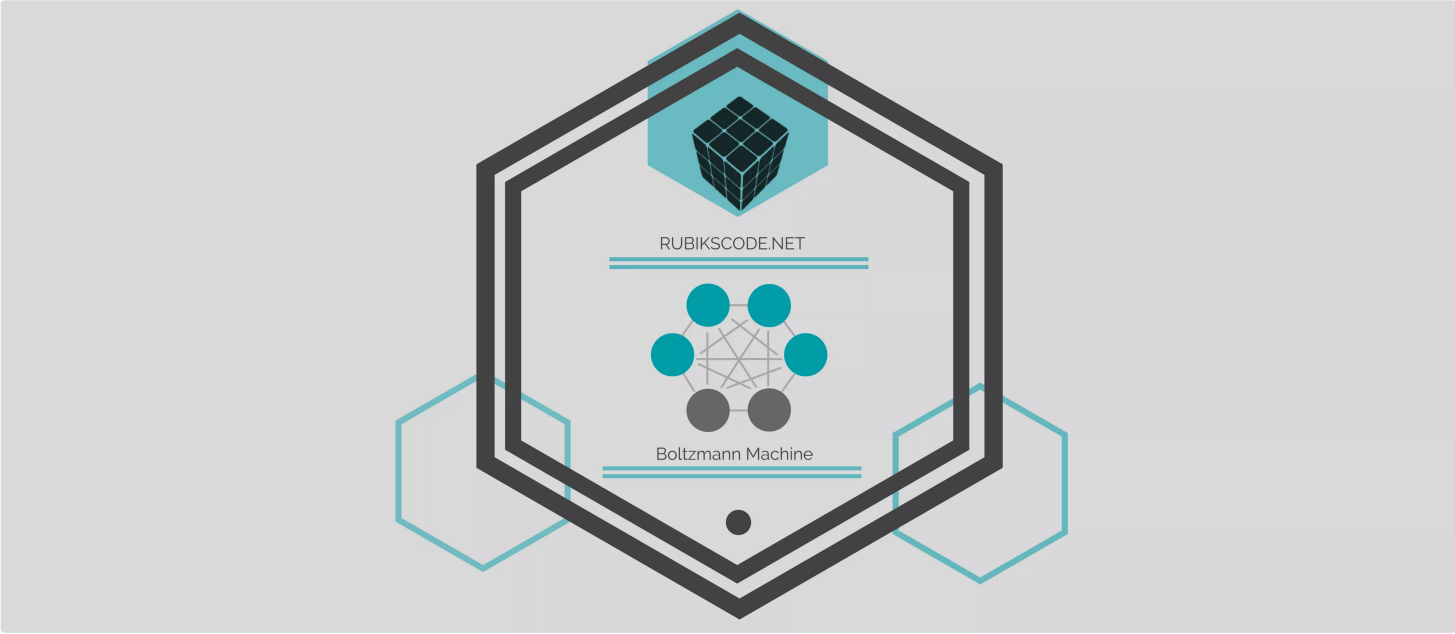
RUBIK'S CODE

Freedom.
Wisdom.
Excellence.

