RUBIK'S CODE

Freedom.
Wisdom.
Excellence.

Implementing Simple Neural Network in C#

JANUARY 29, 2018 — 25 COMMENTS

Code that accompanies this article can be downloaded **here**.

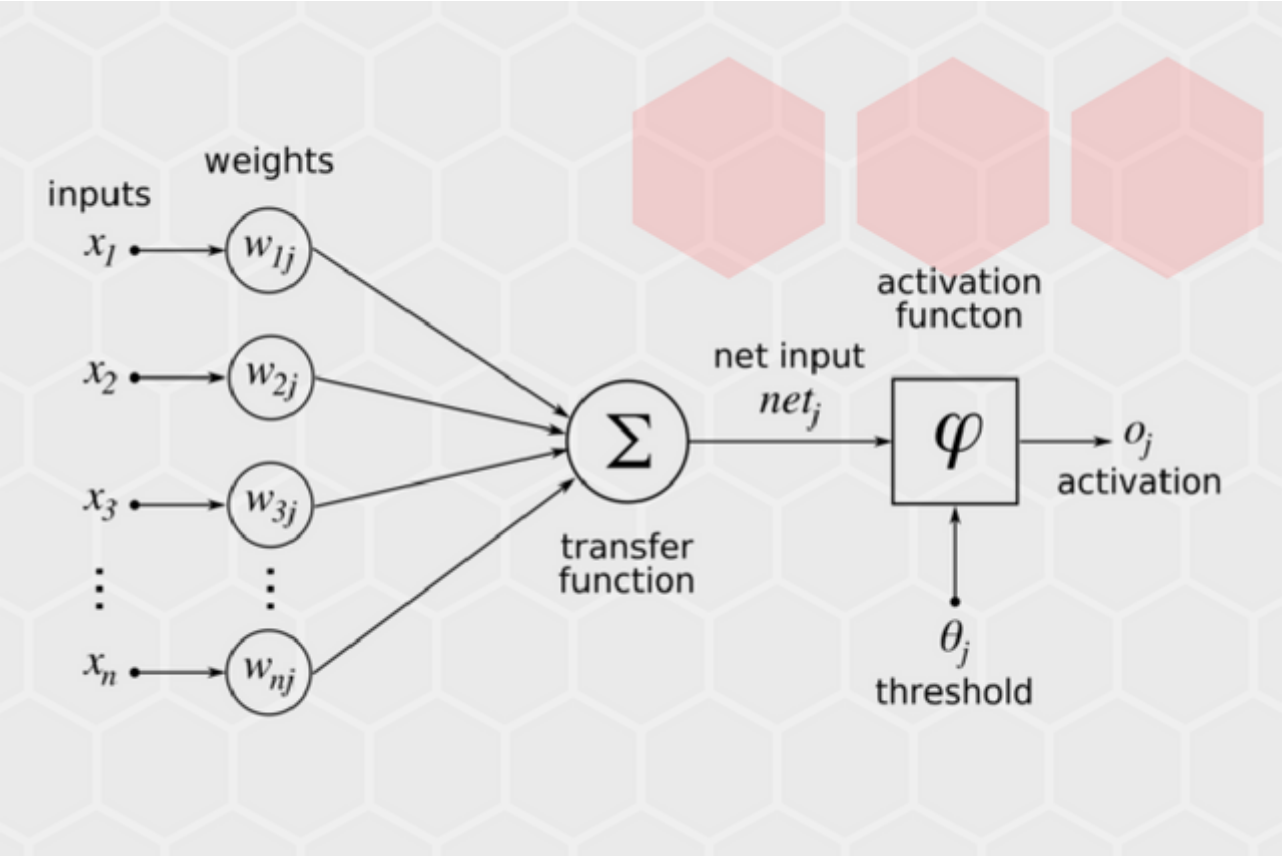
Sometime in the last few weeks, while I was writing the explanations for the **way in which neural networks learn** and **backpropagation algorithm,** I realized how I never tried to implement these algorithms in one of the programming languages. Then it struck me that I've never tried to implement the whole Artificial Neural Network from scratch. I was always using some libraries that were hiding that implementation from me so I could focus on the mathematical model and the problem I was trying to solve. One thing led to another and the decision to implement my own Neural Network from scratch without using third-party libraries was made. Also, I decided to use object-oriented programming language I prefer – C#.

