Problems and Solutions of Visualizing and Analysing Multidimensional Output from MLP Networks - Barycentric Projections.

Filip Piekniewski & Leszek Rybicki

20 November 2003

Abstract

Understanding the process of neural network learning is an important issue in todays cognitive science. The main problem in this field is that the networks' input and output are usually multidimensional. Because of our natural limitation to see (and think) in three-dimensions at most, the analysis of a learning process with a 4-, 10- or 28-dimensional output is difficult, as we can only see sequences of numbers and we don't really know what is happening. That is where visualisation may be of use.

The aim of this paper is to present a very simple idea of visualizing the learning process of a MLP network using projections into a polygon. We will also try to establish differences between several variants of barycentric visualization and between networks based on different activation functions, with different learning algorithms.

1 Introduction

Artificial Neural Networks have become very popular in a wide range of applications including scientific research, military equiptment, household appliances and many other fields that require pattern recognition, data classification and separation. Altough many learning algorithms have been implemented, and the whole theory of neural networks seems to be quite developed, unsolved problems and questions still exist.

One of those problems is to find a good measure, or a method of determining if the network is well trained and ready to solve "real life" problems, or if there are singularities that could render the network useless. The backprop algorithm (and other learning algorithms) minimises a well defined error function (usually MSE[1]). One might ask, since we know the value of the error function, what else do we need?

In fact, mean square error (and other error functions) provides just a piece of statistical information about the learning process, while most important issues are lost. Two different neural networks can have the same MSE value for the same data sets, but have different generalisation abbilities or specialise in different cathegories of data. One of them could have a glitch causing less typical patterns to be classified badly, while perfectly classifying other patterns of the same cathegory. Neither the MSE is nor any other single dimensional real number measure is sufficient to investigate such issues.

To get more information about the learning process, it would be necessary to analyze the whole output data, find out which vectors are problematic, which of them separate well and which don't. Obviously, analyzing thousands of numbers is not a good idea. Economy, biology and physics has a common solution to that - plotting.

[1] Mean Square Error

2 Visualization

From now on, we will focus on MLP networks, used for classification. The problem can be defined as follows:

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

That way, each cathegory is mapped to a nonzero vertex of a k-dimensional simplex (simplest polyhedron in k-dimensional space). For example, if there are two cathegories, then output is two dimensional, and cathegories are mapped on to vertices $(1,0)$ and $(0,1)$ of a triangle that contains also point $(0,0)$. The whole space of possible activations is then a k-dimensional cube, where the $(0,...,0)$ vertex stands for no activation, and the $(1,...,1)$ vertex stands for full activation. For cathegorisation, we expect the data patterns to cluster around vertices that belong to a diagonal hyperplane ($(0,...,1,..0)$, etc.).

Obviously, if there are two or three cathegories, visualization is not a big problem. The real problem occurs when there are more than four cathegories. In such (common) situations "1 to 1" plots are impossible to create. There are, however, many ways of projecting information into less dimensional space, some of them we will use.

First of all, let's review the problem, pointing out important issues:

- We have n vectors (outputs of a neural net) which lay inside a k-dimensional cube.
- We map the cathegories into diagonal vertices.
- The learning process will cause most of the data to gather around these vertices.
- The vertices are in a k-dimensional hyperspace ($k \geq 4$), but they all belong to a diagonal hyperplane

This way, the most effective method of visualization seems to be some kind of a projection.

Now, what interest us most about each vector $O^{(i)}$ is:

- Is it well classified (in other words, if the distance between $O^{(i)}$ and $\overrightarrow{Cat(i)}$ is the smallest among the cathegories)?
- Is it far or near of its assigned cathegory, or does it tend to move towards another cathegory?
- Is it alone or are there any other vactors in its vicinity?

Let's consider a mapping system, as follows. Each of the cathegories is a centre of a Gaussian radial function

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

which will be used for scaling. Categories will be mapped into the corners of a polygon, one to each. Let's say we have k cathegories. Inside a k-gon, we will project each output vector $O^{(i)}$ as:

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

Where $\delta = \sum_{l=1}^{k} G(0,\sigma)(\left\|O^{(i)} - \overrightarrow{Cat(l)}\right\|)$ is a normalization factor, $(O_x^{(i)}, O_y^{(i)})$ is a coordinate of $i-th$ output's projection, $(\overrightarrow{Cat(l)}_x, \overrightarrow{Cat(l)}_y)$ is a coordinate of $l-th$ cathegory projection ($l-th$ vertex of k-gon), $\|\|$ is an euclidian norm i k-dimensional space.

