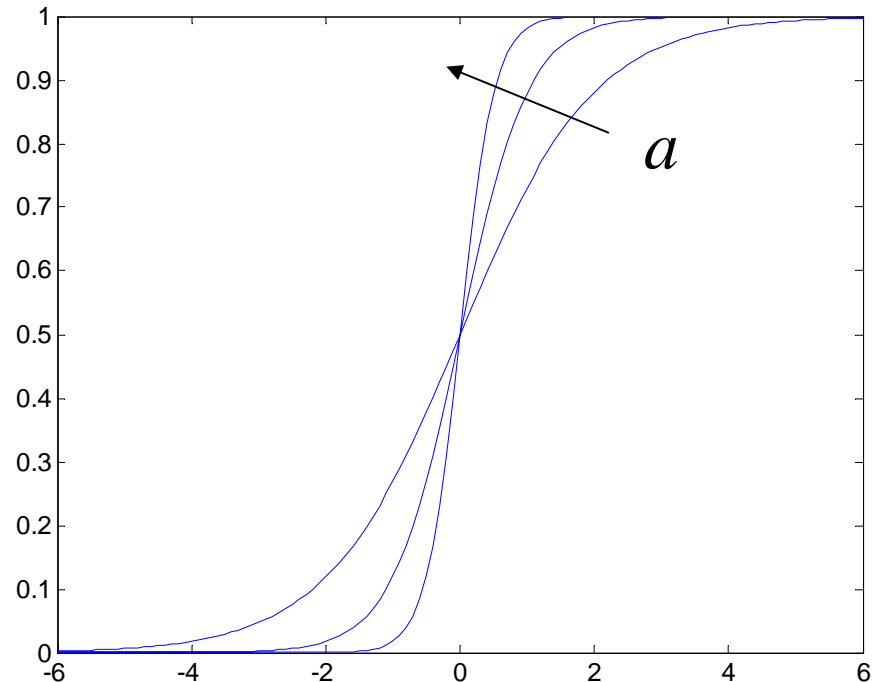


Learning strategies for neuronal nets - the backpropagation algorithm

In contrast to the NNs with thresholds we handled until now NNs are the NNs with non-linear activation functions $f(x)$. The most common function is the *sigmoid function* $\psi(x)$:

$$f(x) = \psi(x) = \frac{1}{1 + e^{-ax}}$$

It approximates with increasing a the step function $\sigma(x)$.



The function $\psi(x)$ is closely related to the tanh function. For $a=1$ applies:

$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1 = 2\psi(2x) - 1$$

If the two parameter a and b are added, a more general form results:

$$\psi_g(x) = \frac{1}{1 + e^{-a(x-b)}} - 1$$

with a controlling the sheerness and with b controlling the position on the x -axis.

The non-linearity of the activation function is the core reason for the existence of the multi-layer perceptron; without this property the multi-layer perceptron would coincide with a trivial linear one-layered network.

The differentiability of $f(x)$ allows us to apply necessary constraints $(\partial(\cdot)/\partial w_{i,j})$ for optimization of the weight coefficients $w_{i,j}$.

The first derivate of the sigmoid function can be ascribed to itself:

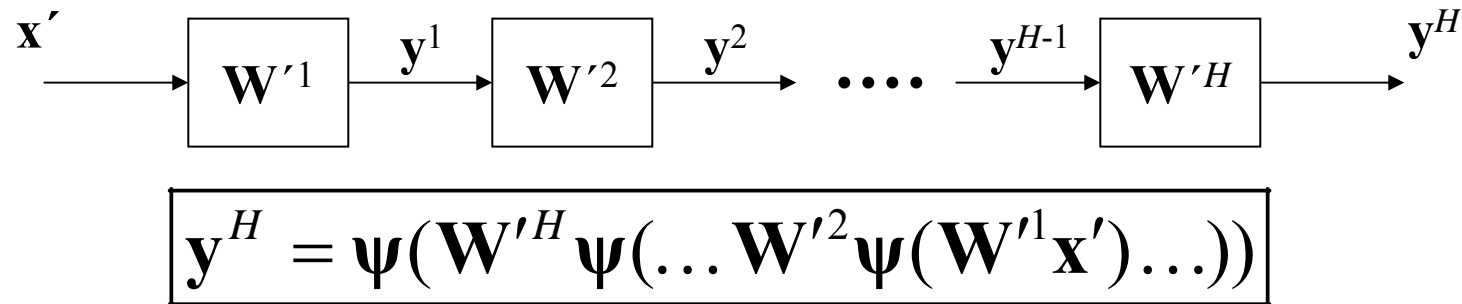
$$\frac{d\psi}{dx} = \frac{-1}{(1 + e^{-x})^2} (-1)e^{-x} = \frac{1}{1 + e^{-x}} \left(1 - \frac{1}{1 + e^{-x}}\right) = \psi(x)(1 - \psi(x))$$

Figure of the extended weight matrix

$$\mathbf{W}' = \begin{bmatrix} w_{1,0} = b_1 & w_{1,1} & w_{1,2} & \cdots & w_{1,N} \\ w_{2,0} = b_2 & w_{2,1} & w_{2,2} & \cdots & w_{1,N} \\ w_{3,0} = b_3 & w_{3,1} & w_{3,2} & \cdots & w_{1,N} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ w_{M,0} = b_M & w_{M,1} & w_{M,2} & \cdots & w_{M,N} \end{bmatrix} = [\mathbf{b}, \mathbf{W}]$$

The neural net with H layers

The multi-layer NN with H layers has a peculiar weight matrix \mathbf{W}'^i in every layer, but identical sigmoid-functions:



The learning is based on adjustment of the weight matrices with minimization of a square error criterion between target value \mathbf{y} and approximation $\hat{\mathbf{y}}$ by the net (supervised learning) in mind. The expected value has to be formed by the available training ensemble of $\{\hat{\mathbf{y}}_j, \mathbf{x}_j\}$:

$$J = \min_{\mathbf{W}^i} E \left\{ \|\hat{\mathbf{y}} - \mathbf{y}\|^2 \right\} = \min_{\mathbf{W}^i} E \left\{ \|\mathbf{y}^H - \mathbf{y}\|^2 \right\}$$

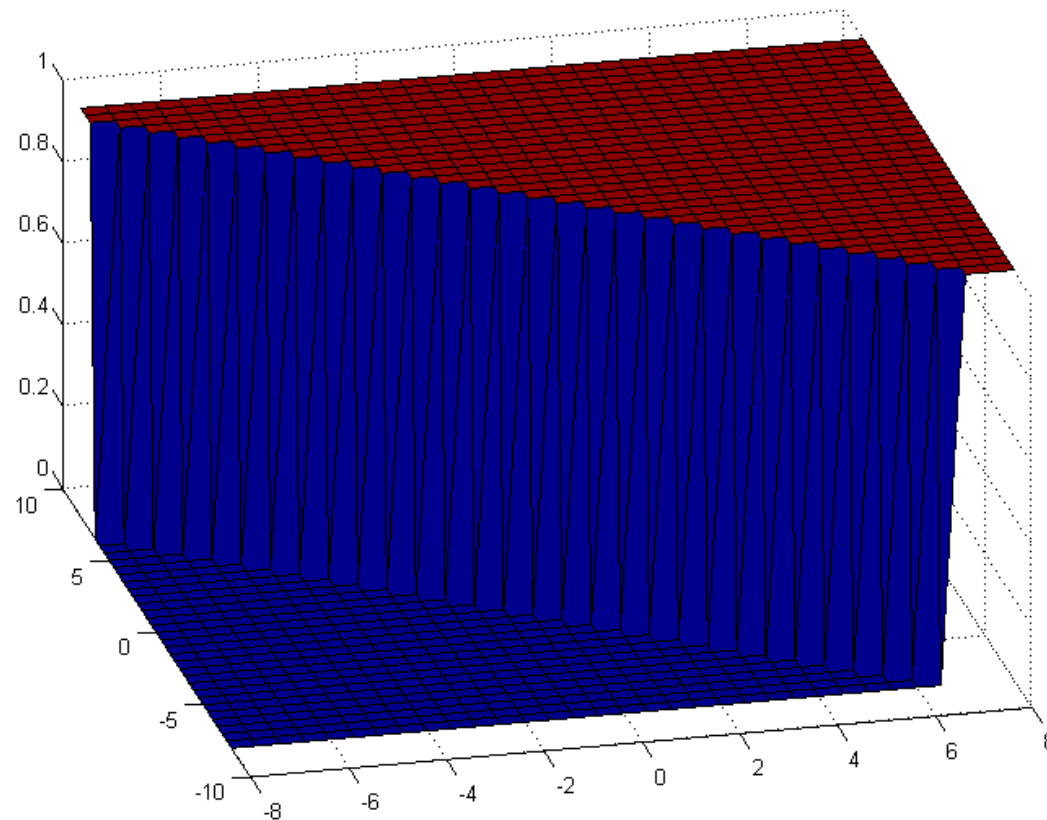
Note: The one-layer nn and the linear polynom classifier are identical, if the activation function is set to $\psi \equiv 1$.

Relations to the concept of function approximation by a linear combination of base functions

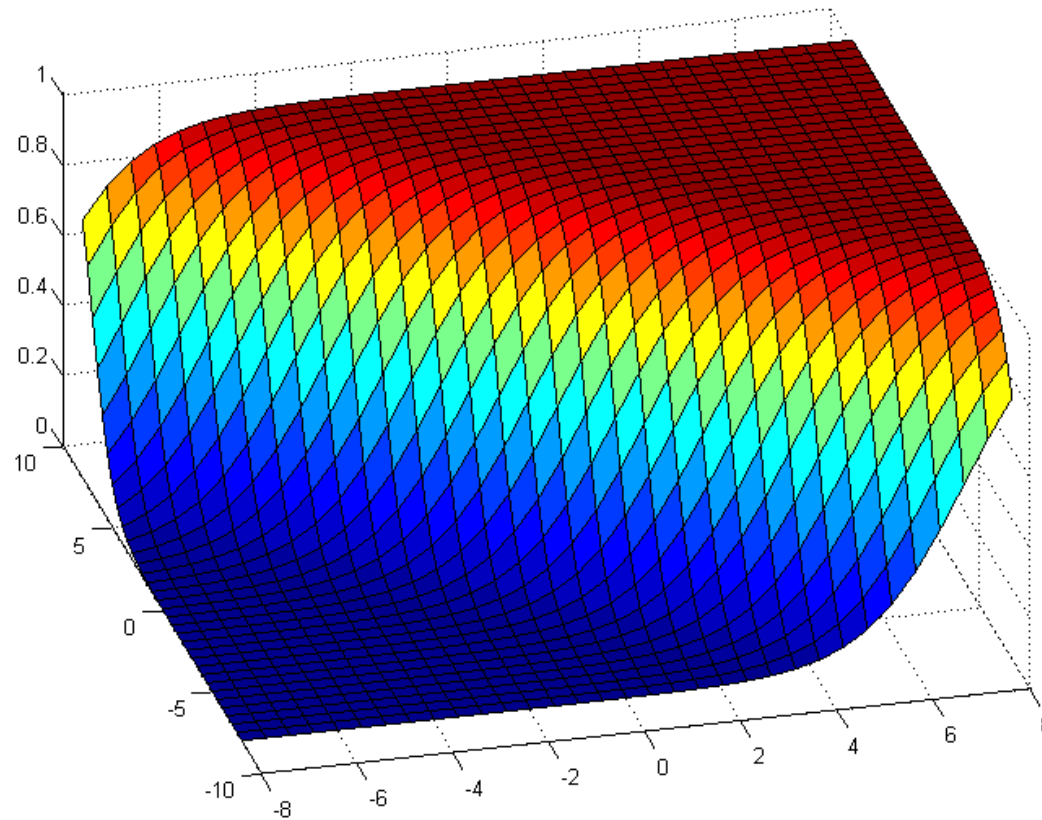
The first layer creates the base functions in form of a hidden vector \mathbf{y}_1 and the second layer forms a linear combination of these base functions.

Therefore the coefficient matrix \mathbf{W}'^1 of the first layer controls the *appearance* of the base functions, while the weight matrix \mathbf{W}'^2 of the second layer contains the coefficients of the *linear combination*. Additionally the matrix is weighted through the activation function.

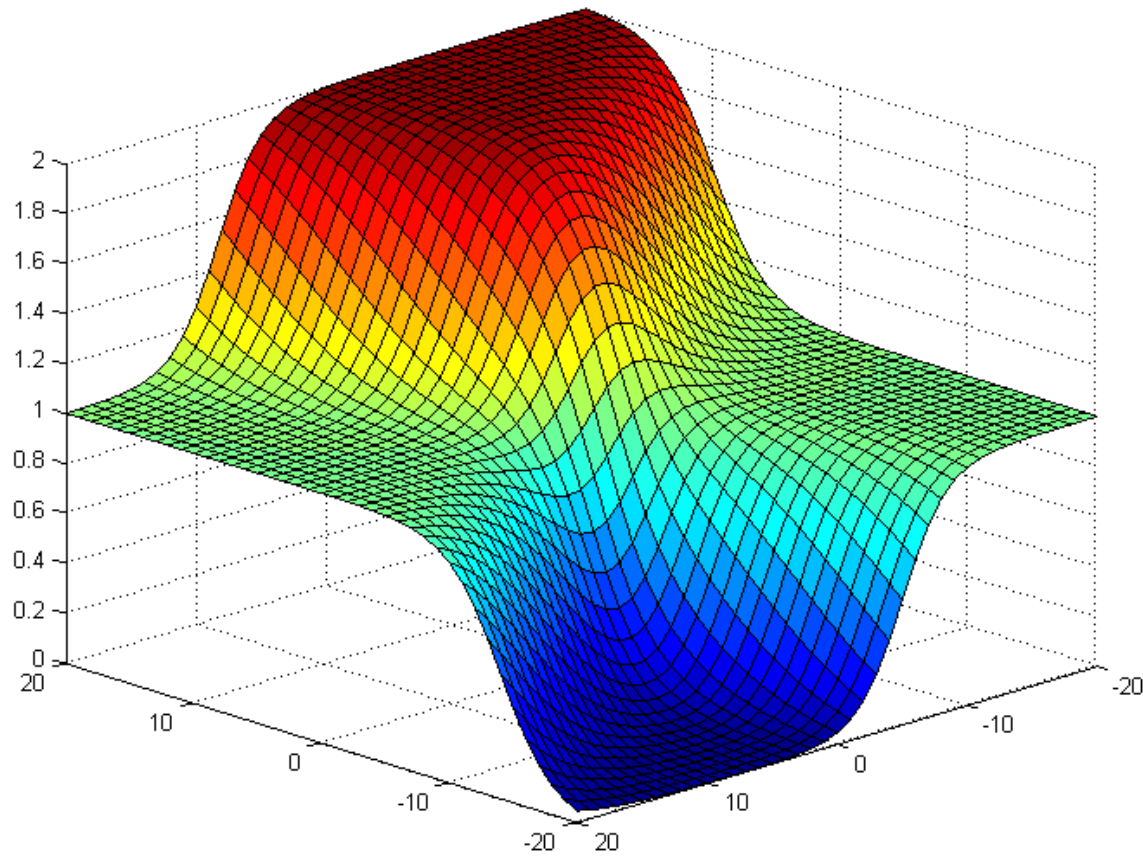
Reaction of a neuron with two inputs and a threshold function σ