This means that a more OO approach was taken and not the usual scripting point of view like we would have by using Python and R. One very good **article** did that implementation that kind of way and I strongly recommend you to skim through it. What I wanted to do is to separate every component and every operation. What was initially just a thought exercise grew into quite a cool mini side-project. So, I decided to share it with the world. Before we dive into the code, I would like to emphasize that this is not really the way you would generally implement the network. More math and forms of matrix multiplication should be used to optimize this entire process.

Apart from that, the implemented network represents a simplified, most basic form of Neural Network. Nevertheless, this way one can see all the components and elements of one Artificial Neural Network and get more familiar with the concepts from previous articles.

Artificial Neural Network Structure

Before we dive into the code, let's run through the structure of ANN. In general, Artificial Neural Networks are biologically motivated, meaning that they are trying to mimic the behavior of the real nervous systems. Just like the smallest building unit in the real nervous system is the *neuron*, the same is with artificial neural networks – the smallest building unit is *artificial neuron*. In a real nervous system, these neurons are connected to each other by synapsis, which gives this entire system enormous processing power, ability to learn and huge flexibility. Artificial neural networks apply the same principle.



By connecting artificial neurons they aim to create a similar system. They are grouping neurons into *layers* and then create connections among neurons from each layer. Also, by assigning *weights* to each *connection,* they are able to filter important from non-important connections. The structure of the artificial neuron is a mirroring structure of the real neuron, too. Since they can have multiple inputs, i.e. input connections, a special function that collects that data is used – *input function*. The function that is usually used as input function in neurons is the function that sums all weighted inputs that are active on input connections – *weighted input function.*

Another important part of each artificial neuron is ***activation function***. This function defines whether this neuron will send any signal to its outputs and which value will be propagated to the outputs. Basically, this function receives value from the input function and according to this value it generates an output value and propagates them to the outputs. If you need more details on the architecture of the artificial neural network, you can find it **here**.

Implementation

So, as you can see from the previous chapter there are a few important entities that we need to pay attention to and that we can abstract. They are neurons, connections, layer, and functions. In this solution, a separate class will implement each of these entities. Then, by putting it all

together and adding **backpropagation algorithm** on top of it, we will have our implementation of this simple neural network.

Input Functions

As mentioned before, crucial parts of the neuron are input function and activation function. Let's examine the input function. First I created an interface for this function so it can be easily changed in the neuron implementation later on:

```
1    public interface IInputFunction
2    {
3        double CalculateInput(List<ISynapse> inputs);
4    }
```

IInputFunction.cs hosted with ❤ by GitHub view raw

These functions have only one method – *CalculateInput,* which receives a list of connections which are described in *ISynapse interface*. We will cover this abstraction later; so far all we need to know is that this interface represents connections among neurons. *CalculateInput* method needs to return some sort of value based on the data contained in the list of connections. Then, I did the concrete implementation of input function – *weighted sum function.*

```
1    public class WeightedSumFunction : IInputFunction
2    {
3        public double CalculateInput(List<ISynapse> inputs)
4        {
5            return inputs.Select(x => x.Weight * x.GetOutput()).Sum();
6        }
7    }
```

WeightedSumFunction.cs hosted with ❤ by GitHub view raw

This function sums weighted values on all connections that are passed in the list.

Activation Functions

Taking the same approach as in input function implementation, the interface for activation functions is implemented first:

```
1    public interface IActivationFunction
2    {
3        double CalculateOutput(double input);
4    }
```

IActivationFunction.cs hosted with ❤ by GitHub view raw

After that, concrete implementations can be done. The *CalculateOutput method* should return the output value of the neuron based on input value that it got from input function. I like to have options, so I've done all functions mentioned in **one of the previous blog posts**. Here is how the step function looks:

```
1   public class StepActivationFunction : IActivationFunction
2   {
3       private double _treshold;
4
5       public StepActivationFunction(double treshold)
6       {
7           _treshold = treshold;
8       }
9
10      public double CalculateOutput(double input)
11      {
12          return Convert.ToDouble(input > _treshold);
13      }
14  }
```

StepActivationFunction.cs hosted with ❤ by GitHub view raw

Pretty straightforward, isn't it? A threshold value is defined during the construction of the object, and then the *CalculateOutput* returns 1 if the input value exceeds the threshold value, otherwise, it returns 0.