A sample plot can be seen in 1. To make the plot more useful, we add some extra information coded as color, like where the sample belongs, and how was

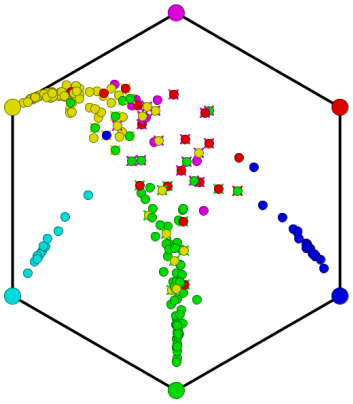

Figure 1: A sample plot, with six cathegories and some data. It visualizes the output of a sigmoidal MLP network with a MSE of 0.214. The plot was scaled by $\sigma = 0.35$. Each dot represents one output vector. The color of the dot represents the class to which it belongs. If a dot is marked with an X, the color of the mark is the cathegory, to which the sample was mistakenly assigned by the network. One glimpse on this plot gives information that cathegories violet, cyan and blue are quite well separeted, while cathegories yellow, red and green mix.

it classified. Further in the paper we will see some other add-ons like network dynamics, and convex hull around each data cluster.

Since the plot mechanism described above is just a projection, some information is lost. Two dots displayed as close to each other can be quite distant in the activation space. If we see dots from different cathegories mix up in the plot, it doesn't always mean they mix in the activation space. These are obvious faults of plotting multidimensional data onto two dimensions. One way to compensate for these faults is to see different plots, and choose the one that gives the best view. This can be done by permuting the vertices of the polygon (which corespond to cathegory centers). Unfortunately it's rather difficult to numerically choose an optimal (most suitable for a human) permutation, so the problem of choosing the best permutation is left to the plot reviewer (user). Permuting cathegories is just one of many methods to make such plots more useful. We introduced some other self-adapting optimizations, that might make

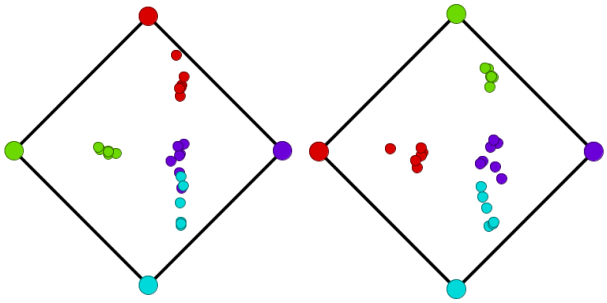

Figure 2: Two plots, showing exactly the same data (sigmoidal MPL output, MSE 0.1, scaled by a factor $\sigma = 0.7$), but with different permutations of cathegory vertices. By looking only at the left picture one might think that violet and cyan vectors mix up. This is not true though, as we can see on the second picture (to the right) where these vectors are well separated.

the plot just a bit better. As we have seen in formula (1), each vector is scaled by a Gaussian function localized in the centre of its cathegory. The σ parameter, responsible for dispersion, if constant for all cathegories and data samples. What if we make σ dependent upon some parameters unique for a cathegory? Lets assume that

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

In this case, the Gaussian dispersion depends on the maximal distance between the centre of cathegory l, and any vector assigned to the cathegory. If the vectors of cathegory l are spread widely all over activation space, the coresponding Gaussian function will have a big dispersion. A big dispersion means that the cathegory will attract other vectors stronger. What consequences does it have for our plot? Since a "wide" cathegory is a stronger attractor, this scaling will reveal the border regions in activation space, while most vectors from the $l - th$ cathegory will be projected close to the polygon corners. This kind of projection is good for analyzing border regions of networks that are already trained, and is rather useles for fresh networks.

