Mon. Not. R. Astron. Soc. **000**, 1–7 (2010) Printed 23 August 2010 (MN LATEX style file v2.2)

# 2D Function Classification using Neural Networks

Pascal S.P. Steger[1⋆], Ruedi Stoop[2], Stefan Martignoli[2]
[1]*Department of Physics, ETH Zürich, CH-8093 Zürich, Switzerland*
[2]*Institute for Neuroinformatics, University Zürich, CH-8093 Zürich, Switzerland*

23 August 2010

**ABSTRACT**
We present an application of neural networks to function classification and regression starting from a two-dimensional dataset. The aim is to apply it on general classification problems. We go the first steps by describing preparation of data and overall procedure, explaining the main part of the program, presenting sample runs and runtime characteristics, and showing limitations and not recognized functions.

We find that (i) all simple logical functions are learned efficiently, (ii) basic function classification is possible for all proposed functions with a two-layer forward connected neural network, (iii) learning efficiency is mostly influenced by the sigmoidal output function form, and learning parameter should be a little less than unity to allow efficient error minimization, (iv) the number of hidden neurons is to be chosen in the realm of 2 times input number such that most information can be stored, but overadaptation to input patterns is prevented.

**Key words:** artificial intelligence: neural networks, regression – methods: numerical

## 1 INTRODUCTION

Computers are used to simplify lives of human beings, in a very broad range of applications. A common task in exact sciences one would like to automate is regression of experimental data in order to quantify dependencies and compare them with models.

Classical approaches of regression theory use polynomial interpolation, trigonometric approximation or spline interpolation on data points (cf. Quarteroni et al. (2002)); in order to find an underlying function, one often has to search for a power law connecting two empirically defined observables.

Methods from artificial intelligence (U. Lämmel (2004), Rojas (1996) for an introduction) should be able to help with classification of the underlying functions, as they render human intuition (It is a sine!) treatable by computer (output: $2 \cdot \cos(1.3x)$).

The special environment we are investigating is classification of functions by neural networks with supervised learning (Jones et al. (1990),Poggio & Girosi (1990)), followed by fitting parameters. A possible application in astrophysics is efficient matching of the spectral energy distributions of distant galaxies to a catalogue of known distributions in order to infer luminosity, age, mass, and redshift.

### 1.1 Regression Methods from Artificial Intelligence

In order to round out the discussion on regression with artificial intelligence, we provide a short overview of different methods together with a possible implementation and their main advantages and shortcomings.

The simplest procedure is to create a two-layer network in order to match the input with an output vector containing information about function type and degree, by entering training data together with target output, and then backpropagating the error. The network is ready to match other, differing inputs after this training epoch.

Implementation is done in accordance with an enhanced Werbos construction, with two forward connected layers. An additional input for mimicking a threshold is not necessary for basic logic functions, and was found to have negligible influence for regression as well. Any division of output space in several unconnected patches can be represented by this type of network. This basic network is our choice no 1 for function classification.

If we do not want to invoke a teaching cycle with known output, we can implement a more general network as a discrete Hopfield network using Hebbian learning (Hopfield (1982)), where vectors of a given length are matched to a predefined set of shapes.

An energy matrix constructed from a training set is applied on a given pattern, evaluating the minimal resulting energy. Our choice against unsupervised learning was motivated by the fact that several functions do generate ambiguities after normalization, that should be taken care of by simulated annealing, which introduces several additional parameters not necessary with supervised learning.

Direct comparison with smeared-out function prototypes in real space is given using a holographic neuron. Its strength lies in the fact that it can recognize translated and rotated versions of the functions as well, given that we let it run on the Fourier components. As shown by Stoop et al. (2003) it is distinguished by a big storage capacity, fast evaluation, and stochastic resonance. An implementation of a holographic neuron is desirable for real world data with errors, which will be done in follow-up work. Core issue at the moment is FFT of 2 dimensional datapoints with noise.

⋆ E-mail: psteger@phys.ethz.ch

© 2010 RAS

With a little modification of the distance function, the genetic algorithm used for the solution of the travelling salesman problem by using coordinates of nodes as genes can be used to find a smooth path through the points. The fit can then be compared to runs on a set of previously run sample plots, where the function is known.

By defining a set of simple approximation and moving prescriptions, one can try to generate a path through the data points that minimizes curvature and distance from the sampling points. With the set of known functions mentioned above, the corresponding function would be determined.

An analog procedure to clustering of 2-dimensional data can be used to work out the groups and subgroups of frequencies in Fourier space, comparing them in a later step to previously computed frequencies of sample functions.

### 1.2 Aim

In this article we analyze a simple neural network that is constructed to classify functions using a given pool of known functions. The results should allow to generalize the method to any number of input variables, with hints as to what parameters should be chosen, real-world application to the mentioned classification problem of astrophysical SED being the ultimate goal.