Other functions are easy as well. Here is the Sigmoid activation function implementation:

```
1   public class SigmoidActivationFunction : IActivationFunction
2   {
3       private double _coeficient;
4
5       public SigmoidActivationFunction(double coeficient)
6       {
7           _coeficient = coeficient;
8       }
9
10      public double CalculateOutput(double input)
11      {
12          return (1 / (1 + Math.Exp(-input * _coeficient)));
```

```
13        }
14    }
```

SigmoidActivationFunction.cs hosted with ♥ by GitHub view raw

And here is Rectifier activation function implementation:

```
1    public class RectifiedActivationFuncion : IActivationFunction
2    {
3        public double CalculateOutput(double input)
4        {
5            return Math.Max(0, input);
6        }
7    }
```

RectifiedActivationFuncion.cs hosted with ♥ by GitHub view raw

So far so good – we have implementations for input and activation function, and we can proceed to implement the trickier parts of the network – neurons and connections.

Neuron

The workflow that a neuron should follow goes like this: Receive input values from one or more weighted input connections. Collect those values and pass them to the activation function, which calculates the output value of the neuron. Send those values to the outputs of the neuron. Based on that workflow abstraction of the neuron this is created:

```
1        public interface INeuron
2        {
3            Guid Id { get; }
4            double PreviousPartialDerivate { get; set; }
5
6            List<ISynapse> Inputs { get; set; }
7            List<ISynapse> Outputs { get; set; }
8
9            void AddInputNeuron(INeuron inputNeuron);
10           void AddOutputNeuron(INeuron inputNeuron);
11           double CalculateOutput();
12
13           void AddInputSynapse(double inputValue);
14           void PushValueOnInput(double inputValue);
15       }
```

INeuron.cs hosted with ♥ by GitHub view raw

Before we explain each property and method, let's see the concrete implementation of a neuron, since that will make the way it works far clearer:

```
1   public class Neuron : INeuron
2   {
3       private IActivationFunction _activationFunction;
4       private IInputFunction _inputFunction;
5
6       /// <summary>
7       /// Input connections of the neuron.
8       /// </summary>
9       public List<ISynapse> Inputs { get; set; }
10
11      /// <summary>
12      /// Output connections of the neuron.
13      /// </summary>
14      public List<ISynapse> Outputs { get; set; }
15
16      public Guid Id { get; private set; }
17
18      /// <summary>
19      /// Calculated partial derivate in previous iteration of training process.
20      /// </summary>
21      public double PreviousPartialDerivate { get; set; }
22
23      public Neuron(IActivationFunction activationFunction, IInputFunction inputFunction)
24      {
25          Id = Guid.NewGuid();
26          Inputs = new List<ISynapse>();
27          Outputs = new List<ISynapse>();
28
29          _activationFunction = activationFunction;
30          _inputFunction = inputFunction;
31      }
32
33      /// <summary>
34      /// Connect two neurons.
35      /// This neuron is the output neuron of the connection.
36      /// </summary>
37      /// <param name="inputNeuron">Neuron that will be input neuron of the newly created conne
38      public void AddInputNeuron(INeuron inputNeuron)
39      {
40          var synapse = new Synapse(inputNeuron, this);
41          Inputs.Add(synapse);
```

```
42            inputNeuron.Outputs.Add(synapse);
43        }
44
45        /// <summary>
46        /// Connect two neurons.
47        /// This neuron is the input neuron of the connection.
48        /// </summary>
49        /// <param name="outputNeuron">Neuron that will be output neuron of the newly created con
50        public void AddOutputNeuron(INeuron outputNeuron)
51        {
52            var synapse = new Synapse(this, outputNeuron);
53            Outputs.Add(synapse);
54            outputNeuron.Inputs.Add(synapse);
55        }
56
57        /// <summary>
58        /// Calculate output value of the neuron.
59        /// </summary>
60        /// <returns>
61        /// Output of the neuron.
62        /// </returns>
63        public double CalculateOutput()
64        {
65            return _activationFunction.CalculateOutput(_inputFunction.CalculateInput(this.Inputs)
66        }
67
68        /// <summary>
69        /// Input Layer neurons just receive input values.
70        /// For this they need to have connections.
71        /// This function adds this kind of connection to the neuron.
72        /// </summary>
73        /// <param name="inputValue">
74        /// Initial value that will be "pushed" as an input to connection.
75        /// </param>
76        public void AddInputSynapse(double inputValue)
77        {
78            var inputSynapse = new InputSynapse(this, inputValue);
79            Inputs.Add(inputSynapse);
80        }
81
82        /// <summary>
83        /// Sets new value on the input connections.
84        /// </summary>
85        /// <param name="inputValue">
```

```
86        /// New value that will be "pushed" as an input to connection.
87        /// </param>
88        public void PushValueOnInput(double inputValue)
89        {
90            ((InputSynapse)Inputs.First()).Output = inputValue;
91        }
92    }
```