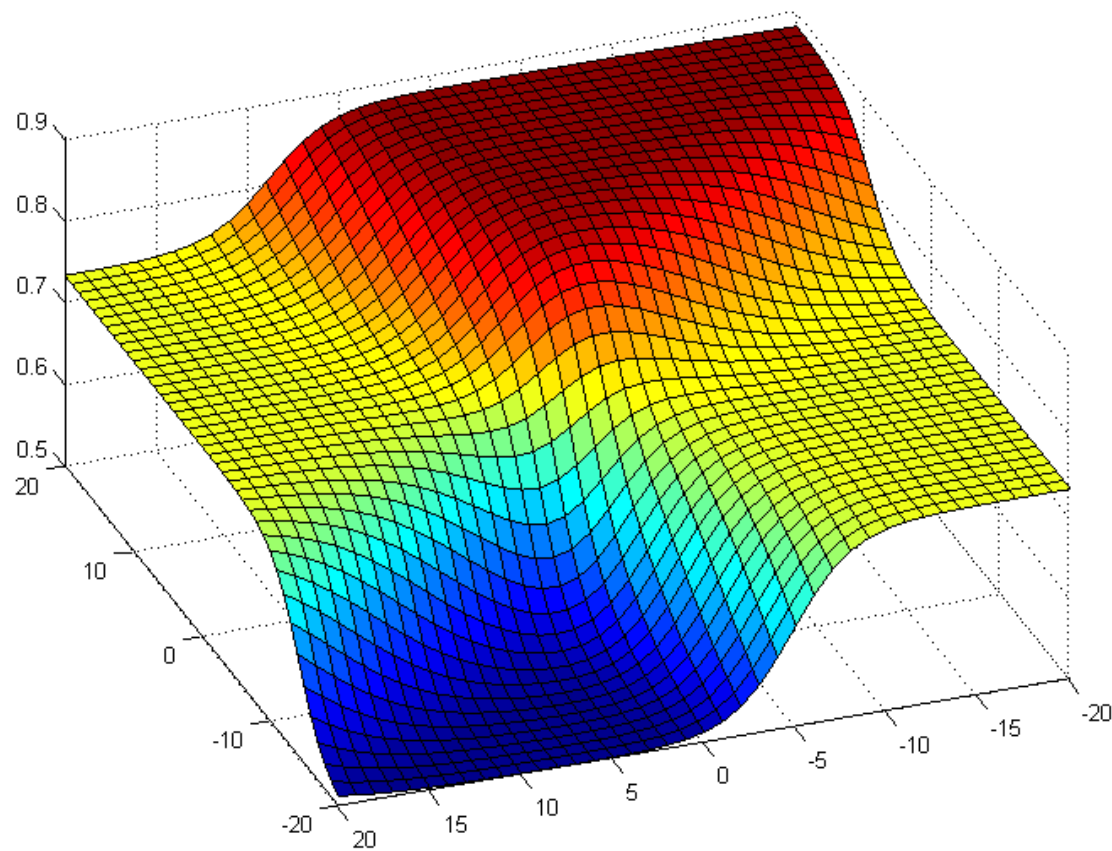
Reaction of a neuron with two inputs and a sigmoid function ψ



Overlapping two neurons with two inputs each and a sigmoid function



Reaction of a neuron in the *second* layer to two neurons of first layer after valuation by sigmoid function



The backpropagation learning algorithm

- The class map is done by a multi-layer perceptron, which produces a 1 on the output in the regions of \mathbf{x} , that are populated by the samples of the according class, and produces a 0 in the areas allocated by other classes. In the areas in-between and outside an interpolation resp. extrapolation is done.
- The network ist trained based on the optimization of the squared quality factor:

$$J = E \left\{ \left\| \hat{\mathbf{y}}(\mathbf{W}''^i, \mathbf{x}') - \mathbf{y} \right\|^2 \right\}$$

- This term is non-linear both in the elements of the input-vector \mathbf{x}' , and in the weight coefficients $\{w'_{ij}{}^h\}$.

- From insertion of the function map of the multi-layer perceptron results:

$$J = E \left\{ \left\| \underbrace{\psi(\mathbf{W}'^H \dots \psi(\mathbf{W}'^2 \psi(\mathbf{W}'^1 \mathbf{x}') \dots))}_{\hat{\mathbf{y}}} - \mathbf{y} \right\|^2 \right\}$$

- Sought is the *global* minimum. It is not clear whether such an element exists or how many *local* minima exist.

The backpropagation algorithm solves the problem iteratively with a gradient algorithm. One iterates according to:

$$\boxed{\mathbf{W}' \leftarrow \mathbf{W}' - \alpha \nabla \mathbf{J}} \quad \text{with: } \mathbf{W}' = \{ \mathbf{W}'^1, \mathbf{W}'^2, \dots, \mathbf{W}'^H \}$$

$$\text{and the gradients: } \nabla \mathbf{J} = \frac{\partial J}{\partial \mathbf{W}'} = \left\{ \frac{\partial J}{\partial w'_{nm}} \right\}$$

- The iteration terminates, when the gradient disappears at a (local or even global) minimum.
- A proper choice of α is difficult. Small values increase the number of required iterations. High values reduce the probability of running into a local minimum, risking the method to diverge and not finding the minimum (or giving rise to oscillations).

- The iteration has only *linear* convergence (gradient algorithm).
- Calculating the gradient:
For determination of the composed function

$$\hat{\mathbf{y}}(\mathbf{x}') = \Psi(\mathbf{W}'^H \dots \Psi(\mathbf{W}'^2 \Psi(\mathbf{W}'^1 \mathbf{x}') \dots))$$

- the chain rule is needed.
- The error caused by a sample results in:

$$J_j = \frac{1}{2} \left\| \hat{\mathbf{y}}(\mathbf{x}_j) - \mathbf{y}_j \right\|^2 = \frac{1}{2} \sum_{k=1}^{N^H} (\hat{y}_k(j) - y_k(j))^2$$

- The expected value $E\{\dots\}$ of the gradient has to be approximated by a mean value of all sample gradients:

$$\nabla \mathbf{J} = \frac{1}{n} \sum_{j=1}^n \nabla \mathbf{J}_j$$

- We have to distinguish between
 - individual learning based on the last sample

$$[\mathbf{x}_j, \mathbf{y}_j] \Rightarrow \nabla \mathbf{J}_j$$

- and cumulative learning (batch learning), based on

$$\nabla \mathbf{J} = \frac{1}{n} \sum_{j=1}^n \nabla \mathbf{J}_j$$

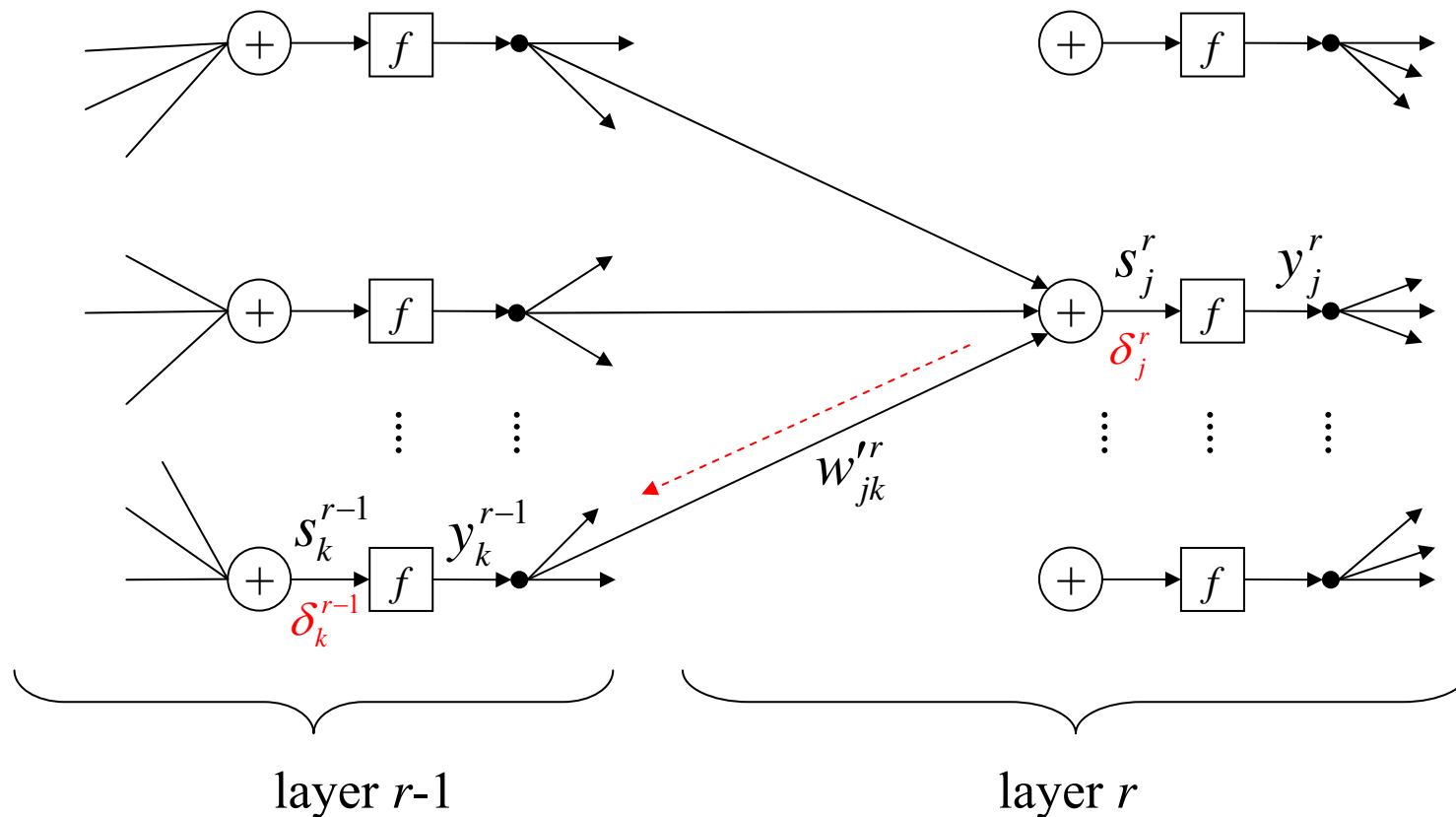
- Partial derivations for *one* layer:
For the r -th layer applies:

$$y_m^r = \psi(s_m^r) = \psi \left(\sum_{n=0}^{N^r} w_{nm}^{\prime r} x_n^{\prime r} \right)$$

$x_n^{\prime r}$ n-th input of layer r

y_m^r m-th output of layer r

Definition of the variables of two layers for the backpropagation algorithm



- First of all we calculate the effect of the last hidden layer on the starting layer $r=H$.

Using partial derivations of the quality function J (actually J_j , but j is omitted for simplification reasons) and applying the chain rule results:

$$\frac{\partial J}{\partial w'_{nm}} = \frac{\partial J}{\partial s_n^H} \cdot \frac{\partial s_n^H}{\partial w'_{nm}} = -\delta_n^H \underbrace{\frac{\partial s_n^H}{\partial w'_{nm}}}_{=y_m^{H-1}}$$

- introducing the *sensibility* of cell n

$$\delta_n = -\frac{\partial J}{\partial s_n}$$

- results:
$$\delta_n^H = -\frac{\partial J}{\partial s_n^H} = -\frac{\partial J}{\partial \hat{y}_n^H} \cdot \frac{\partial \hat{y}_n^H}{\partial s_n^H} = \frac{(y_n - \hat{y}_n) f'(s_n^H)}{\frac{\partial \left(\frac{1}{2} \sum_{k=1}^{N^H} (\hat{y}_k - y_k)^2 \right)}{\partial \hat{y}_n}}$$
- and thus for update of weights:

$$w_{neu} = w_{alt} + \Delta w \quad \text{with} \quad \Delta w = -\alpha \frac{\partial J}{\partial w}$$

$$\Delta w'_{nm} = -\alpha \delta_n^H y_m^{H-1} = -\alpha (y_n - \hat{y}_n) f'(s_n^H) y_m^{H-1}$$

- For all other hidden layers $r < H$ the thoughts behind are a little more complex. Due to dependencies amongst each other, the values of s_j^{r-1} have an effect on all elements s_k^r of the following layer. Using the chain rule results in:

$$\underbrace{\frac{\partial J}{\partial s_j^{r-1}}}_{\delta_j^{r-1}} = \sum_k \underbrace{\frac{\partial J}{\partial s_k^r}}_{\delta_k^r} \frac{\partial s_k^r}{\partial s_j^{r-1}}$$

- With

$$\frac{\partial s_k^r}{\partial s_j^{r-1}} = \frac{\partial \left(\sum_m w_{km}'^r \overbrace{y_m^{r-1}}^{f(s_m^{r-1})} \right)}{\partial s_j^{r-1}} = w_{kj}'^r f'(s_j^{r-1})$$

- results:

$$\delta_j^{r-1} = f'(s_j^{r-1}) \left[\sum_k w_{kj}'^r \delta_k^r \right]$$

- i.e. the errors “backpropagate” from the starting layer to lower layers!

