields are declared with public scope for simplicity. The Random object is used when converting a node probability to a concrete zero or one value. Variables numVisible and numHidden (OK, OK, I know they're objects) hold the number of hidden and visible nodes. Integer array visValues holds the zero or one values of the visible nodes. Note that you can use a Boolean type if you wish. Double array visBiases holds the bias values associated with each visible node. Double array visProbs holds visible node probabilities. Note that the visProbs array isn't necessary because node values can be computed on the fly; however, storing the probability values is useful if you want to examine the behavior of the RBM during runtime.

The other four Machine class fields are:

```

1 public int[] hidValues;
2 public double[] hidProbs;
3 public double[] hidBiases;
4 public double[][] vhWeights;

```

Arrays hidValues, hidBiases, and hidProbs are the node values, associated bias values, and node probabilities, respectively. The vhWeights object is an array-of-arrays style matrix where the row index corresponds to a visible node and the column index corresponds to a hidden node.

The key class method computes the values of the hidden nodes using values in a parameter that corresponds to the visible nodes. That method's definition begins with:

```

1 public int[] HiddenFromVis(int[] visibles)
2 {
3     int[] result = new int[numHidden];
4     ...

```

Next, the calculations of the hidden-layer nodes are done node by node:

```

1  for (int h = 0; h < numHidden; ++h) {
2      double sum = 0.0;
3      for (int v = 0; v < numVisible; ++v)
4          sum += visibles[v] * vhWeights[v][h];
5      sum += hidBiases[h]; // Add the hidden bias
6      double probActiv = LogSig(sum); // Compute prob
7      double pr = rnd.NextDouble(); // Determine 0/1
8      if (probActiv > pr) result[h] = 1;
9      else result[h] = 0;
10 }

```

The code mirrors the explanation of the input-output mechanism explained earlier. Function `LogSig` is a private helper function, because the Microsoft .NET Framework doesn't have a built-in logistic sigmoid function (at least that I'm aware of).

The key method concludes by returning the computed hidden node values to the caller:

```

1  ...
2      return result;
3  }

```

The rest of the demo code implements the CD-1 training algorithm as described earlier. The code isn't trivial, but if you examine it carefully, you should be able to make the connections between RBM concepts and implementation.

Wrapping Up

Restricted Boltzmann machines are simple and complicated at the same time. The RBM input-output mechanism is both deterministic and probabilistic (sometimes called stochastic), but is relatively easy to understand and implement. The more difficult aspect of RBMs, in my opinion, is understanding how they can be useful.

As a standalone software component, an RBM can act as a lossy compression machine to reduce the number of bits needed to represent some data, or can act as a probabilistic factor analysis component that identifies core concepts in a data set. When concatenated, RBMs can create a deep neural network structure called a "deep belief network" that can make predictions.

RBM's were invented in 1986 by my Microsoft colleague Paul Smolensky, but gained increased attention relatively recently when the CD-1 training algorithm was devised by researcher and Microsoft collaborator Geoffrey Hinton. Much of the information presented in this article is based on personal conversations with Smolensky, and the 2010 research paper, "A Practical Guide to Training Restricted Boltzmann Machines," by Hinton.

Dr. James McCaffrey works for Microsoft Research in Redmond, Wash. He has worked on several Microsoft products including Internet Explorer and Bing. Dr. McCaffrey can be reached at jammc@microsoft.com.

REFERENCE