Neuron.cs hosted with ❤ by GitHub view raw

Each neuron has its unique identifier – *Id*. This property is used in backpropagation algorithm later. Another property that is added for backpropagation purposes is the *PreviousPartialDerivate*, but this will be examined in detail further on. A neuron has two lists, one for input connections – *Inputs,* and another one for output connections – *Outputs.* Also, it has two fields, one for each of the functions described in previous chapters. They are initialized through the constructor. This way, neurons with different input and activation functions can be created.

This class has some interesting methods, too. *AddInputNeuron* and *AddOutputNeuron* are used to create a connection among neurons. The first one adds input connection to some neuron and the second one adds output connection to some neuron. *AddInputSynapse* adds *InputSynapse* to the neuron, which is a special type of connection. These are special connections that are used just for the input layer of the neuron, i.e. they are used only for adding input to the entirety of the system. This will be covered in more detail in the next chapter.

Last but not least, the *CalculateOutput* method is used to activate a chain reaction of output calculation. What will happen when this function is called? Well, this will call input function, which will request values from all input connections. In turn, these connections will request output values from input neurons of these connections, i.e. output values of neurons from the previous layer. This process will be done until input layer is reached and input values are propagated through the system.

Connections

Connections are abstracted trough *ISynapse* interface:

```
1    public interface ISynapse
2    {
3        double Weight { get; set; }
4        double PreviousWeight { get; set; }
5        double GetOutput();
6
```

```
7        bool IsFromNeuron(Guid fromNeuronId);
8        void UpdateWeight(double learningRate, double delta);
9    }
```

ISynapse.cs hosted with ❤ by GitHub view raw

Every connection has its weight represented through the property of the same name. Additional property *PreviousWeight* is added and it is used during backpropagation of the error through the system. Update of the current weight and storing of the previous one is done in helper function *UpdateWeight.* There is another helper function – *IsFromNeuron*, which detects if a certain neuron is an input neuron to the connection. Of course, there is a method that gets an output value of the connection – G*etOutput.*