- All learning samples are crossed without any changes of the weight coefficients, and the gradient $\nabla \mathbf{J}$ results by forming the mean value of the $\nabla \mathbf{J}_j$. Both values can be used for update of the parameter matrix \mathbf{W}' of the perceptron:

$$\mathbf{W}'_{k+1} = \mathbf{W}'_k - \alpha \nabla \mathbf{J}_j \quad \text{individual learning}$$

$$\mathbf{W}'_{k+1} = \mathbf{W}'_k - \alpha \nabla \mathbf{J} \quad \text{batch learning}$$

- The gradients $\nabla \mathbf{J}$ and $\nabla \mathbf{J}_j$ differ. The latter is the mean value of the first ($\nabla \mathbf{J} = E\{\nabla \mathbf{J}_j\}$), or: the first value is random value of the latter.

The whole individual learning procedure consists of these steps:

- Choose interim values for the coefficient matrices $\mathbf{W}^1, \mathbf{W}^2, \dots, \mathbf{W}^H$ of all layers
- Given a new observation $[\mathbf{x}'_j, \hat{\mathbf{y}}_j]$
- Calculate the discrimination function $\hat{\mathbf{y}}_j$ from the given observation \mathbf{x}'_j and the current weight matrices (forward calculation)
- Calculate the error between estimation $\hat{\mathbf{y}}$ and target vector \mathbf{y} :

$$\Delta \mathbf{y} = \hat{\mathbf{y}} - \mathbf{y}$$

- Calculate the gradient $\nabla \mathbf{J}$ regarding to all perceptron weights (error backpropagation). To do so calculate first δ_j^H of the starting layer according to:

$$\delta_j^H = -\frac{\partial J}{\partial s_n^H} = -\frac{\partial J}{\partial \hat{y}_n^H} \cdot \frac{\partial \hat{y}_n^H}{\partial s_n^H} = (y_j - \hat{y}_j) f'(s_j^H)$$

- and calculate from this backwards all values of the lower layers according to:

$$\delta_j^{r-1} = f'(s_j^{r-1}) \left[\sum_k w_{kj}^{r'} \delta_k^r \right] \quad \text{für } r = H, H-1, \dots, 2$$

- considering the first derivate of the sigmoid function $f(s) = \psi(s)$:

$$f'(s) = \frac{d\psi(s)}{ds} = \psi(s)(1 - \psi(s)) = y(1 - y)$$

- (Parallely) correct all perceptron weights according to:

$$w'_{nm} \leftarrow w'_{nm} - \alpha \frac{\partial J}{\partial w'_{nm}} = w'_{nm} - \alpha \delta_n^r y_m^{r-1} \quad \begin{array}{l} r = 1, 2, \dots, H \\ \text{für } n = 1, 2, \dots, N^H \\ m = 1, 2, \dots, M^H \end{array}$$

- For individual learning the weights $\{w'_{nm}{}^h\}$ are corrected with $\nabla \mathbf{J}$ for every sample, while for batch learning all learning samples have to be considered, in order to determine the averaged gradient $\nabla \mathbf{J}$ from the sequence $\{\nabla \mathbf{J}\}$, before the weights can be corrected according to:

$$w'_{nm} \leftarrow w'_{nm} - \sum_i \alpha \delta_n^r(i) y_m^{r-1}(i) \quad \begin{array}{l} r = 1, 2, \dots, H \\ \text{für } n = 1, 2, \dots, N^H \\ m = 1, 2, \dots, M^H \end{array}$$

Backpropagation algorithm for matrices

- Choose interim values for the coefficient matrices $\mathbf{W}^1, \mathbf{W}^2, \dots, \mathbf{W}^H$ of all layers
- Given a new observation $[\mathbf{x}'_j, \hat{\mathbf{y}}_j]$
- Calculate the discrimination function $\hat{\mathbf{y}}_j$ from the given observation \mathbf{x}'_j and the current weight matrices (forward calculation). Store all values of \mathbf{y}^r and \mathbf{s}^r in all layers inbetween $r = 1, 2, \dots, H$.
- Calculate the error between estimation $\hat{\mathbf{y}}$ and target vector \mathbf{y} on the output:
$$\Delta \mathbf{y} = \hat{\mathbf{y}} - \mathbf{y}$$
- Calculate the gradient $\nabla \mathbf{J}$ regarding all perceptron weights (error backpropagation). To do so first calculate δ^H of the starting layer according to:

$$\delta^H = -\frac{\partial J}{\partial \mathbf{s}^H} = -\frac{\partial J}{\partial \hat{\mathbf{y}}^H} \cdot \frac{\partial \hat{\mathbf{y}}^H}{\partial \mathbf{s}^H} = (\mathbf{y} - \hat{\mathbf{y}}) \circ f'(\mathbf{s}^H)$$

- and calculate from this backwards all values of the lower layers according to:

$$\delta^{r-1} = f'(\mathbf{s}^{r-1}) \circ (\mathbf{W}'^{1^T} \delta^r) \quad \text{für } r = H, H-1, \dots, 2$$

- considering the first derivate of the sigmoid function $f(s) = \psi(s)$:

$$f'(s) = \frac{d\psi(s)}{ds} = \psi(s)(1 - \psi(s)) = y(1 - y)$$

- **Individual learning:** (parallely) correct all perceptron weight matrices with $\nabla \mathbf{J}$ for every sample according to:

$$\mathbf{W}'^r \leftarrow \mathbf{W}''^r - \alpha \frac{\partial J}{\partial \mathbf{W}''^r} = \mathbf{W}''^r - \alpha \boldsymbol{\delta}^r \mathbf{y}^{(r-1)T} \quad \text{für } r = 1, 2, \dots, H$$

- **Cumulative learning:** all samples have to be considered, in order to determine the averaged value $\nabla \mathbf{J}$ from the sequence of the $\{\nabla \mathbf{J}\}$, before the weights can be corrected according to:

$$\mathbf{W}'^r \leftarrow \mathbf{W}''^r - \alpha \sum_j \boldsymbol{\delta}_j^r \mathbf{y}_j^{(r-1)T} \quad \text{für } r = 1, 2, \dots, H$$

Properties:

- The backpropagation algorithm is simple to implement, but very complex especially for large coefficient matrices, which causes the number of samples to be large as well. The dependency of the method on the starting values of the weights, the correction factor α , and the order of processing the samples have also an adverse effect.
- Linear dependencies remain unconsidered, just like for recursive training of the polynomial classifier.
- The gradient algorithm has the advantage that it can be used for very large problems.
- The dimension of the coefficient matrix \mathbf{W} results from the dimensions of the input vector, the masked layers, and the output vector $(N, N^1, \dots, N^{H-1}, K)$ as:

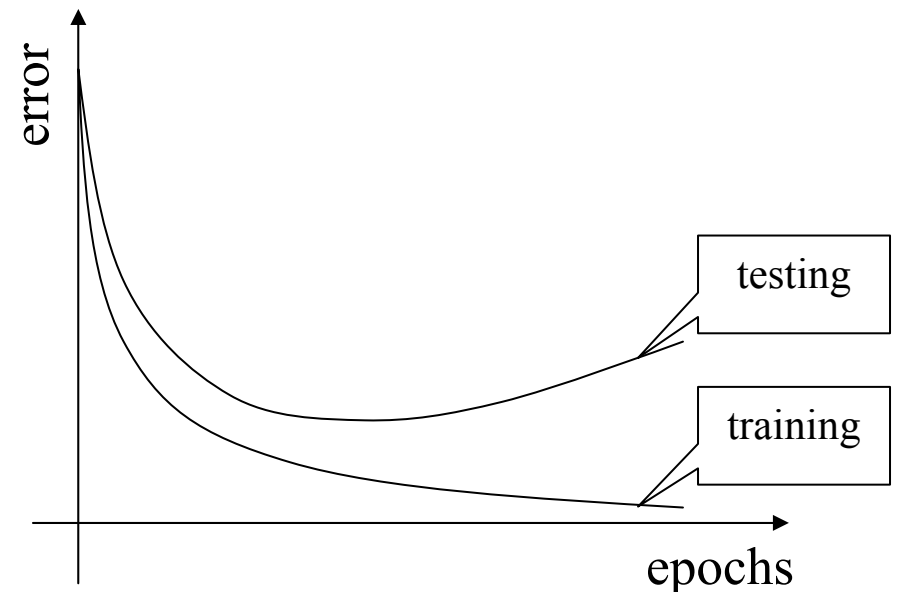
$$T = \dim(\mathbf{W}) = \sum_{h=1}^H (N^{h-1} + 1)N^h \quad \text{with } N^0 = N \quad \text{and} \quad N^H = K$$

$$N = \dim(\mathbf{x}) \quad \text{feature space}$$

$$K = \dim(\hat{\mathbf{y}}) \quad \text{number of classes}$$

Dimensioning the net

- The considerations for designing the multi-layer perceptron with threshold function (σ resp. sign) gives quite a good idea how many hidden-layers and how many neurons should be used for a MLPC for backpropagation learning (assumed one has a certain idea of distribution of clusters).
- The net is to be chosen as simple as possible. The higher the dimension the higher the risk of overfitting, accompanied by a loss of ability to generalize. Also higher dimension allows a lot of local maximas, which can cause the algorithm to get stuck!
- For a given number of samples, the learning phase should be stopped at a certain point. If not stopped the net is overfitted to the existing data and loses its ability to generalize.



Complexity of backpropagation algorithm

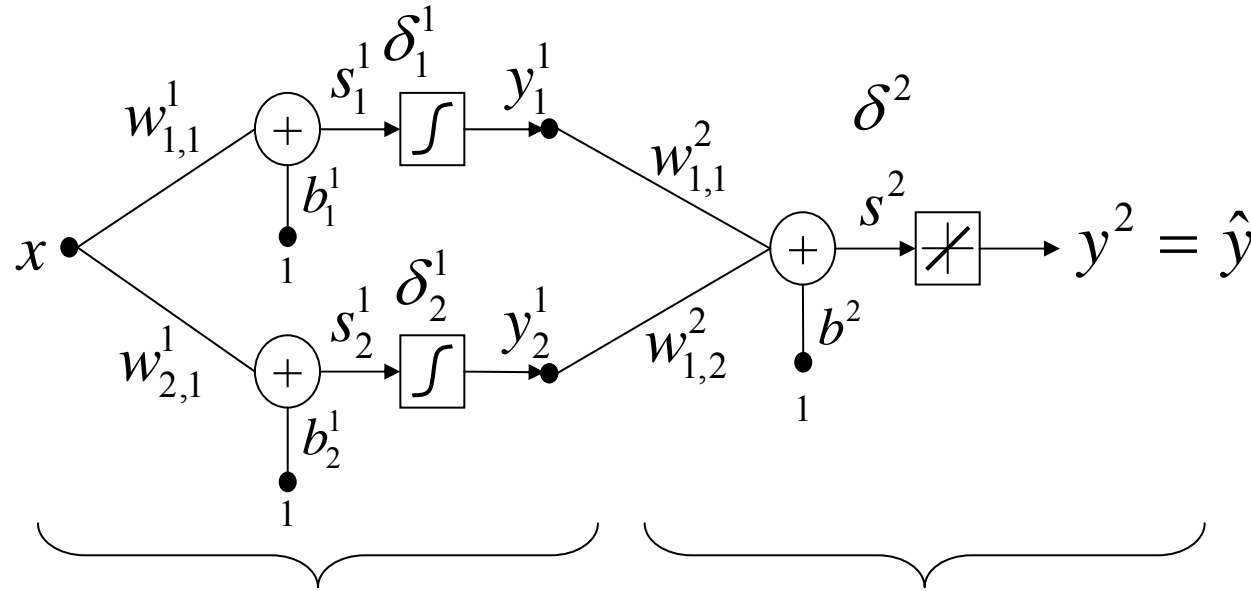
- For $T = \dim(\mathbf{W})$ number of weights and bias terms, linear complexity in the number of weights can be easily understood, since $O(T)$ computation steps are needed for forward simulation, $O(T)$ steps are needed for backpropagation of the error and also $O(T)$ operations for correction of weights, altogether: $O(T)$.
- If gradients would be determined experimentally by finite differences (on order to do so, one would have to increment each weight and determine the effect on the quality criterion by forward calculation) by calculating a differential quotient according to (i.e. no analytical evaluation):

$$\frac{\partial J}{\partial w_{ji}^r} = \frac{J(w_{ji}^r + \varepsilon) - J(w_{ji}^r)}{\varepsilon} + O(\varepsilon)$$

- T forward calculations of complexity $O(T)$ would result and therefore a overall (?) complexity of $O(T^2)$

Backpropagation for training a two-layer network for function approximation

(Matlab demo: „Backpropagation Calculation)



$$\mathbf{y}^1 = \boldsymbol{\psi}(\mathbf{W}^1 x + \mathbf{b}^1) = \boldsymbol{\psi}(\mathbf{W}^1 \mathbf{x}^1) = \boldsymbol{\psi}(\mathbf{s}^1)$$

$$\text{mit: } \mathbf{W}^1 = \begin{bmatrix} w_{1,1}^1 \\ w_{2,1}^1 \end{bmatrix} \text{ und } \mathbf{b}^1 = \begin{bmatrix} b_1^1 \\ b_2^1 \end{bmatrix}$$

$$\hat{y} = y^2 = (\mathbf{W}^2 \mathbf{y}^1 + b^2)$$

$$\text{mit: } \mathbf{W}^2 = \begin{bmatrix} w_{1,1}^2 & w_{1,2}^2 \end{bmatrix}$$

- Sought is an approximation of the function

$$y(x) = 1 + \sin\left(\frac{\pi}{4} x\right) \quad \text{for} \quad -2 \leq x \leq 2$$

Backpropagation for training a two-layer network for function approximation

- Backpropagation: For output layer or second layer with linear activation function results due to $f'=1$:

$$\delta^2 = (y - \hat{y})$$

- Backpropagation: For first layer with sigmoid function results:

$$\delta_j^1 = f'(s_j^1) [w'_{1,j} \delta^2] = y_j^1 (1 - y_j^1) [w_{1,j}^2 \delta^2] \quad \text{for } j = 1, 2$$

- Correction of weights in the output layer:

$$\Delta w_{1,j}^2 = \alpha \delta^2 y_j^1 \quad \text{and} \quad \Delta b^2 = \alpha \delta^2 \quad \text{for } j = 1, 2$$

- Correction of weights in the input layer:

$$\Delta w_{j,1}^1 = \alpha \delta_j^1 x \quad \text{and} \quad \Delta b_j^1 = \alpha \delta_j^1 \quad \text{for } j = 1, 2$$

nnd1bc

File Edit View Insert Tools Window Help

Neural Network DESIGN Backpropagation Calculation

Last Error: ???