Implementing Restricted Boltzmann Machine with .NET Core

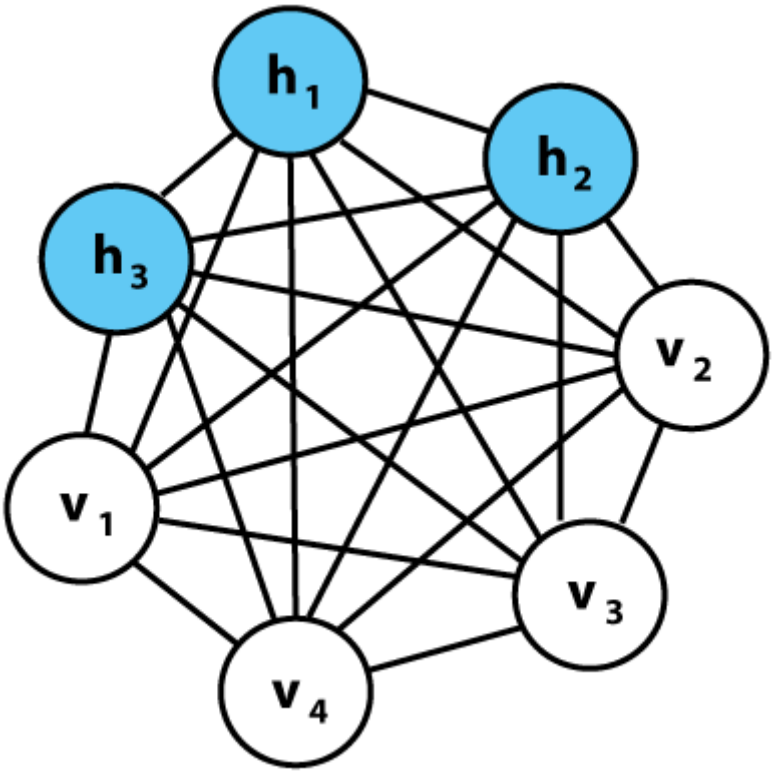
OCTOBER 15, 2018 — 2 COMMENTS



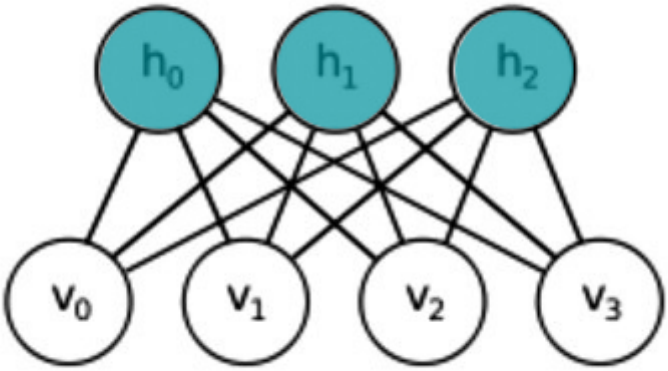
The code that accompanies this article can be downloaded here.

In the previous article, we had a chance to see what is the Restricted Boltzmann Machine and how it functions and learns. The path was bumpy because first, we needed to learn what are Energy-Based Models, the group of machine learning models that Restricted Boltzmann Machine is a part of. Then we explored functionalities of general Boltzmann machine, and finally, we examined an optimized version of it – Restricted Boltzmann Machine. To sum it up here, Energy-Based Models is a set of deep learning models which utilize physics concept of energy.

They determine dependencies between variables by associating a scalar value, which represents that energy to the complete system. To be more precise, this scalar value actually represents a measure of the probability that the system will be in a certain state. The Boltzmann Machine is one type of Energy-Based Models. They consist of symmetrically connected neurons. These neurons have a binary state, i.e they can be either on or off. The decision regarding the state is made stochastically. Since all neurons are connected to each other, calculating weights for all connections is resource demanding, so this architecture needed to be optimized.



In the end, we ended up with Restricted Boltzmann Machine, an architecture which is having two layers of neurons, visible and hidden, as you can see on the image below. Hidden neurons are connected only to the visible ones, and vice-versa, meaning there are no connections between layers in the same layer. This architecture is simple and pretty flexible. In fact, if you stack up several Restricted Boltzmann Machines, you will get so-called Deep Belief Networks, but this topic is out of the scope for this article.



Learning process of Restricted Boltzmann Machine is separated into two steps: Gibbs Sampling and Contrastive Divergence. More on the topic you can find in the previous article. Here we will only mention that first, we need to calculate probabilities that neuron from the hidden layer is activated based on the input values on the visible layer – Gibbs Sampling. Using this value, we will either turn on the neuron or not.

Contrastive Divergence Simplified

After Gibbs Sampling is performed we will use Contrastive Divergence to update the weights. This process is a bit tricky to be explained, so I decided to give it a full chapter in this article. We will use a simple example that will hopefully simplify this explanation. Let's consider the situation in which we have the visible layer with four nodes in the visible layer and hidden layer with three nodes. For example, lets say that input values on the visible layer are [0, 1, 1, 0].