Here is the implementation of the connection:

```
1    public class Synapse : ISynapse
2    {
3        internal INeuron _fromNeuron;
4        internal INeuron _toNeuron;
5
6        /// <summary>
7        /// Weight of the connection.
8        /// </summary>
9        public double Weight { get; set; }
10
11       /// <summary>
12       /// Weight that connection had in previous itteration.
13       /// Used in training process.
14       /// </summary>
15       public double PreviousWeight { get; set; }
16
17       public Synapse(INeuron fromNeuraon, INeuron toNeuron, double weight)
18       {
19           _fromNeuron = fromNeuraon;
20           _toNeuron = toNeuron;
21
22           Weight = weight;
23           PreviousWeight = 0;
24       }
25
26       public Synapse(INeuron fromNeuraon, INeuron toNeuron)
27       {
28           _fromNeuron = fromNeuraon;
```

```
29            _toNeuron = toNeuron;
30
31            var tmpRandom = new Random();
32            Weight = tmpRandom.NextDouble();
33            PreviousWeight = 0;
34        }
35
36        /// <summary>
37        /// Get output value of the connection.
38        /// </summary>
39        /// <returns>
40        /// Output value of the connection.
41        /// </returns>
42        public double GetOutput()
43        {
44            return _fromNeuron.CalculateOutput();
45        }
46
47        /// <summary>
48        /// Checks if Neuron has a certain number as an input neuron.
49        /// </summary>
50        /// <param name="fromNeuronId">Neuron Id.</param>
51        /// <returns>
52        /// True - if the neuron is the input of the connection.
53        /// False - if the neuron is not the input of the connection.
54        /// </returns>
55        public bool IsFromNeuron(Guid fromNeuronId)
56        {
57            return _fromNeuron.Id.Equals(fromNeuronId);
58        }
59
60        /// <summary>
61        /// Update weight.
62        /// </summary>
63        /// <param name="learningRate">Chossen learning rate.</param>
64        /// <param name="delta">Calculated difference for which weight of the connection needs to
65        public void UpdateWeight(double learningRate, double delta)
66        {
67            PreviousWeight = Weight;
68            Weight += learningRate * delta;
69        }
70    }
```

Notice the fields *_fromNeuron* and *_toNeuron*, which define neurons that this synapse connects.

Apart from this implementation of the connection, there is another one that I've mentioned in the previous chapter about neurons. It is *InputSynapse* and it is used as an input to the system. The weight of these connections is always 1 and it is not updated during the training process. Here is the implementation of it:

```
 1     public class InputSynapse : ISynapse
 2    {
 3        internal INeuron _toNeuron;
 4
 5        public double Weight { get; set; }
 6        public double Output { get; set; }
 7        public double PreviousWeight { get; set; }
 8
 9        public InputSynapse(INeuron toNeuron)
10        {
11            _toNeuron = toNeuron;
12            Weight = 1;
13        }
14
15        public InputSynapse(INeuron toNeuron, double output)
16        {
17            _toNeuron = toNeuron;
18            Output = output;
19            Weight = 1;
20            PreviousWeight = 1;
21        }
22
23        public double GetOutput()
24        {
25            return Output;
26        }
27
28        public bool IsFromNeuron(Guid fromNeuronId)
29        {
30            return false;
31        }
32
33        public void UpdateWeight(double learningRate, double delta)
34        {
35            throw new InvalidOperationException("It is not allowed to call this method on Input C
36        }
```

```
37    }
```

InputSynapse.cs hosted with ❤ by GitHub view raw

Layer

Implementation of the neural layer is quite easy:

```
1    public class NeuralLayer
2    {
3        public List<INeuron> Neurons;
4
5        public NeuralLayer()
6        {
7            Neurons = new List<INeuron>();
8        }
9
10       /// <summary>
11       /// Connecting two layers.
12       /// </summary>
13       public void ConnectLayers(NeuralLayer inputLayer)
14       {
15           var combos = Neurons.SelectMany(neuron => inputLayer.Neurons, (neuron, input) => new
16           combos.ToList().ForEach(x => x.neuron.AddInputNeuron(x.input));
17       }
18   }
```

NeuralLayer.cs hosted with ❤ by GitHub view raw

It contains the list of neurons used in that layer and the C*onnectLayers method*, which is used to glue two layers together.