Input: $p = 1.0$

Target: $t = 1 + \sin(p \cdot \pi / 4) = 1.707$

Simulate:
 $a1 = \text{logsig}(W1 \cdot p + b1) = [0.321; 0.368]$
 $a2 = \text{purelin}(W2 \cdot a1 + b2) = 0.446$
 $e = t - a2 = 1.261$

Backpropagate:
 $s2 = -2 \cdot \text{dpurelin}(n2) / \text{dn2} \cdot e = -2.522$
 $s1 = \text{dlogsig}(n1) / \text{dn1} \cdot W2 \cdot s2 = [-0.049; 0.100]$

Update:
 $W1 = W1 - \text{lr} \cdot s1 \cdot p' = [-0.265; -0.420]$
 $b1 = b1 - \text{lr} \cdot s1 = [-0.475; -0.140]$
 $W2 = W2 - \text{lr} \cdot s2 \cdot a1' = [0.171; -0.077]$
 $b2 = b2 - \text{lr} \cdot s2 = 0.732$

Continue
 Reset
 Random
 Contents
 Close

Chapter 11

Starting Matlab demo
[matlab-BPC.bat](#)

Backpropagation for training a two-layer network for function approximation

(Matlab demo: „Function Approximation“)

- Sought is an approximation of the function


$$y(x) = 1 + \sin\left(i \cdot \frac{\pi}{4} x\right) \quad \text{for} \quad -2 \leq x \leq 2$$

- With increasing value of i (difficulty index) the MLPC network requirements increase. The approximation based on the sigmoid function needs more and more layers in order to depict several periods of the sinus function.
- Problem: Convergence to local minima, even when the network is big enough to represent the function
 - The network is too small, so that approximation is poor, i.e. the global minimum can be reached, but approximation quality is poor ($i=8$ and network 1-3-1)
 - The network is big enough, so that approximation is good, but it converges towards a local minimum ($i=4$ and network 1-3-1)

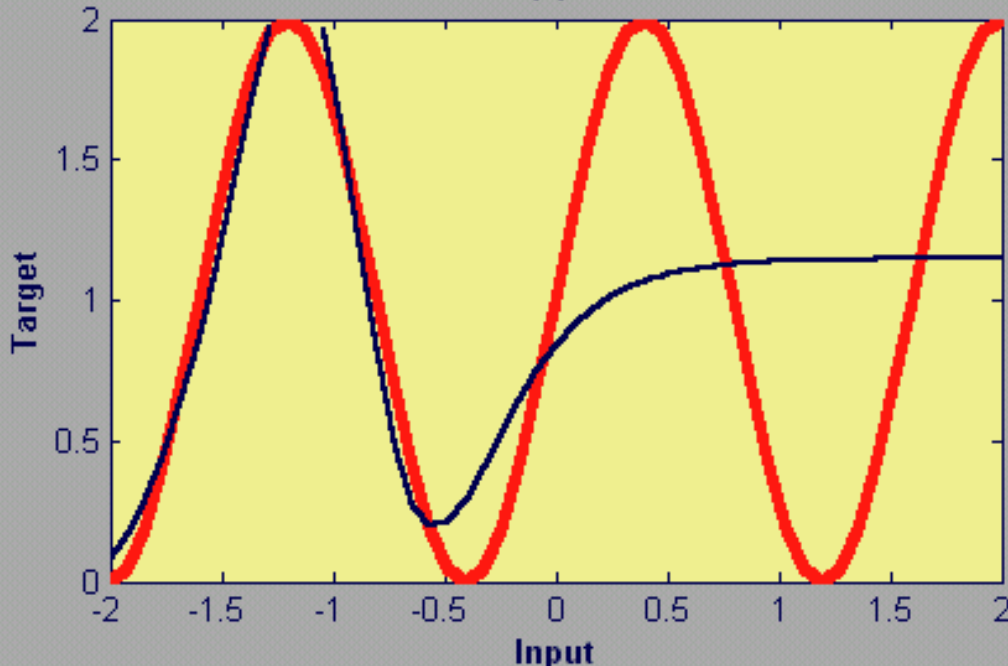
nnd11fa

File Edit View Insert Tools Window Help

Neural Network DESIGN Function Approximation



Function Approximation



Click the [Train] button to train the logsig-linear network on the function at left.

Use the slide bars to choose the number of neurons in the hidden layer and the difficulty of the function.

Train

Contents

Close

Chapter 11

Number of Hidden Neurons S1: 2

1 9


Difficulty Index: 5

1 9

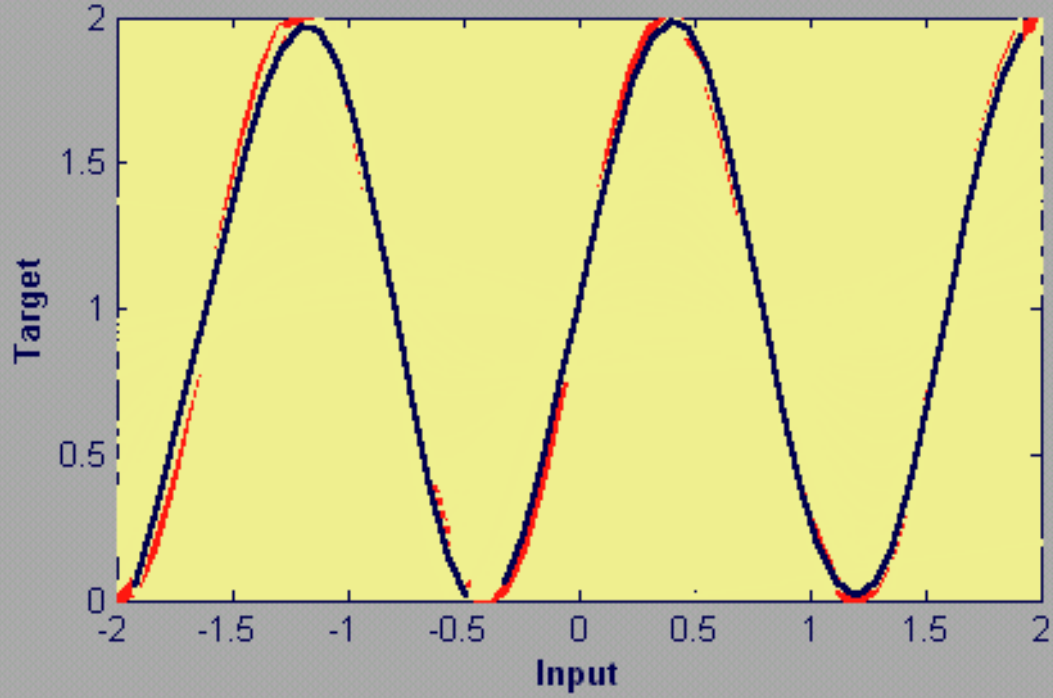
nnd11fa

File Edit View Insert Tools Window Help

Neural Network DESIGN Function Approximation



Function Approximation



Click the [Train] button to train the logsig-linear network on the function at left.

Use the slide bars to choose the number of neurons in the hidden layer and the difficulty of the function.

Train

Contents

Close

Chapter 11

Number of Hidden Neurons S1: 4

1 9

Difficulty Index: 5

1 9

Starting Matlab demo
[matlab-FA.bat](#)

Model big enough but only local minimum

The screenshot shows a software window titled "nnd11fa" with a menu bar (File, Edit, View, Insert, Tools, Window, Help). The main area is titled "Neural Network DESIGN" and "Function Approximation". A central plot, titled "Function Approximation", shows a target function (black solid line) and a network approximation (red dashed line) on a yellow background. The x-axis is labeled "Input" and ranges from -2 to 2. The y-axis is labeled "Target" and ranges from 0 to 2. Below the plot are two sliders: "Number of Hidden Neurons S1:" with a value of 3, and "Difficulty Index:" with a value of 4. To the right of the plot is a text box with instructions: "Click the [Train] button to train the logsig-linear network on the function at left. Use the slide bars to choose the number of neurons in the hidden layer and the difficulty of the function." Below the text are three buttons: "Train", "Contents", and "Close". At the bottom right, it says "Chapter 11".

Generalization ability of the network

- Assuming that the network is trained for 11 samples

$$\{y_1, x_1\}, \{y_2, x_2\}, \dots, \{y_{11}, x_{11}\}$$

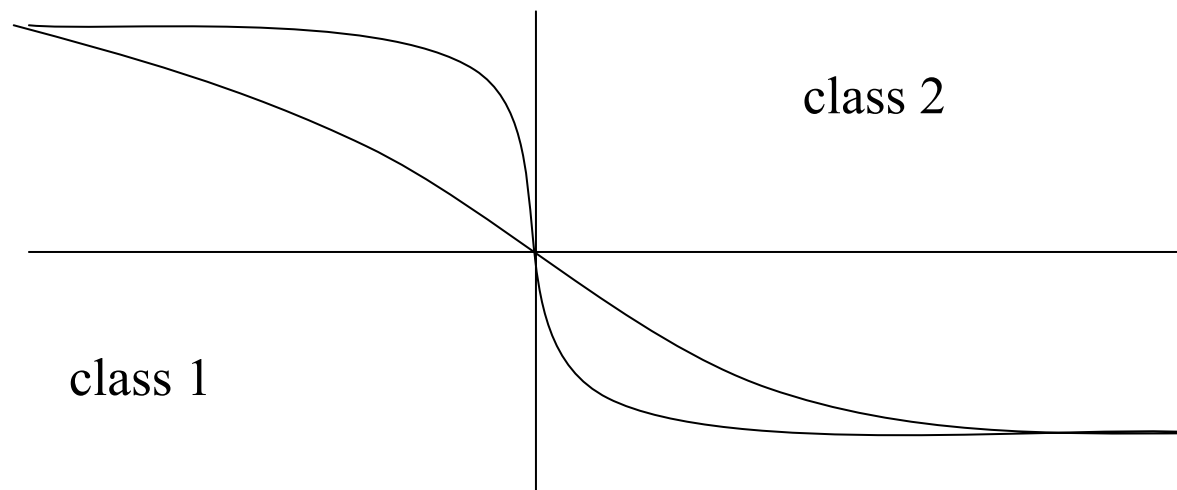
- How well does the network approximate the function for samples not learned depending on the complexity of the network?

Starting Matlab demo (Hagan 11-21)
[matlab-GENERAL.bat](#)

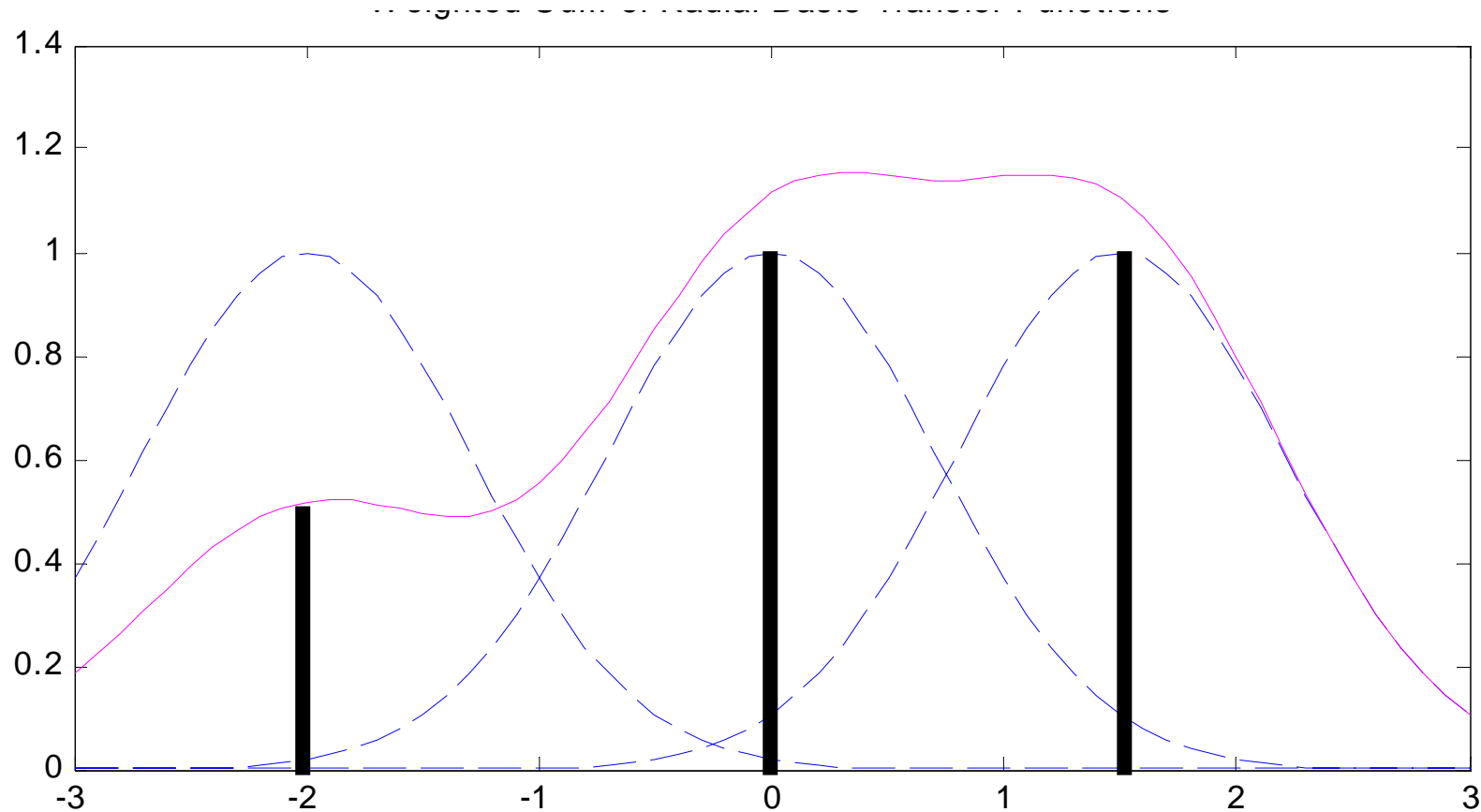
- Rule: The network should have fewer parameters than the available input/output pairs

Improving the ability to generalize of a net by adding noise

- If only few samples are available, the generalization ability of a nn can be improved by extending the number of samples e.g. by normally distributed noise. i.e. adding further samples, which can be found in the nearest neighbourhood with high probability.
- In doing so the intra class areas are broadened and the borders between classes are sharpened.

