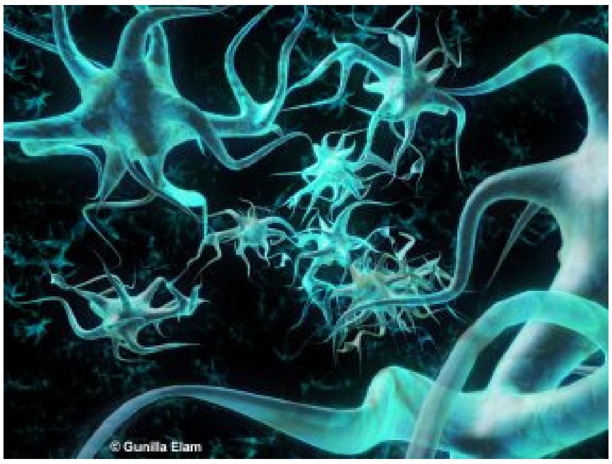
Neural Networks on C#

By **Andrew Kirillov**

Posted: **19 Nov 2006**
Views: **140,524**
Bookmarked: **304 times**

The articles describes a C# library for neural network computations, and their application for several problem solving.

Download source files - 251 Kb
Download demo project - 181 Kb



Introduction

It is known fact, that there are many different problems, for which it is difficult to find formal algorithms to solve them. Some problems cannot be solved easily with traditional methods; some problems even do not have a solution yet. For many such problems, neural networks can be applied, which demonstrate rather good results in a great range of them. The history of neural networks starts in 1950-ies, when the simplest neural network's architecture was presented. After the initial work in the area, the idea of neural networks became rather popular. But then the area had a crash, when it was discovered that neural networks of those times are very limited in terms of the amount of tasks they can be applied to. In 1970-ies, the area got another boom, when the idea of multi-layer neural networks with the back propagation learning algorithm was presented. From that time, many different researchers have studied the area of neural networks, what lead to a vast range of different neural architectures, which were applied to a great range of different problems. For now, neural networks can be applied to such tasks, like classification, recognition, approximation, prediction, clusterization, memory simulation, and many other different tasks, and their amount is growing.

In this article, a C# library for neural network computations is described. The library implements several popular neural network architectures and their training algorithms, like Back Propagation, Kohonen Self-Organizing Map, Elastic Network, Delta Rule Learning, and Perceptron Learning. The usage of the library is demonstrated on several samples:

Component Catalog
Free Magazines

FEATURE ZONES
Product Showcase
IBM DeveloperWorks
WhitePapers / Webcasts
.NET Bug Tracking
ASP.NET Web Hosting

- Classification (one-layer neural network trained with perceptron learning algorithms);
- Approximation (multi-layer neural network trained with back propagation learning algorithm);
- Time Series Prediction (multi-layer neural network trained with back propagation learning algorithm);
- Color Clusterization (Kohonen Self-Organizing Map);
- Traveling Salesman Problem (Elastic Network).

The attached archives contain source codes for the entire library, all the above listed samples, and some additional samples which are not listed and discussed in the article.

The article is not intended to provide the entire theory of neural networks, which can be found easily on the great range of different resources all over the Internet, and on CodeProject as well. Instead of this, the article assumes that the reader has general knowledge of neural networks, and that is why the aim of the article is to discuss a C# library for neural network computations and its application to different problems.