Simple Artificial Neural Network

Now, let's put all that together and add backpropagation to it. Take a look at the implementation of the Network itself:

```
1    public class SimpleNeuralNetwork
2    {
3        private NeuralLayerFactory _layerFactory;
4
5        internal List<NeuralLayer> _layers;
6        internal double _learningRate;
7        internal double[][] _expectedResult;
```

```
 8
 9        /// <summary>
10        /// Constructor of the Neural Network.
11        /// Note:
12        /// Initialy input layer with defined number of inputs will be created.
13        /// </summary>
14        /// <param name="numberOfInputNeurons">
15        /// Number of neurons in input layer.
16        /// </param>
17        public SimpleNeuralNetwork(int numberOfInputNeurons)
18        {
19            _layers = new List<NeuralLayer>();
20            _layerFactory = new NeuralLayerFactory();
21
22            // Create input layer that will collect inputs.
23            CreateInputLayer(numberOfInputNeurons);
24
25            _learningRate = 2.95;
26        }
27
28        /// <summary>
29        /// Add layer to the neural network.
30        /// Layer will automatically be added as the output layer to the last layer in the neura
31        /// </summary>
32        public void AddLayer(NeuralLayer newLayer)
33        {
34            if (_layers.Any())
35            {
36                var lastLayer = _layers.Last();
37                newLayer.ConnectLayers(lastLayer);
38            }
39
40            _layers.Add(newLayer);
41        }
42
43        /// <summary>
44        /// Push input values to the neural network.
45        /// </summary>
46        public void PushInputValues(double[] inputs)
47        {
48            _layers.First().Neurons.ForEach(x => x.PushValueOnInput(inputs[_layers.First().Neuro
49        }
50
51        /// <summary>
```

```
52        /// Set expected values for the outputs.
53        /// </summary>
54        public void PushExpectedValues(double[][] expectedOutputs)
55        {
56            _expectedResult = expectedOutputs;
57        }
58
59        /// <summary>
60        /// Calculate output of the neural network.
61        /// </summary>
62        /// <returns></returns>
63        public List<double> GetOutput()
64        {
65            var returnValue = new List<double>();
66
67            _layers.Last().Neurons.ForEach(neuron =>
68            {
69                 returnValue.Add(neuron.CalculateOutput());
70            });
71
72            return returnValue;
73        }
74
75        /// <summary>
76        /// Train neural network.
77        /// </summary>
78        /// <param name="inputs">Input values.</param>
79        /// <param name="numberOfEpochs">Number of epochs.</param>
80        public void Train(double[][] inputs, int numberOfEpochs)
81        {
82            double totalError = 0;
83
84            for(int i = 0; i < numberOfEpochs; i++)
85            {
86                for(int j = 0; j < inputs.GetLength(0); j ++)
87                {
88                    PushInputValues(inputs[j]);
89
90                    var outputs = new List<double>();
91
92                    // Get outputs.
93                    _layers.Last().Neurons.ForEach(x =>
94                    {
95                        outputs.Add(x.CalculateOutput());
```

```
                });

                // Calculate error by summing errors on all output neurons.
                totalError = CalculateTotalError(outputs, j);
                HandleOutputLayer(j);
                HandleHiddenLayers();
            }
        }
    }

    /// <summary>
    /// Hellper function that creates input layer of the neural network.
    /// </summary>
    private void CreateInputLayer(int numberOfInputNeurons)
    {
        var inputLayer = _layerFactory.CreateNeuralLayer(numberOfInputNeurons, new RectifiedA
        inputLayer.Neurons.ForEach(x => x.AddInputSynapse(0));
        this.AddLayer(inputLayer);
    }

    /// <summary>
    /// Hellper function that calculates total error of the neural network.
    /// </summary>
    private double CalculateTotalError(List<double> outputs, int row)
    {
        double totalError = 0;

        outputs.ForEach(output =>
        {
            var error = Math.Pow(output - _expectedResult[row][outputs.IndexOf(output)], 2);
            totalError += error;
        });

        return totalError;
    }

    /// <summary>
    /// Hellper function that runs backpropagation algorithm on the output layer of the netw
    /// </summary>
    /// <param name="row">
    /// Input/Expected output row.
    /// </param>
    private void HandleOutputLayer(int row)
    {
```

```
            _layers.Last().Neurons.ForEach(neuron =>
            {
                neuron.Inputs.ForEach(connection =>
                {
                    var output = neuron.CalculateOutput();
                    var netInput = connection.GetOutput();

                    var expectedOutput = _expectedResult[row][_layers.Last().Neurons.IndexOf(neu

                    var nodeDelta = (expectedOutput - output) * output * (1 - output);
                    var delta = -1 * netInput * nodeDelta;

                    connection.UpdateWeight(_learningRate, delta);

                    neuron.PreviousPartialDerivate = nodeDelta;
                });
            });
        }

        /// <summary>
        /// Hellper function that runs backpropagation algorithm on the hidden layer of the netw
        /// </summary>
        /// <param name="row">
        /// Input/Expected output row.
        /// </param>
        private void HandleHiddenLayers()
        {
            for (int k = _layers.Count - 2; k > 0; k--)
            {
                _layers[k].Neurons.ForEach(neuron =>
                {
                    neuron.Inputs.ForEach(connection =>
                    {
                        var output = neuron.CalculateOutput();
                        var netInput = connection.GetOutput();
                        double sumPartial = 0;

                        _layers[k + 1].Neurons
                        .ForEach(outputNeuron =>
                        {
                            outputNeuron.Inputs.Where(i => i.IsFromNeuron(neuron.Id))
                            .ToList()
                            .ForEach(outConnection =>
                            {
```

```
                                sumPartial += outConnection.PreviousWeight * outputNeuron.Previo
                            });
                        });

                        var delta = -1 * netInput * sumPartial * output * (1 - output);
                        connection.UpdateWeight(_learningRate, delta);
                    });
                });
            }
        }
    }
```