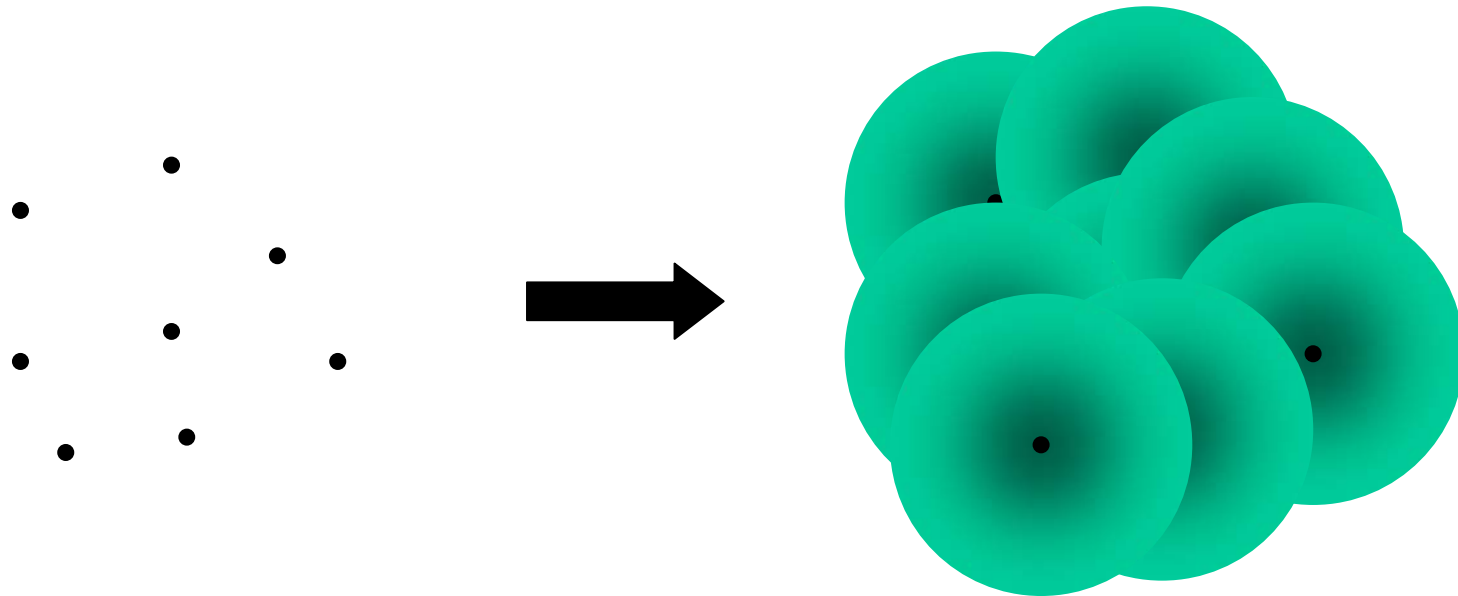


Interpolation by overlapping of normal distributions

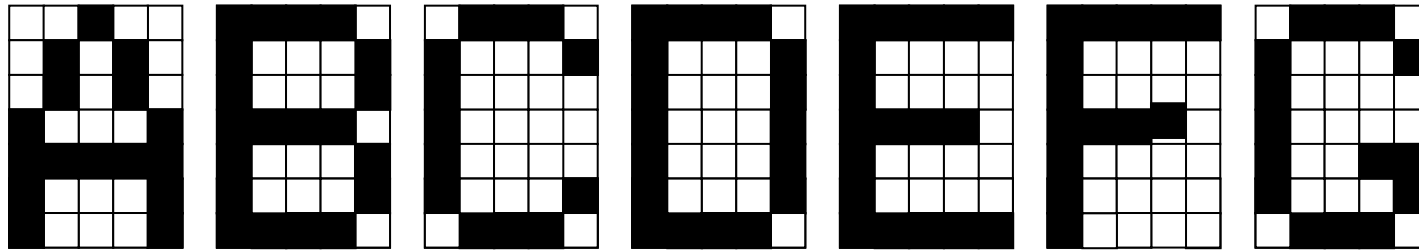


$$y(x) = \sum \alpha_i f_i(x - t_i)$$

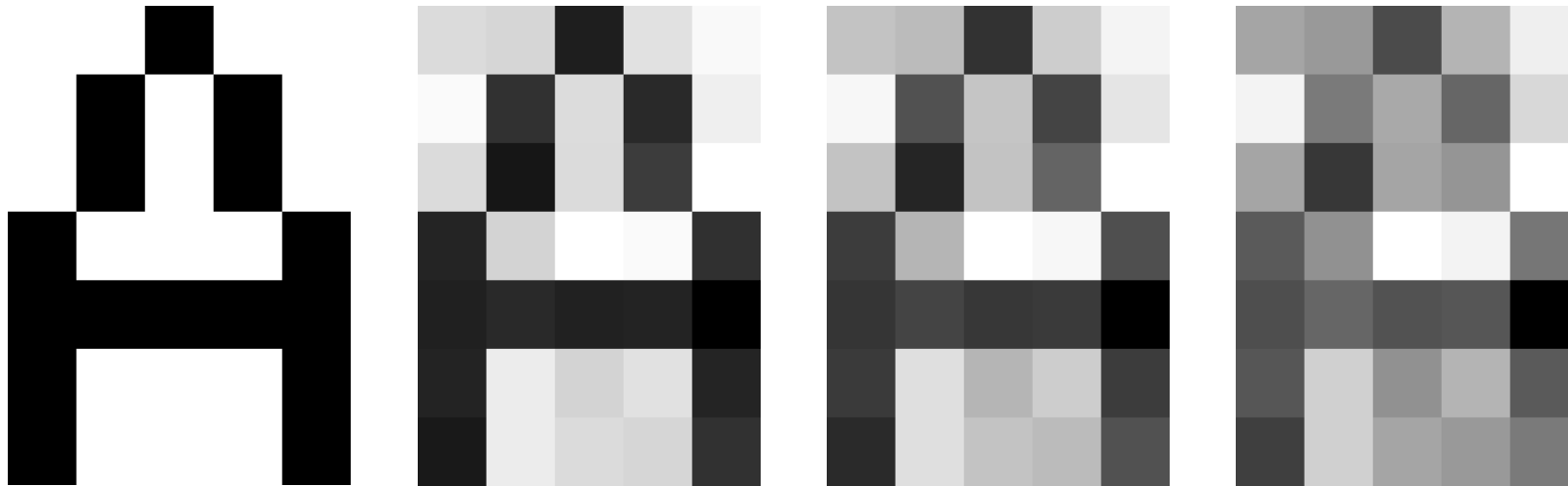
Interpolation by overlapping of normal distributions



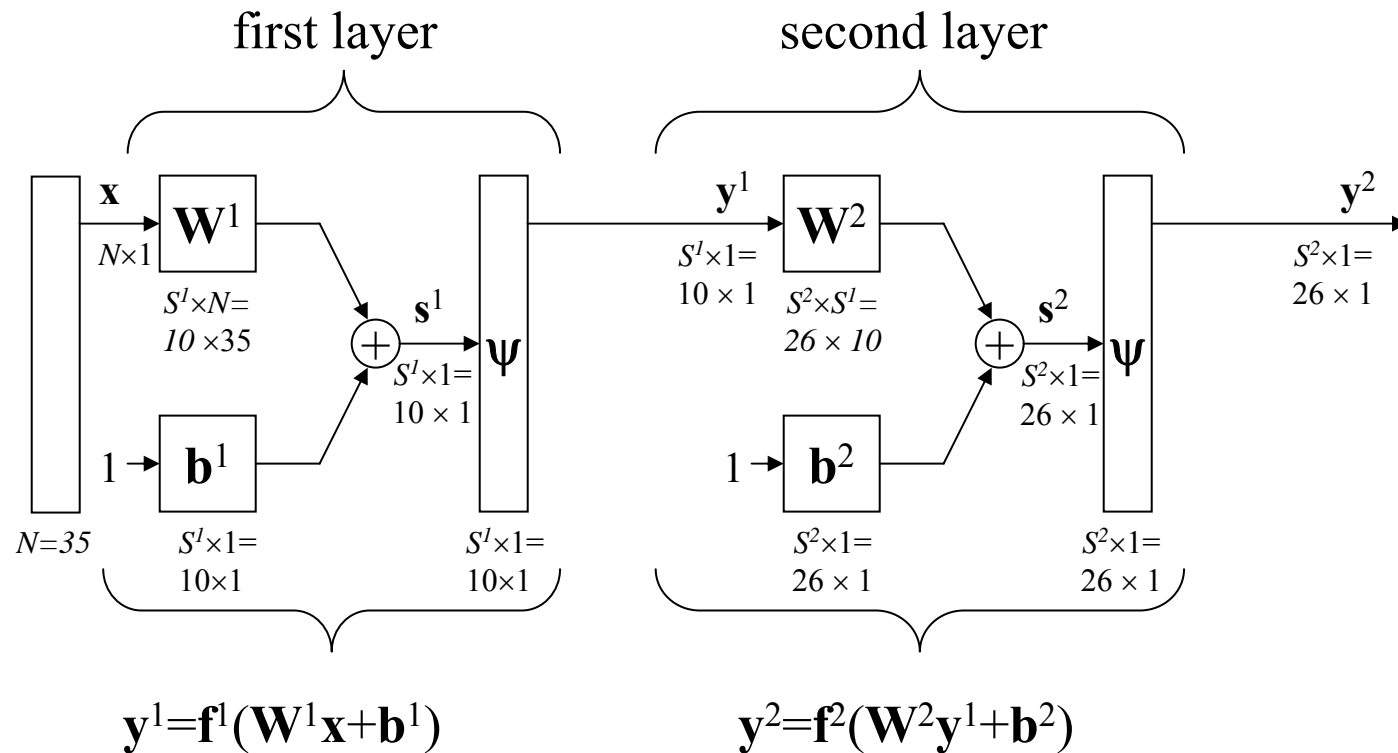
Character recognition task for 26 letters of size 7×5



- with gradually increasing additive noise:



Two-layer neural net for symbol recognition with 35 input values (pixel), 10 neurons in the hidden layer and 26 neurons in the output layer



$$\mathbf{y}^2 = \Psi(\mathbf{W}^2 \Psi(\mathbf{W}^1 \mathbf{x} + \mathbf{b}^1) + \mathbf{b}^2)$$

Demo with MATLAB

- Opening Matlab
 - Demo in Toolbox Neural Networks
 - „Character recognition“ (command line)
 - Two networks are trained
 - network 1 without noise
 - network 2 with noise
 - Network 2 has better results than network 1 considering an independent set of test data

Starting Matlab demo
[matlab-CR.bat](#)

Possibilities to accelerate the adaption

Adaption of NN is simply a general parameter optimization problem. Accordingly a variety of other optimization methods from numerical mathematics could be applied in principle. This can lead to significant improvements (convergence speed, convergence area, stability, complexity).

In general conclusions cannot be made easily, since the events can differ a lot and compromises have to be made!

- Heuristic improvements of the gradient algorithm (increment control)
 - gradient algorithm (steepest descent) with momentum term (update of weights depends not only on gradient but also on former update)
 - using an adaptive learning factor
- Conjugate gradient algorithm

- Newton and Semi-Newton-algorithms (using the second derivatives of the error function, e.g. in form of Hesse matrix or its estimation)
 - Quickprop
 - Levenberg-Marquardt-algorithm

Taylor expansion of quality function in \mathbf{w} :

$$J(\mathbf{w} + \Delta\mathbf{w}) = J(\mathbf{w}) + \nabla\mathbf{J}^T \Delta\mathbf{w} + \frac{1}{2} \Delta\mathbf{w}^T \mathbf{H} \Delta\mathbf{w} + \text{terms of higher order}$$

with: $\nabla\mathbf{J} = \frac{\partial J}{\partial \mathbf{w}}$ gradient vector

and: $\mathbf{H} = \left\{ \frac{\partial^2 J}{\partial w_i \partial w_j} \right\}$ Hesse matrix

- Pruning techniques. Starting with a sufficiently big network, neurons, that have no or little effect on the quality function, are taken off the net, which can reduce overfitting of data.

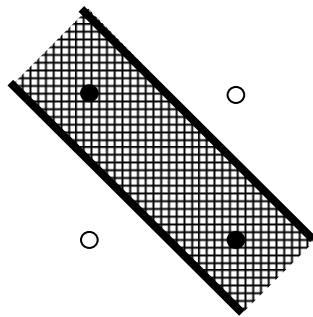
Demo with MATLAB

(Demo of Theodoridis)

- C:\...\matlab\PR\startdemo
- Example 2 (XOR) with (2-2-1 and 2-5-5-1)
- Example 3 with three-layer network [5,5]
(3000 epochs, learning rate 0,7, momentum 0,5)

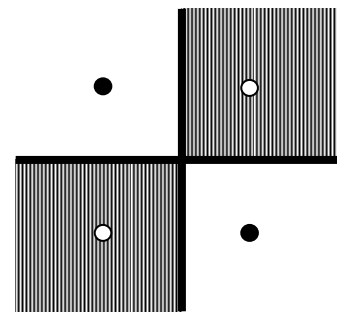
Start der Matlab-Demo
[matlab-theodoridis.bat](#)

Two solutions of the XOR-problem:



Convex area implemented with two-layer net (2-2-1). Possible mal-classification for variance:

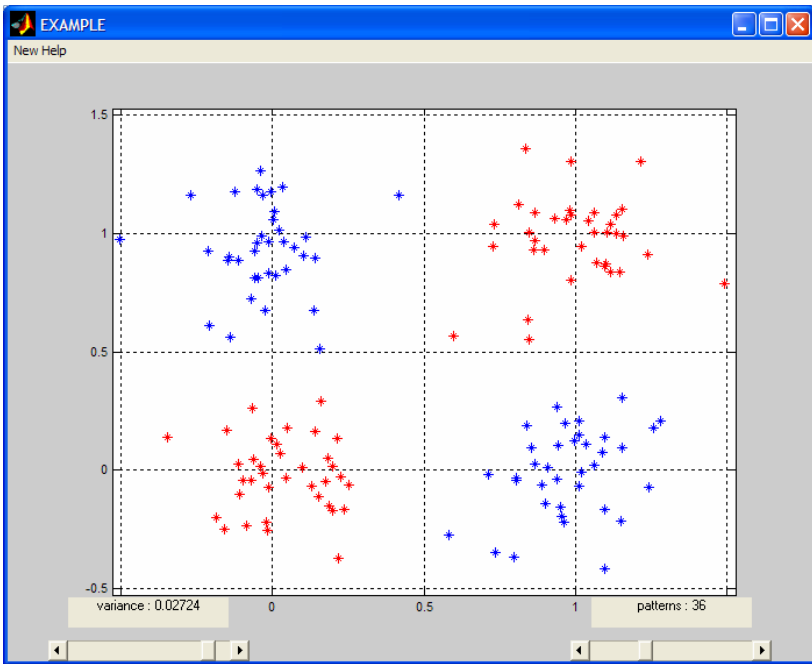
$$\|\mathbf{n}\| > \frac{1}{4}\sqrt{2} \approx 0,35$$



