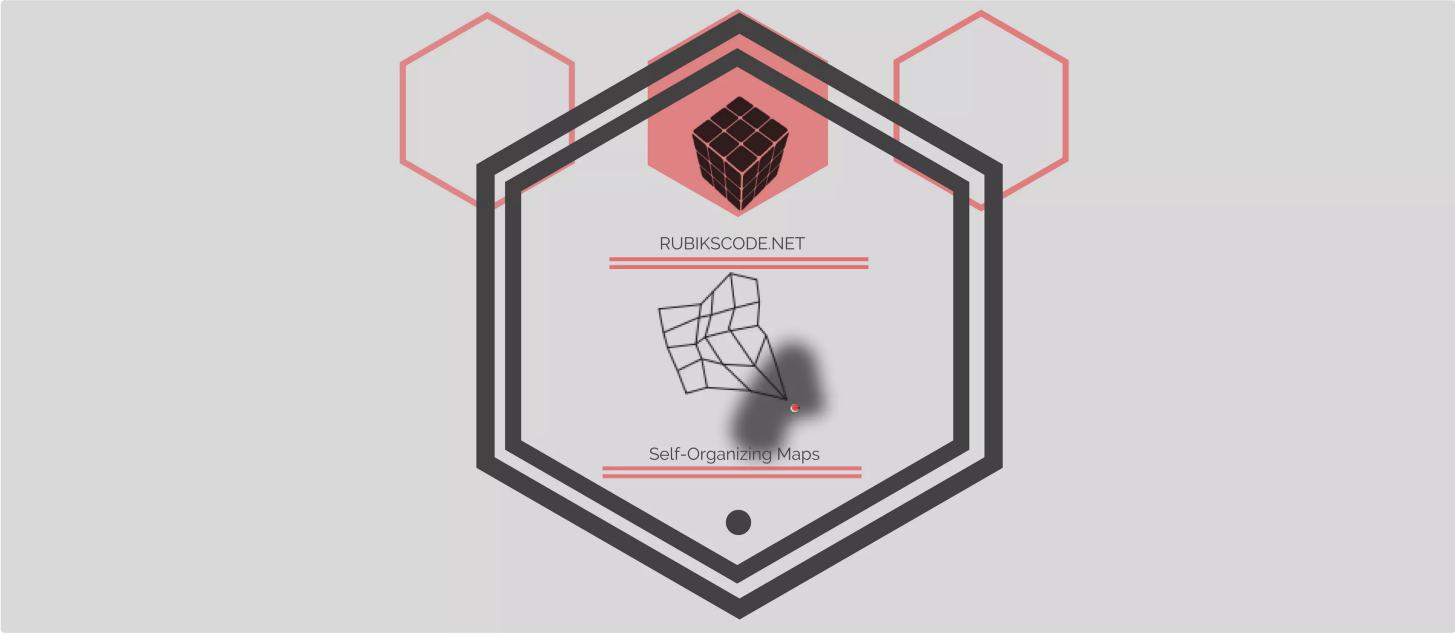
RUBIK'S CODE

Freedom.
Wisdom.
Excellence.