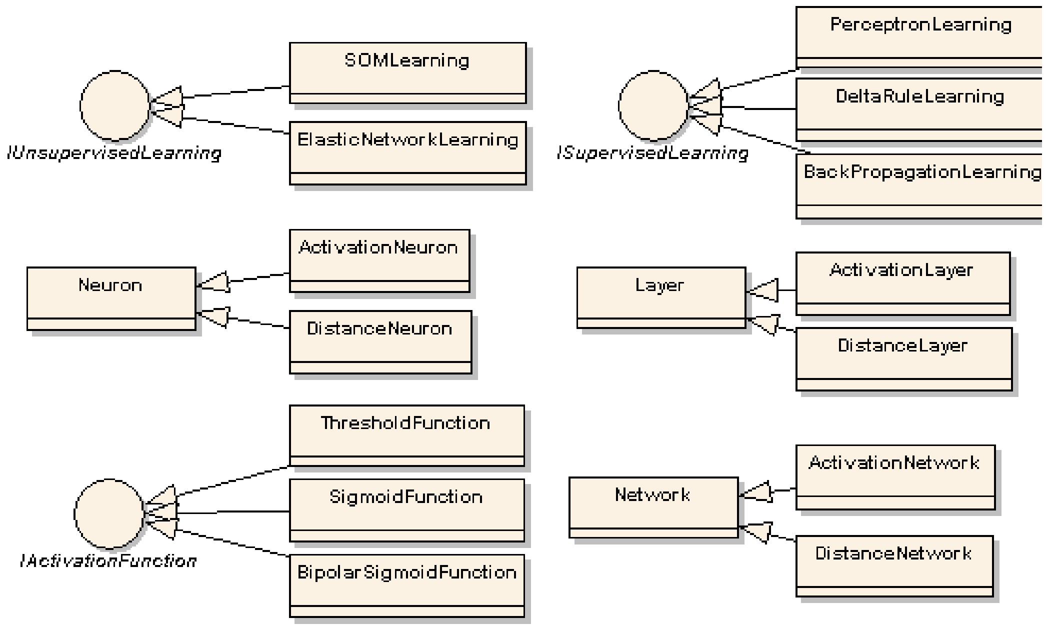
Using the library

Designing the library, one of the main ideas was to make it flexible, reusable, and easy to use and understand. Instead of combining several neural network entities into a single class and making a mess, which leads to loosing flexibility and clarity in the code and design, all entities were split into distinct classes, making them easer to understand and reuse. Some neural networks libraries tend to combine the entity of neuron's network together with the learning algorithm, what makes it hard to develop another learning algorithm which can be applied to the same neural network architecture. Some other libraries and applications do not extract such entities, like neurons, layers of neurons, or a network of layers, but implement the entire neuron network architecture in a single class. In some cases, it is arguable what is better, because there may be such unusual neural network architectures, where it is hard to split the network into layers and neurons. In some other cases, networks do not tend to multi-layer architecture, so it may be useless to have an additional entity like layer. But in most cases, it is favorable to split all these entities into distinct classes, what leads not only to easier understanding, but also allows reusing of all these components and building new neural networks architectures from smaller generic pieces.

The library contains six main entities:

- `Neuron` - a base abstract class for all neurons, which encapsulates such common entities like a neuron's weight, output value, and input value. Other neuron classes inherit from the base class to extend it with additional properties and specialize it.
- `Layer` - represents a collection of neurons. This is a base abstract class, which encapsulates common functionality for all neuron's layers.
- `Network` - represents a neural network, what is a collection of neuron's layers. This is a base abstract class, which provides common functionality of a generic neural network. To implement a specific neural network architecture, it is required to inherit the class, extending it with specific functionalities of any neural network architecture.
- `IActivationFunction` - activation function's interface. Activation functions are used in activation neurons - the type of neuron, where the weighted sum of its inputs is calculated and then the value is passed as input to the activation function, and the output value becomes the output value of the neuron.
- `IUnsupervisedLearning` - interface for unsupervised learning algorithms - the type of learning algorithms where a system is provided with sample inputs only during the learning phase, but not with the desired outputs. The aim of the system is to organize itself in such a way to find correlation and similarities between data samples.
- `ISupervisedLearning` - interface for supervised learning algorithms - the type of learning algorithms where a system is provided with sample inputs,

with desired output values during the learning phase. The aim of the system is to generalize learning data, and learn to provide the correct output value when it is presented with the input value only.



The library provides the following neural network architectures:

