Visual comparison of performance for different activation functions in MLP networks

Filip Piękniewski
Faculty of Mathematics and
Computer Science
Nicolaus Copernicus University
Toruń, Poland
E-mail: philip@mat.uni.torun.pl

Leszek Rybicki
Faculty of Mathematics and
Computer Science
Nicolaus Copernicus University
Toruń, Poland
E-mail: lrybicki@mat.uni.torun.pl

***Abstract*— Multi Layer Perceptron networks have been successful in many applications, yet there are many unsolved problems in the theory. Commonly, sigmoidal activation functions have been used, giving good results. The backpropagation algorithm might work with any other activation function on one condition though - it has to have a differential. In this paper we investigate some possible activation functions and compare the results they give on some sample data sets.**

I. Introduction

MLP networks are popular due to their simplicity in implementation and quite satisfying results they give. In some cases though, MLP networks fail to provide a good solution. This can happen due to bad architecture, insufficient number of neurons or simply insufficient number of training epochs. To overcome such problems a lot of work needs to be done before the training process can be started. This includes constructing a good architecture, initialization of weights, choosing the best representation of data etc.

Another thing that affects the training process is the choice of the transfer function. The back-propagation algorithm is quite universal in this case, any function can be chosen as long as it has a derivative. There are however some obvious conditions that a good activation function should meet.

First, the function and it's derivative should be easy to compute. Since a look up table with an interpolation scheme can be used, the computation problem can be easily overcome. Secondly, the function should have an extensive linear part in order to speed up training and obtain satisfying convergence in less epochs.

Commonly the logistic sigmoidal function is used, because it's derivative is one of the easiest to compute, but there is a whole class of other sigmoidal functions that can be implemented, broadening the selection of potential transfer functions that can be used to solve a given problem. In fact our work shows that the log-exp function can give significantly better results than the logistic sigmoidal function, but still a lot of research needs to be done in that field.

The transfer functions have yet another important interpretation. They can be regarded as fuzzy logical rules that separate different categories. By changing the transfer function we change the decision profile. That, combined with the differences in training process, can have a great impact on the neural network performance and even the network's ability to solve certain problems.

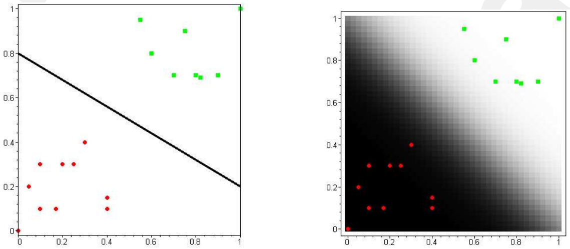

Fig. 1. Linearly separable data and the separating line (left). Note that we don't know if the line provides an optimal solution, which is not the case if the sigmoid transfer function is used (right). Value of the function marked by lightness.

The rest of the paper is organized as follows. In the next section we inspect in more detail the geometric aspects of neural networks, and in section 3 we introduce a way of visualizing neural network response, which gives a good insight into the way a network separates data. Next we discuss some examples of activation functions, present some sample results and conclude.

II. Geometry of decision

Geometrically speaking, the weight set of a single binary neuron defines a half-space in the hyperspace of possible input vectors. In fact, a threshold neuron is an implementation of the characteristic function of this half-space. A set is an n-dimensional half-space if it's a set of vectors satisfying:

$$x_1w_1 + x_2w_2 + \ldots + x_nw_n > \Theta$$

for a collection $(w_1, \ldots, w_n)$ and $\Theta \in R$. A single McCulloch-Pitts perceptron works by calculating the weighted sum of its inputs and passing it to a threshold function, returning either one or zero, like this:

$$\phi(x_1, \ldots, x_n) = \begin{cases} 0 & ; \sum_1^n w_i x_i \leq \Theta \\ 1 & ; \sum_1^n w_i x_i > \Theta \end{cases}$$

where $(x_1, \ldots, x_n)$ is the input pattern, $(w_1, \ldots, w_n)$ are the neuron's weights. We can (and often do) interpret this as a YES/NO answer or a binary categorial recognition.