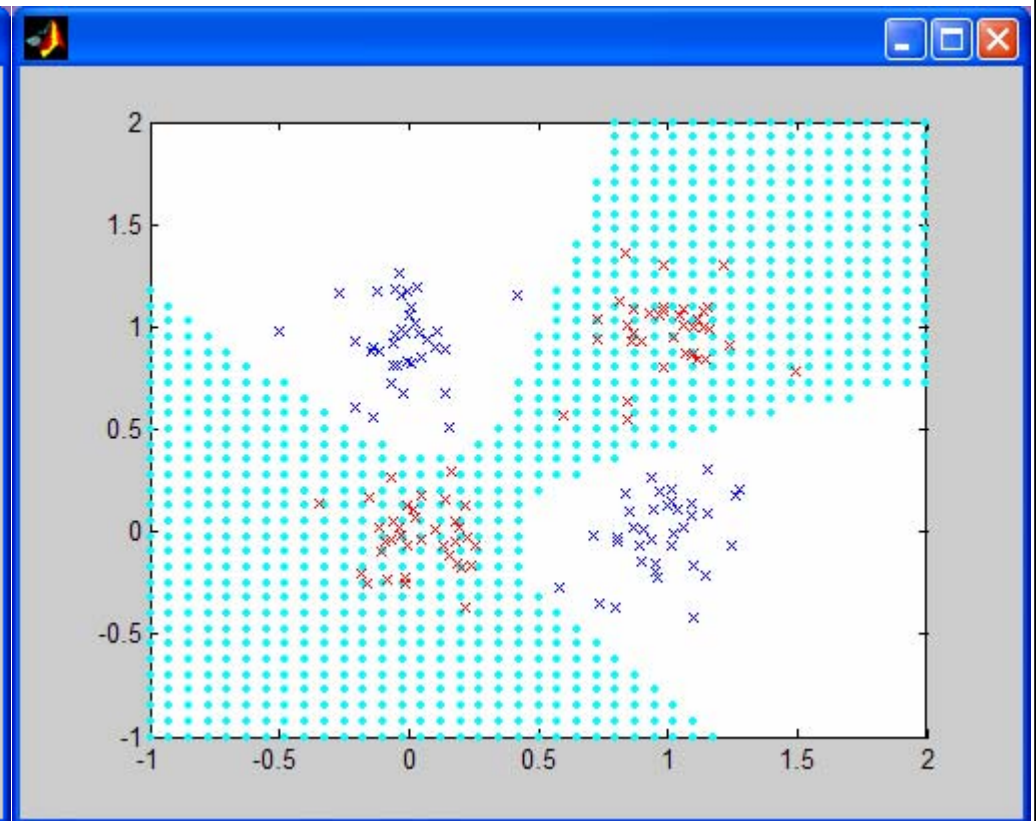
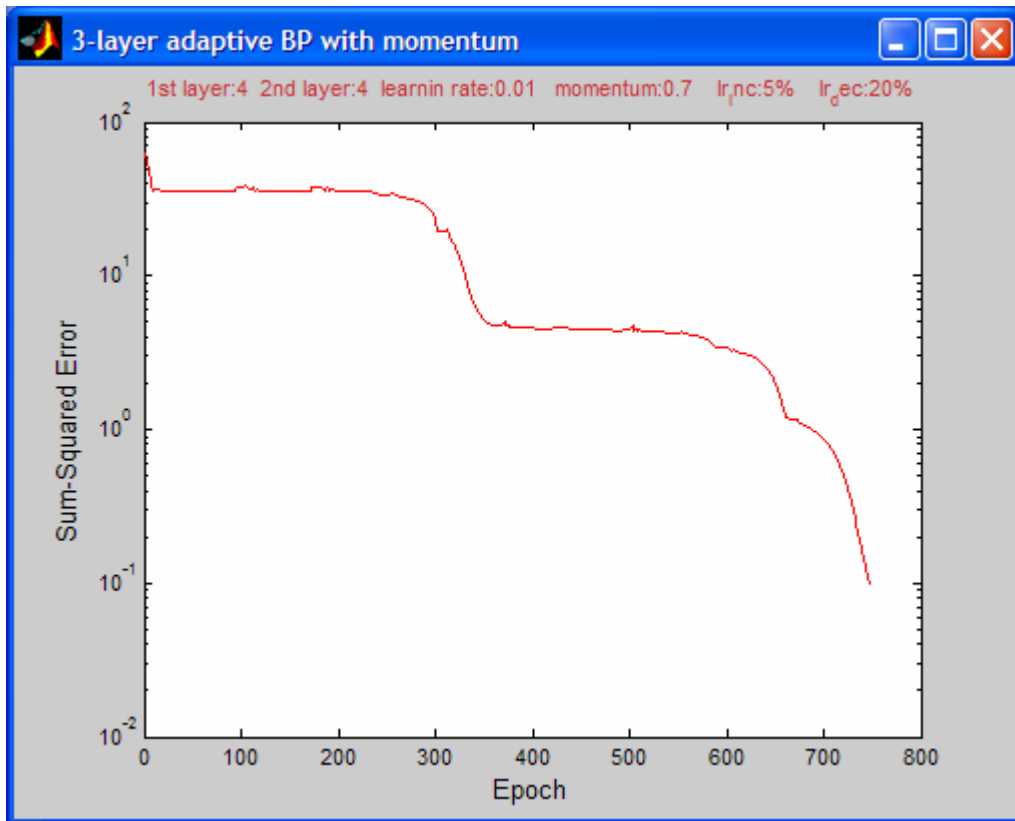
2-2-2-1 does not converge!

Union of 2 convex areas implemented with a three-layer net (2-5-5-1). Possible mal-classification for variance:

$$\|\mathbf{n}\| > 0,5$$

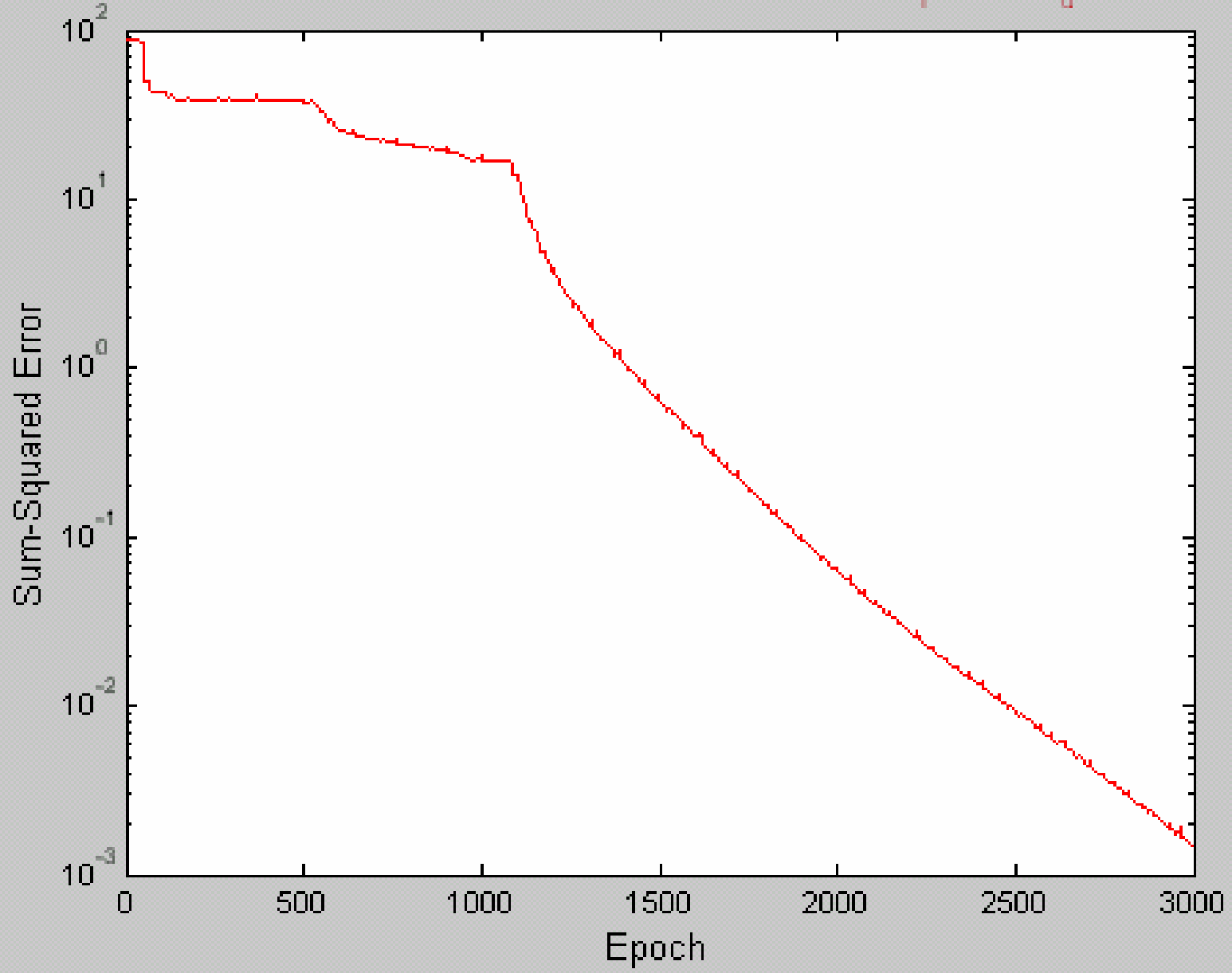


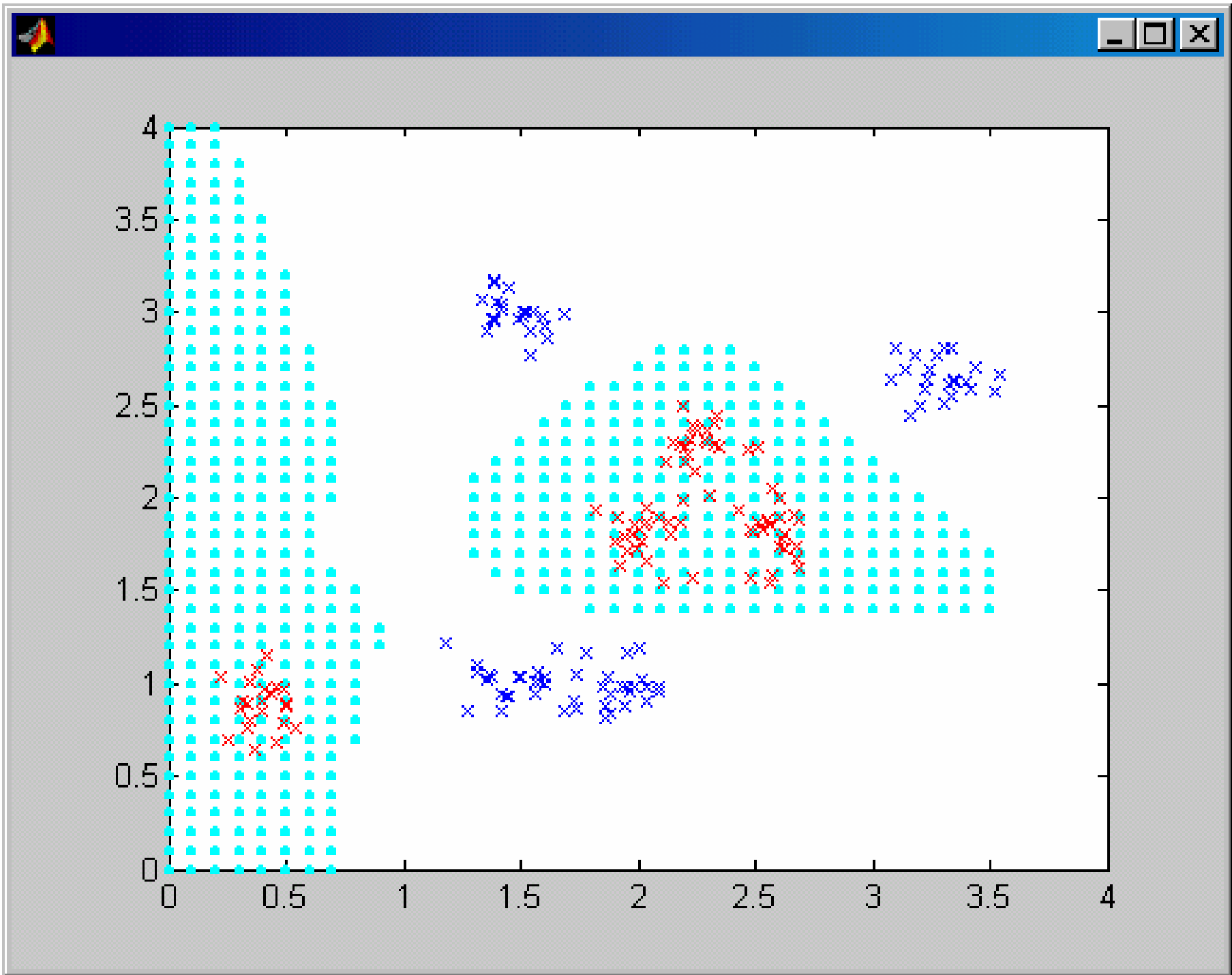
XOR with high noise in order to activate second solution! With two lines a error-free (exact) combination cannot be reached and the gradient is still different from 0!



