ARTIFICIAL NEURAL NETWORKS

Multilayer Perceptron

Arkadiusz Gabryś
Matrikel-Nr. 21895081
Department of Compuer Science

April 6, 2015

Abstract — Content of this document describes simple and typical multilayer perceptron. Guides from neuron cell to full neural network mathematical model and shows implementation example with both network modeling and training.

Contents

1 Introduction

Many tasks involving mathematical models and equations cannot be solve by algorithm because of its complexity or limitations in computer calculations. Also such problems exist which are too difficult to be describe by math itself. Sometimes it is even not known if mathematical model exist.

Some of this tasks are trivial for humans and by simulating how the human brain works it is possible to create algorithm that is able to *learn* how to solve a problem. *Learning* mean that mathematical model modify itself in *learning process* to meet certain conditions defined by *training set*. What is *training set* and how *learning process* is done will be covered in next chapters.

Now it is necessary to point that the way brain works is not fully known. But to create simple model our today knowledge is sufficient enough. Structure and mechanism of single neuron is know so based on that mathematical model can be create. Such model is just a generalization of something very complex. Yet good enough to reach our goals and break the bounders of what computer is capable of.

Algorithms uses neural network approach are relativity easy to understand and write in compare to what they can achieve. They are also quite fast. Because of that they are very popular in the filed of machines learning and problem solving.

Popularity of this method has led to creation of many variants and approaches in neural networks field. There exist supervised and unsupervised learning methods, linear and recursive networks. Each approach can be implemented in many different ways.

In this paper the simplest approach with basic *Error Back Propagation* algorithm will be discussed. First the structure and function of single human neuron will be briefly discussed to show how mathematical model is created. After that will be shown how single neurons can be organized together in networks and then in layers. Finally the simplest model of *Multilayer Perceptron* will be discuss along with *Error Back Propagation* learning algorithm. At the end example implementation in *C#* will be presented both with hints, observations and conclusion.