The aim of a neuron training process is to determine a vector of weights (with its corresponding hyperplane) that would best separate sample input vectors of different categories. This is not always possible as shown by the infamous XOR problem. We'll return to this later. If it *is* possible to separate the samples, we want the solution to be as general as possible. This means that if we add a vector that was not in the initial training set, we expect it to be classified properly without re-training the perceptron. This, of course, depends on the choice of samples. If the samples are near the border hyperplane (in the input space), there is no problem. But if the samples are grouped around their category centers, as it would be with many statistically collected datasets (where the typical specimens are more likely to be selected than the others), there is nothing that would tell us where to put the hyperplane for optimal generalization. There is no measure of distance that could be minimized.

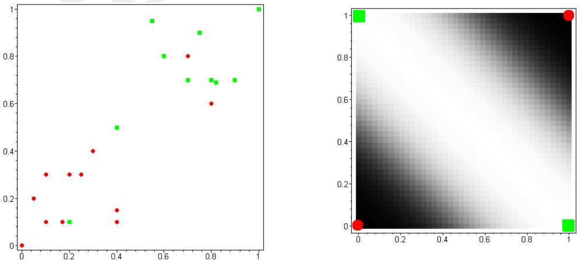

Fig. 2. An example that is not linearly separable (left) and the infamous XOR problem (right), with values of two sigmoidal functions (density from black to white), that transform the problem to make it separable.

The modification of the neuron to provide not only a binary (or bipolar) response, but a fuzzy value, requires specifying a *transfer function*, $f(x)$. The value returned by the neuron is then defined as follows:

$$\phi(x_0, \ldots, x_n) = f(\sum_{i=0}^{n} w_i x_i)$$

Please note the lack of the threshold constant (Θ). This is compensated for by an input (x_0) with a constant value of 1 (often called a *bias*). The weight associated with this input allows to control the influence of the bias on the output. This might not seem like a simplification, but it allows to regulate the bias weight together with other weights with a gradient minimizing algorithm.

Not much is required from a transfer function. They usually satisfy $f(0) = 0$ (or $f(0) = \frac{1}{2}$) and they're usually nondecreasing functions. It isn't required but often practiced that a transfer function has a range within $(0, 1)$ or $(-1, 1)$. For practical purposes mentioned later, it's practical if the function is differentiable. It also helps if the differential is not constant. The similarity to distribution functions is not quite coincidental.

Equipped with a transfer function, the neuron's output can be interpreted in terms of fuzzy logic. It's no longer a 'yes' or a 'no', but a 'maybe', 'rather not' or 'almost sure' answer, thus allowing us to maximize generalization, provided that the training set of the input vectors is representative. The optimal weights for a training set

$$T = \{(x_1^i, \ldots, x_n^i; c^i), i = 1 \ldots m\}$$

where $c^i \in \{0, 1\}$ is the category of the $i-$th sample, are the ones that minimize the mean square error:

$$E = \sum_{i=1}^{m} (\phi(x_0^i, \ldots, x_n^i) - c^i)^2$$

What if there are more than two categories to separate? Simple. For k categories, train k perceptions, each to separate one category from the others. This parallel network of neurons returns a vector $(\phi_1, \ldots, \phi_k)$ of responses, where the category returned by the network is the number of the output with the highest value (the *winner*. This requires a different method of training.

Training the neurons separately would require a lot of effort to lower the activations of many neurons while increasing one. We don't need that. If the winner wins, we don't need to punish the other neurons. Thus we train the network as a whole, punishing only the neuron that won, but shouldn't have. Since this method will not be the one used in the following sections, we won't get into any more details.