Using formulas from the previous article, we will calculate the activation probability for each neuron in the hidden layer. If this probability is high, neuron from the hidden layer will be activated, otherwise, it will be off. For example, based on current weights and biases we get that values of the hidden layer are [0, 1, 1]. This is the moment when we calculate the so-called positive gradient.

Now, we use the outer product of visible layer neuron states [0, 1, 1, 0] and hidden layer neuron states [0, 1, 1]. Outer product is defined like this:

```
v[0]*h[0] v[0]*h[1] v[0]*h[2]
v[1]*h[0] v[1]*h[1] v[1]*h[2]
v[2]*h[0] v[2]*h[1] v[2]*h[2]
v[3]*h[0] v[3]*h[1] v[3]*h[2]
```

where *v* represents a neuron from the visible layer and *h* represents a neuron from the hidden layer. As a result, we get this:

```
0 0 0
0 1 1
0 1 1
0 0 0
```

This matrix is actually corresponding to all connections in this system. Meaning, the first element can be observed as some kind of property or action on the connection between *v[0]* and *h[0]*. In fact, it is exactly that! Wherever we have value 1 in the matrix we add the learning rate to the weight of the connection between two neurons. So, in our example we will do so for connections between *v[1]h[1], v[1]h[2], v[2]h[1]* and *v[2]h[2].*

Awesome! We performed the first step. Now, we are once again using formulas from the previous article to calculate probabilities for the neurons in the visible layer, using values from the hidden layer. Based on these probabilities we calculate temporary Contrastive Divergence states for the visible layer – *v'[n].* For example, we get the values [0, 0, 0, 1]. Finally, we calculate probabilities for the neurons in the hidden layer once again, only this time we are using Contrastive Divergence states of the visible layer calculated previously. We calculate Contrastive Divergence states for the hidden layer – – *h'[n],* and for this example get the results [0, 0, 1].

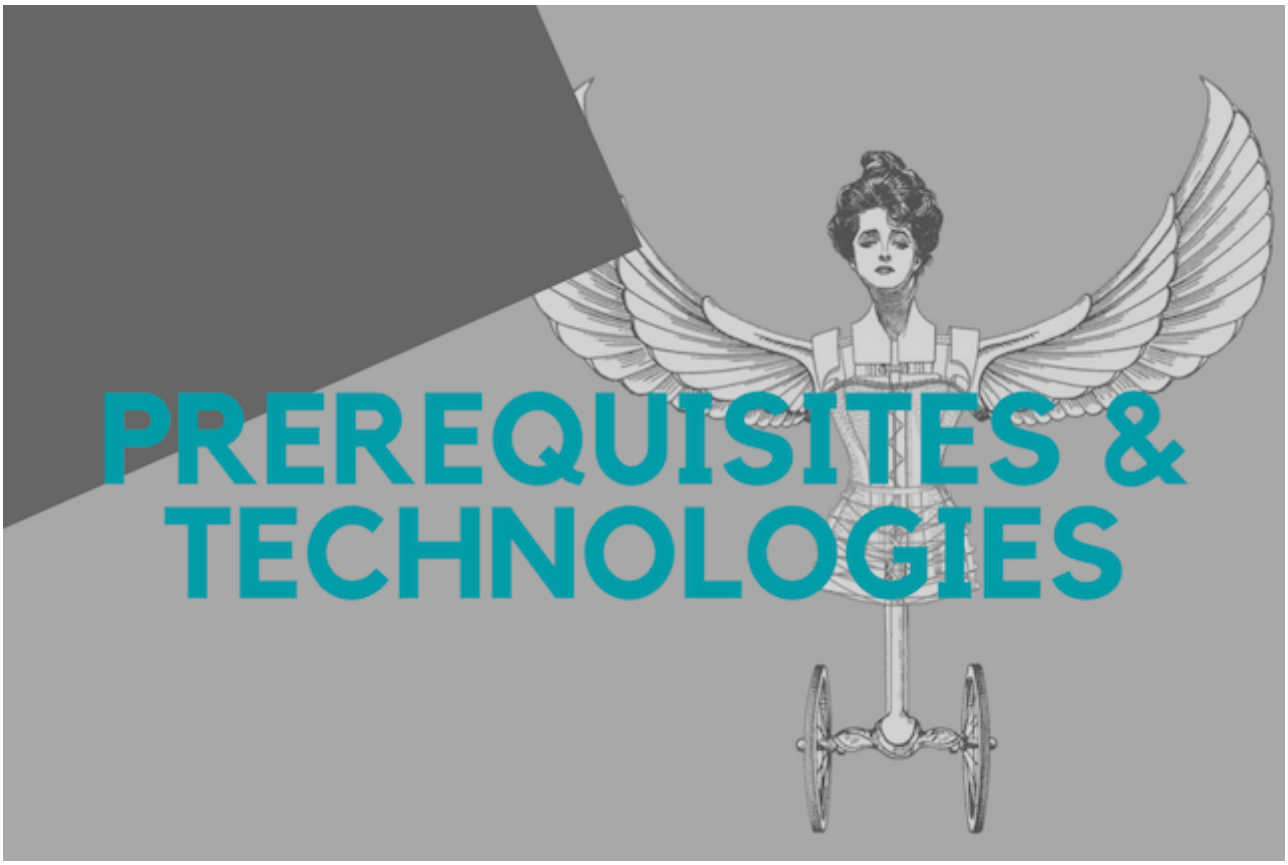
We use the outer product of visible layer neuron Contrastive Divergence states [0, 0, 0, 1] and hidden layer neuron states [0, 0, 1] to get this matrix:

```
0 0 0
0 0 0
0 0 0
0 0 1
```