2 Neuron

2.1 Cell

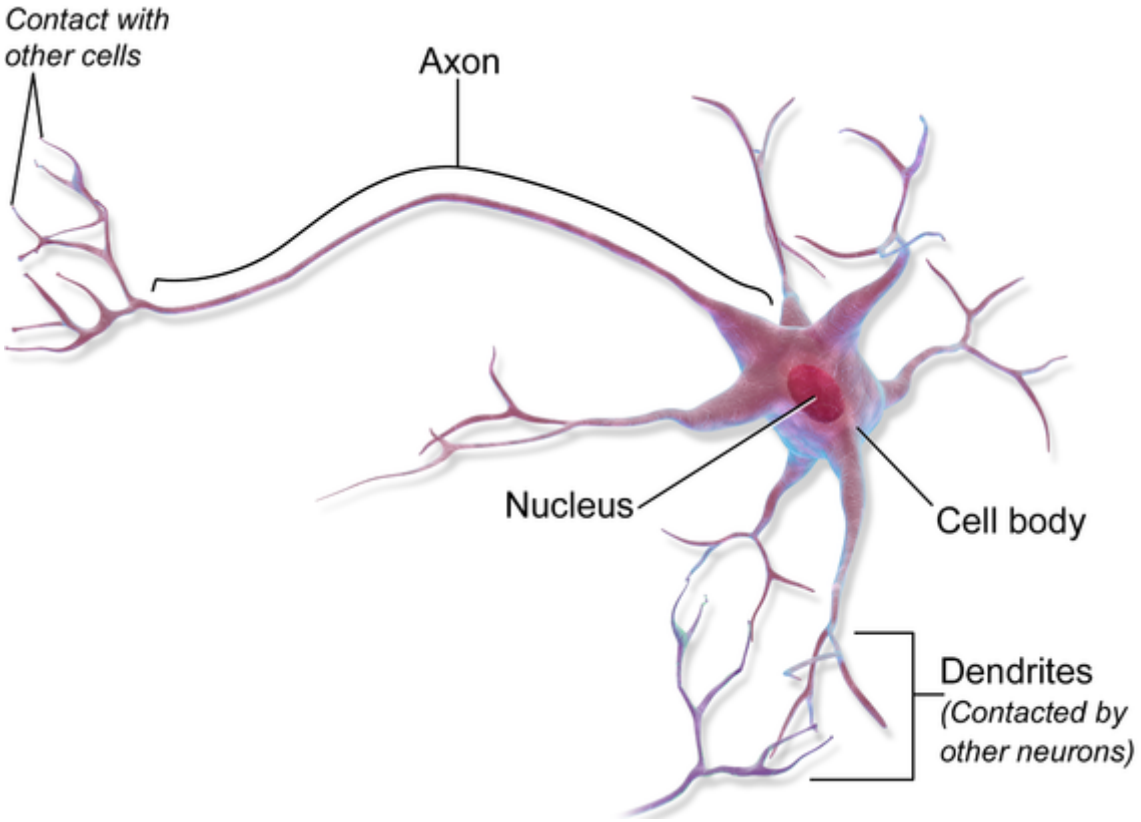

Figure 1: Multipolar neuron cell *(sizes and shapes vary)*

Without going into a details. There exist a few type of neural cell. In our mathematical model we use *Multipolar neural cell* which is shown on Figure 1. This one constitute the majority of neurons in the brain and include motor neurons and interneurons.

The way it works is such that it gather input signals through *dendrites* and produce one output signal through *axon*. The input signals are sum by *nucleus* and only if specific value is obtains the *axon* is activated. Both input and output signal are also modified while they are transmitted to *nucleus* or next cell. Modification depends on *nucleus* and thickness of *neurolemma (which is not shown on the figure)*. This signal changing part is a place where the *neurons network* knowledge is contained.

2.2 Model

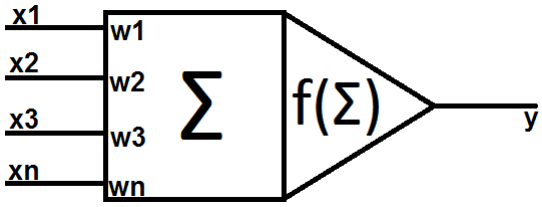

Figure 2: Basic neuron/perceptron model

Such *multipolar neuron cell* mechanism can be easily model. Each *neuron* input is represented by x_n and multiply by corresponding factors w_n (1) which represents change in *neuron* input signal. All signals are sum (2) (that represents *nucleus* function) and then processed by the *activation function* (3) (which is representation of the active or inactive *axon* and also serves other purposes - more about that later in this chapter).

$$\begin{aligned} x_n \times w_n &= \Delta_n && (1) \\ \sum_{i=1}^{n} \Delta_i &= \Sigma && (2) \\ f(\Sigma) &= y && (3) \end{aligned}$$

$$f\left(\sum_{i=1}^{n} x_i w_i\right) = y$$

Equation 1: Basic neuron/perceptron model

Most part of the model is simple and obvious yet the *activation function* need to be discuss in more details.

Activation function may take various form - from simple identity function to step function or more complicated continuous functions. It all depends on the output we want to obtain and/or the problem specification. In today's applications mostly sigmoidal continuous functions are used:

$$y = sgm(x) = \frac{1}{1 + e^{-\lambda s}}$$

Equation 2: Sigmoidal activation function

Most of *activation functions* are more or less similar. The x is the sum calculated by neuron in step (2). On λ depends slope of the function, in many cases this factor is omitted.

