technische universität
dortmund

Computational Intelligence

Winter Term 2009/10

Prof. Dr. Günter Rudolph

Lehrstuhl für Algorithm Engineering (LS 11)

Fakultät für Informatik

TU Dortmund

- Single-Layer Perceptron
  - Accelerated Learning
  - Online- vs. Batch-Learning
- Multi-Layer-Perceptron
  - Model
  - Backpropagation

Acceleration of Perceptron Learning

Assumption: $x \in \{ 0, 1 \}^n \Rightarrow ||x|| \geq 1$ für alle $x \neq (0, ..., 0)'$

If classification incorrect, then $w'x < 0$.

Consequently, size of error is just $\delta = -w'x > 0$.

$\Rightarrow w_{t+1} = w_t + (\delta + \varepsilon)\, x$ for $\varepsilon > 0$ (small) corrects error in a *single* step, since

$$
\begin{aligned}
w'_{t+1}x &= (w_t + (\delta + \varepsilon)\, x)'\, x \\
&= \underbrace{w'_t\, x} + (\delta + \varepsilon)\, x'x \\
&= -\delta + \delta\, ||x||^2 + \varepsilon\, ||x||^2 \\
&= \underbrace{\delta\, (||x||^2 - 1)}_{\geq 0} + \underbrace{\varepsilon\, ||x||^2}_{> 0} > 0 \qquad ☑
\end{aligned}
$$

Generalization:

Assumption: $x \in \mathbb{R}^n$ $\Rightarrow ||x|| > 0$ für alle $x \neq (0, ..., 0)'$

as before: $w_{t+1} = w_t + (\delta + \varepsilon)\, x$ for $\varepsilon > 0$ (small) and $\delta = - w'_t\, x > 0$

$$\Rightarrow \quad w'_{t+1}x = \delta \underbrace{(||x||^2 - 1)}_{< 0 \text{ possible!}} + \varepsilon \underbrace{||x||^2}_{> 0}$$

Idea: Scaling of data does not alter classification task!

Let $\ell = \min \{ ||x|| : x \in B \} > 0$

Set $\hat{x} = \frac{x}{\ell}$ $\Rightarrow$ set of scaled examples $\hat{B}$

$\Rightarrow || \hat{x} || \geq 1 \quad \Rightarrow \quad || \hat{x} ||^2 - 1 \geq 0 \quad \Rightarrow \quad w'_{t+1}\, \hat{x} > 0$ ☑

There exist numerous variants of Perceptron Learning Methods.

Theorem: (Duda & Hart 1973)

If rule for correcting weights is $w_{t+1} = w_t + \gamma_t\, x$ (if $w'_t\, x < 0$)

1. $\forall\, t \geq 0 : \gamma_t \geq 0$
2. $\displaystyle\sum_{t=0}^{\infty} \gamma_t = \infty$
3. $\displaystyle\lim_{m \to \infty} \frac{\sum_{t=0}^{m} \gamma_t^2}{\left(\sum_{t=0}^{m} \gamma_t\right)^2} = 0$

then $w_t \to w^*$ for $t \to \infty$ with $\forall\, x'w^* > 0$. ∎

e.g.: $\gamma_t = \gamma > 0$ or $\gamma_t = \gamma\, /\, (t+1)$ for $\gamma > 0$

as yet: *Online Learning*

→ Update of weights after each training pattern (if necessary)

now: *Batch Learning*

→ Update of weights only after test of all training patterns

→ Update rule:

$$w_{t+1} = w_t + \gamma \sum_{\substack{w'_t x < 0 \\ x \in B}} x \qquad (\gamma > 0)$$

vague assessment in literature:

- advantage : „usually faster“
- disadvantage : „needs more memory“ ← just a single vector!

find weights by means of optimization

Let $F(w) = \{ x \in B : w'x < 0 \}$ be the set of patterns incorrectly classified by weight w.