Implementing Self-Organizing Maps with .NET Core

SEPTEMBER 17, 2018 — 2 COMMENTS

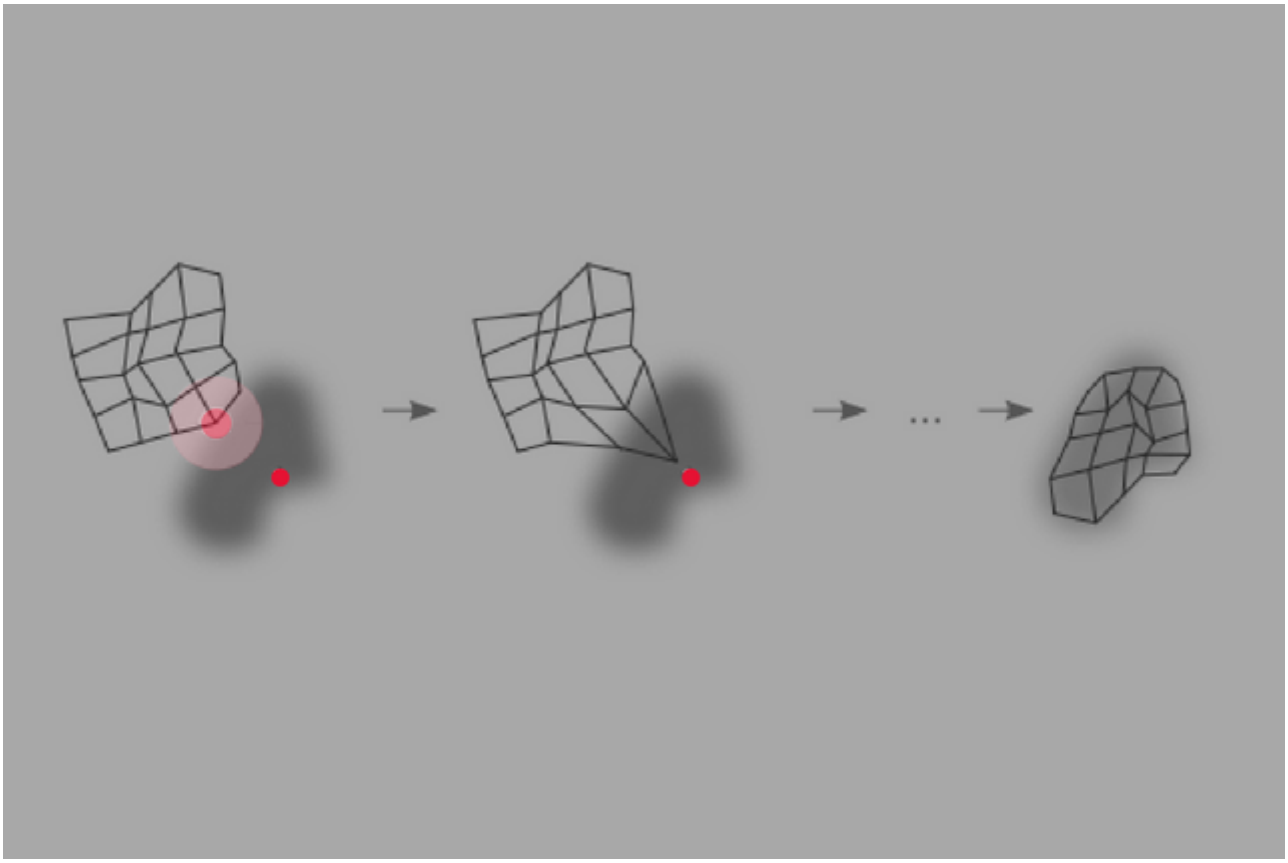


The code that accompanies this article can be downloaded here.

In the previous articles, we explored what Self-Organizing Maps are and how you can implement them using Python and TensorFlow. One of the most interesting things about these networks is that they utilize unsupervised learning, a different type of learning than we got a chance to see during our trip through the world of artificial neural networks. In this type of learning, neural networks don't get the expected result during the training process. Instead, they figure out the relationship between input data on their own. Self-Organizing Maps use this approach for clustering and classification purposes and they are quite good at it.

Apart from that, we had to redefine the concepts of neurons, connection and weights. They have a different meaning in the Self-Organizing Maps world. Neurons are grouped into two collections. The first collection represents input neurons and their number corresponds to the number of features in the dataset.

The second collection represents output neurons, which are usually organized as one or two-dimensional arrays and are triggered only by certain input values. The location of each neuron is also important' this is another thing that is different from the standard feed-forward artificial neural networks. Not only does each neuron have a location, but it is considered that neurons that lie close to each other have similar properties and actually represent a cluster.



Another interesting fact is that every input neuron is connected to every output neuron. This makes the learning process vastly different from what we are used to. These are the main steps of this process for Self-Organizing Maps:

1. Weight initialization
2. The input vector is selected from the dataset and used as an input for the network
3. BMU is calculated
4. The radius of neighbors that will be updated is calculated

5. Each weight of the neurons within the radius is adjusted to make them more like the input vector
6. Steps from 2 to 5 are repeated for each input vector of the dataset

You can check out one of the previous articles in which this process is explained in great detail.