The rest of the paper is laid out as follows: In section 2 we lay down all methods used: our choice of network structure, preparation of input, the learning method and output generation together with implementation details. Section 3 contains results on application of the network to basic logic functions, our predefined set of functions, and general data. The available parameter space is explored for limitations, regression errors, and runtime characteristics. We draw conclusions and summarize in section 4.

## 2 METHODS

### 2.1 Input Data

#### *2.1.1 Function Set*

The functions we want to classify can be drawn from a big class of functions, we representatively choose some simple function templates $f_{TD}(x) : \mathbb{R} \to \mathbb{R}$, where index $T$ stands for "type", and $D$ encodes the "degree", for implementation simplicity chosen as integer values. We work with

$$f_{1n}(x) = x^n, \quad (1)$$
$$f_{2n}(x) = \exp(nx), \quad (2)$$
$$f_{3n}(x) = x^{1/(n+1)}, \quad (3)$$
$$f_{4n}(x) = \log(nx), \quad (4)$$
$$f_{5n}(x) = \cos(n\pi x), \quad (5)$$
$$f_{6n}(x) = \exp(-(2n(x-0.5))^2), \quad (6)$$

where $n \in \{1, 2, 3, 4\}$. Fig. 1 shows the function templates, normalized to lie in $[0, 1] \times [0, 1]$. The similarity between $f_{1n}$, $f_{3n}$ and $f_{2n}$, $f_{4n}$ can give rise to classification problems which will be dealt with later.

#### *2.1.2 Sampling of Input*

Input is given as an unordered list of $\mathbf{r}_i = (x_i, y_i) \in \mathbb{R}^2$ values for datapoints, with optional errors $\mathbf{e}_i = (e_i^x, e_i^y) \in \mathbb{R}^2$ in the $x$- and $y$-direction. We do not want to perfectly fit the actual points, probably with a polynomial of high order, but rather find a curve that smoothly runs from point to point. This can be achieved with a smearing of the inputs. Assuming Poissonian errors we could sample each input by a set of $N$ points with Gaussian distribution in both dimensions around the exact input value. This would additionally account for the fact that an accumulation of points is rated higher, and that big errors dilute the importance of the respective point.

| input | | | normalized | | |
|---|---|---|---|---|---|
| (87.901,16.033) | ± | (1.217,3.629) | (0.935,0.069) | ± | (0.014,0.042) |
| (47.201,56.778) | ± | (5.137,0.619) | (0.466,0.542) | ± | (0.059,0.007) |
| (18.216,20.656) | ± | (1.553,2.618) | (0.132,0.123) | ± | (0.018,0.030) |
| ( 6.733,49.695) | ± | (1.757,2.502) | (0.000,0.460) | ± | (0.020,0.029) |
| (46.678,69.808) | ± | (2.745,2.905) | (0.460,0.693) | ± | (0.032,0.034) |
| (67.733,29.033) | ± | (0.358,0.330) | (0.703,0.220) | ± | (0.004,0.004) |
| (93.503,20.258) | ± | (1.670,3.388) | (1.000,0.119) | ± | (0.019,0.039) |
| (58.550,96.246) | ± | (4.789,0.806) | (0.597,1.000) | ± | (0.055,0.009) |
| (71.007,64.515) | ± | (0.622,1.421) | (0.741,0.632) | ± | (0.007,0.016) |
| (89.003,57.577) | ± | (5.159,3.849) | (0.948,0.551) | ± | (0.059,0.045) |
| (23.714,10.043) | ± | (2.548,2.890) | (0.196,0.000) | ± | (0.029,0.034) |
| (29.676,38.286) | ± | (3.537,3.023) | (0.264,0.328) | ± | (0.041,0.035) |

**Table 1.** Sample input before and after normalization.

We restrict ourself to equidistant sampling in $x$-direction such that only $y$ values are to be learned, yielding a reduction of the network size by a factor of roughly 2. This in turn implicates a reduction of factor 4 for the connection weight matrices, ultimately effectuating a speed-up in computation.

Several approaches sketched in the introduction work in Fourier space, e.g. by clustering frequencies. This is not necessary for the simple forward connected neural network that we chose here, a FFT thus being developed only for use by a holographic neuron in follow-up work.

#### *2.1.3 Normalization of Input*

The raw data is transformed to lie in the range $[0, 1]^2$ by translating with

$$\mathbf{r}_i' = \mathbf{r}_i - \tilde{\mathbf{r}}, \quad (7)$$
$$\tilde{\mathbf{r}} = \begin{pmatrix} \min_i x_i \\ \min_i y_i \end{pmatrix} \quad (8)$$

and scaling by

$$\mathbf{r}_i'' = \mathbf{s} \cdot \mathbf{r}_i', \quad (9)$$
$$\mathbf{e}_i'' = \mathbf{s} \cdot \mathbf{e}_i', \quad (10)$$
$$\mathbf{s} = \begin{pmatrix} s^x \\ s^y \end{pmatrix} = \begin{pmatrix} 1/(\max_i x_i - \min_i x_i) \\ 1/(\max_i y_i - \min_i y_i) \end{pmatrix}, \quad (11)$$

A sample normalization acting upon a vector of positions with errors is shown in table 1.