Even with transfer functions, the problem of linear separability remains. Our weight set may be the optimal for a problem, but some inputs may be mis-categorized. If the dataset is not linearly separable, there might exist several sets of weights that minimize the mean square error function and the value of the transfer function doesn't help in case of the badly classified vectors - it can be as low or as high as it gets.

It has been proven [4] that increasing the dimensionality of the problem eventually makes it separable. It's like giving the network a hint that does not contain the answer but helps to find one. XOR is not separable, but it gets separable if you add the value of AND or OR as an extra input. This kind of reasoning led to the creation of network constructing algorithms like the towering or pyramid algorithm where newly added neurons are being connected not only to the inputs, but also to the currently existing neurons. But constructing a network via network building algorithms doesn't prove efficient when it comes to separating multiple categories. Instead, a more general approach is used.

Let's sum it up. A threshold neuron is an instance of the characteristic function of a half-space. A neuron with a transfer function implements a transformation $\phi : \mathbb{R}^n \rightarrow \mathbb{R}$ creating a fuzzy hyperplane that can be interpreted as a fuzzy logic rule separating the data. A set of parallel neurons corresponds to a function $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^m$, an inter-dimensional transformation with a transfer function applied to each vector in all dimensions. The trained network is expected to transform each sample vector to a vicinity of an axis in the resulting hyperspace corresponding to the sample's assigned category. If the neurons are connected serially, on the other hand, the network computes a function $\phi : \mathbb{R}^n \rightarrow \mathbb{R}$ that tends to increase near the vectors it is trained to recognize, giving nice separability and generalization, but only for one category.

A network of neurons connected parallelly into layers and then serially between layers is referred to as a Multi Layer Perceptron. MLPs can't be trained like single neurons (or like parallel networks for that matter) because they have many layers, all of which should be trained. They can be trained part by part in a constructive algorithm but it's inefficient since they work as a whole and no single part of the structure is meant to recognize any particular part of the data, although specialization might appear.

A trained Multi Layer Perceptron implements a transformation of $\mathbb{R}^n$ into $\mathbb{R}^m$ that moves an input vector to its corresponding desired output. The transformation is the cumulative result of transformations made by particular layers. The idea of training an MLP is based on calculating the error (squared difference between output and desired output) and propagating it back - from the output layer to the layer connected to the input. This is done with several assumptions:

- the transfer function is continuous
- the differential of the transfer function exists
- the differential has a maximum in the vicinity of zero

The weights in subsequent layers are changed depending on to the value of the differential of the transfer function for the current activation of the neuron. Note that the weights are not changed if the output and the desired output coincide.

The geometric interpretation of the algorithm involves the *energy landscape*. The energy landscape is simply the multidimensional plot of the energy (error) function. It can be visualized as an area with hills and valleys. The back-propagation algorithm is like a ball that starts with a random position and always tends to roll down to a local valley (minimum). The word *local* is a key word here.

The rest of the paper will focus on Multi Layer Perceptrons used to separate multiple categories of data, trained with the back-propagation algorithm.

III. Barycentric plotting

In order to investigate neural net performance, we use a simple projection scheme - barycentric plotting. In this way we get a lot more information about the network, than by simply watching MSE or other error measures. Before we explain the plot mechanism we have to make some assumptions about the problems being solved:

- the input consists of n vectors $E^{(i)} \in \mathbb{R}^s$, each assigned to one of k categories. Vector $E^{(i)}$ is assigned to category $Cat(i) \in \{1 \ldots k\}$
- the network consists of two layers. There are s inputs, some number h of hidden neurons, and k output neurons.
- the network is trained to activate the t-th output neuron, if the input vector $E^{(i)}$ is assigned to category t, while others should not be activated. The desired output vector corresponding to the i-th category will be denoted by $\overrightarrow{Cat(i)}$ as opposed to the actual network output $O^{(i)}$.