Another variant of adaptive, cathegory dependent scaling is done by making σ depend on the avarage distance of a vector from its assigned cathegory. This

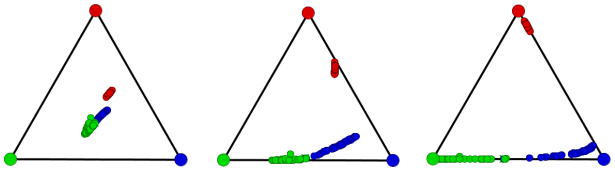

Figure 3: An example of three plots created using the same data (iris data parsed by a MLP network, MSE=0.086) but with different scaling options. The one on the left is scaled with $\sigma = 2.0$, constant for all cathegories (1), the one in the middle is scaled using the maximum formula (2) with σ_0 set to 2.0 , third one was scaled using average formula (3) again with $\sigma_0 = 2.0$.

can be done as follows:

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

As opposed to the previous method, let's suppose, that most vectors are properly classified around their common cathegory, and a few are not. The average distance between a sample and the cathegory centre is small, and so the σ parameter is small. It's as if we treated those wrongly classified vectors as data errors, rather than network errors. This gives a slightly different projection, that displays badly classified samples.

3 Additional visual guides

Adaptive scaling methods and vertex permutations are helpful when it comes to enhancing the informational value of the plot. We suggest a couple more enhancements.

It's not uncommon for a single data sample to float away from its cathegory. The network would keep the MSE low by keeping the rest of the samples close to their proper cathegories. While this might be satisfactory from a statistical point of view, the consequences might be catastrophical in real life applications. Imagine a patient with very uncommon symptoms (for genetic reasons for example) being cathegorised to (and treated for) a different disease.

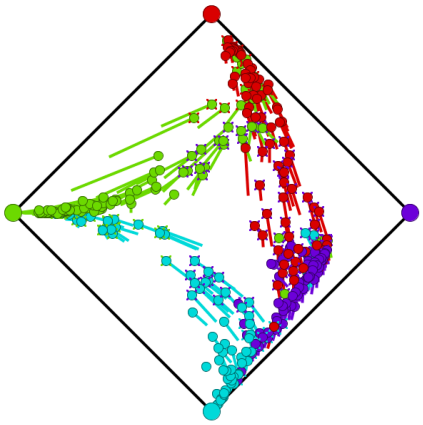

Figure 4: The dynamics of a network's output in early stages of the learning process (backprop algorithm with momentum), Hayashi data set, MSE=0.3, scaled adaptively (avarage) $\sigma_0 = 0.5$.

A renegade sample could not be noticed as its corresponding dot could be covered by other dots on the plot. To resolve this, as well as provide a certain per-cathegory measure of data similarity, we suggest completing the plot with convex hulls marking the borders of each cathegory. If a data sample happens to stand out, the hull will expand to contain it, making it clear that the cathegory does mix with another one.

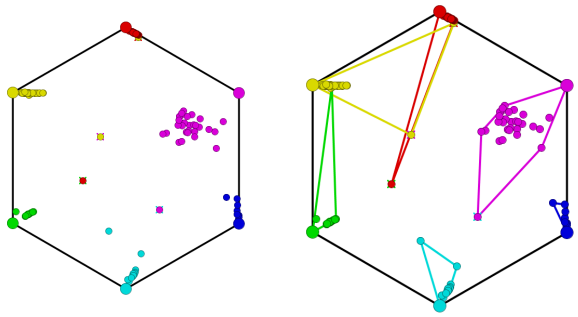

Figure 5: The green cathegory is well separated... or is it?

From a not-overtrained, well-generalizing neural net, we expect the convex hull of a given cathegory to remain within a vicinity of the cathegory's vertex. If

two convex hulls overlap, it suggests that their cathegories might be confused by the network (but not certainly). If the convex hull is stretched away from the cathegory vertex, but the samples remain tucked tight, it means that the net has trouble separating the cathegory and its architecture and/or teaching process has to be reviewed.

To provide extra information about the overlapping of cathegories, we have added simple voronoi border capability to the plot. The borders separate only the cathegory centers, but give a good view of relations between cathegories. One thing has to be set straight though. The fact that a sample is not in its assigned voronoi segment, doesn't mean it's not classified properly. The position of a sample in the plot depends on multiple factors, including relative weights of the output synapse, adaptive scaling (σ parameters),

Therefore we decided to mark the badly classified samples with an X. The color of the X represents the color of the cathegory it was (mis)assigned to. Within the plotting program, the user can hover the mouse over any dot to chceck it's input values and sequence number in the input data source. This gives extra information about misleading data or patterns misassigned in the process of data preparation. The X mark is independent of scaling, because its color is the color of the vertex the dot approaches when σ approaches one.