### 2.2 Neural Network

#### *2.2.1 Basic Structure*

From the available methods described in the appendix, we chose to implement a simple forward connected network with two hidden layers (cf. 2), following the script Stoop (2010), p. 101. This network can divide the available parameter space in several simply

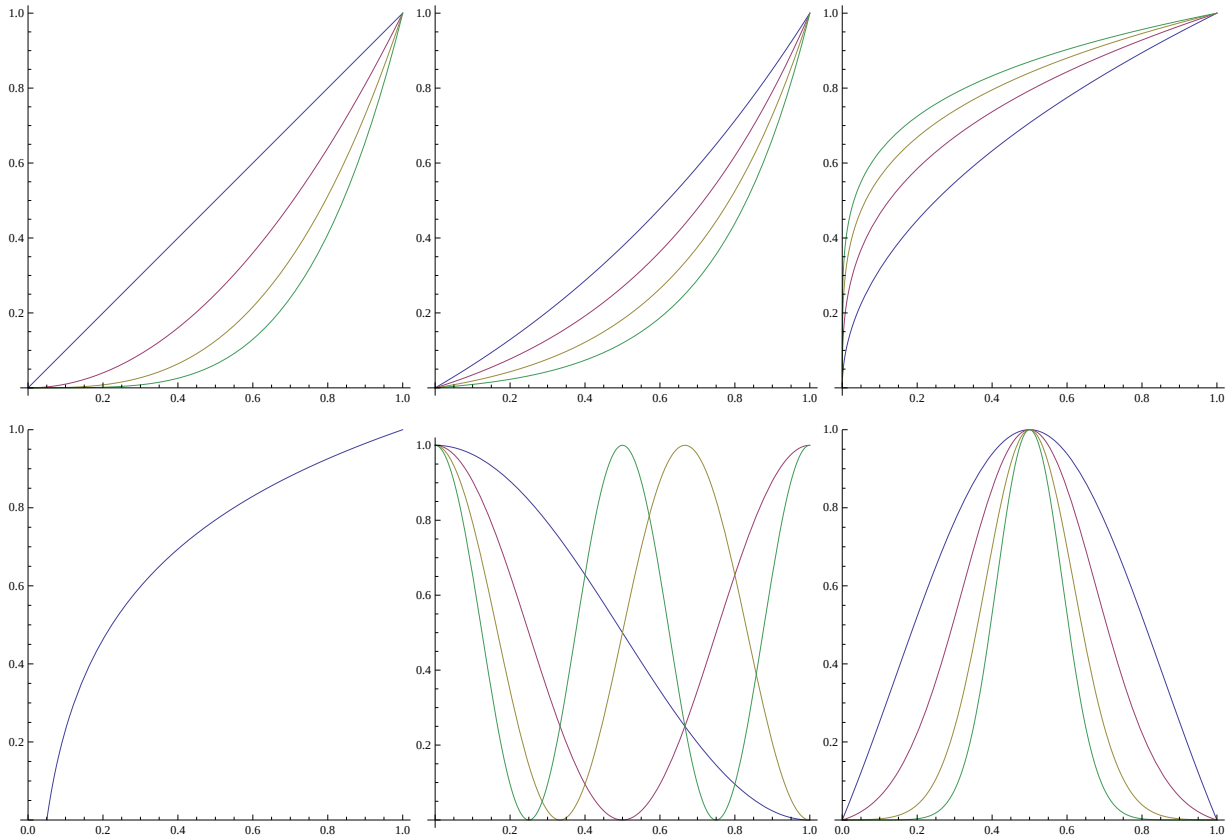

**Figure 1.** Function templates used for supervised learning. $f_{1n}, f_{2n}, f_{3n}$ are shown in the upper row, $f_{4n}, f_{5n}, f_{6n}$ in the lower one. Normalization of logarithmic plots reduces them to the same shape.

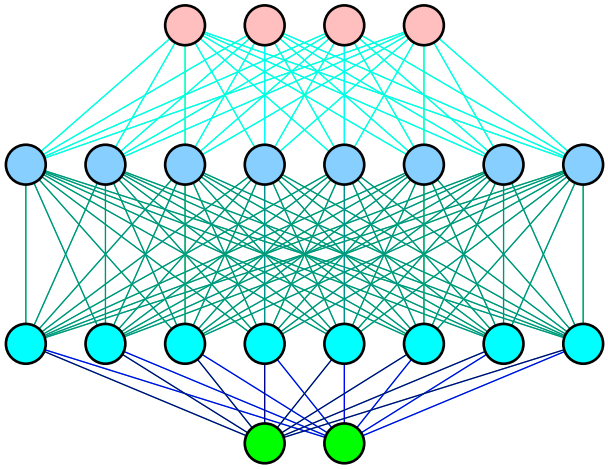

**Figure 2.** Basic topology of a forward connected network with two hidden layers. The shown configuration was used to map random input with 4 points at fixed $x$-coordinates to type and degree of the function, with the number of hidden neurons chosen such that the number of configurations to learn (4*4+1) is twice the number of neurons. Ability to classify noisy data is preserved.

connected domains, which we need in order to distinguish between a large number of function types $f_{T,D}$. Additional hidden layers do not enhance the functionality in a qualitative way; as a matter of fact the learning rate for backpropagation would be compounded.