3-layer adaptive BP with momentum

1st layer:5 2nd layer:5 learnin rate:0.7 momentum:0.5 lr_{nc}:5% lr_{dc}:20%





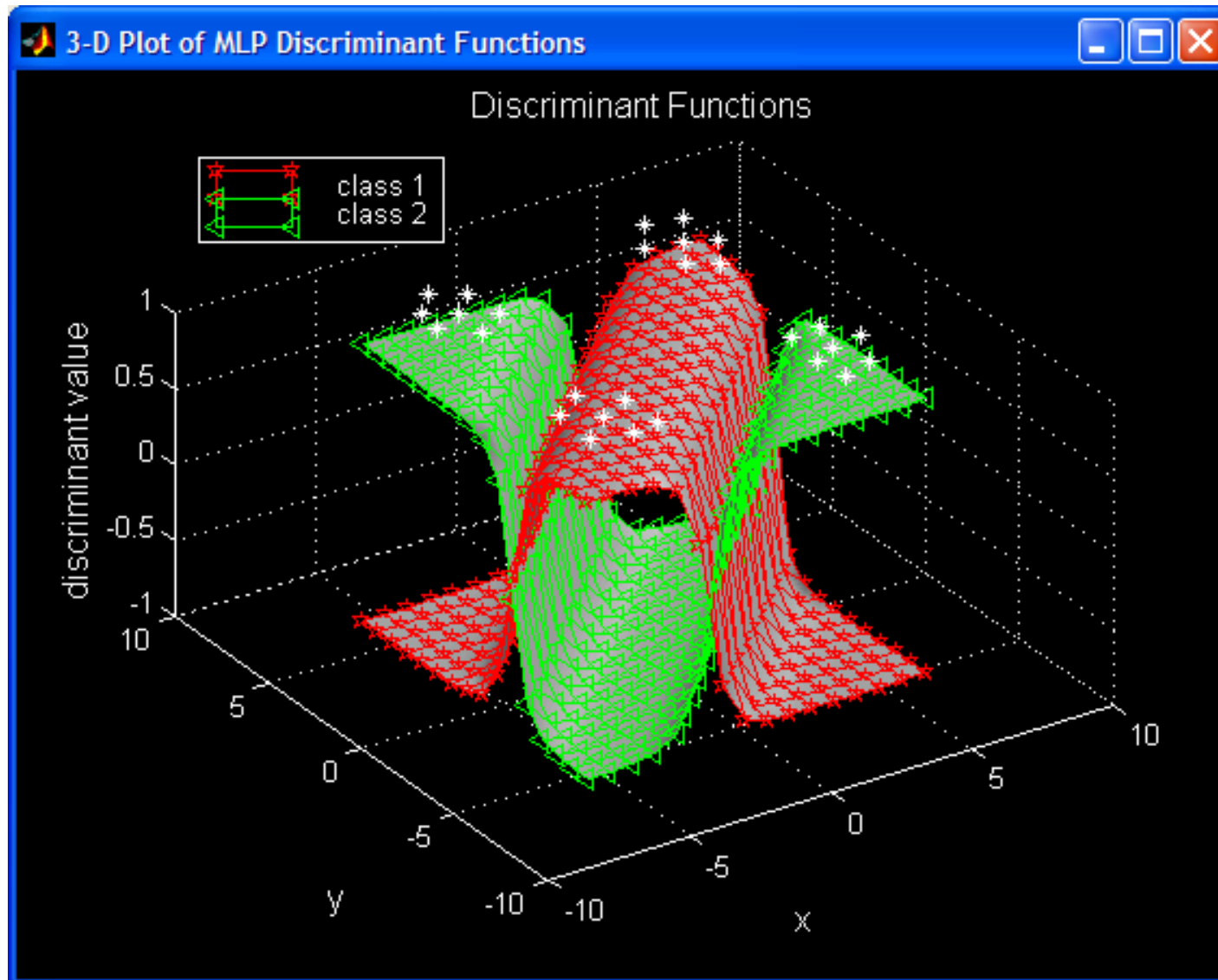
Demo with MATLAB

(Klassifikationgui.m)

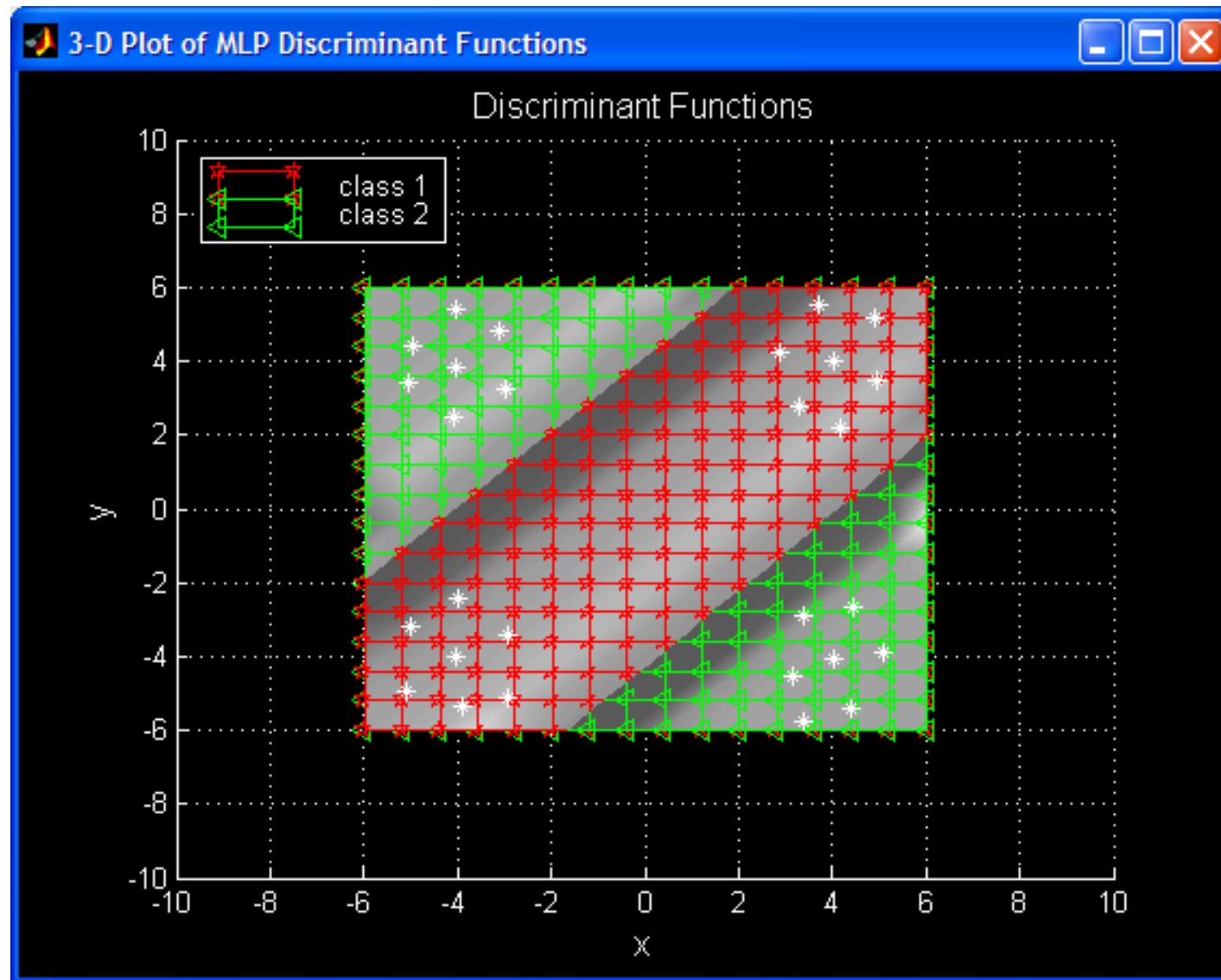
- Opening Matlab
 - first invoke setmypath.m, then
 - C:\Home\ppt\Lehre\ME_2002\matlab\KlassifikatorEntwurf-WinXX\Klassifikationgui
 - set of data: load Samples/xor2.mat
Setting: 500 epochs, learning rate 0.7
 - load weight matrices: models/xor-trivial.mat
 - load weight matrices : models/xor-3.mat
- Two-class problem with banana shape
 - load set of data Samples/banana_test_samples.mat
 - load presetting from: models/MLP_7_3_2_banana.mat

Starting Matlab-Demo
[matlab-Klassifikation_gui.bat](#)

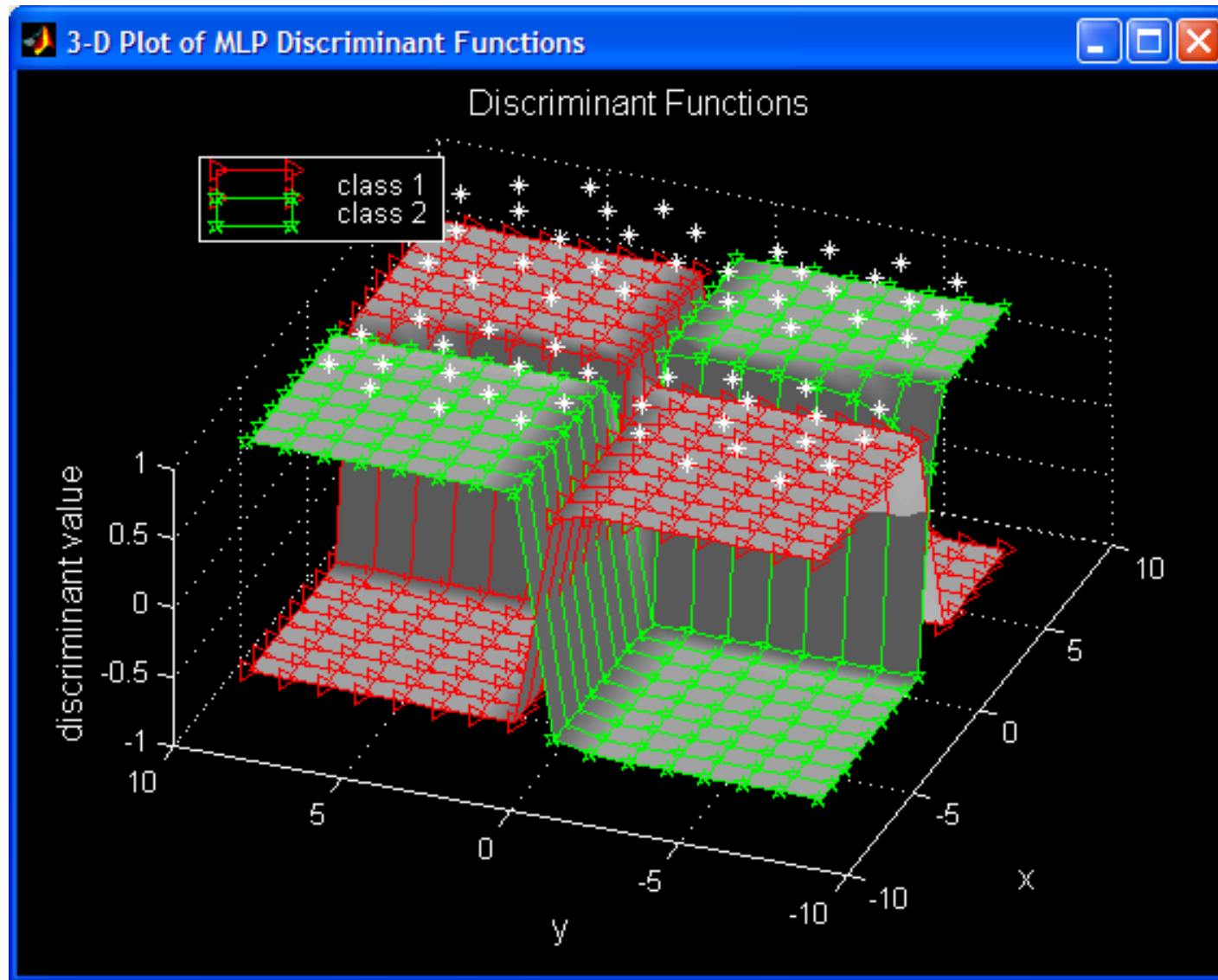
Solution of the XOR-problem with two-layer nn



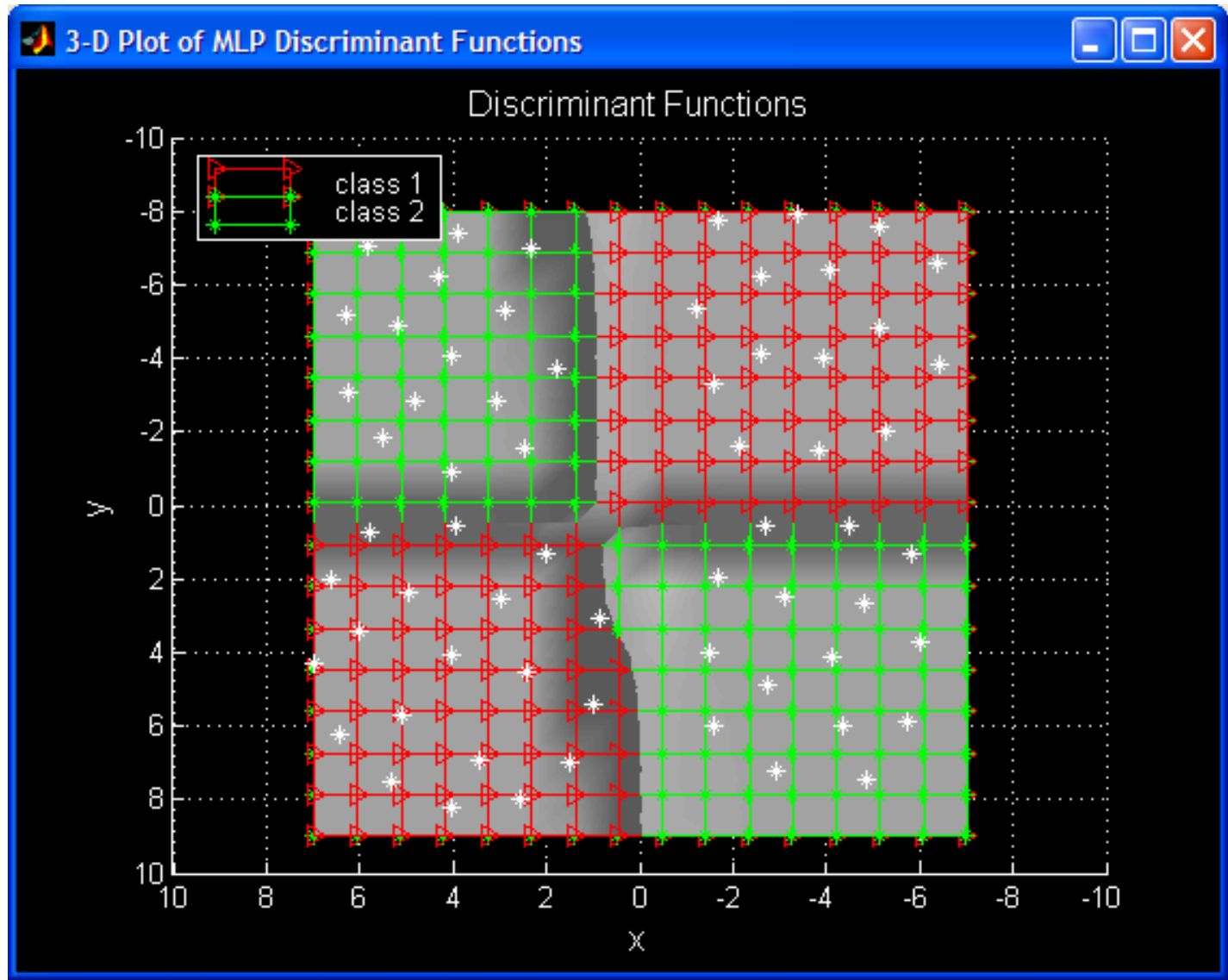
Solution of the XOR-problem with two-layer nn