Objective function: $$f(w) = -\sum_{x \in F(w)} w'x \rightarrow \min!$$

Optimum: $f(w) = 0$ iff $F(w)$ is empty

Possible approach: *gradient method*

$$\boxed{w_{t+1} = w_t - \gamma \nabla f(w_t) \qquad (\gamma > 0)}$$

converges to a local minimum (dep. on w_0)

Gradient method

$w_{t+1} = w_t - \gamma \nabla f(w_t)$

Gradient points in direction of steepest ascent of function f(·)

Gradient $$\nabla f(w) = \left(\frac{\partial f(w)}{\partial w_1}, \frac{\partial f(w)}{\partial w_2}, \ldots, \frac{\partial f(w)}{\partial w_n} \right)$$

Caution:
Indices i of w_i here denote components of vektor w; they are **not** the iteration counters!

$$\frac{\partial f(w)}{\partial w_i} = -\frac{\partial}{\partial w_i} \sum_{x \in F(w)} w'x = -\frac{\partial}{\partial w_i} \sum_{x \in F(w)} \sum_{j=1}^{n} w_j \cdot x_j$$

$$= -\sum_{x \in F(w)} \underbrace{\frac{\partial}{\partial w_i} \left(\sum_{j=1}^{n} w_j \cdot x_j \right)}_{x_i} = -\sum_{x \in F(w)} x_i$$

Gradient method

thus:

gradient $$\nabla f(w) = \left(\frac{\partial f(w)}{\partial w_1}, \frac{\partial f(w)}{\partial w_2}, \ldots, \frac{\partial f(w)}{\partial w_n} \right)'$$

$$= \left(-\sum_{x \in F(w)} x_1, -\sum_{x \in F(w)} x_2, \ldots, -\sum_{x \in F(w)} x_n \right)'$$

$$= -\sum_{x \in F(w)} x$$

$$\Rightarrow \quad w_{t+1} = w_t + \gamma \sum_{x \in F(w_t)} x$$

gradient method ⇔ batch learning

How difficult is it

(a) to find a separating hyperplane, provided it exists?

(b) to decide, that there is no separating hyperplane?

Let $B = P \cup \{ -x : x \in N \}$ (only positive examples), $w_i \in \mathbb{R}$, $\theta \in \mathbb{R}$, $|B| = m$

For every example $x_i \in B$ should hold:

$x_{i1} w_1 + x_{i2} w_2 + ... + x_{in} w_n \geq \theta$ $\rightarrow$ trivial solution $w_i = \theta = 0$ to be excluded!

Therefore additionally: $\eta \in \mathbb{R}$

$x_{i1} w_1 + x_{i2} w_2 + ... + x_{in} w_n - \theta - \eta \geq 0$

Idea: η maximize $\rightarrow$ if $\eta^* > 0$, then solution found

Matrix notation:

$$A = \begin{pmatrix} x_1' & -1 & -1 \\ x_2' & -1 & -1 \\ \vdots & \vdots & \vdots \\ x_m' & -1 & -1 \end{pmatrix} \qquad z = \begin{pmatrix} w \\ \theta \\ \eta \end{pmatrix}$$

Linear Programming Problem:

$$\left.\begin{array}{l} f(z_1, z_2, \ldots, z_n, z_{n+1}, z_{n+2}) = z_{n+2} \quad \to \quad \text{max!} \\ \\ \text{s.t.} \quad Az \geq 0 \end{array}\right\}$$

calculated by e.g. Kamarkar-algorithm in **polynomial time**

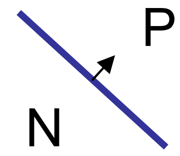
If $z_{n+2} = \eta > 0$, then weights and threshold are given by z.

Otherwise separating hyperplane does not exist!

What can be achieved by adding a layer?

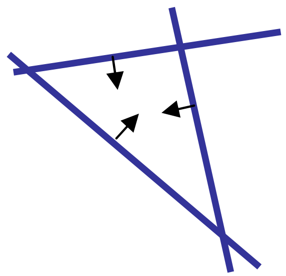
- Single-layer perceptron (SLP)