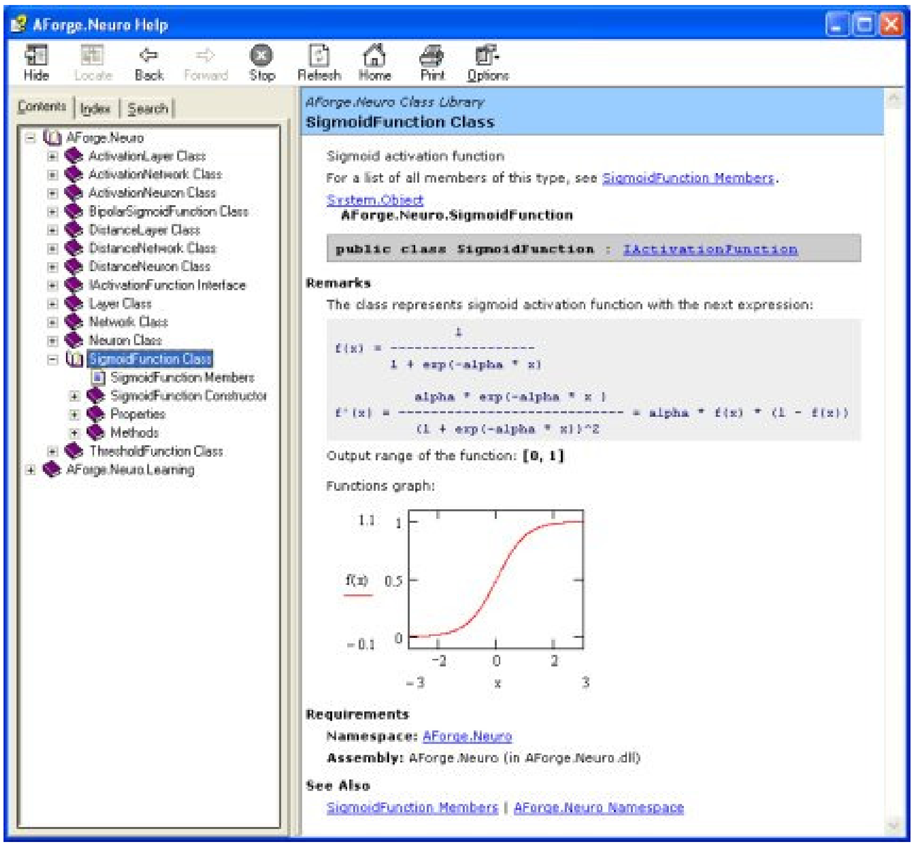
- `Activation Network` - the neural network where each neuron computes its output as the activation function's output, and the argument is a weighted sum of its inputs combined with the threshold value. The network may consist of a single layer, or of multiple layers. Trained with supervised learning algorithms, the network allows to solve such tasks as approximation, prediction, classification, and recognition.
- `Distance Network` - the neural network where each neuron computes its output as a distance between its weight values and input values. The network consists of a single layer, and may be used as a base for such networks like Kohonen Self Organizing Map, Elastic Network, and Hamming Network.

Different learning algorithms are used to train different neural networks, and are used to solve different problems:

- `Perceptron Learning` [1] - the algorithm may be considered as the first neural network learning algorithm, and its history starts from 1957. The algorithm may be used with a one-layer activation network, where each neuron has a threshold activation function. The range of its applications are rather small and limited the with classification of linearly separable data.
- `Delta Rule Learning` [2] - the algorithm is a next step after the perceptron learning algorithm. It utilizes the activation function's derivative, and may be applicable to single-layer activation networks only, where each neuron has a continuous activation function instead of a threshold activation function. The most popular continuous activation function is the unipolar and bipolar sigmoid function. Because the algorithm may be applied to one-layer networks only, it is limited to some classification and recognition tasks mostly.
- `Back Propagation Learning` [3] - this is one of the most popular and known algorithms for multi-layer neural network learning. Initially, it was described in 1974, and from that time, it was extensively studied and applied to a broad range of different tasks. Because the algorithm is able to train multi-layer neural networks, the range of its applications is very great, and includes such tasks as approximation, prediction, object recognition, etc.
- `SOM Learning` [4] - this algorithm was developed by Kohonen, and may be considered as one of the most famous unsupervised learning algorithms for clusterization problems. It treats neural network as a 2D map of nodes,

where each node may represent a separate class. The algorithm organizes a network in such a way, that it becomes possible to find the correlation and similarities between data samples.

- Elastic Network Learning [5] - the algorithm is similar to the idea of the SOM learning algorithm, but it treats network neurons not as a 2D map of nodes, but as a ring. During the learning procedure, the ring gets some shape, which represents a solution. One of the most common demonstrations of this learning algorithm is the Traveling Salesman Problem (TSP).



AForge.Neuro Class Library

SigmoidFunction Class

Sigmoid activation function

For a list of all members of this type, see SigmoidFunction Members.

System.Object
  AForge.Neuro.SigmoidFunction

```
public class SigmoidFunction : IActivationFunction
```

Remarks

The class represents sigmoid activation function with the next expression:

```
                 1
f(x) = ------------------
        1 + exp(-alpha * x)

           alpha * exp(-alpha * x )
f'(x) = ---------------------------- = alpha * f(x) * (1 - f(x))
           (1 + exp(-alpha * x))^2
```

Output range of the function: **[0, 1]**

Functions graph:

Requirements

Namespace: AForge.Neuro

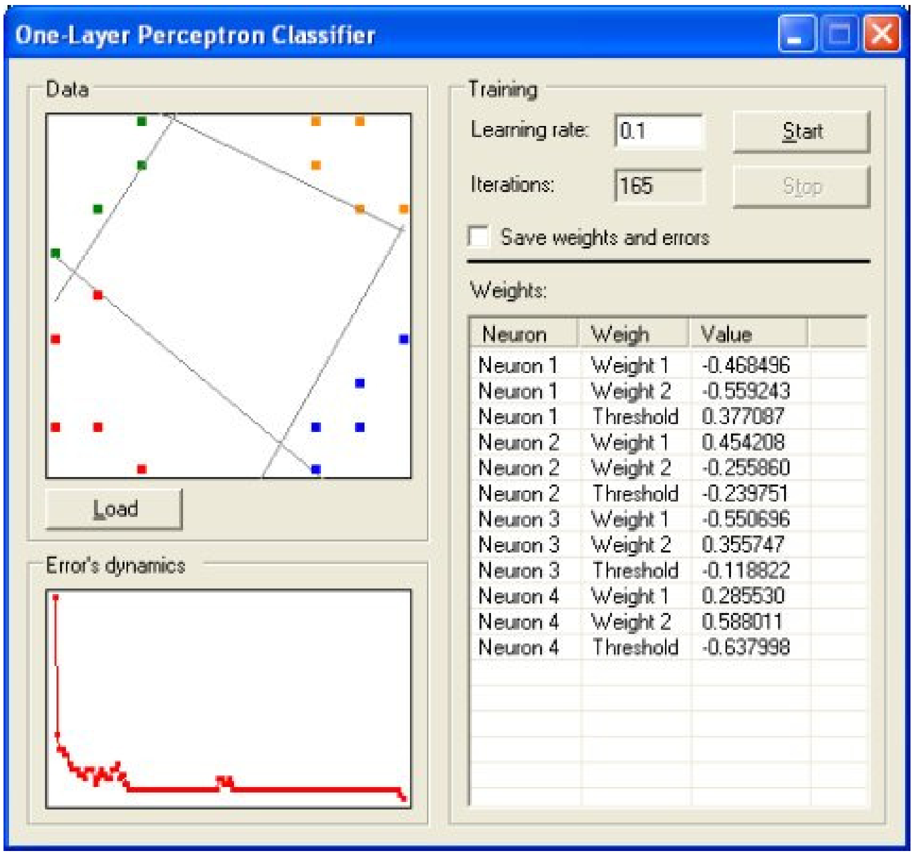
Assembly: AForge.Neuro (in AForge.Neuro.dll)

See Also

SigmoidFunction Members | AForge.Neuro Namespace

For additional source of information, the library is provided with help information, which is distributed in HTML help format.

Classification



Learning rate: 0.1

Iterations: 165

☐ Save weights and errors

Neuron	Weigh	Value
Neuron 1	Weight 1	-0.468496
Neuron 1	Weight 2	-0.559243
Neuron 1	Threshold	0.377087
Neuron 2	Weight 1	0.454208
Neuron 2	Weight 2	-0.255860
Neuron 2	Threshold	-0.239751
Neuron 3	Weight 1	-0.550696
Neuron 3	Weight 2	0.355747
Neuron 3	Threshold	-0.118822
Neuron 4	Weight 1	0.285530
Neuron 4	Weight 2	0.588011
Neuron 4	Threshold	-0.637998

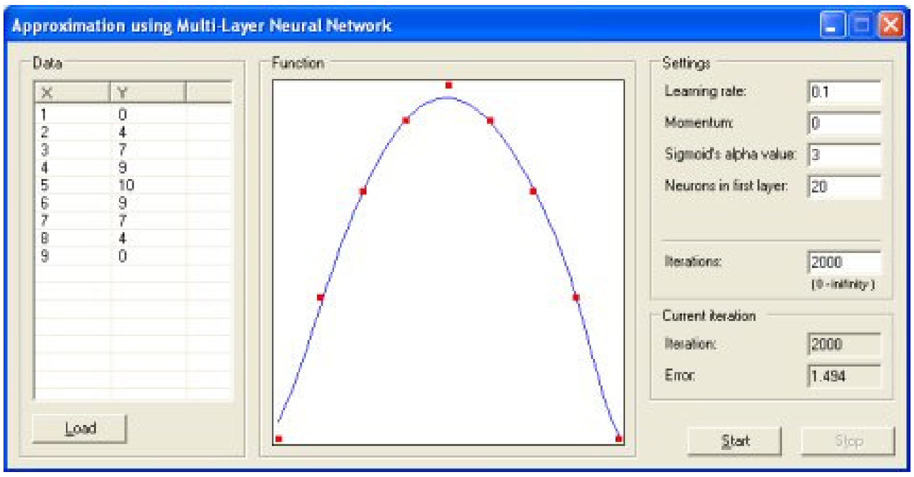
This sample demonstrates the use of a one-layer activation network with a threshold activation function and the use of the perceptron learning algorithm. The neuron's amount of the network is equal to the amount of different data classes, and each neuron is trained to classify certain classed only. Passing a data sample to the trained network, one neuron of the network should get activated (produce output equal to 1), but all other neurons should get deactivated (produce output equal to 0). The class of the data sample is determined by the activated neuron's number. In the case, if several neurons become activated or none of them, the network is unable to classify the presented data sample correctly. In the case of 2D data samples, it is very easy to visualize the network, because weights and threshold value of each neuron represents a line, which separates one class from all others.