#### *2.2.2 Parameters*

Parameters in the network include number of hidden neurons, distribution of starting connection weights, learning rate, output function, and number of iterations. These will be investigated later on. A parameter that is not investigated concerns the method of error propagation through the network. Backpropagation with momentum, resilient propagation, backpercolation, and QuickProp are only a few which would enable faster convergence.

### 2.3 Output

#### *2.3.1 Representation in Network*

The main output of the program is an equation $f(x)$ for the best fitting function. This is achieved in two steps: first, the basic function type and degree are determined, then they are put back into unnormalized coordinates.

Our six function classes and all of the four degrees are stored equidistantly in $[0, 1]$ so as to maximize assignment quality. Another trick to get better results with noisy data is the order in which the input functions are stored: $f_{1D}, f_{2D}$ and $f_{3D}, f_{4D}$ do lie next to each other, which ensures that the outputs for similar curves do not differ much. This enhances the probability that rounding to the next integer for function type ends up mostly in the surroundings of a function that looks like the meant one.

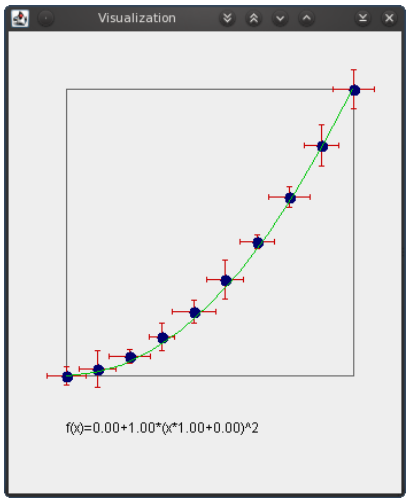


**Figure 3.** Sample visualization for fitting of an $x^2$

### 2.4 Error

The basic error function to be minimized in order to find the function type and degree is

$$e = \sum_{i=1}^{N} (y_i - f(x_i))^2, \tag{12}$$

taking into account the $y$-axis error only.

If we wanted a fit that includes $x$-direction as well for a real-world application, we would use

$$e = \sum_{i=1}^{N} (x_i - x_{m,i})^2 + (y_{m,i} - f(x_{m,i})^2 \tag{13}$$

where $(x_{m,i}, y_{m,i})$ denotes the coordinates of the nearest point on the fitting curve to a given data point.

#### *2.4.1 Fitting, Backtransformation*

During normalization, scaling and translation parameters are stored for later use. The fitted formula $\tilde{f}_{TD}(x)$ is then transformed back

$$f(x) = t_y + s_y * \tilde{f}_{TD}(s_x * x + t_x) \tag{14}$$

### 2.5 Visualization

Visualization of the input data with error bars and fitting formula is performed in the normalized coordinates, ranging over $[0, 1] \times [0, 1]$ in the two-dimensional plane. See fig. 3 for a sample output.

### 2.6 Implementation

#### *2.6.1 Computation environment*

The program is written in JAVA, where we profited from programming and debugging aids in the ECLIPSE environment. Visualization is accomplished with SWING API. The presented tests were run on an AMD Phenom 9750 processor using 64bit registers and a total of 8GB RAM. Threading was not implemented, since the crucial part of the program being backpropagation with online updating would corrupt parallel computations.

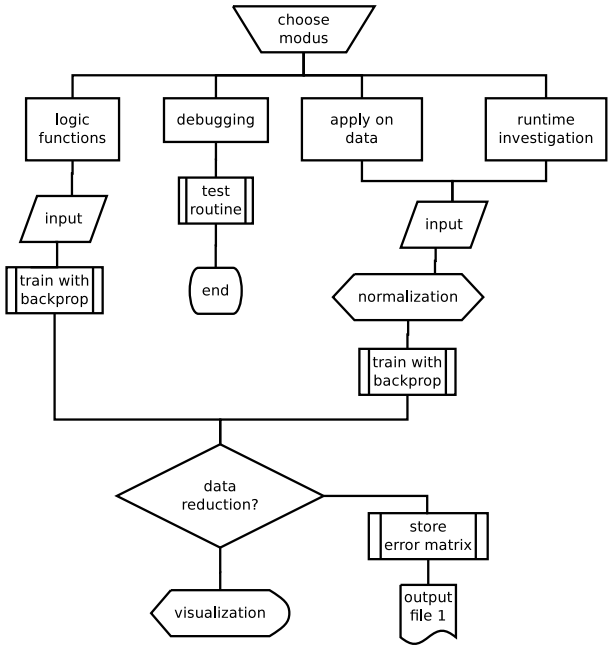


**Figure 4.** Basic architecture of Java program

#### *2.6.2 Program Architecture*

Fig. 4 gives an overview of the program architecture: after choice of basic program mode, the training data is generated, the network trained following Stoop (2010), and afterwards applied different test input. Further data reduction needs error output to external files; in all other cases online output of the found function is sufficient.