Thus, the geometrical interpretation of a neural network's output can be easily made. A well trained network should implement a transformation that maps some k sets from an n-dimensional space into k diagonal vertices of a k-dimensional unit hypercube. The geometry of data clusters in the input space might be very sophisticated, as we only know that there are k categories spread somewhere around the n-dimensional space. The transformation the network implements changes the dimensionality of the problem, and sets the samples from each category scattered around the corresponding vertices of the hypercube.

Although the dimensionality is often reduced by the network, it still might be far too large for visualization, since we have k (usually more than 3) categories. To overcome this problem, we project the hypercube's diagonal vertices into a polygon (k-gon) as follows:

$$\begin{aligned} O_x^{(i)} &= \frac{1}{\delta}\sum_{l=1}^{k} G\left(\left(\left\|O^{(i)} - \overrightarrow{Cat(l)}\right\|\right); 0, \sigma\right) \cdot \overrightarrow{Cat(l)}_x \\ O_y^{(i)} &= \frac{1}{\delta}\sum_{l=1}^{k} G\left(\left(\left\|O^{(i)} - \overrightarrow{Cat(l)}\right\|\right); 0, \sigma\right) \cdot \overrightarrow{Cat(l)}_y, \end{aligned} \tag{1}$$

where $\delta = \sum_{l=1}^{k} G\left(\left(\left\|O^{(i)} - \overrightarrow{Cat(l)}\right\|\right); 0, \sigma\right)$ is a normalizing factor, $(O_x^{(i)}, O_y^{(i)})$ are coordinates of the i-th output's projection, $(\overrightarrow{Cat(l)}_x, \overrightarrow{Cat(l)}_y)$ are coordinates of the l-th category projection (l-th vertex of the k-gon), $\|\|$ is the Euclidean norm in a k-dimensional space. $G(x; 0, \sigma)$ is a scaling function. A simple Gaussian kernel is used:

$$G(x; a, \sigma) = e^{-\frac{(x-a)^2}{2\sigma^2}}$$

By making the dispersion parameter σ smaller, we can investigate the misclassified samples more thoroughly, while by increasing it, we can get an overall view of the network's separation capabilities. The dispersion parameter can also depend on the classification properties for each category. Such adaptive scaling schemes might clear the plot up, and make important information more visible. We introduce two adaptive schemes:

- max-scaling, in which σ is proportional to maximum distance of sample from its assigned category:

$$\sigma_{(l)} = \sigma_0 \max_{i \in \mathbb{N}, Cat(i) = l} \left\|O^{(i)} - \overrightarrow{Cat(l)}\right\| \tag{2}$$

- avg-scaling, in which σ is proportional to average distance of samples from their assigned category:

$$\sigma_{(l)} = \sigma_0 \left(\frac{1}{M} \sum_{Cat(i) = l} \left\|O^{(i)} - \overrightarrow{Cat(l)}\right\| \right) \tag{3}$$

With the plot mechanism described above, we investigated some differences between networks of the same architecture, but with different activation functions.

IV. Transfer functions

The choice of a transfer function for a problem depends on its nature, the required precision and speed, and the chosen training algorithm. It's useless to try to train a network based on functions with badly-behaved second derivatives with a second-order backprop algorithm. Sometimes some experimentation is required - with an MLP trainer and a barycentric plotter for example.

The transfer function of a threshold neuron is the threshold function. Since the derivative is the Dirac δ, we cannot train a network composed of such neurons with a gradient algorithm like the one described above. The energy landscape is not continuous.

Neurons with a transfer function $f(x) = x$ are often called linear units. They just calculate the weighted sum of the inputs (a scalar product of the input vector and the weight vector). A layer of such neurons implements a linear transformation via multiplying the input vector by the matrix of the weights. Multiple layers are useless since the products of weight matrices is a matrix - a multi-layer network composed only of linear neurons can be replaced with a single layer with weights based on a simple calculation. The problem of linear separability returns. Still, linear layers are useful due to the linear nature of the transformation they induce.