⊟ Collapse

```
// prepare learning data

double[][] input = new double[samples][];
double[][] output = new double[samples][];
// ... preparing the data ...


// create perceptron

ActivationNetwork network = new ActivationNetwork( new ThresholdFunction( )
                                                   2, classesCount );
// create teacher

PerceptronLearning teacher = new PerceptronLearning( network );
// set learning rate

teacher.LearningRate = learningRate;
// loop

while ( ... )
{
    // run epoch of learning procedure

    double error = teacher.RunEpoch( input, output );
    ...
}
```

Despite the network's architecture simplicity, it can be used for many different classification/recognition tasks. The only limitation of this architecture is that the network may classify only linearly separable data.

Approximation



Approximation using Multi-Layer Neural Network

Data

X	Y
1	0
2	4
3	7
4	9
5	10
6	9
7	7
8	4
9	0

Settings

Learning rate:	0.1
Momentum:	0
Sigmoid's alpha value:	3
Neurons in first layer:	20
Iterations:	2000 (0 - infinity)

Current iteration

Iteration:	2000
Error:	1.494

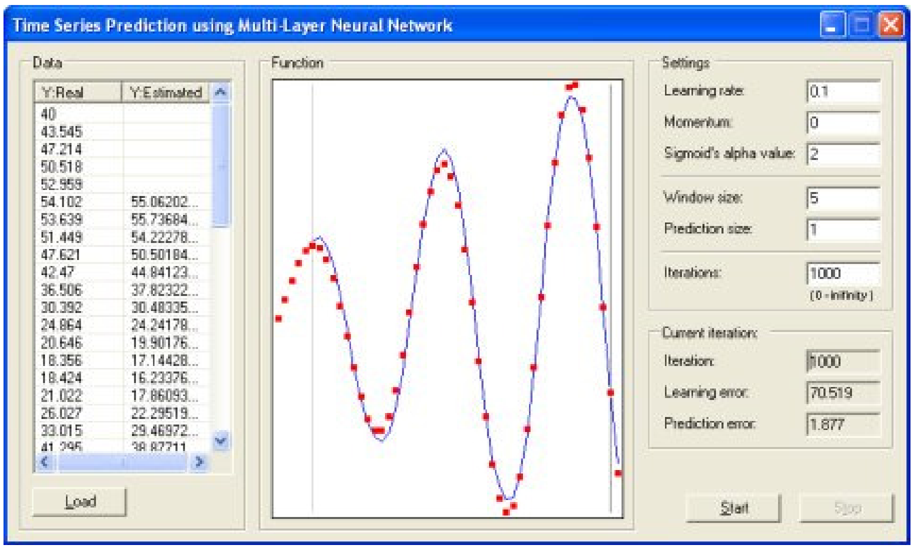
This sample demonstrates the use of multi-layer neural networks trained with the back propagation algorithm, which is applied to a function's approximation problem. Suppose that we know the function's value only in a limited amount of points, but we would like to calculate the function in some other points as well, which are in the range of `min` and `max X` values, for which we know the value of the function. During the training phase, the network is trained to produce the correct function's value for the presented `X` value. After the training is done, the network is used to calculate the function's value for different other values which were not available for the network during the training procedure.