## 3 RESULTS

### 3.1 Logic Functions

We let the neural network learn basic logic functions with two inputs as a means to rule out basic errors. In the end, quite good approximations were found with a network incorporating more than $2 * 2 + 1$ hidden neurons in each layer, with strongest replication of equal output values and weakest coincidence for the XOR and NOT(XOR), see table 3.1.

### 3.2 Network applied on Training Set

Using the neural network with 10 sample points, each 20 neurons on the first and second hidden layer, learning rate 1.0 and sigmoid parameter 0.1, we get the values in table 3 when applied to the input dataset.

Most of the functions are classified correctly once the output values are rounded to the nearest integer. An exception being degrees from the logarithmic functions which are mapped to the same value, an expression of the fact that a simple scaling and translation is undone by normalization of the input values.

$$\log(D \cdot x) = \log(x) + \log(D) \rightarrow \log(x) \tag{15}$$

We define the quality of approximation in analogy to the logic function example as as the difference to target value squared, either

| $(a,b) = (0,0)$ | $(0,1)$ | $(1,0)$ | $(1,1)$ | error $e$ |
|---|---|---|---|---|
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0000 |
| 1.0 | 0.0 | 0.0 | 0.0 | 0.0005 |
| 0.0 | 1.0 | 0.0 | 0.0 | 0.0005 |
| 1.0 | 1.0 | 0.0 | 0.0 | 0.0004 |
| 0.0 | 0.0 | 1.0 | 0.0 | 0.0004 |
| 1.0 | 0.0 | 1.0 | 0.0 | 0.0003 |
| 0.0 | 1.0 | 1.0 | 0.0 | **0.0021** |
| 1.0 | 1.0 | 1.0 | 0.0 | 0.0007 |
| 0.0 | 0.0 | 0.0 | 1.0 | 0.0009 |
| 1.0 | 0.0 | 0.0 | 1.0 | **0.0010** |
| 0.0 | 1.0 | 0.0 | 1.0 | 0.0004 |
| 1.0 | 1.0 | 0.0 | 1.0 | 0.0005 |
| 0.0 | 0.0 | 1.0 | 1.0 | 0.0003 |
| 1.0 | 0.0 | 1.0 | 1.0 | 0.0004 |
| 0.0 | 1.0 | 1.0 | 1.0 | 0.0006 |
| 1.0 | 1.0 | 1.0 | 1.0 | 0.0000 |

**Table 2.** Output of neural network of size $(n_{\rm in}, n_{\rm hidden,1}, n_{\rm hidden,2}, n_{\rm out}) = (2,8,8,1)$ on all logic functions $f(a,b)$ after 1000 learning steps with $\eta = 1.0$. Error is taken to be square of norm, $e = \sum_{i=1}^{N}(o-t)^2$.

| type | degree | found type | found degree |
|---|---|---|---|
| 1 | 1 | 0.968 | 1.018 |
| 1 | 2 | 1.000 | 2.022 |
| 1 | 3 | 1.024 | 3.035 |
| 1 | 4 | 1.019 | 3.982 |
| 2 | 1 | 2.003 | 1.016 |
| 2 | 2 | 2.019 | 2.012 |
| 2 | 3 | 1.983 | 2.989 |
| 2 | 4 | 2.010 | 3.938 |
| 3 | 1 | 2.955 | 1.041 |
| 3 | 2 | 3.039 | 2.008 |
| 3 | 3 | 2.972 | 3.029 |
| 3 | 4 | 3.029 | 3.901 |
| 4 | $1\ldots4$ | **4.001** | **2.587** |
| 5 | 1 | 5.010 | 1.075 |
| 5 | 2 | 5.026 | 2.065 |
| 5 | 3 | 4.997 | 3.086 |
| 5 | 4 | 4.991 | 3.974 |
| 6 | 1 | 5.906 | 1.077 |
| 6 | 2 | 5.965 | 2.089 |
| 6 | 3 | 5.974 | 3.061 |
| 6 | 4 | 5.938 | 3.946 |

**Table 3.** Output of neural network applied on training data set.

from given output directly or after rounding to nearest $n \in \mathbb{N}$.

$$e_1(T,D) \equiv (T - T_{\rm tar})^2 + (D - D_{\rm tar})^2 \quad (16)$$

$$e_2(T,D) \equiv (\mathrm{round}(T) - T_{\rm tar})^2 + (\mathrm{round}(D) - D_{\rm tar})^2 \quad (17)$$

$T$ denotes type of function, $D$ is the degree. In notation of eq. 1 the according function reads $f_{TD}(x)$.