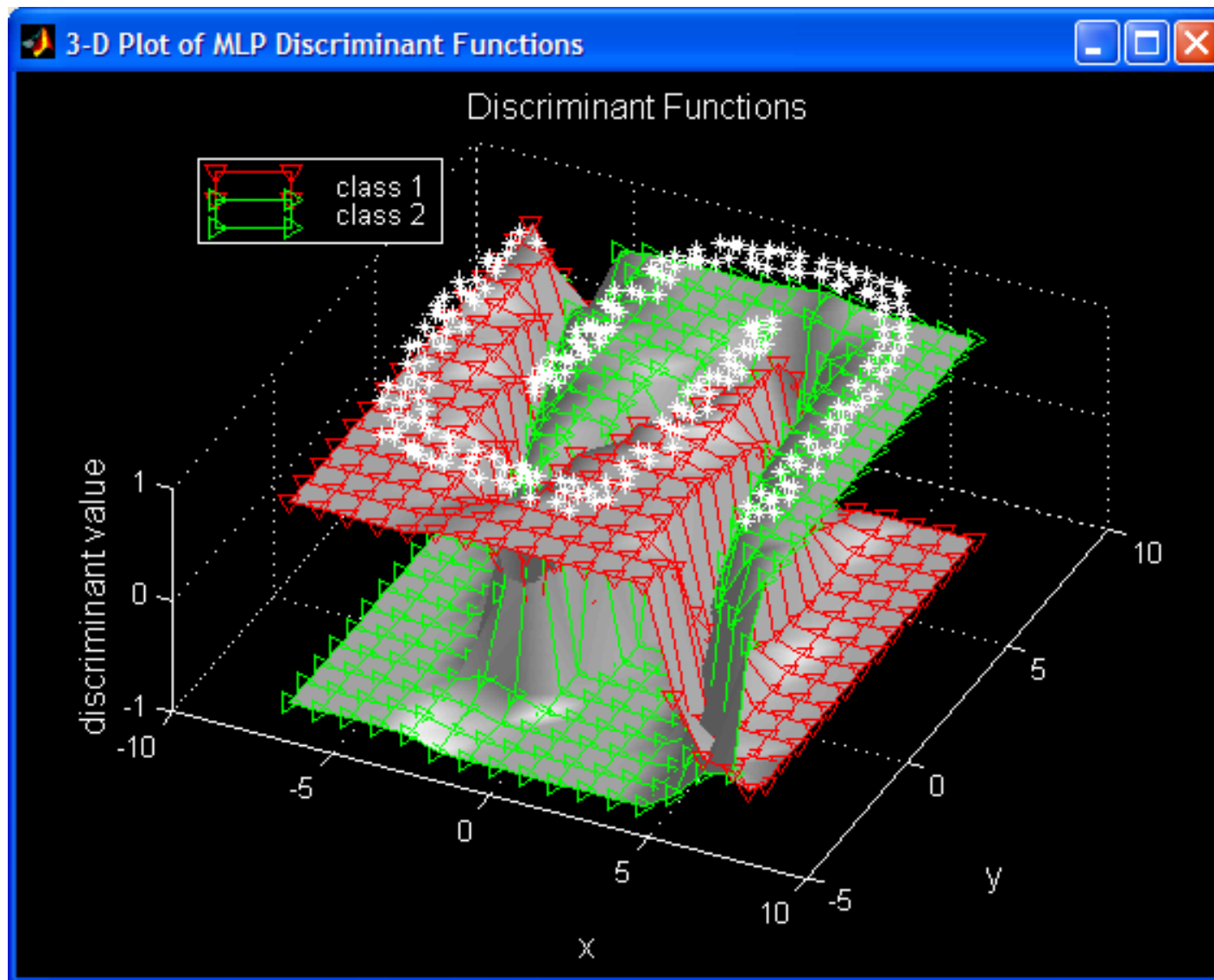


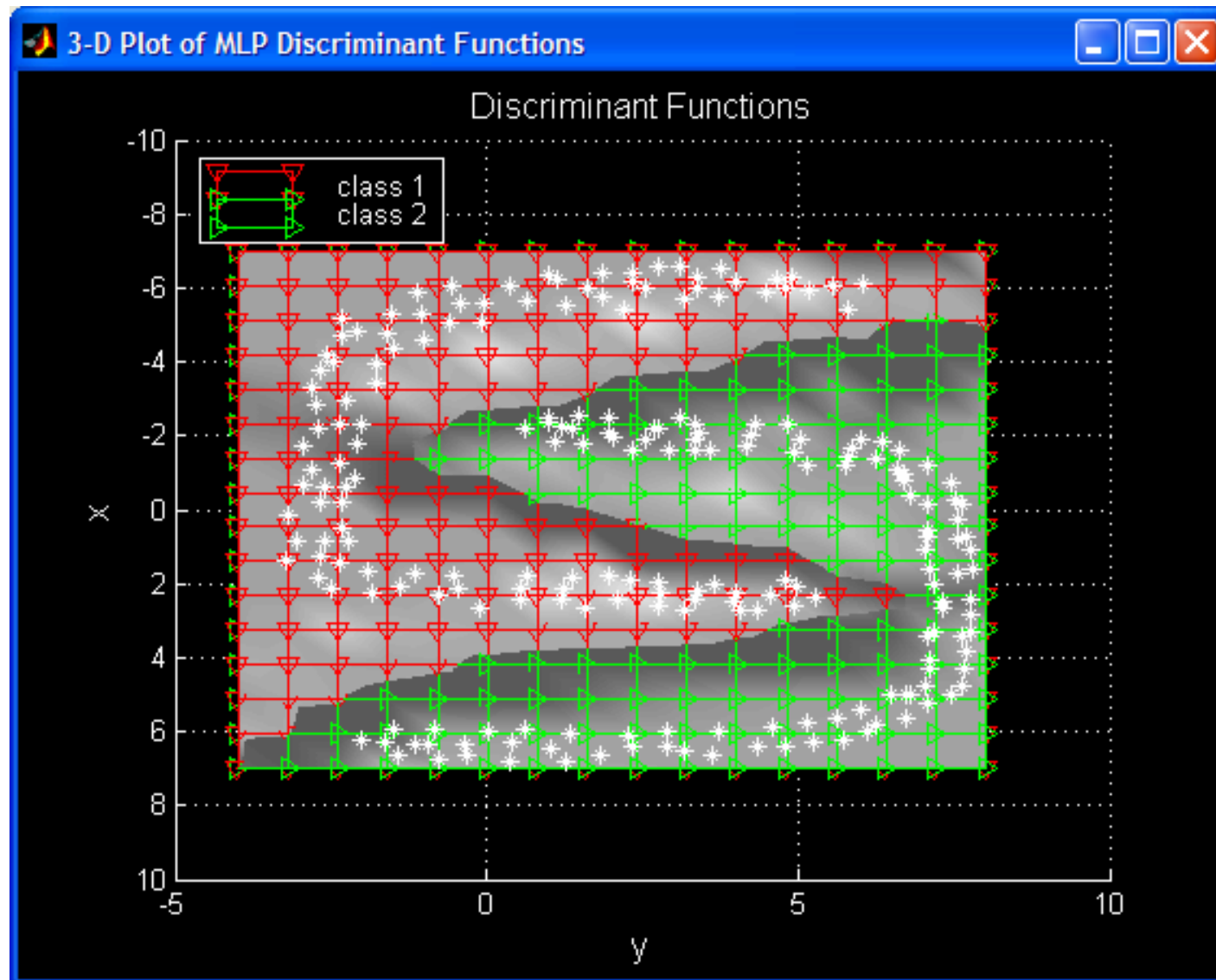
Solution of the XOR-problem with three-layer nn



Solution of the XOR-problem with three-layer nn







Convergence behaviour of backpropagation algorithm

- Backpropagation with common gradient descent (steepest descent)

Starting Matlab demo
[matlab-BP-gradient.bat](#)

- Backpropagation with conjugate gradient descent (conjugate gradient) – significantly better convergence!

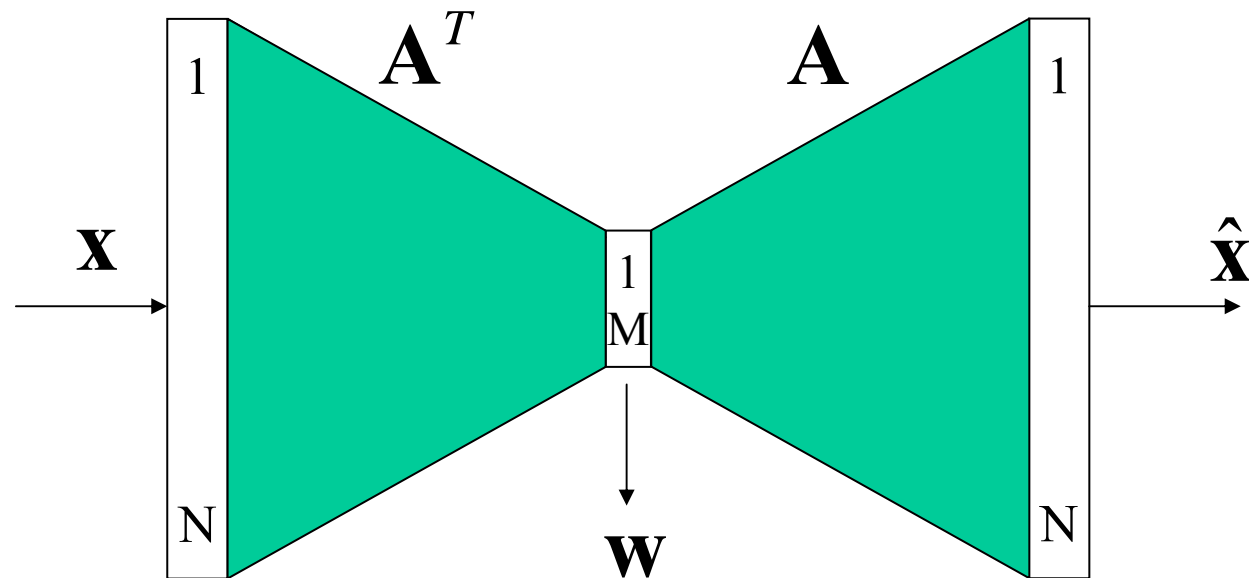
Starting Matlab demo
[matlab-BP-CGgradient.bat](#)

NN and their properties

- „Quick and Dirty“. A nn design is simple to implement and results in solutions are by all means usable (a suboptimal solution is better than just any solution).
- Particularly large problems can be solved.
- All strategies only use “local” optimization functions and generally do not reach a global optimum, which is a big draw back for using neural nets.

Using a NN for iterative calculation of Principal Component Analysis (KLT)

- Instead of solving the KLT explicitly by finding Eigen values, a nn can be used for iterative calculation.
- A two-layer perceptron is used. The output of the hidden layer \mathbf{w} is the feature vector sought.



- The first layer calculates the feature vector as:

$$\mathbf{w} = \mathbf{A}^T \mathbf{x}$$

- Supposing a linear *activation function* is used. The second layer implements the reconstruction of \mathbf{x} with the transposed weight matrix of the first layer

$$\hat{\mathbf{x}} = \mathbf{A}\mathbf{w}$$

- Target of optimization is to minimize:

$$J = E \left\{ \|\hat{\mathbf{x}} - \mathbf{x}\|^2 \right\} = E \left\{ \|\mathbf{A}\mathbf{w} - \mathbf{x}\|^2 \right\} = E \left\{ \|\mathbf{A}\mathbf{A}^T \mathbf{x} - \mathbf{x}\|^2 \right\}$$

- The following learning rule:

$$\mathbf{A} \leftarrow \mathbf{A} - \alpha(\mathbf{A}\mathbf{w} - \mathbf{x})\mathbf{w}^T = \mathbf{A} - \alpha\nabla\mathbf{J}$$

- leads to a coefficient matrix \mathbf{A} , which can be written as a product of two matrices (without proof!):

$$\mathbf{A} = \mathbf{B}_M \mathbf{T}$$

- \mathbf{B}_M is a $N \times M$ -matrix of the M dominant Eigenvectors of $E\{\mathbf{x}\mathbf{x}^T\}$ and \mathbf{T} a orthonormal $M \times M$ -matrix, which causes a rotation of the coordinate system, within the space that is spanned by the M dominant Eigenvectors of $E\{\mathbf{x}\mathbf{x}^T\}$.

- The learning strategy does not include translation. i.e. this strategy calculates the Eigenvectors of the moment matrix $E\{\mathbf{x}\mathbf{x}^T\}$ and **not** of the covariance matrix $\mathbf{K} = E\{(\mathbf{x}-\boldsymbol{\mu}_x)(\mathbf{x}-\boldsymbol{\mu}_x)^T\}$. This can be implemented by subtracting a recursively estimated expected value $\boldsymbol{\mu}_x = E\{\mathbf{x}\}$ before starting the recursive estimation of the feature vector. The same effect can be retrieved by using an extended observation vector:

$$\tilde{\mathbf{x}} = \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix} \quad \text{instead of} \quad \mathbf{x}$$