⊟ Collapse

```
// prepare learning data

double[][] input = new double[samples][];
double[][] output = new double[samples][];
// ... preparing the data ...

// create multi-layer neural network

ActivationNetwork    network = new ActivationNetwork(
    new BipolarSigmoidFunction( sigmoidAlphaValue ),
    1, neuronsInFirstLayer, 1 );
// create teacher

BackPropagationLearning teacher = new BackPropagationLearning( network );
// set learning rate and momentum

teacher.LearningRate = learningRate;
teacher.Momentum     = momentum;
// loop

while ( ... )
{
    // run epoch of learning procedure

    double error = teacher.RunEpoch( input, output ) / samples;
    ...
}
```

The network architecture may be used for approximation of not just two-dimensional functions. A function of arbitrary dimensions may be approximated with a multi-later neural network. But do not forget that the amount of the network's layers and neurons and such parameters like sigmoid's alpha value may affect the learning speed very drastically. Some incorrect settings of your network may lead to the inability to learn anything at all. So, be careful and try to experiment more.

Time Series Prediction



Y:Real	Y:Estimated
40	
43.545	
47.214	
50.518	
52.959	
54.102	55.06202...
53.639	55.73684...
51.449	54.22278...
47.621	50.50184...
42.47	44.84123...
36.506	37.82322...
30.392	30.48335...
24.864	24.24178...
20.646	19.90176...
18.356	17.14428...
18.424	16.23376...
21.022	17.86093...
26.027	22.29519...
33.015	29.46972...
41.295	38.87711

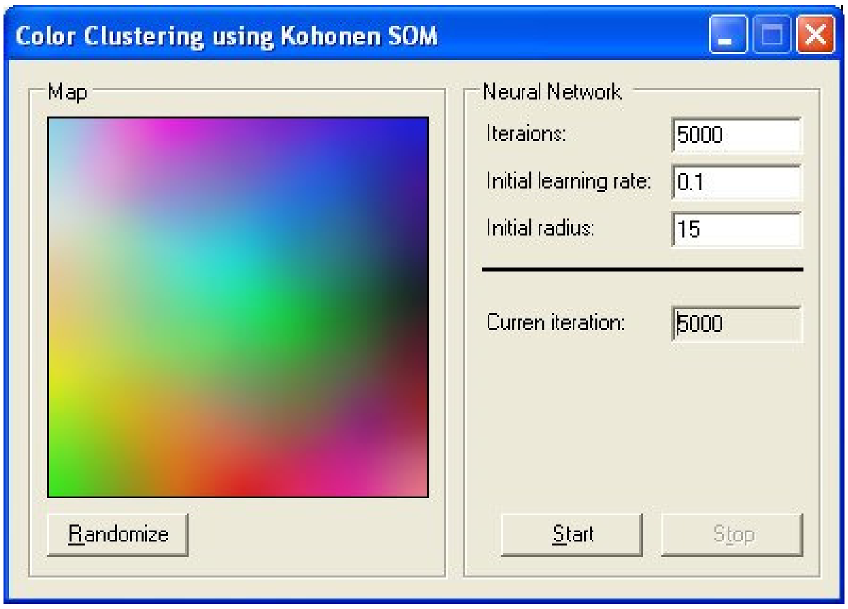
Learning rate: 0.1
Momentum: 0
Sigmoid's alpha value: 2
Window size: 5
Prediction size: 1
Iterations: 1000 (0 - infinity)

Iteration: 1000
Learning error: 70.519
Prediction error: 1.877

This sample also demonstrates a multi-layer neural network with back propagation learning algorithm, but applied to a different task - time series prediction. The problem of time series prediction is very important and a very popular problem, and many researchers work in the area trying many different algorithms and methods for the task. It is easy to explain the popularity of the problem by taking a look at areas where it can be applied. One of the popular areas is trades - if you can predict future values of stocks or rates of currency exchanges, then ... if you can do it really well, then you may become a rich man. But, let's return to neural networks. During the training phase, certain amount of previous values of the time series are presented to the network, and the network is trained to predict the next value of the time series. The more training samples you have, the better prediction models you may get. Also, a very important parameter is window size - how many values from the history are used to predict the future one. The larger the window size you have, the better model you may get, but may not - depending on the time series, and require experiments. But, a larger window size will decrease the amount of training samples you need, so it is a trade off value.

The sample code for this problem is the same like in the previous example, only the procedure of preparing the learning data is changed.

Color Clusterization



Iteraions: 5000
Initial learning rate: 0.1
Initial radius: 15
Curren iteration: 5000

This is a very simple sample, which demonstrates the process of the Kohonen Self-Organizing Map's organization. A square SOM network is created, which consists of 10000 neurons with randomly initialized weights. Each neuron has three weights, which are interpreted as `RGB` values for visualization. So, the initial visualization of the network will show some sort of noisy colored map. During the learning procedure, random colors are picked and passed to the network's input one by one. Repeating learning iterations, the neural network organizes itself in such a way that it no longer looks like a random picture during visualization, but it gets a certain structure - colors, which are close on the RGB palette, also placed closer on the Kohonen map too.