### 3.3 Parameter Dependence

How does the performance of our network depend on the parameters? We divide between parameters inherent to the neural network, as learning rate, number of iterations, stretching of the sigmoid function and measure of noise on the input?

#### *3.3.1 Noisy Input*

Noisy input is generated by taking the trained functions and adding noise, characterized by maximal disturbance $\delta$. Noise does only induce misclassification for $f_{1D}, f_{2D}, f_{3D}, f_{5D}$ if it is bigger than $\delta = 0.08$. For $f_{4D}$ we do not expect small errors as all degrees should be mapped to one value. Interestingly there are no measurable effects of noise for the Gaussian distributions $f_{6D}$, at least up to $\delta = 0.1$ they are well separated.

#### *3.3.2 Learning Rate and Number of Iterations*

The error in classification does depend on the number of iterations, see fig. 6 for the exact dependence broken down into the different function classes. We see that error is reduced in two steps: after the first 2000 iterations, approximately 3 out of 24 functions are classified wrongly, this even holds when rounding is applied. After 20000 steps, this is further reduced and almost every time the classification is right for functions in $f_{1n}, f_{2n}, f_{5n}, f_{6n}$. The errors showing up in $f_{4n}$ are due to the fact that the degree cannot be learned after normalization happens for the logarithmic function, the type being mixed with $f_{3n}$ from the obvious marginal cases where logarithmic and root functions look similar.

Learning rate and number of iterations is investigated next, with results summarized in table 4. The trend for better quality of fit with growing number of iterations discussed beforehands is visible when considering the evolution for fixed $\eta$. Reading the table along the other dimension, we find that worst recognition is obtained with least lerning: the maxima of each row lie in the realm of small learning rates. It is rapidly improved once $\eta = 0.3$ is encountered and stays mostly constant for larger values. Best results are obtained for $\eta = 0.9$ and not $\eta = 1.0$, probably triggered by the fact that this additional damping of oscillations around minima in the error landscape during adjustment of connection weights helps to converge faster. For the first part of convergence (the 2000 steps mentioned before) this is not relevant, though; it is only after 4000 steps that full propagation of the adjustments causes a temporal rise of classification error.

#### *3.3.3 Shape of Sigmoid Function*

So far, the intuitive parameters for convergence were examined. A simple adaptation of the network working on logical functions to the function classification scheme by mere scaling did not work, even when experimenting with different choices of learning rate and number of iterations. The reason was that the biggest impact on learning efficiency comes from stretching the sigmoidal output function of each neuron, which is in fact so strong that it easily can suppress any progression to completion. The first runs of the neural network yielded no result due to the fact that random initial values for the connection weights add up for a large number of neurons in one layer and are mapped by the vanilla sigmoid function to a value $1 + \epsilon$, $\epsilon \ll 1 < /math >$. Only if the sigmoid function is

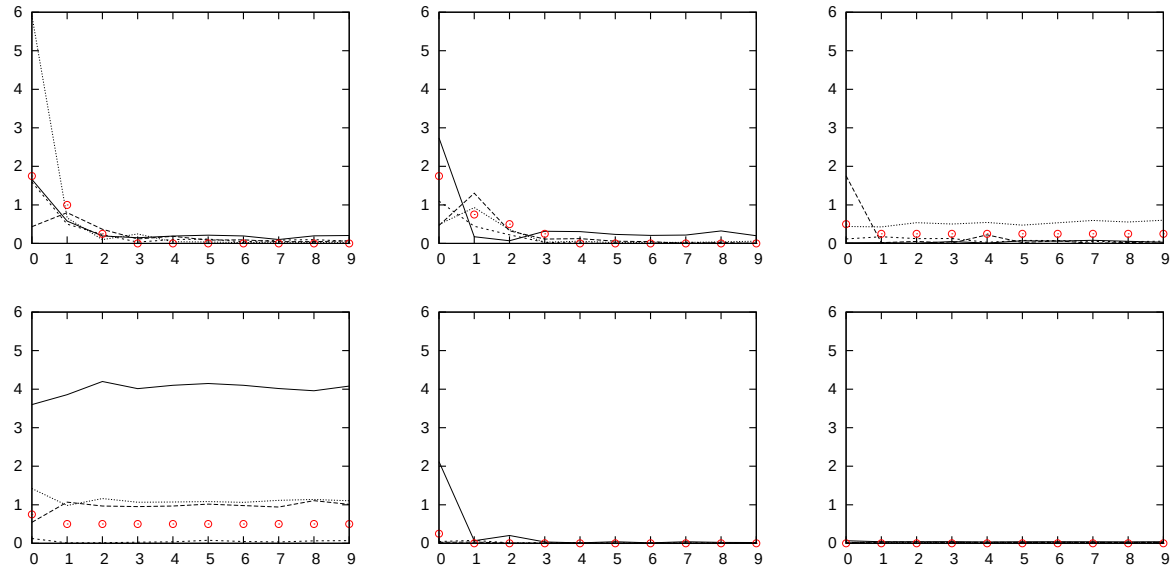

**Figure 5.** Classification error $e_1$ for each function as a function of noise is shown in black, where the label for the abscisse is $10(1-\delta)$ . Every panel holds all $f_{TD}$ for fix $T$. The red circles indicate $e2$ made by taking the rounded values for $T$ and $D$, averaged over all functions of a given type.