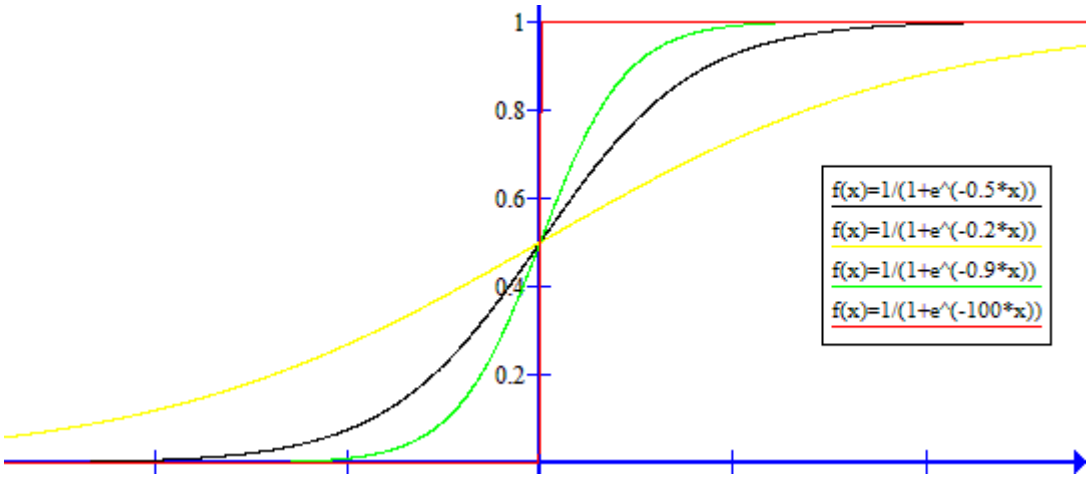

Figure 3: Sigmoidal activation function graph with different lambda factors

As can be seen on the Figure 3 *activation function* strongly depends on λ. For $\lambda \to \infty$ function become step function and for $\lambda \to 0$ function become linear.

This is just one simple example. Choosing *activation function* for specific application is not a trivial task and reader should look for different materials to achieve best result because this paper does not describe that topic.

3 Network

For one neuron there is one output value so we can easily see that one is not enough to solve many problems. Combining neurons into networks and layers will make them more useful. But one single neuron is still able to solve some class of problems.

Typically each neuron has one constant input value. Graphical interpretation of neuron can show why and also what class of problems can be solve with one neuron only.

Consider a neuron with three inputs $n = 3$ and with random weights w_n. So we have four axis, three for input data and one to show neuron output. Without one constant input graph of such neuron would be just a random set of points in 4D space.

But if we set one of the inputs constant then we will have only three axis, two for data and one for output value. Because of this one constant input instead of set of random points we will obtain 3D plane (Figure 4).

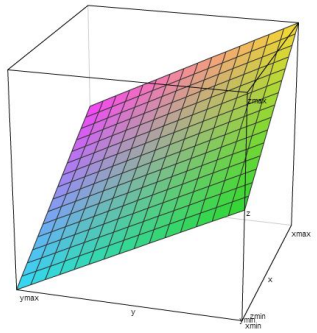

Figure 4: Single neuron plane

So our single neuron is able to model a nD planes. Such plane allows us to distinguish between three possibilities: data are greater then model, smaller or equal. It is enough to solve all linearly separable problems (e.g. logical AND, OR functions). But not all of the problems are linearly separable.

3.1 Layers

Before solving non linearly separable problems it have to be shown how more then one output can be obtain.

By putting same input values into multiple neurons with different set of weights w_n a more then one output can be obtain. Of course the single constant input still need to be provide for each of the neurons.

There are no differences between just using three separate neurons with different weights. So neurons in such configuration cannot solve problems beyond single neuron capabilities.

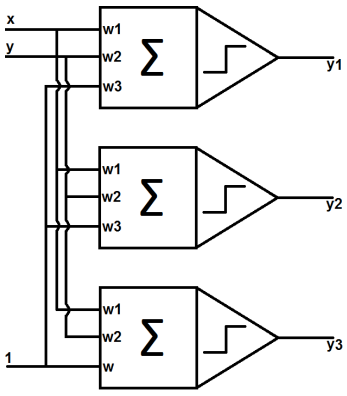

Figure 5: One layer of neurons - two inputs with common constant one

3.2 MLNN/PLP

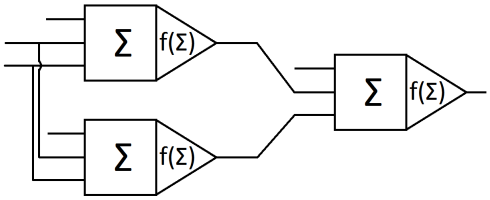

Figure 6: Very simple multilayer network

Non linearly separable problems can be solve by connecting multiple layers of neurons. Figure 6 shows the simplest possible structure of *Multilayer Perceptron*. Such structure is called *Multilayer perceptron* or *Feedforward neural network* or just *Multilayer neural network*.