This is how we defined the negative gradient. Similar to the previous situation, wherever we have value 1 in this matrix we will subtract the learning rate to the weight between two neurons. So, in our example, we will subtract the learning rate from the weights of the connection between neurons *v[4]h[3].*

Now let's see how we can implement Restricted Boltzmann Machine using .NET technologies.

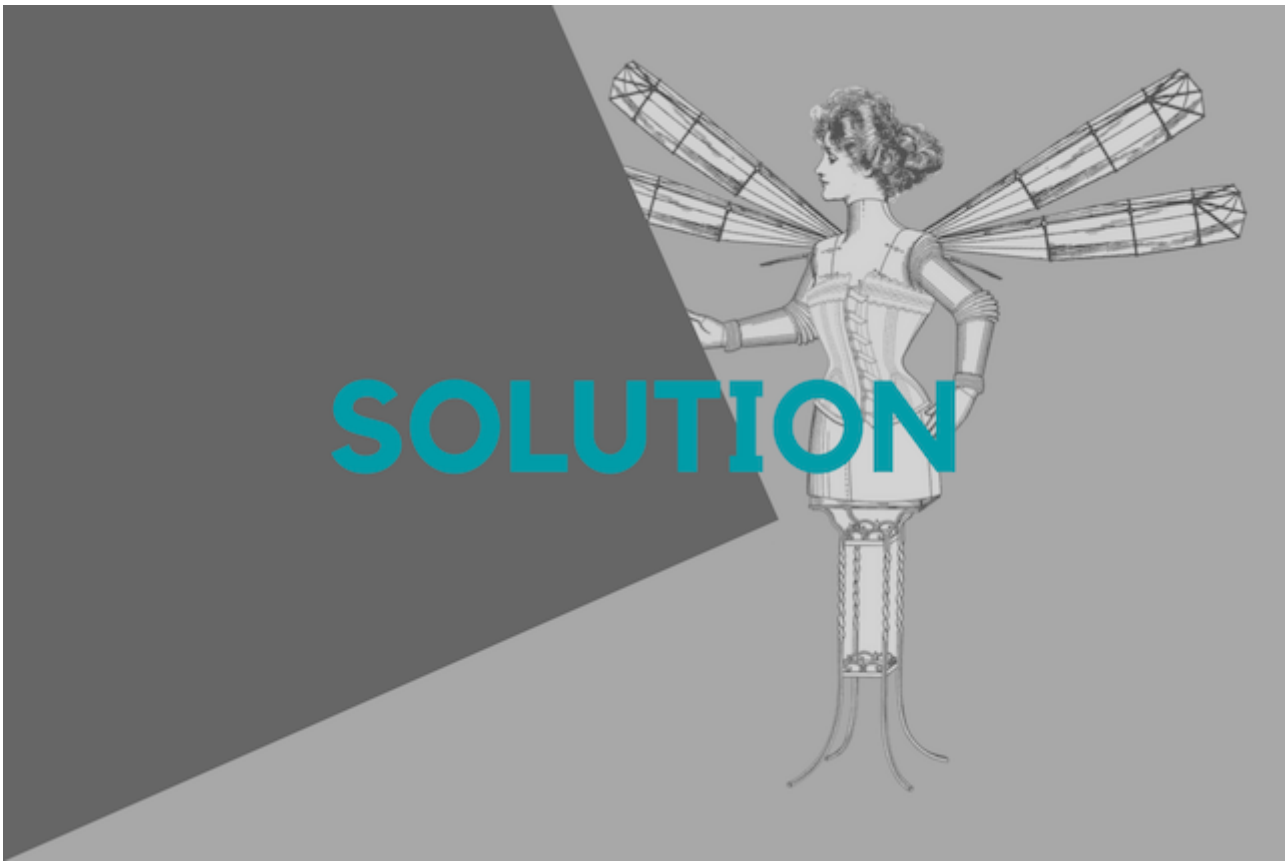
Prerequisites & Technologies



The solution itself is built using .NET Core and C#. It is created as a class library – *RestrictedBoltzmannMachine.* Apart from that, unit tests are written using Xunit and Moq in *RestrictedBoltzmannMachineTests* library. This is the list of technologies used for developing this solution:

- .NET Core 2.1
- C# 7.3
- Xunit 2.4.0
- Moq 4.10.0

Solution



The *RestrictedBoltzmannMachine* library is divided into five big classes. Each of these classes controls a different aspect of the Restricted Boltzmann Machine. If we take a look into the structure of the system, we can see that we need to model neurons, connections, visible and hidden layer. In addition to that, we need to coordinate the entire learning process using these entities. To sum it up, these classes are a part of this solution:

- Neuron – Models neurons in both visible and hidden layer.
- Connection – Models connections between neurons.
- Layer – An abstract class that implements basic layer operations, and from which hidden and visible layer are deriving from.
- Sigmoid – Models sigmoid function.
- RBM – Models the Restricted Boltzmann Machine itself.

It is important to note that this solution covers only the creation of two-layers Restricted Boltzmann Machine has only binary state neurons. So, let's take a look at the implementations of different classes.

Neuron

The neuron is, along with weighted connections, the main building unit of this system. We describe this unit using interface *INeuron* and we implement it in the class *Neuron.* The object of this class represents one neuron in either visible or hidden layer.