  ⇒ Hyperplane separates space in two subspaces



- Two-layer perceptron

  ⇒ arbitrary convex sets can be separated



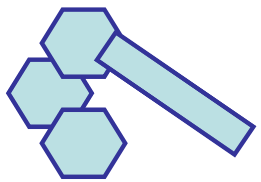
connected by AND gate in 2nd layer

- Three-layer perceptron

  ⇒ arbitrary sets can be separated (depends on number of neurons)-

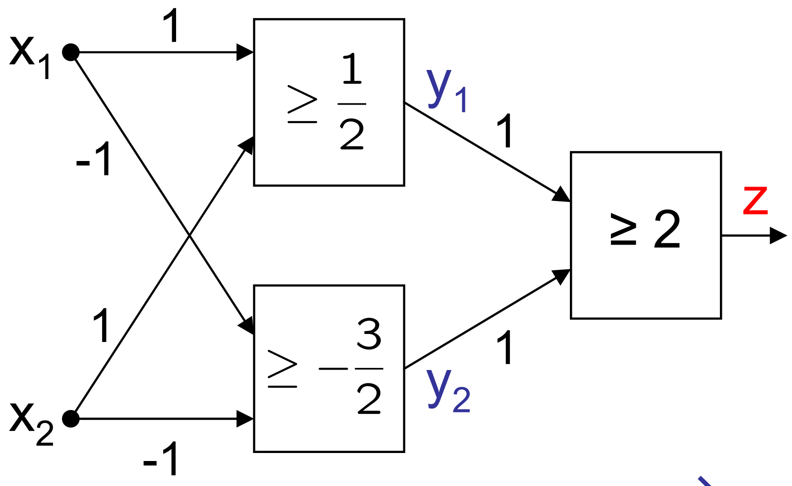
  several convex sets representable by 2nd layer,

  these sets can be combined in 3rd layer

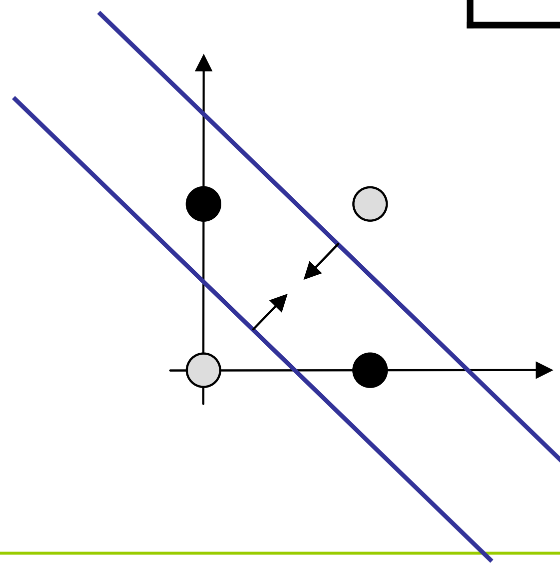


⇒ more than 3 layers not necessary!