Feedforward neural network is in our case the best naming convention because it tells exactly how each layer is connected. *Feedforward* means that each output signal is connected only to next layer input. So no recursive connections, loops etc. Such network structure is most popular because we now how to train neurons in that structure. More about that in next section.

In general each multilayer network can be divided into three parts:

- Input layer - used to prepare input data for network e.g. scaling or changing data distribution;
- Hidden layers - actual *feedforward network* with neurons;
- Output layer - provides output data, can do some additional output preparation e.g. rescaling, swapping data;

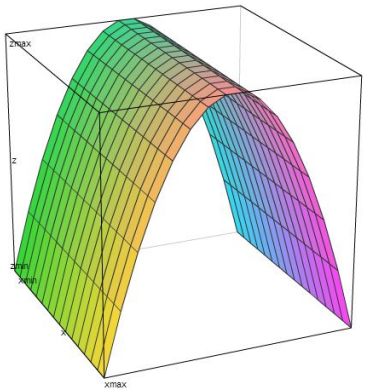

Figure 7: Graph of three neurons combined in two layer structure

Input layer is in many cases omitted as well as *output layer*.

The most important *hidden layer* in *feedforward network* has only one rule: *connect output from m layer to $m+1$ layer*. Not all of the output has to be connected to all neurons in next layer but this is usually the case. Still each of the neurons has to have its one constant input. Graph of such network (figure 7) will show why non linearly separable problems now can be model by such network.

Now there is a question how many neurons and layers is needed.
There are two simple rules:

1. As small number of layers as possible
2. As small number of neurons as possible

And there are many approaches to solve this problem:

1. Analytical calculations
2. Building network from small number of neurons and measuring results
3. Building network from large number of neurons and measuring results
4. Generic algorithms
5. Using theoretical knowledge about the problem

None of this methods will be discussed here.