```
1   public interface INeuron
2   {
3       bool State { get; set; }
4       bool CDState { get; set; }
5       double Bias { get; }
6       List<IConnection> Inputs { get; }
7       List<IConnection> Outputs { get; }
8
9       void AddInputNeuron(INeuron inputNeuron, double weight);
10      void AddOutputNeuron(INeuron outputNeuron, double weight);
11  }
12
13  public class Neuron : INeuron
14  {
15      public bool State { get; set; }
16      public bool CDState { get; set; }
17      public double Bias { get; }
18      public List<IConnection> Inputs { get; }
19      public List<IConnection> Outputs { get; }
20
21
22      public Neuron(bool initialState, double bias)
23      {
24          State = initialState;
25          Bias = bias;
26
27          Inputs = new List<IConnection>();
28          Outputs = new List<IConnection>();
29      }
30
31      public void AddInputNeuron(INeuron inputNeuron, double weight)
32      {
33          var connection = new Connection(weight, inputNeuron, this);
34          Inputs.Add(connection);
35          inputNeuron.Outputs.Add(connection);
36      }
37
38      public void AddOutputNeuron(INeuron outputNeuron, double weight)
39      {
```

```
            var connection = new Connection(weight, this, outputNeuron);
            Outputs.Add(connection);
            outputNeuron.Inputs.Add(connection);
        }
    }
```

RBMNeuron.cs hosted with ❤ by GitHub view raw

As you can see, every neuron has its binary state – *State* and its bias. Also, every neuron contains the list of input and list of output connections. In this implementation, we haven't made a separation between visible and hidden neuron, which might seem weird at first. This is done so because we are planning to use this implementation for Deep Belief Networks later. However, we made this kind of separation on the layer level of abstraction.