XOR with 3 neurons in 2 steps

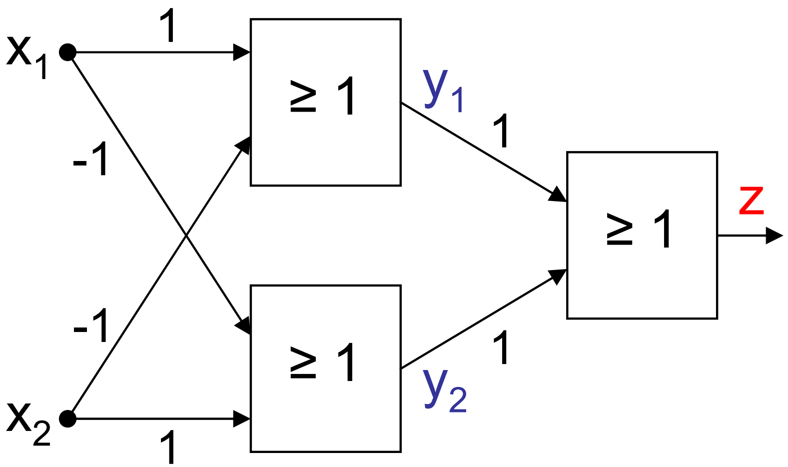


x_1	x_2	y_1	y_2	z
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0



convex set

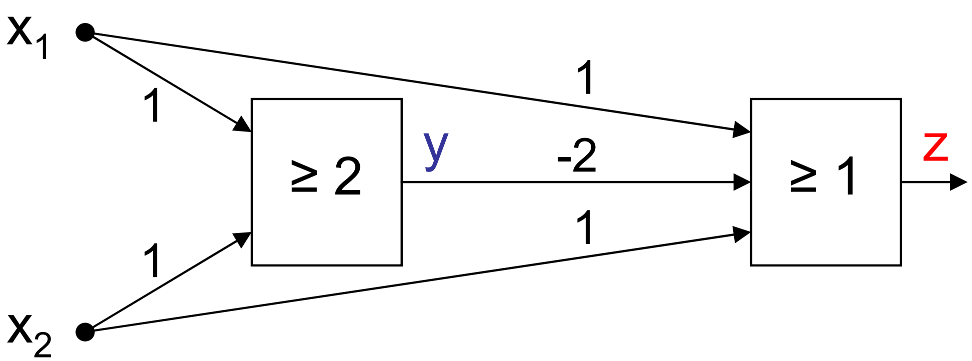
XOR with 3 neurons in 2 layers



x_1	x_2	y_1	y_2	z
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	0	0	0

without AND gate in 2nd layer

XOR mit 2 Neuronen möglich



x_1	x_2	y	$-2y$	x_1-2y+x_2	z
0	0	0	0	0	0
0	1	0	0	1	1
1	0	0	0	1	1
1	1	1	-2	0	0

BUT: this is not a layered network (no MLP) !

Evidently:

MLPs deployable for addressing significantly more difficult problems than SLPs!

But:

How can we adjust all these weights and thresholds?

Is there an efficient learning algorithm for MLPs?

History:

Unavailability of efficient learning algorithm for MLPs was a brake shoe ...

... until **Rumelhart, Hinton** and **Williams (1986): Backpropagation**

Actually proposed by **Werbos (1974)**

... but unknown to ANN researchers (was PhD thesis)

Quantification of classification error of MLP

- Total Sum Squared Error (TSSE)

$$f(w) = \sum_{x \in B} \| \underbrace{g(w;x)}_{\substack{\text{output of net} \\ \text{for weights w and input x}}} - \underbrace{g^*(x)}_{\substack{\text{target output of net} \\ \text{for input x}}} \|^2$$

- Total Mean Squared Error (TMSE)

$$f(w) = \frac{1}{|B| \cdot \ell} \sum_{x \in B} \| g(w;x) - g^*(x) \|^2 = \underbrace{\frac{1}{|B| \cdot \ell}}_{\textit{const.}} \cdot \text{ TSSE}$$

training patters

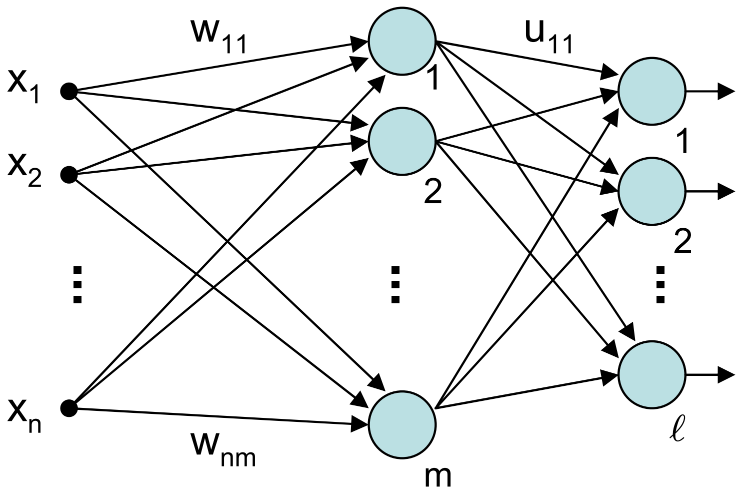
output neurons

⇒ leads to same solution as TSSE

Learning algorithms for Multi-Layer-Perceptron (here: 2 layers)

idea: minimize error!

$f(w_t, u_t) = \text{TSSE} \quad \to \quad \text{min!}$



Gradient method

$$u_{t+1} = u_t - \gamma \nabla_u f(w_t, u_t)$$

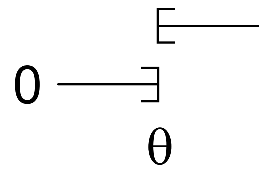
$$w_{t+1} = w_t - \gamma \nabla_w f(w_t, u_t)$$

$$a(x) = \begin{cases} 1 & \text{if } x > \theta \\ 0 & \text{otherwise} \end{cases}$$

BUT:

f(w, u) cannot be differentiated!