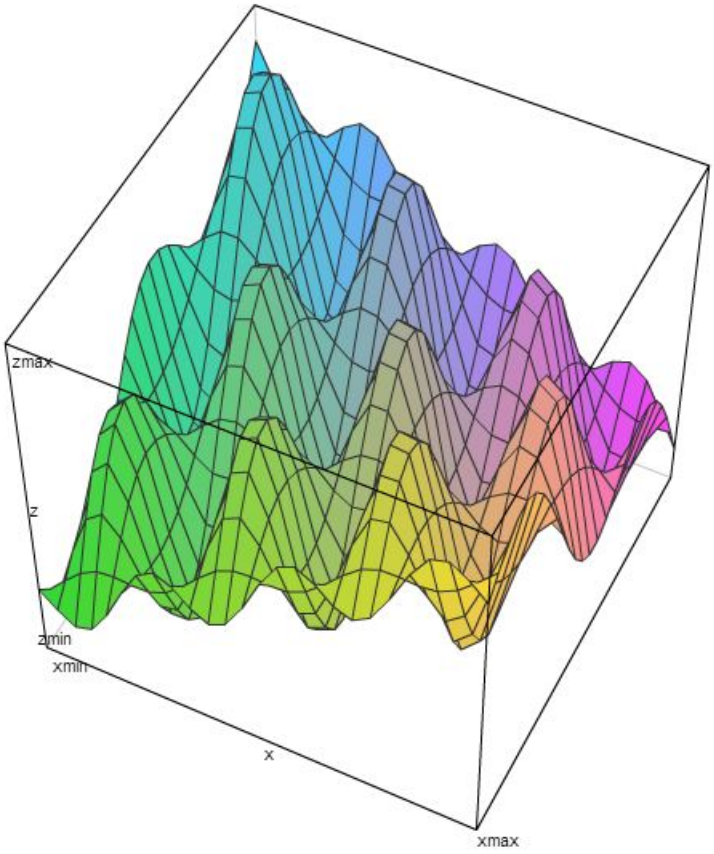

Figure 8: Complicated model modeled by MLP

3.3 Training

Now the EBP *(Error Back Propagation)* algorithm will be discussed. Training of multilayer network is numerical optimization of goal function $Q(k)$ where EBP algorithm in one of the gradient based methods.

$$w_i^{(n)}(k) = w_i^{(n)}(k-1) - \alpha \frac{\partial Q(k)}{w_i^{(n)}}$$

Equation 3: Error Back Propagation

Using this formula we will for each iteration k modify previous weight $w_i^{(n)}(k-1)$ of i neuron in n layer by negative gradient $\alpha \frac{\partial Q(k)}{w_i^{(n)}}$ where α is *learn factor* (determine *speed* of learning). Goal function $Q(k)$ is usually square of the network result error.

To know network error the sample set with both inputs and results is needed. Such form of network training is called supervised. Creating good sample set is a very large topic and wont be discussed here.

4 Implementation

In this section will be presented simple *C#* program as an example implementation of presented topics. By design it implements the simplest multilayer network with two inputs and one output to model boolean functions. Program is capable to learn non linearly separable XOR function and show this process.

4.1 User interface

User interface presents the network structure.

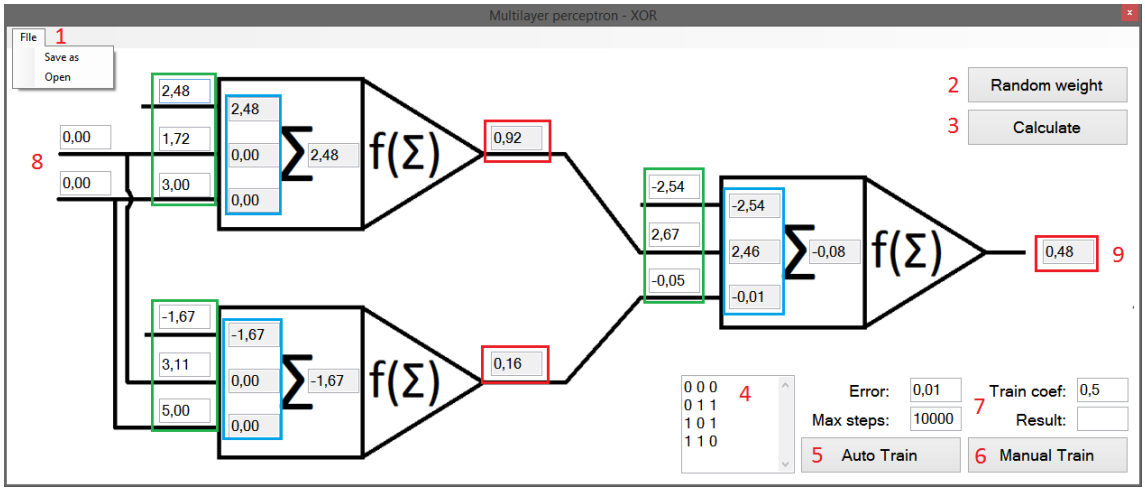

Figure 9: User interface

Elements of interface:

1. File menu - save and open network configuration with training data
2. Random weight - set random values for neurons weights
3. Calculate - calculate network output for given input values (8)
4. Training data - first two values are network inputs third value is desired network output
5. Auto train - train network using given Training data
6. Manual train - perform one iteration of EBP algorith with given input (8) and output given in Result field (7) as training data
7. Training parameters:
   (a) Error - desired error value for Auto training (5)
   (b) Max steps - limits number of Auto training iterations
   (c) Train coef - corresponds to *learn factor* α
   (d) Result - desired network output value for Manual training
8. Network inputs - used for Manual training (6) and for manual network output calculation (3)
9. Network output - calculated when button Calculate is pressed (3)

Green elements are neurons weights, **blue** are multiplication of weights and inputs, **red** are neurons outputs.