While implementing Self-Organizing Maps with TensorFlow and Python was a lot of fun, I decided to do a similar thing with the C#. This way I am hoping to make these concepts more understandable to the .NET developers.

Prerequisites & Technologies

The solution itself is built using .NET Core and C#. It is created as a class library – *SOM.* Apart from that, unit tests are written using Xunit and Moq in *SOMTests* 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 *SOM* library is divided into three big classes. Each of these classes controls a different aspect of the Self-Organizing Maps. If we take a look into the Self-Organizing Map structure, we can see that we need to model an input layer, an output layer and weighted connections. In addition to that, we need to coordinate the entire learning process using these entities. To sum up, these classes are a part of this solution:

- Vector – Models all vectors in the system, like input neurons and weighted connections.
- Neuron – Models one neuron in the output matrix.
- SOMap – Models Self-Organizing Map itself.

It is important to note that this solution covers only the creation of two-dimensional Self-Organizing maps that receives a one-dimensional array as an input. So, let's take a look at the implementations of different classes.

Vector

his class models all the vectors in the system. It covers the input layer, weighted connections and the input itself. It is modeled as a list of double values, with one extra functionality – *EuclidianDistance.* This function is calculating the Euclidean distance between two vectors. For

testing purposes, we added the interface which looks like this:

```
1    public interface IVector : IList<double>
2    {
3        double EuclidianDistance(IVector vector);
4    }
```

IVector.cs hosted with ❤ by GitHub view raw

As you can see, this interface represents the integration of *IList<double>* interface with the additional functionality that we mentioned previously. The implementation of this interface can be found in *Vector* class:

```
1    public class Vector : List<double>, IVector
2    {
3        public double EuclidianDistance(IVector vector)
4        {
5            if (vector.Count != Count)
6                throw new ArgumentException("Not the same size");
7
8            return this.Select(x => Math.Pow(x - vector[this.IndexOf(x)], 2)).Sum();
9        }
10   }
```

Vector.cs hosted with ❤ by GitHub view raw

In order to cover everything defined in *IList<double>* interface, this class inherits *List<double>* class and implements *IVector* interface. The first thing that implementation of the *EuclidianDistance* function does is a check of the sizes of two vectors. Vectors must have the same size, which means that they have to be in the same dimension if we want to calculate the Euclidean distance for them. We cannot do this on one two-dimensional and one three-dimensional array, and that is why this check is performed. After that, this function does the calculation itself.

Neuron

Quite obviously, this class is in charge of modeling one of the neurons from the output layer of the Self-Organizing Map. The objects of this class will represent one of the elements in the matrix. The neuron itself is described by the interface *INeuron*:

```
1    public interface INeuron
2    {
3        int X { get; set; }
```

```
4        int Y { get; set; }
5        IVector Weights { get; }
6
7        double Distance(INeuron neuron);
8        void SetWeight(int index, double value);
9        double GetWeight(int index);
10       void UpdateWeights(IVector input, double distanceDecay, double learningRate);
11   }
```

INeuron.cs hosted with ❤ by GitHub view raw

As you can see, every neuron has an *X* and a *Y* coordinate. These properties determine the position of the neuron in the matrix. Neurons that are close to each other have similar characteristics, which means that these properties are very important. Apart from that, every neuron contains *Vector* object for weighted connections. This object represents all the weighted connections that are attached to this neuron. Also, this interface describes a set of functions that each neuron should implement:

- *Distance* – Calculates the distance from that neuron to another neuron in the matrix.
- *SetWeight* – Sets a value to a weight defined by the index.
- *GetWeight* – Retrieves a value of a weight defined by the index.
- *UpdateWeights* – Updates weights of a neuron based on the input, learning rate and the distance decay.