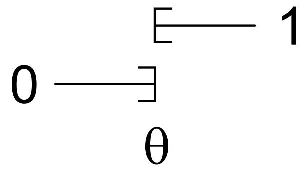
Why? → Discontinuous activation function a(.) in neuron!

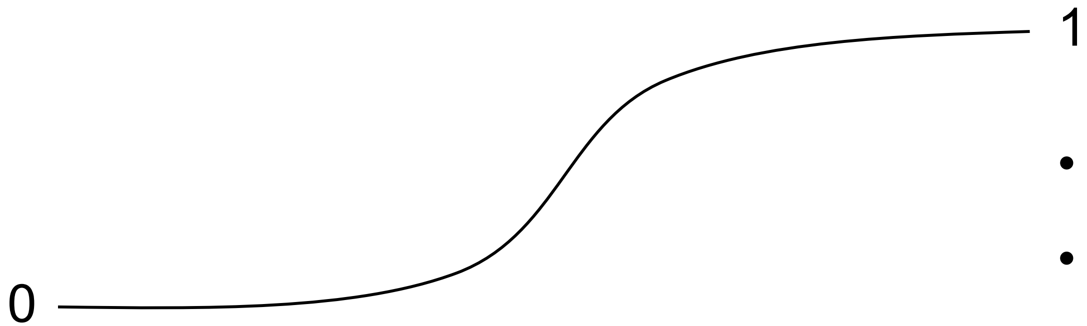


idea: find **smooth** activation function similar to original function !

Learning algorithms for Multi-Layer-Perceptron (here: 2 layers)

<u>good idea:</u> sigmoid activation function (instead of signum function)





- monotone increasing
- differentiable
- non-linear
- output $\in$ [0,1] instead of $\in$ { 0, 1 }
- threshold θ integrated in activation function

e.g.:

- $a(x) = \dfrac{1}{1+e^{-x}}$ $\qquad a'(x) = a(x)(1 - a(x))$
- $a(x) = \tanh(x)$ $\qquad a'(x) = (1 - a^2(x))$

values of derivatives directly determinable from function values

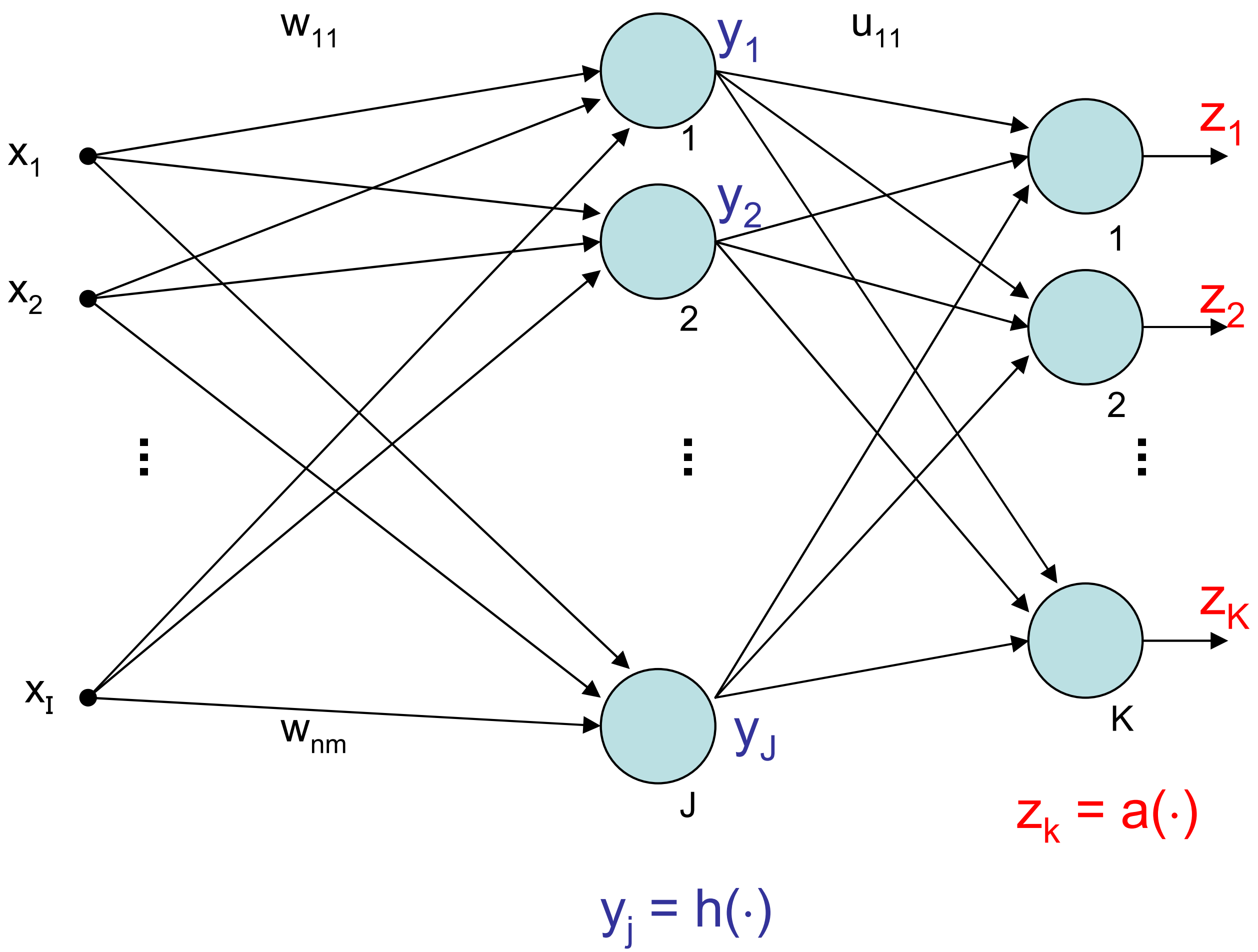
Learning algorithms for Multi-Layer-Perceptron (here: 2 layers)