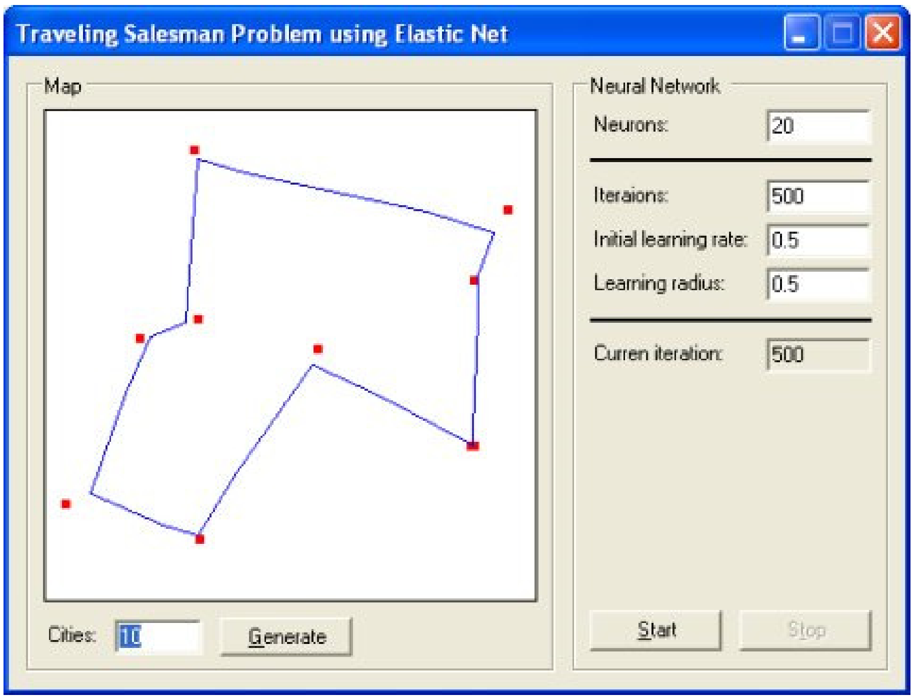
⊟ Collapse

```
// set neurons weights randomization range

Neuron.RandRange = new DoubleRange( 0, 255 );
// create network

DistanceNetwork network = new DistanceNetwork( 3, 100 * 100 );
// create learning algorithm

SOMLearning trainer = new SOMLearning( network );
// input

double[] input = new double[3];
// loop

while ( ... )
{
    // update learning rate and radius

    // ...


    // prepare network input

    input[0] = rand.Next( 256 );
    input[1] = rand.Next( 256 );
    input[2] = rand.Next( 256 );

    // run learning iteration

    trainer.Run( input );

    ...
}
```

Playing with this sample, you will find that the neural network creates different pictures always, and it is rather interesting to observe the process of the map's organization. May be an idea for screen saver ...

Instead of colors, any other object could be used, which could be described with a real-valued vector of features. Passing these feature vectors to the Kohonen map during the learning procedure, will lead to its organization, so after the training procedure is finished, you can take a look at the 2D map, and discover which objects are similar and close to each other according to their features.

Traveling Salesman Problem



Neurons: 20
Iteraions: 500
Initial learning rate: 0.5
Learning radius: 0.5
Curren iteration: 500

The traveling salesman problem demonstrates the use of the Elastic Network, which is similar to SOM in the concept of self-organizing, but differs in the neural network's interpretation. SOM interprets a neural network as 2D map of nodes, but the Elastic Network interprets it as a ring of nodes. During the training phase, feature vectors are presented to the network one by one, what makes the network to get some sort of a shape, which represents a solution. In the case of the TSP problem, each neuron of the network has two weights, which represent `(X, Y)` coordinates. During the training phase, coordinates of arbitrary cities are passed to the network's input one by one, and the network organizes its weights in such a way, that they represent a path of the traveling agent.

⊟ Collapse

```
// set random generators range

Neuron.RandRange = new DoubleRange( 0, 1000 );
// create network

DistanceNetwork network = new DistanceNetwork( 2, neurons );
// create learning algorithm

ElasticNetworkLearning trainer = new ElasticNetworkLearning( network );
// input

double[] input = new double[2];
// loop

while ( ... )
{
    // set network input

    int currentCity = rand.Next( citiesCount );
    input[0] = map[currentCity, 0];
    input[1] = map[currentCity, 1];

    // run one training iteration

    trainer.Run( input );
    ...
}
```

The disadvantage of this method is that it does not provide an exact solution - the neuron's weights may be close to city coordinates, but may be not equal to them. Another disadvantage of this method is that it does not produce a good result on a large number of cities. But still, the method is a nice demonstration of the neural network's application.

