

Computational Intelligence

Winter Term 2025/26

Prof. Dr. Günter Rudolph

Computational Intelligence

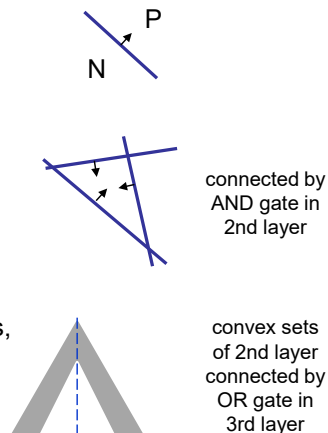
Fakultät für Informatik

TU Dortmund

- Multi-Layer-Perceptron
 - Model
 - Backpropagation
- Typical Fields of Application
 - Classification
 - Prediction
 - Function Approximation

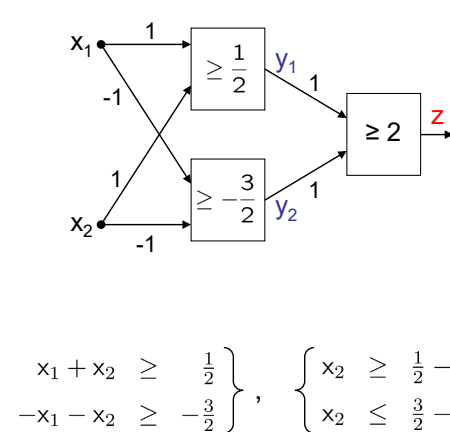
What can be achieved by adding a layer?

- Single-layer perceptron (SLP)
 - ⇒ Hyperplane separates space in two subspaces
- Two-layer perceptron
 - ⇒ arbitrary convex sets can be separated
- Three-layer perceptron
 - ⇒ arbitrary sets can be partitioned into convex subsets, convex subsets representable by 2nd layer, resulting sets can be combined in 3rd layer

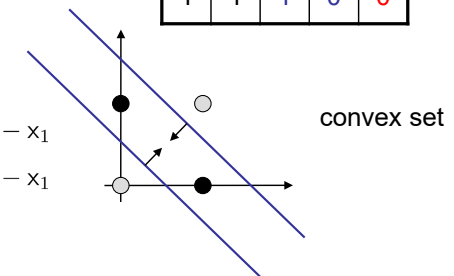


⇒ more than 3 layers not necessary (in principle)

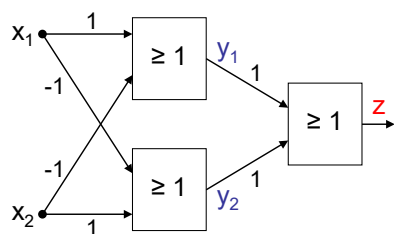
XOR with 3 neurons in 2 steps



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



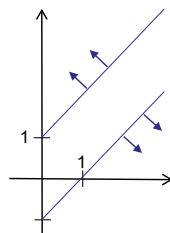
XOR with 3 neurons in 2 layers



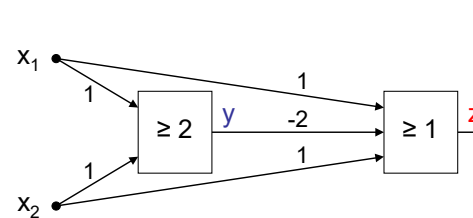
x_1	x_2	y_1	y_2	z
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	0	0	0

without AND gate in 2nd layer

$$\left. \begin{aligned} x_1 - x_2 &\geq 1 \\ x_2 - x_1 &\geq 1 \end{aligned} \right\}, \quad \left\{ \begin{aligned} x_2 &\leq x_1 - 1 \\ x_2 &\geq x_1 + 1 \end{aligned} \right.$$



XOR can be realized with only 2 neurons!



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)

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

- Total Mean Squared Error (TMSE)

$$f(w) = \frac{1}{|B| \cdot \ell} \sum_{x \in B} \|g(w; x) - g^*(x)\|^2 = \underbrace{\frac{1}{|B| \cdot \ell}}_{\substack{\text{const.} \\ \Rightarrow \text{leads to same} \\ \text{solution as TSSE}}} \cdot \text{TSSE}$$

$\nearrow$ # training patters $\nearrow$ # output neurons

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

idea: minimize error!

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

Gradient method

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

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

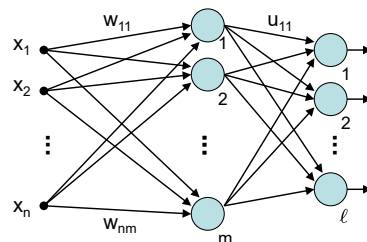
BUT:

$f(w, u)$ cannot be differentiated!

Why? $\rightarrow$ Discontinuous activation function $a(\cdot)$ in neuron!

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

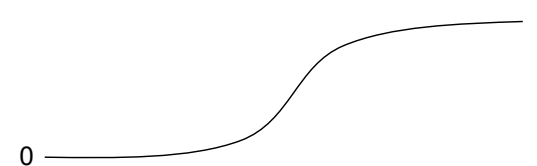
idea: find **smooth** activation function similar to original function !