The first transfer function analyzed with our test is the logistic sigmoid function:

$$SG_\beta(x) = \frac{1}{1 + e^{-\beta x}}$$

The logistic function is actually $tanh(x)$ transposed from the range $(-1, 1)$ to $(0, 1)$. As β approaches infinity, the function approaches the threshold function. The differential has an interesting property:

$$SG'(x) = SG(x)(1 - SG(x))$$

This is often used in back-propagation.

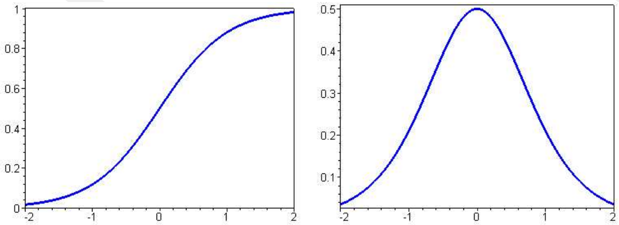

Fig. 3. Logistic transfer function (left) and its differential (right).

Another bad idea is trying to train the network to return the extreme values - 0 and 1. Notice that σ never reaches these and the overall effect is that the weights increase to infinity very quickly. That, in an implementation, results in a displeasing effect of a sudden chaos when an overflow occurs. Good target values are 0.2 and 0.8.

The implementation of the logistic function requires calculating the value of the exponential function, but in many implementations, the function is read from a previously prepared array. This speeds up the training process, although the tabelarisation involves a choice between low memory usage and good precision.

The logistic transfer function, like the following transfer functions, has been tested on a 3-category 300-vector dataset representing three four-dimensional gaussians with mean values: $a_1 = [1, 0, 0, 0]$, $a_2 = [0, 0, 0.5, 1]$, $a_3 = [0, 0, 0, 1]$ and dispersions: $\sigma_1 = [0.5, 0.3, 0.3, 0.3]$, $\sigma_2 = [0.1, 0.4, 0.4, 0.4]$, $\sigma_3 = [0.7, 0.3, 0.3, 0.1]$ respectively. For best comparison, the presented plots were created after 100 and 200 cycles (epochs) of training with learning rate set to 0.03, momentum set to 0. Adaptive dispersion was used, the *maximum* version. The network architecture was 4-6-3 in all cases, with the initial weights in the range of $[-0.01, 0.01]$.

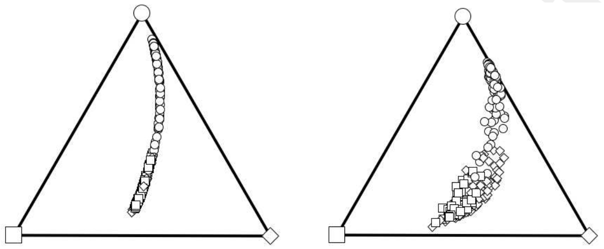

Fig. 4. The logistic function in test. Left: after 100 cycles, MSE=0.136. Right: after 200 cycles, MSE=0.109.

The result in this case was predictable. The learning process was a slow but steady drop of the global error while more and more patterns joined their proper categories. The plot shows a situation that occurs in the beginning of the training process (and lasts until about 160 cycles). When one of the categories gets recognized by the network, the elements of the plot start forming a prolonging pattern. The MSE drops below 0.1 shortly after the checkpoint of 200 cycles.

The semilinear function can be regarded either as a rough approximation of the sigmoid function or as a compromise between the linear and the threshold function. It has certain properties of all the three. It is the distribution function of a uniform statistical distribution on a segment.

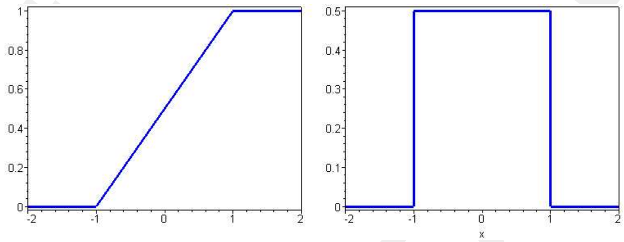

Fig. 5. Semilinear transfer function (left) and its differential (right).