Conclusion

The five presented samples demonstrate that the initial aim of the library was achieved - it is flexible, reusable, and it is easy to use it for different tasks. Although, there is still much work to do, because of a great range of different neural network architectures and their learning algorithms, but still - the library can be used for many different problems, and can be extended to solve even more. I hope the library will become useful not only in my further research work, but other different researchers will find it interesting and useful.

References

1. Perceptron, from Wikipedia, the free encyclopedia.
2. Delta Rule, from Wikipedia, the free encyclopedia.
3. Back Propagation, from Wikipedia, the free encyclopedia.
4. Self-Organizing Map, from Wikipedia, the free encyclopedia.
5. Elastic Net Tutorial, Andrei Cimponeriu, October 12, 1999.

History

- [19.11.2006] - Initial publication of the article.

License

About the Author

Andrew Kirillov

Occupation: Software Developer (Senior)
Location: Latvia

Other popular Algorithms & Recipes articles:

- Symbolic Differentiation
  This article demonstrates differentiating expressions using a stack and displaying the input expression and its derivative.
- Manipulating colors in .NET - Part 1
  Understand and use color models in .NET
- C#: A-Star is born
  Understand graphs and A* path-finding algorithm with C#
- Fast mathematical expressions parser
  Code for a fast math parser library.
- State of the Art Expression Evaluation
  A tutorial on how to realize an expression evaluator in CSharp with ANTLR

Article Top ***Rate this Article for us!*** *Poor* ○ ○ ○ ○ ○ *Excellent* **Vote**

FAQ Noise Tolerance Medium Search Messages **Set Options**

Layout Normal Per page 25

New Message Msgs 1 to 25 of 138 (Total in Forum: 138) (Refresh) First Prev Next

Clasifing grayscale 20x20 images	**WhatEverMD**	**11:33 25 Feb '09**
Re: Clasifing grayscale 20x20 images	Andrew Kirillov	11:45 25 Feb '09
Re: Clasifing grayscale 20x20 images	WhatEverMD	12:17 25 Feb '09
Re: Clasifing grayscale 20x20 images	Andrew Kirillov	12:24 25 Feb '09
Re: Clasifing grayscale 20x20 images	WhatEverMD	13:21 25 Feb '09
Source available for the Demos?	**Lee Humphries**	**5:23 26 Jan '09**
Re: Source available for the Demos?	Andrew Kirillov	7:46 26 Jan '09
Re: Source available for the Demos?	Lee Humphries	22:28 26 Jan '09
Great work - but how to use more than one Input in approximatin and time series.	**john depp**	**21:24 12 Jan '09**
Re: Great work - but how to use more than one Input in approximatin and time series.	Andrew Kirillov	7:45 13 Jan '09
How can i use this algorithm after learning~?	**Chon John**	**1:53 8 Jan '09**
Re: How can i use this algorithm after learning~?	Andrew Kirillov	7:56 8 Jan '09
impressive	**learonlizheng110**	**5:11 18 Nov '08**
Re: impressive	Andrew Kirillov	8:33 18 Nov '08
Can Neural Networks Lib be achieved multi-objective optimization	**beardgh**	**3:25 7 Nov '08**
Urgent Question	**navas13**	**6:30 2 Sep '08**
Great article, but you have thread safety problems	**dietsche**	**2:58 24 Jul '08**
Re: Great article, but you have thread safety problems	Andrew Kirillov	11:17 24 Jul '08
Image Classification/Layout Classificaton & Backpropogation	**ankswe**	**14:36 11 Jun '08**
Please help..	**madhugeeth**	**11:44 5 Jun '08**
Re: Please help..	Andrew Kirillov	11:47 5 Jun '08
Re: Please help..	madhugeeth	11:53 5 Jun '08
Re: Please help..	Andrew Kirillov	12:18 5 Jun '08
Re: Please help..	madhugeeth	12:21 5 Jun '08
please help	**begumtt**	**23:10 25 May '08**

Last Visit: 18:17 16 Mar '09 Last Update: 13:48 17 Mar '09 **1** 2 3 4 5 Next »

General News Question Answer Joke Rant Admin

PermaLink | Privacy | Terms of Use
Last Updated: 19 Nov 2006
Editor: Smitha Vijayan