Another thing we can note in our *Neuron* class is that it has a property *CDState.* This is not a real state of the neuron, it is just a temporary field that is used during the Contrastive Divergence process, that we explored earlier. Apart from that, we can see two functions *AddInputNeuron* and *AddOutputNeuron* which as their names suggest are used to connect this neuron with another neuron. In these functions, we create objects of the *Connection* class, which we will now explore.

Connection

Weighted connections are another inevitable part of our Restricted Boltzmann Machine. This entity is modeled through the interface *IConnection* and the class *Connection.* Here is how that looks like:

```
public interface IConnection
{
    double Weight { get; set; }
    INeuron InputNeruon { get; }
    INeuron OutputNeruon { get; }
    void UpdateWeight(double learningRate);
}

public class Connection : IConnection
{
    public double Weight { get; set; }

    public INeuron InputNeruon { get; }
    public INeuron OutputNeruon { get; }

```

```
16        public Connection(double weight, INeuron inputNeuron, INeuron outputNeuron)
17        {
18            Weight = weight;
19            InputNeruon = inputNeuron;
20            OutputNeruon = outputNeuron;
21        }
22
23        public void UpdateWeight(double learningRate)
24        {
25            if (InputNeruon.State & OutputNeruon.State)
26            {
27                Weight += learningRate;
28            }
29
30            if (InputNeruon.CDState & OutputNeruon.CDState)
31            {
32                Weight -= learningRate;
33            }
34        }
35    }
```

RBMConnection.cs hosted with ❤ by GitHub view raw

So, connections have input and output neuron, as well as the weight. We need to note *UpdateWeight* method. This method is adding or subtracting learning rate from the current weight value depending on the states of input and output neuron. Observe that if *CDStates* of neurons are both 1, we are subtracting the learning rate.

Layer

Now, when we have all the necessary blocks we can construct layer. This is an abstract class because here we want to make difference between visible and hidden layer. Why do we want to do that here? Well, for example, we want to push input values only to the visible layer. We will see how that look like in a second, but first, let's examine *Layer* class:

```
1    public abstract class Layer
2    {
3        public List<INeuron> Neurons { get; }
4
5        protected Sigmoid _sigmoid;
6        protected Random _random;
7
8        public Layer(int numberOfNeurons)
```

```
 9       {
10           Neurons = new List<INeuron>();
11           _random = new Random();
12
13           for (int i = 0; i < numberOfNeurons; i++)
14           {
15               Neurons.Add(new Neuron(GenerateRandomBool(), _random.NextDouble()));
16           }
17       }
18
19       public void ConnectLayers(Layer layer)
20       {
21           foreach(var thisNeuron in this.Neurons)
22           {
23               foreach(var layerNeruon in layer.Neurons)
24               {
25                   layerNeruon.AddInputNeuron(thisNeuron, _random.NextDouble());
26               }
27           }
28       }
29
30       private bool GenerateRandomBool()
31       {
32           return _random.NextDouble() > 0.5;
33       }
34
35       public abstract void CalculateCDState();
36   }
```

RBMLayer.cs hosted with ❤ by GitHub view raw

A layer is essentially a set of neurons, and that is exactly why this class has a list of *INeurons.* Apart from that, it has two protected fields which are used in *VisibleLayer* and *HiddenLayer* classes. These fields are helpers fields, and they are used for performing the sigmoid function and for generating random numbers. *GenerateRandomBool* is helpers method too.