4.2 Program structure

Whole project is accessible on https://github.com/gabr/kisem_xor
It is based on two classes *XORForm* and *Network*:

- *XORForm*:
  - Implements User interface actions
  - Contains *Network* class
- *Network*:
  - Contains all neurons weights w, inputs u and it sum of weighted inputs S
  - Properties *Inputs* and *Output* gives access to first and last layer of network
  - *_connections* is a list of tuples where one tuple represents one connection between neurons
  - *_rand* is a object used to generate random values of weights
  - Methods implements User interface actions and the *Train()* method implements EBP algorith

Whole classes structure can be seen on Figure 10.

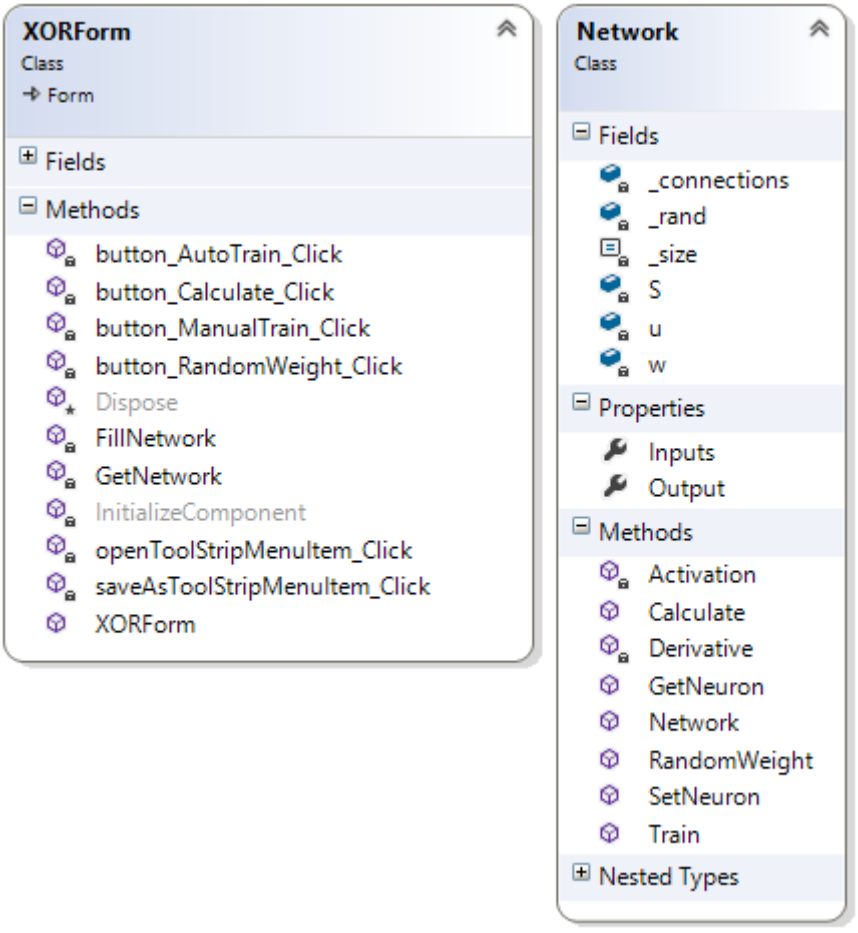

Figure 10: Classes

4.3 Usage

Usage of program on example of training boolean function XOR:

1. Open program
2. Set random weights
3. Provide true table for XOR function in training data window
4. Set desired error value and maximum iterations value
5. Set learning factor
6. Press Auto Train button

The resulted calculated weights can be saved using File menu *File* → *Save as*.