The implementation of this interface is a part of the *Neuron* class and it looks like this:

```
1    public class Neuron : INeuron
2    {
3        public int X { get; set; }
4        public int Y { get; set; }
5        public IVector Weights { get; }
6
7        public Neuron(int numOfWeights)
8        {
9            var random = new Random();
10           Weights = new Vector();
11
12           for (int i = 0; i < numOfWeights; i ++)
13           {
14               Weights.Add(random.NextDouble());
15           }
16       }
```

```
17
18        public double Distance(INeuron neuron)
19        {
20            return Math.Pow((X - neuron.X), 2) + Math.Pow((Y - neuron.Y), 2);
21        }
22
23        public void SetWeight(int index, double value)
24        {
25            if (index >= Weights.Count)
26                throw new ArgumentException("Wrong index!");
27
28            Weights[index] = value;
29        }
30
31        public double GetWeight(int index)
32        {
33            if (index >= Weights.Count)
34                throw new ArgumentException("Wrong index!");
35
36            return Weights[index];
37        }
38
39        public void UpdateWeights(IVector input, double distanceDecay, double learningRate)
40        {
41            if(input.Count != Weights.Count)
42                throw new ArgumentException("Wrong input!");
43
44            for(int i = 0; i < Weights.Count; i++)
45            {
46                Weights[i] += distanceDecay * learningRate * (input[i] - Weights[i]);
47            }
48
49        }
50    }
```

Neuron.cs hosted with ♥ by GitHub view raw

In the constructor of this class, weights of the connections are initialized to random values. *Distance* method is implemented in a way that it returns the Euclidean distance between neurons in the matrix, based on the coordinates *X* and *Y*. The *SetWeight* and *GetWeight* check the index value within the range and after that, either set or retrieve the value of the weight. *UpdateWeights* method first checks if the input is in the proper dimension, and then it uses a formula defined in this article, to update weights on each connection of the neuron.

SOMap

Finally, let's check out the implementation of the *SOMap* class. This class wraps other elements into a cohesive unity, utilizes them and implements the learning process on top of that. We tried to keep the exposed API similar to the one we can see in the TensorFlow implementation. Here is how that implementation looks:

```
using System;
using SOM.NeuronNamespace;
using SOM.VectorNamespace;

namespace SOM
{
    public class SOMap
    {
        internal INeuron[,] _matrix;
        internal int _height;
        internal int _width;
        internal double _matrixRadius;
        internal double _numberOfIterations;
        internal double _timeConstant;
        internal double _learningRate;

        public SOMap(int width, int height, int inputDimension, int numberOfIterations, doub
        {
            _width = width;
            _height = height;
            _matrix = new INeuron[_width, _height];
            _numberOfIterations = numberOfIterations;
            _learningRate = learningRate;

            _matrixRadius = Math.Max(_width, _height) / 2;
            _timeConstant = _numberOfIterations / Math.Log(_matrixRadius);

            InitializeConnections(inputDimension);
        }

        public void Train(Vector[] input)
        {
            int iteration = 0;
            var learningRate = _learningRate;

            while (iteration < _numberOfIterations)
            {
```

```
38                  var currentRadius = CalculateNeighborhoodRadius(iteration);
39
40                  for (int i = 0; i < input.Length; i++)
41                  {
42                      var currentInput = input[i];
43                      var bmu = CalculateBMU(currentInput);
44
45                      (int xStart, int xEnd, int yStart, int yEnd) = GetRadiusIndexes(bmu, cur
46
47                      for (int x = xStart; x < xEnd; x++)
48                      {
49                          for (int y = yStart; y < yEnd; y++)
50                          {
51                              var processingNeuron = GetNeuron(x, y);
52                              var distance = bmu.Distance(processingNeuron);
53                              if (distance <= Math.Pow(currentRadius, 2.0))
54                              {
55                                  var distanceDrop = GetDistanceDrop(distance, currentRadius);
56                                  processingNeuron.UpdateWeights(currentInput, learningRate, d
57                              }
58                          }
59                      }
60                  }
61                  iteration++;
62                  learningRate = _learningRate * Math.Exp(-(double)iteration / _numberOfIterat
63              }
64          }
65
66          internal (int xStart, int xEnd, int yStart, int yEnd) GetRadiusIndexes(INeuron bmu, (
67          {
68              var xStart = (int)(bmu.X - currentRadius - 1);
69              xStart = (xStart < 0) ? 0 : xStart;
70
71              var xEnd = (int)(xStart + (currentRadius * 2) + 1);
72              if (xEnd > _width) xEnd = _width;
73
74              var yStart = (int)(bmu.Y - currentRadius - 1);
75              yStart = (yStart < 0) ? 0 : yStart;
76
77              var yEnd = (int)(yStart + (currentRadius * 2) + 1);
78              if (yEnd > _height) yEnd = _height;
79
80              return (xStart, xEnd, yStart, yEnd);
81          }
```

```
82
83          internal INeuron GetNeuron(int indexX, int indexY)
84          {
85              if (indexX > _width || indexY > _height)
86                  throw new ArgumentException("Wrong index!");
87
88              return _matrix[indexX, indexY];
89          }
90
91          internal double CalculateNeighborhoodRadius(double itteration)
92          {
93              return _matrixRadius * Math.Exp(-itteration/_timeConstant);
94          }
95
96          internal double GetDistanceDrop(double distance, double radius)
97          {
98              return Math.Exp(-(Math.Pow(distance, 2.0) / Math.Pow(radius, 2.0)));
99          }
100
101         internal INeuron CalculateBMU(IVector input)
102         {
103             INeuron bmu = _matrix[0, 0];
104             double bestDist = input.EuclidianDistance(bmu.Weights);
105
106             for (int i = 0; i < _width; i++)
107             {
108                 for (int j = 0; j < _height; j++)
109                 {
110                     var distance = input.EuclidianDistance(_matrix[i, j].Weights);
111                     if( distance < bestDist)
112                     {
113                         bmu = _matrix[i, j];
114                         bestDist = distance;
115                     }
116                 }
117             }
118
119             return bmu;
120         }
121
122         private void InitializeConnections(int inputDimension)
123         {
124             for (int i = 0; i < _width; i++)
125             {
```

```
126                     for (int j = 0; j < _height; j++)
127                     {
128                         _matrix[i, j] = new SOM.NeuronNamespace.Neuron(inputDimension) { X = i,
129                     }
130                 }
131             }
132         }
133     }
```