This class defines *ConnectLayers* method, which is used to connect all the neurons from the two layers. It is considered that layer on which this action is called is a visible layer. The abstract method that is defined is called *CalculateCDState.* This method is overridden in *VisibleLayer* and *HiddenLayer* classes and it is used to calculate intermediate state during Contrastive Divergence.

Visible Layer

The first concrete layer implementation that we are going to examine is one of the visible layer, defined in the *VisibleLayer* class. Its implementing abstract *Layer* class and defines one extra method – *PushValuesToInput*. This method is setting states of neurons in the visible layer according to the input row.

```
1   public class VisibleLayer : Layer
2   {
3       public VisibleLayer(int numberOfNeurons) : base(numberOfNeurons)
4       {
5       }
6
7       public override void CalculateCDState()
8       {
9           foreach (var neuron in Neurons)
10          {
11              var probability = _sigmoid.CalculateOutput(neuron.Outputs.Where(x => x.InputNeruo
12              neuron.CDState = probability < _random.NextDouble();
13          }
14      }
15
16      public void PushValuesToInput(bool[] input)
17      {
18          if (input.Length != Neurons.Count)
19          {
20              throw new ArgumentException("Input has invalid size");
21          }
22
23          for (int i = 0; i < input.Length; i++)
24          {
25              Neurons[i].State = input[i];
26          }
27      }
28  }
```

RBMVisibleLayer.cs hosted with ❤ by GitHub view raw

Hidden Layer

Implementation of the hidden layer is located in the *HiddenLayer* class. This class implements the abstract *Layer* class and overrides method *CalculateCDState.* Apart from that, it defines method – *CalculateState.* This method is used to calculate the state of the neurons in the hidden layer based on the values of values in the visible layer. Here it is:

```
public class HiddenLayer : Layer
{
    public HiddenLayer(int numberOfNeurons) : base(numberOfNeurons)
    { }

    public void CalculateState()
    {
        foreach(var neuron in Neurons)
        {
            var probability = _sigmoid.CalculateOutput(neuron.Inputs.Where(x => x.InputNeruon
            neuron.State = probability < _random.NextDouble();
        }
    }

    public override void CalculateCDState()
    {
        foreach (var neuron in Neurons)
        {
            var probability = _sigmoid.CalculateOutput(neuron.Inputs.Where(x => x.InputNeruon
            neuron.CDState = probability < _random.NextDouble();
        }
    }
}
```