The formula:

$$SL(x) = \begin{cases} 0 & ; x < a - \Delta_x \\ \frac{x+\Delta_x - a}{2\Delta_x} & ; a - \Delta_x \leq x < a + \Delta_x \\ 1 & ; a + \Delta_x \leq x \end{cases}$$

where Δ_x is responsible for the slant and a centers the function. Both can be disposed of, theoretically, since the relative size of the weights controls the slope and the weight of the bias node centers the function. In practice, it's very important to adjust the function to the initial weight dispersion and the learning rate. If the initial weights are larger than Δ_x, you get the behavior of the threshold function. If the initial learning rate is too big, the network might destabilize and end the training process prematurely, returning only 0-s and 1-s. Still, destabilization during the training process is possible.

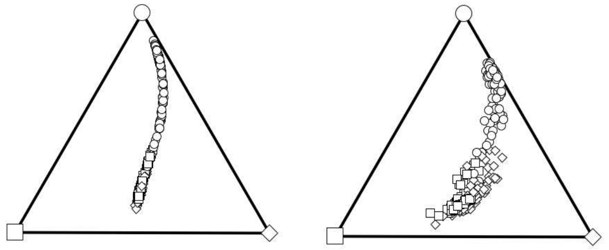

Fig. 6. The semilinear function in test. Left: after 100 cycles, MSE=0.135. Right: after 200 cycles, MSE=0.113.

The experiment shows a very slow decrease of the MSE, with the in-line effect discussed before breaking after 190 cycles. After that, the MSE doesn't seem to drop below the 0.1 barrier for at least 200 more cycles. Tests on other data samples show better separation capabilities, especially when multiple categories are involved.

The semiquadratic function is based on the triangular uncertainty (figure 7) and approximates the sigmoid function with two pieces of the parabola and two constant parts.

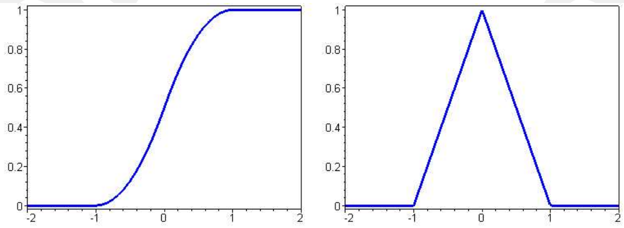

Fig. 7. Semiquadratic transfer function (left) and its differential (right).

The implementation is fast, since the only required operations are purely arithmetic and logical.

The semiquadratic function excels both in speed of calculation and in speed of training, breaking the 0.1 MSE barrier before the 100-epoch checkpoint, then steadily proceeding with training. The plot shows a tendency of the samples to form a triangle that appears very early in this case.

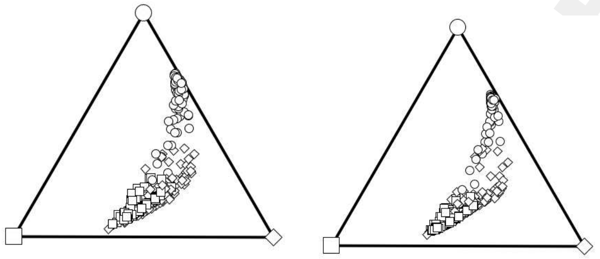

Fig. 8. The semiquadratic function in test. Left: after 100 cycles, MSE=0.1. Right: after 200 cycles, MSE=0.093.

The last function tested, referred to as log-exp, requires a longer explanation. The differential is actually the difference of two sigmoids, forming a bell-shaped plot depending on a parameter that distances them. The function itself is a perfectly smooth one with a controllable slope.