<ins>Gradient method</ins>

$f(w_t, u_t) = \text{TSSE}$

$u_{t+1} = u_t - \gamma \nabla_u f(w_t, u_t)$

$w_{t+1} = w_t - \gamma \nabla_w f(w_t, u_t)$



x_i : inputs

y_j : values after first layer

z_k: values after second layer

$$y_j = h\left(\sum_{i=1}^{I} w_{ij} \cdot x_i\right) = h(w_j' \, x)$$

output of neuron j after 1st layer

$$z_k = a\left(\sum_{j=1}^{J} u_{jk} \cdot y_j\right) = a(u_k' \, y)$$

output of neuron k after 2nd layer

$$= a\left(\sum_{j=1}^{J} u_{jk} \cdot h\left(\sum_{i=1}^{I} w_{ij} \cdot x_i\right)\right)$$

error of input x:

$$f(w, u; x) = \sum_{k=1}^{K} (\underbrace{z_k(x)}_{\text{output of net}} - \underbrace{z_k^*(x)}_{\text{target output for input x}})^2 = \sum_{k=1}^{K} (z_k - z_k^*)^2$$

error for input x and target output z*:

$$f(w,u;x,z^*) = \sum_{k=1}^{K} \Bigg[\underbrace{a\left(\sum_{j=1}^{J} u_{jk} \cdot \underbrace{h\left(\overbrace{\sum_{i=1}^{I} w_{ij} \cdot x_i}^{w_j'x}\right)}_{y_j}\right)}_{z_k} - z_k^*(x) \Bigg]^2$$

total error for all training patterns (x, z*) ∈ B:

$$f(w,u) = \sum_{(x,z^*)\in B} f(w,u;x,z^*) \qquad \text{(TSSE)}$$

gradient of total error:

$$\nabla f(w,u) = \sum_{(x,z^*)\in B} \nabla f(w,u;x,z^*)$$

vector of partial derivatives w.r.t. weights u_{jk} and w_{ij}

thus:

$$\frac{\partial f(w,u)}{\partial u_{jk}} = \sum_{(x,z^*)\in B} \frac{\partial f(w,u;x,z^*)}{\partial u_{jk}}$$

and

$$\frac{\partial f(w,u)}{\partial w_{ij}} = \sum_{(x,z^*)\in B} \frac{\partial f(w,u;x,z^*)}{\partial w_{ij}}$$

assume: $$a(x) = \frac{1}{1+e^{-x}} \quad \Rightarrow \quad \frac{d\,a(x)}{dx} = a'(x) = a(x)\cdot(1-a(x))$$

and: $$h(x) = a(x)$$

chain rule of differential calculus:

$$[\,p(q(x))\,]' = \underbrace{p'(q(x))}_{\text{outer derivative}} \cdot \underbrace{q'(x)}_{\text{inner derivative}}$$

$$f(w, u; x, z^*) = \sum_{k=1}^{K} [\, a(u_k' y) - z_k^* \,]^2$$

partial derivative w.r.t. u_{jk}:

$$\frac{\partial f(w, u; x, z^*)}{\partial u_{jk}} = 2\,[\, a(u_k' y) - z_k^* \,] \cdot a'(u_k' y) \cdot y_j$$

$$= 2\,[\, a(u_k' y) - z_k^* \,] \cdot a(u_k' y) \cdot (1 - a(u_k' y)) \cdot y_j$$

$$= \underbrace{2\,[\, z_k - z_k^* \,] \cdot z_k \cdot (1 - z_k)}_{\text{“error signal“ } \delta_k} \cdot y_j$$

partial derivative w.r.t. w_{ij}:

$$\frac{\partial f(w,u;x,z^*)}{\partial w_{ij}} = 2\sum_{k=1}^{K}[\underbrace{a(u_k'y)}_{z_k} - z_k^*]\cdot \underbrace{a'(u_k'y)}_{z_k\,(1-z_k)}\cdot u_{jk}\cdot \underbrace{h'(w_j'x)}_{y_j\,(1-y_j)}\cdot x_i$$

$$= 2\cdot\sum_{k=1}^{K}[z_k - z_k^*]\cdot z_k\cdot(1-z_k)\cdot u_{jk}\cdot y_j\,(1-y_j)\cdot x_i$$

factors reordered

$$= x_i\cdot \underbrace{y_j\cdot(1-y_j)\cdot\sum_{k=1}^{K}\underbrace{2\cdot[z_k - z_k^*]\cdot z_k\cdot(1-z_k)\cdot u_{jk}}_{\text{error signal } \delta_k \text{ from previous layer}}}_{\text{error signal } \delta_j \text{ from “current“ layer}}$$

Generalization (> 2 layers)

$$\left.\begin{array}{l}\text{Let neural network have L layers } S_1, S_2, \ldots S_L. \\ \text{Let neurons of all layers be numbered from 1 to N.}\end{array}\right\} \begin{array}{l} j \in S_m \rightarrow \\ \text{neuron j is in} \\ \text{m-th layer}\end{array}$$

All weights w_{ij} are gathered in weights matrix W.

Let o_j be output of neuron j.

error signal:

$$\delta_j = \begin{cases} o_j \cdot (1 - o_j) \cdot (o_j - z_j^*) & \text{if } j \in S_L \text{ (output neuron)} \\ o_j \cdot (1 - o_j) \cdot \sum\limits_{k \in S_{m+1}} \delta_k \cdot w_{jk} & \text{if } j \in S_m \text{ and } m < L \end{cases}$$

correction:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \gamma \cdot o_i \cdot \delta_j$$

<u>in case of online learning</u>:
correction after **each** test pattern presented

error signal of neuron in inner layer determined by

- error signals of all neurons of subsequent layer and
- weights of associated connections.

$\Downarrow$

- First determine error signals of output neurons,
- use these error signals to calculate the error signals of the preceding layer,
- use these error signals to calculate the error signals of the preceding layer,
- and so forth until reaching the first inner layer.

$\Downarrow$

thus, error is propagated backwards from output layer to first inner
$\Rightarrow$ **backpropagation** (of error)

⇒ other optimization algorithms deployable!

in addition to **backpropagation** (gradient descent) also:

- **Backpropagation with Momentum**
  take into account also previous change of weights:

$$\Delta w_{ij}^{(t)} = -\gamma_1 \cdot o_i \cdot \delta_j - \gamma_2 \cdot \Delta w_{ij}^{(t-1)}$$

- **QuickProp**
  assumption: error function can be approximated locally by quadratic function,
  update rule uses last two weights at step t – 1 and t – 2.

- **Resilient Propagation (RPROP)**
  exploits sign of partial derivatives:
  2 times negative or positive ⇒ increase step!
  change of sign ⇒ reset last step and decrease step!
  typical values: factor for decreasing 0,5 / factor of increasing 1,2

- **evolutionary algorithms**
  individual = weights matrix

later more about this!