All enhancements and visual guides described so far are meant to emphasize certain qualities of trained networks or compensate for the limitations of dimension-reducing projection. To research the dynamic aspects of the teaching process, the reviewer has to either watch the changing plot during training, or look at multiple plots representing the states of the network in subsequent stages of the process. Since training a big network with a large portion of data can be a process of slow changes over a long time, watching the dots move can be boring, tiresome or even (you can trust us on that) put you to sleep. Displaying multiple plots requires additional space and doesn't provide any visual clue on how far a given dot has moved (or if it has moved at all). It's not a very good idea to print a sequence of plots either.

Our solution to that is rather simple. We mark the movement of a dot between a cetrain (user-defined) number of training epochs as a line ending with the dot. This might not say much about an individual dot, but the whole plot seems to be enriched by another dimension: time. Extra information given by such a plot is of multiple sort. We know that the teaching process is going well if the dots are moving towards their assigned cathegories, but that would be a pretty optymistic situation and doesn't really happen throughout most of the process.

The network is separating well if a group of dots belonging to the same cathegory is moving simultaniously. Any dot described as a "renegade" before, or a dot that is to become a "renegade" would most certainly move in a different direction. In some situations you see that all cathegories seem to be moving towards one that DOesn't move much. That means that the network has trouble separating that cathegory and tends to erradicate the mSE by warping the whole output in that direction. A situation like that suggests that maybe crossvalidation or data preseparation might be required. The network might be too small or a comitee of networks would solve the problem.

4 Implementation and Results

The basic idea of visualisation doesn't depend on any specific network architecture. To the visualiser, the neural network is the proverbial black box, from which it expects as much and as little as to have an output of a certain dimension and to learn in epochs. In our case the neural network can even run on a remote server.

Our implementation consists of the visualiser module (of course) and an example of a black-box neural network teaching module based on JOONE[2] implemented in Java, for portability reasons. The choice of JOONE as the back-end was also dictated by an

[2] Java Object Oriented Neural Engine

extensive choice of varoius transfer functions implemented and the ease of implementing new ones. This way a network can have an arbitrary number of layers of different sizes with di erent transfer functions, all controlled by a single backprop algorithm.

A layer is a set of neurons implementing a common transfer function. The following example focuses on three transfer functions - linear, sigmoid and semiquadratic. A linear neuron s output is the sum of its inputs. A multilayer network based solely on linear neurons is obsolete, as it could be replaced by a single layer. Another limitation of the linear transfer function is the infamous linear separability problem that slowed down research for some time.

The sigmoid function is defined as

$$f(x) = \frac{1}{1 + e^{gx}}$$

where $g \in \mathbb{R}$ is a parameter responsible for the inclination of the function's plot in the vicinity of 0. Sigmoid functions are not linear (thus solving the separability problem) and have other great properties, such as an easy to implement rep- resentation of differential.

The semi-quadratic function is designed to have certain desired properties of the sigmoid function plus an even easier implementation of its differential. A simple example of such a function (one implemented in out back-end network runner) is defined as:

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

The differential function of a semi-quadratic function is partialy linear, wich should improve training speed.

It has to be mentioned that there are two equiponderant ways of defining activation functions. Either the activation varies from -1 to 1 or from 0 to 1. The $[-1, 1]$ equivalent of the sigmoid function is the hyperbolic tangent (tanh) function. The semiquadratic functions could obviously be easily redefined to match the $[-1, 1]$ activation. The question is which activation function should one use? The anwser is not obvious. In our case, we would like most of the activations to be zero, and since tanh is a bit more linear in the proximity of zero than a typical sigmoid (a defined above) then that could be an argument for $[-1, 1]$ activations. Though changing the activation range could cause perfomance problems in some cases, it doesn't affect the computational power of a neural net. We decided to use the $[0, 1]$ activations, but our neural engine supports any range.