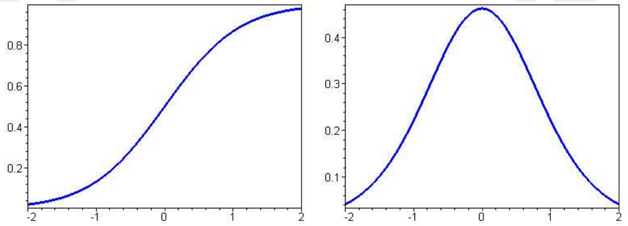

Fig. 9. Logarithmic-exponential transfer function (left) and its differential (right).

The formula is:

$$LE(x) = 1 - \frac{1}{2b} ln \left[\frac{1 + e^{a-cx+b}}{1 + e^{a-cx-b}} \right]$$

where, a is responsible for centering the function, b distances the sigmoids and c is a coefficient that controls the slope. For best results, it's advisable to set it to a value that maximizes the value of the differential at 0 (or rather at a).

The implementation requires calculating the exponential function more than once, plus the natural logarithm. The differential isn't much better either. The training process, unless accelerated with lookup tables, is therefore slow, but surprisingly effective.

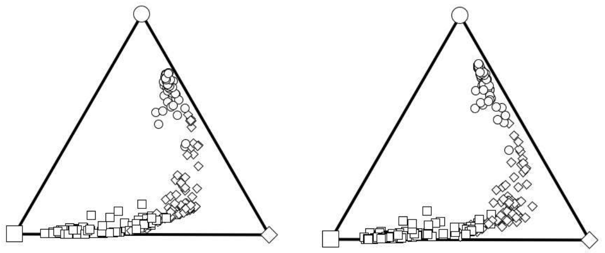

Fig. 10. The log-exp function in test. Left: after 100 cycles, MSE=0.058. Right: after 200 cycles, MSE=0.048.

Testing the network with these transfer functions shows fast convergence. The smoothness of the function allows the hyperplanes to adjust their position with ease. The geometry of the transformation is clearly visible. The previously mentioned in-line and triangle-shape effects don’t occur. Instead, the samples tend to move towards their categories without wandering, until after 240 epochs, no badly classified samples remain. This might not be unexpected in this case, as the data was composed of gaussians, but tests on other datasets show that this kind of functions, despite the relative amount and complexity of calculations required, gives results far better than the commonly-used logistic sigmoid function or the proposed semiquadratic function.

V. Summary

Transfer functions affect not only the geometry of the transformation induced by the network, but also the amount of calculations required and the speed of the training process as a result. The optimal choice differs with various problems, but some general rules can be stated. Understanding the process of training MLPs combined the ancient scientific method of trial and error, equipped with a barycentric plotter, helps to chose the best transfer function and a suitable architecture.

VI. Acknowledgements

We thank our mentor, prof. Włodzisław Duch, who had suggested to take on this project, for a lot of useful remarks and a great amount of optimism. Special thanks go to dr Tomasz Schreiber who helped us with various theoretical issues, and spent some time with us on searching for mistakes.

References

[1] W. Duch, *Uncertainty of data, fuzzy membership functions, and multilayer perceptrons* (2003, subm. to IEEE Transactions on Neural Networks)

[2] W. Duch, *Coloring black boxes: visualization of neural network decisions.* Int. Joint Conf. on Neural Networks, Portland, Oregon, 2003, Vol. I, pp. 1735-1740

[3] P. Pereto, *An introduction to the Modeling of Neural Networks* Cambridge University Press 1992

[4] Simon Haykin, *Neural Networks, a Comprehensive Foundation*. Macmillan College Publishing Company, New York 1994

[5] Stanisław Osowski, *Sieci neuronowe do przetwarzania informacji*. Politechnika Warszawska, Warszawa 2000

[6] Stanisław Osowski, *Sieci neuronowe w ujęciu algorytmicznym*. Wyd. Nauk-Tech, Warszawa 1996