SOMap.cs hosted with ♥ by GitHub view raw

The only publicly exposed members of this class are the constructors and the *Train* method. The constructor is used to receive dimensions and initialize the output matrix, as well as to handle some of the initial values for learning rate and a number of iterations. The *Train* method is one interesting method which implements the unsupervised learning process of Self-Organizing Maps.

It is separated into several private methods that handle different aspects of this process. In essence, this method is one big loop that runs for a defined number of iterations. In each iteration, the first thing that is done is a calculation of neighborhood radius. This is done in *CalculateNeighborhoodRadius* method. This is driven by the number of iterations that had been run so far as well as the total number of iterations and dimensions of the output matrix.

After this number is calculated, the BMU is determined for each input vector. For this purpose, *CalculateBMU* method is used. Afterward, the indexes of the neurons that are within radius are calculated. For each neuron within this radius, the distance from the BMU is calculated, which is used for calculating the new value for the distance decay. Finally, this value, along with the value of the learning rate is used to update the weights on each connection of the neuron. At the end of the *Train* method, the learning rate is updated after each iteration.

Using the Library

The API of this library is pretty simple and straightforward. All you have to do is model your input as *Vector* objects, create a *SOMap* object and call the *Train* method – something along these lines:

```
1    var inputVector = new Vector();
2    inputVector.Add(2);
3    inputVector.Add(2);
4
5    var input = new Vector[10]
6    {
7        inputVector,
8        inputVector,
9        inputVector,
10       inputVector,
11       inputVector,
12       inputVector,
13       inputVector,
14       inputVector,
15       inputVector,
16       inputVector
17   };
18
19
20   var som = new SOMap(2, 2, inputVector.Count, 100, 0.5);
```

```
21    som.Train(input);
```

SOMUseCase.cs hosted with ❤ by GitHub view raw

First, the test input vector is created and a complete input dataset is created by repeating that value. These are just test values, used as an example. Then a 2×2 Self-Organizing Map is created using *SOMap* constructor. We also define that training will take 100 iterations and that the start learning rate should be 0.5. After that, the *Train* method is called with input we generated at the beginning.

Conclusion

The goal of this article was to make the concepts of Self-Organizing Maps 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 Self-Organizing Maps, such as neurons and vectors, and how to implement the unsupervised 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 see discover how to solve a real-world problem using Self-Organizing Maps.

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**.