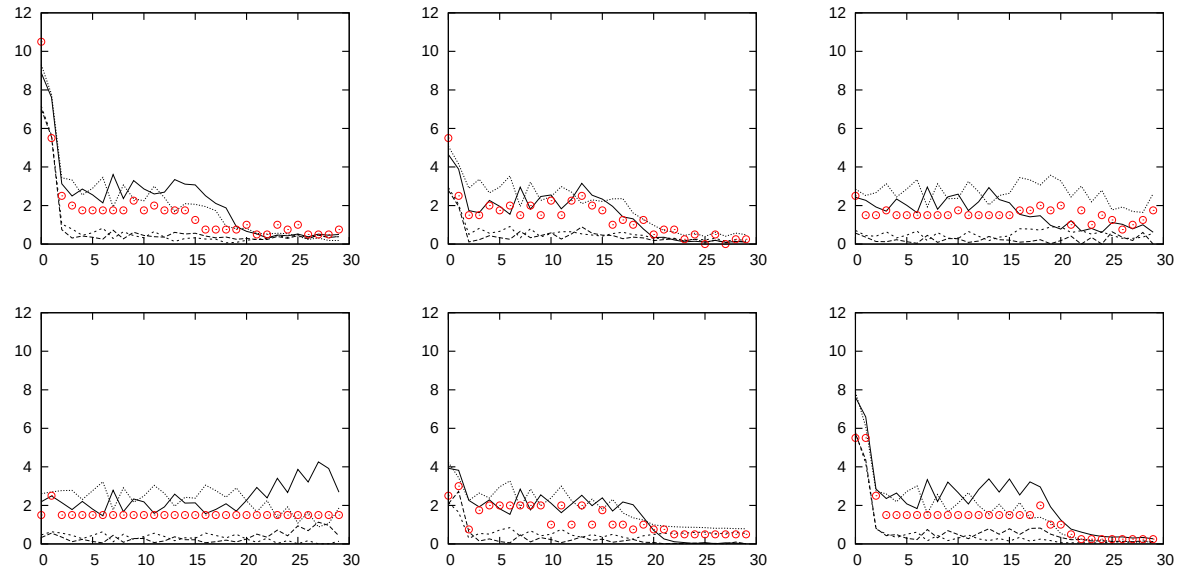

**Figure 6.** Classification error $e_1$ for each function as a function of learning steps is shown in black. Every panel holds all $f_{TD}$ for fix $T$. The values on the abscissa are scaled in 1000-steps. The red circles indicate $e2$ made by taking the rounded values for $T$ and $D$, averaged over all functions of a given type.

stretched by setting small $a$ in

$$\mathrm{sig}_a(x) \equiv \frac{1}{1+\exp(-ax)} \tag{18}$$

the changes in weight get bigger than machine precision. Our preliminary choice of $a = 0.1$ still requires the high number of 25000 iterations per input pattern to get approximately 90% of the functions recognized correctly. In fig. 7 the effect of simoidal steepness $a$ on classification error after $10^6$ iterations. As long as $a < 0.9$, there is hope for actual recognition, best for some values around $a \approx 0.2$.

#### *3.3.4 Number of Hidden Neurons*

The number of hidden neurons on both layers are kept at the same value. Varying this number has both impacts on quality of fit and runtime, see 5.

### 3.4 Limitations

Limitations of our code mainly arise from the degeneracies of several functions. Additional heuristic knowledge needs to be implemented in order to let the network propose ”simple” function forms.