SimpleNeuralNetwork.cs hosted with ❤ by GitHub view raw

This class contains a list of neural layers and a layer factory, a class that is used to create new layers. During construction of the object, initial input layer is added to the network. Other layers are added through the function *AddLayer*, which adds a passed layer on top of the current layer list. The *GetOutput* method will activate the output layer of the network, thus initiating a chain reaction through the network. Also, this class has a few helper methods such as *PushExpectedValues,* which is used to set desired values for the training set that will be passed during training, as well as *PushInputValues,* which is used to set certain input to the network.

The most important method of this class is the *Train* method. It receives the training set and the number of epochs. For each epoch, it runs the whole training set through the network as explained in **this article**. Then, the output is compared with desired output and functions *HandleOutputLayer* and *HandleHiddenLayer* are called. These functions implement backpropagation algorithm as described in **this article**.

Typical Workflow

Typical workflow can be seen in one of the tests implemented in the code on the **repository** – *Train_RuningTraining_NetworkIsTrained.* It goes something like this:

```
var network = new SimpleNeuralNetwork(3);

var layerFactory = new NeuralLayerFactory();
network.AddLayer(layerFactory.CreateNeuralLayer(3, new RectifiedActivationFuncion(), new Weig
network.AddLayer(layerFactory.CreateNeuralLayer(1, new SigmoidActivationFunction(0.7), new We

network.PushExpectedValues(
    new double[][] {
        new double[] { 0 },
```

```
            new double[] { 1 },
            new double[] { 1 },
            new double[] { 0 },
            new double[] { 1 },
            new double[] { 0 },
            new double[] { 0 },
        });

    network.Train(
        new double[][] {
            new double[] { 150, 2, 0 },
            new double[] { 1002, 56, 1 },
            new double[] { 1060, 59, 1 },
            new double[] { 200, 3, 0 },
            new double[] { 300, 3, 1 },
            new double[] { 120, 1, 0 },
            new double[] { 80, 1, 0 },
        }, 10000);

    network.PushInputValues(new double[] { 1054, 54, 1 });
    var outputs = network.GetOutput();
```

Workflow.cs hosted with ❤ by GitHub view raw

Firstly, a neural network object is created. In the constructor, it is defined that there will be three neurons in the input layer. After that, two layers are added using function *AddLayer* and layer factory. For each layer, the number of neurons and functions for each neuron are defined. After this part is completed, the expected outputs are defined and the *Train* function with input training set and the number of epochs is called.

Conclusion

This implementation of the neural network is far from optimal. You will notice plenty of nested for loops which certainly have bad performance. Also, in order to simplify this solution, some of the components of the neural network were not introduced in this first iteration of implementation, momentum and bias, for example. Nevertheless, it was not a goal to implement a network with high performance, but to analyze and display important elements and abstractions that each Artificial Neural Network have.
Thanks 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**.