RBMHiddenLayer.cs hosted with ❤ by GitHub view raw

RBM

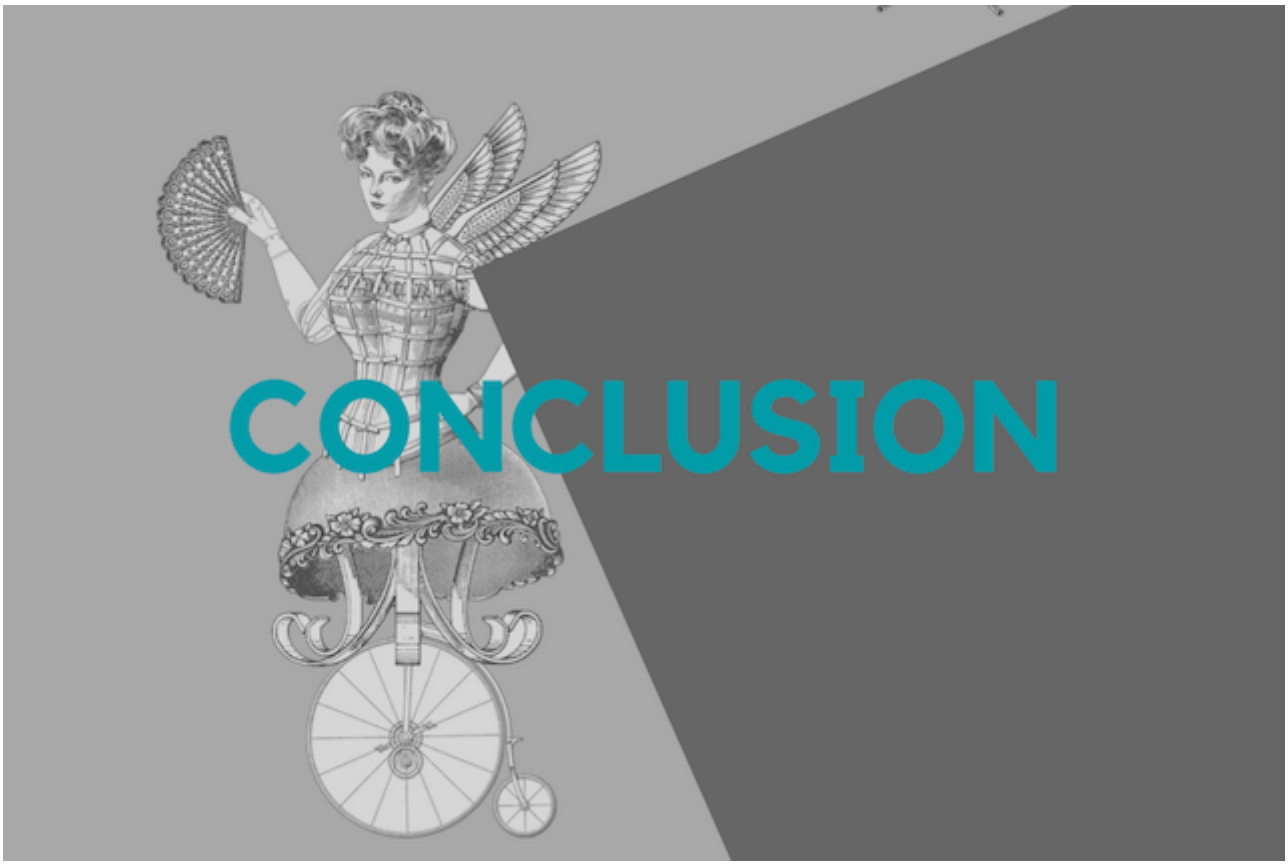
Finally, let's explore the class that utilizes all previous classes – *RBM.* This class is having fields for the visible layer, the hidden layer and for the learning rate. Also, it has *Train* method. In this method, we can see how all pieces are combined together.

```
public class RBM
{
    private HiddenLayer _hiddenLayer;
    private VisibleLayer _visibleLayer;
    private float _learningRate;

    public RBM(int hiddenLayerSize, int visibleLayerSize, float learningRate)
    {
        _learningRate = learningRate;
        _hiddenLayer = new HiddenLayer(hiddenLayerSize);
        _visibleLayer = new VisibleLayer(visibleLayerSize);

```

```
13            _visibleLayer.ConnectLayers(_hiddenLayer);
14        }
15
16        public void Train(bool[][] input, int numberOfIterations)
17        {
18            for (int n = 0; n < numberOfIterations; n++)
19            {
20                for (int i = 0; i < input.GetLength(1); i++)
21                {
22                    _visibleLayer.PushValuesToInput(input[i]);
23                    _hiddenLayer.CalculateState();
24                    _visibleLayer.CalculateCDState();
25                    _hiddenLayer.CalculateCDState();
26
27                    UpdateWeights();
28                }
29            }
30        }
31
32        private void UpdateWeights()
33        {
34            foreach(var inputNeuron in _visibleLayer.Neurons)
35            {
36                inputNeuron.Outputs.ForEach(x => x.UpdateWeight(_learningRate));
37            }
38        }
39    }
```

RBM.cs hosted with ❤ by GitHub view raw

Conclusion



The goal of this article was to make the concepts of Restricted Boltzmann Machine understandable to .NET developers. Apart from being a highly fun experience and experiment, I hope that some developers will benefit from the concepts explored here. We were able to see how to model and implement the main parts of the Restricted Boltzmann Machine, such as neurons and layers, and how to implement their learning process. If you want to explore this field even further, now you can use a library that we developed here. In the next article, we will do a similar thing, only using TensorFlow and Python.

Thank you for reading!

This article is a part of Artificial Neural Networks Series, which you can check out **here**.

Read more posts from the author at **Rubik's Code**.