| $N_{it} =$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $\eta = 0.1$ | 98.3 | 85.0 | 45.3 | 34.3 | 33.8 | 33.8 | 33.7 | 33.7 | 33.8 | 34.1 |
| $\eta = 0.2$ | 77.6 | 35.1 | 34.2 | 33.7 | 33.7 | 33.5 | 33.3 | 32.7 | 29.3 | 20.1 |
| $\eta = 0.3$ | 34.4 | 33.6 | 33.4 | 32.5 | 22.3 | 14.1 | 13.5 | 13.2 | 12.5 | 11.8 |
| $\eta = 0.4$ | 33.9 | 33.6 | 33.0 | 21.8 | 14.2 | 12.8 | 11.4 | 10.5 | 10.3 | 10.4 |
| $\eta = 0.5$ | 33.8 | 33.9 | 33.3 | 19.5 | 13.4 | 12.6 | 11.3 | 10.2 | 10.5 | 10.2 |
| $\eta = 0.6$ | 33.7 | 33.4 | 19.5 | 12.8 | 12.5 | 13.2 | 10.2 | 10.1 | 9.3 | 9.8 |
| $\eta = 0.7$ | 33.6 | 31.0 | 12.9 | 11.5 | 11.1 | 10.3 | 10.3 | 8.9 | 9.6 | 9.4 |
| $\eta = 0.8$ | 34.2 | 33.2 | 13.5 | 12.0 | 11.4 | 10.4 | 10.1 | 10.8 | 8.5 | 9.9 |
| $\eta = 0.9$ | 33.4 | 14.1 | 11.6 | 10.6 | 9.3 | 8.5 | 7.8 | 7.1 | 6.6 | 6.5 |
| $\eta = 1.0$ | 34.3 | 14.3 | 12.0 | 10.7 | 12.5 | 10.3 | 9.3 | 9.5 | 10.4 | 9.7 |

**Table 4.** Overall classification error as function of learning rate $\eta \in [0.1, 1.0]$ and number of iterations $N_{\rm it} = n_{\rm it}/10^4 \in [1, 10]$.

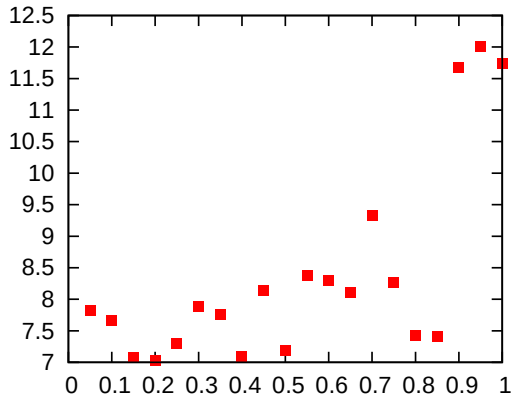


**Figure 7.** Total classification error $e$ as a function of sigmoid parameter $a$, with fixed $\eta = 0.9$ and $n_{it} = 10^6$.

| $n_{\rm hid,1,2}$ | $e$ | $t$[s] |
|---|---|---|
| 1 | 86.23 | 1.7 |
| 2 | 30.56 | 1.9 |
| 3 | 14.84 | 2.5 |
| 4 | 12.38 | 2.9 |
| 5 | 9.31 | 3.5 |
| 6 | 10.79 | 4.0 |
| 7 | 10.00 | 4.7 |
| 8 | 9.32 | 5.3 |
| 9 | 9.33 | 6.0 |
| 10 | 7.32 | 6.7 |
| 15 | 7.64 | 11.3 |
| 20 | 6.58 | 17.5 |

**Table 5.** Dependence of quality of fit after $10^5$ iterations on number of hidden neurons, together with runtime characteristics.

## 4 DISCUSSION

### 4.1 Summary

We presented the architecture of a program to simulate a neural network with two hidden layers, aiming at classification of a set of basic functions. Introducing a measure for the quality of fit allowed us to investigate dependence on different parameters. Learning efficiency is foremost influenced by the sigmoidal output function form. Little noise does not influence most fits. Learning parameter and number of iterations need a short sample run to find the optimal values; the learn parameter should be a little less than unity to allow efficient error minimization. A tradeoff between accuracy of fit for the training functions, generality and computing resources leads to a choice for the number of hidden neurons; twice the input number is a starting point.

### 4.2 Enhancements

The environment is prepared to scale to more sampling points, random input, to find a whole linear superposition of fitting formulae to approach the data even further. The proposed methods can moreover be generalized for fitting parametric functions to three-dimensional datasets. A possible extension would search for formulae of arbitrary two-dimensional surfaces, or even try to classify three-dimensional objects in topological classes.

## 5 ACKNOWLEDGMENTS

I would like to thank Ruedi Stoop for stimulating discussions and for creating such a pleasant atmosphere in class. Stefan Martignoli gave numerous hints how to implement algorithms in MATHEMATICA, which was helpful to get a working knowledge on backpropagation.

## REFERENCES

Hopfield J. J., 1982, Proceedings of the National Academy of Sciences of the USA, 79, pp. 2554
Jones R. D., Lee Y. C., Barnes C. W., Flake G. W., Lee K., Lewis P. S., Qian S., 1990, Proceedings of the International Joint Conference on Neural Networks, pp p. I–649
Poggio T., Girosi F., 1990, Proc. IEEE, 78, 1484
Quarteroni A., Sacco R., Saleri F., 2002, Numerische Mathematik 2. Springer Berlin
Rojas R., 1996, Neural Networks - A Systematic Introduction. Springer-Verlag, Berlin, New-York
Stoop R., 2010, Theorie und Simulation Neuronaler Netze
Stoop R., Buchli J., Keller G., Steeb W.-H., 2003, Phys. Rev. E
U. Lämmel J. C., 2004, Künstliche Intelligenz. Carl Hanser Verlag München Wien