The basic question we want to ask is wether the network architecture affects visualisation. The simple answer is yes. Figure 6 shows the same dataset fed to three networks differing only in transfer functions. The networks were trained for a given amount of epochs and stopped, showing different results in MSE. The datafile is glass.dat. Previous experiments show that cathegory 5 (green) is the hardest to separate from cathegory 6 (yellow) using sigmoid networks. Cathegory 4 (cyan) on the other hand, seems to be immediately secognised and separated by the network.

If we change the hidden layer to semiquadratic, not much happens (not worth showing in the figure) until we change the output layer to linear. A 9-10-6 network with a sigmoid input layer, a semiquadratic hidden layer and a linear output seems to solve the problem immediately, with MSE dropping to 0.17 in the first training epochs. A suggestion that the effect might be caused by the linear layer alone (which tends to train easily on liearly separable problems) turns out to be false, as shown by the plot on the left - a sigmoid-linear network has worse results than an all-sigmoid network.

The efficiency of the three networks is different, as we can see by MSE alone, but that's not the only difference. Please note what happens to the previously mentioned cathegory represented as green - the sigmoid-sigmoid-linear network doesn't seem to separate it from yellow at all. The all-sigmoid network, given more time, separates it, but not cleanly. The sigmoid-semiquadratic-linear network, on the other

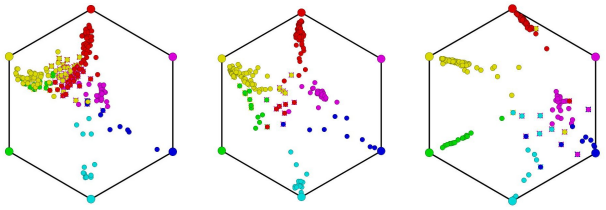

Figure 6: Three different networks were trained to separate the same data set for a given amount of epochs (all in their early training). All three networks have three layers sized 9, 10 and 6. The plot on the left is produced by a network with a linear last layer, others sigmoid, MSE is 0.25. The middle plot is produced by a sigmoid-only network, MSE=0.18. The plot on the right represents a network with its first layer sigmoid second semiquadratic, third - linear, MSE=0.16. Dispersion is set to average in all cases.

hand, has no problems separating this troublesome cathegory, having more problems with the cathegory marked as cyan.

As it was mentioned many times before - some applications require a neural network to separate the data cleanly, without renegade samples. Visualising outputs os networks of different archtectures and/or transfer functions can help decide on the best network for the job or suggest a different solution - a set of networks specialised in different cathegories, assigned by their plots. Sometimes one cathegory requires a different approach than the others.

It's important to mention that these particular architectures give the described results for the one dataset and the test was done solely to demonstrate a practical use for visualisation. A different dataset could favorise a different set of transfer functions. A different set of functions sould give better results in this case. The three networks were chosen mostly due to the clearly visible differences in the plots.

5 Acknowledgements

We thank our mentor, proffessor W"lodzis"law Duch, who had suggested us to take on this project. Great thanks goes to dr Norbert Jankowski, who provided us with practical information about the implementation of the visualisation module. Dr Tomasz Schreiber gained our gratitude by helping with various theoretical issues and helping us solve optimalisation problems. This project would never have worked without the help from Paolo Marrone, author of the Java Object Oriented Neural Network Engine, key component to the back-end of our experimental software.

References

[1] W. Duch *Uncertainty of data, fuzzy membership functions, and multi-layer perceptrons*

[2] W. Duch *Hidden secrets of feedforward neural networks*

[3] Paolo Marrone *Java object oriented neural engine* www.joone.org

[4] Stanisław Osowski *Sieci neuronowe do przetwarzania informacji*

[5] J. Korbicz, A. Obuchowicz, D. Uciński *Sztuczne sieci neuronowe, Podstawy i zastosowania*

[6] Robert A. Kosiński *Sztuczne sieci neuronowe, Dynamika nieliniowa i chaos*

[7] Stanisław Osowski *Sieci neuronowe w ujęciu algorytmicznym*

[8] W. Duch, J. Korbicz, L. Rutkowski, R. Tadeusiewicz *Biocybernetyka i in"rynieria biomedyczna 2000 - tom 6 Sieci neuronowe* Akademicka oficyna wydawnicza Exit, Warszawa 2000.