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

good idea: sigmoid activation function (instead of signum function)

$$0 \begin{cases} \text{---} 1 \\ \text{---} \theta \end{cases}$$



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

e.g.:

$$\begin{aligned} \bullet \quad a(x) &= \frac{1}{1 + e^{-x}} & a'(x) &= a(x)(1 - a(x)) \\ \bullet \quad a(x) &= \tanh(x) & a'(x) &= (1 - a^2(x)) \end{aligned} \quad \left. \vphantom{\begin{aligned} \bullet \quad a(x) &= \frac{1}{1 + e^{-x}} \\ \bullet \quad a(x) &= \tanh(x) \end{aligned}} \right\} \begin{array}{l} \text{values of derivatives directly} \\ \text{determinable from function} \\ \text{values} \end{array}$$

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

Gradient method

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

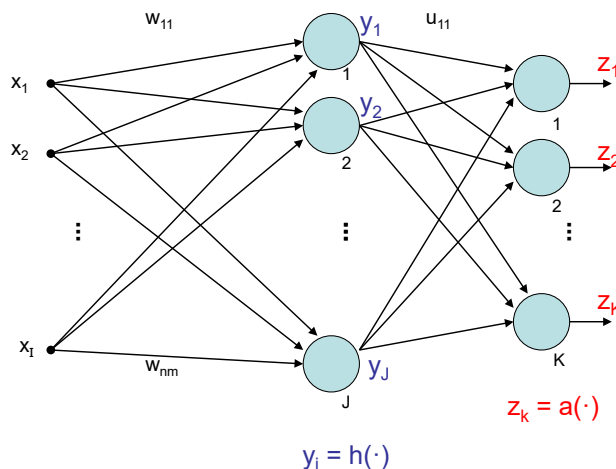
$$u_{t+1} = u_t - \gamma \nabla_u f(w_t, u_t)$$

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

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

$$\begin{aligned} z_k &= a\left(\sum_{j=1}^J u_{jk} \cdot y_j\right) = a(u'_k y) \\ &= a\left(\sum_{j=1}^J u_{jk} \cdot h\left(\sum_{i=1}^I w_{ij} \cdot x_i\right)\right) \end{aligned}$$

output of neuron k
after 2nd layer

error of input x :

$$f(w, u; x) = \sum_{k=1}^K (z_k(x) - z_k^*(x))^2 = \sum_{k=1}^K (z_k - z_k^*)^2$$

$\uparrow$ output of net $\uparrow$ target output for input x

error for input x and target output z^* :

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

total error for all training patterns $(x, z^*) \in B$:

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

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

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} :

$$\begin{aligned} \frac{\partial f(w, u; x, z^*)}{\partial u_{jk}} &= 2 [a(u'_k y) - z_k^*] \cdot a'(u'_k y) \cdot y_j \\ &= 2 [a(u'_k y) - z_k^*] \cdot a(u'_k y) \cdot (1 - a(u'_k y)) \cdot y_j \\ &= 2 \underbrace{[z_k - z_k^*] \cdot z_k \cdot (1 - z_k)}_{\text{"error signal" } \delta_k} \cdot y_j \end{aligned}$$

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

$$\begin{aligned} \frac{\partial f(w, u; x, z^*)}{\partial w_{ij}} &= 2 \sum_{k=1}^K \underbrace{[a(u'_k y) - z_k^*]}_{z_k} \cdot \underbrace{a'(u'_k y)}_{z_k(1-z_k)} \cdot \underbrace{u_{jk}}_{y_j(1-y_j)} \cdot h'(w'_j x) \cdot x_i \\ &= 2 \cdot \sum_{k=1}^K [z_k - z_k^*] \cdot z_k \cdot (1 - z_k) \cdot u_{jk} \cdot y_j \cdot (1 - y_j) \cdot x_i \\ &\xrightarrow{\text{factors reordered}} = x_i \cdot y_j \cdot (1 - y_j) \cdot \underbrace{\sum_{k=1}^K 2 \cdot [z_k - z_k^*] \cdot z_k \cdot (1 - z_k) \cdot u_{jk}}_{\text{error signal } \delta_k \text{ from previous layer}} \\ &\quad \underbrace{\phantom{\sum_{k=1}^K 2 \cdot [z_k - z_k^*] \cdot z_k \cdot (1 - z_k) \cdot u_{jk}}}_{\text{error signal } \delta_j \text{ from "current" layer}} \end{aligned}$$

Generalization (> 2 layers)

Let neural network have L layers $S_1, S_2, \dots, S_L$.

Let neurons of all layers be numbered from 1 to N.

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

Let o_j be output of neuron j.

$j \in S_m \rightarrow$
neuron j is in
m-th layer

error signal:

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

correction:

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

in case of online learning:
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.

⇓

- 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.

⇓

thus, error is propagated backwards from output layer to first inner layer
⇒ **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 size!

change of sign → reset last step and decrease step size!

typical values: factor for decreasing 0,5 / factor for increasing 1,2

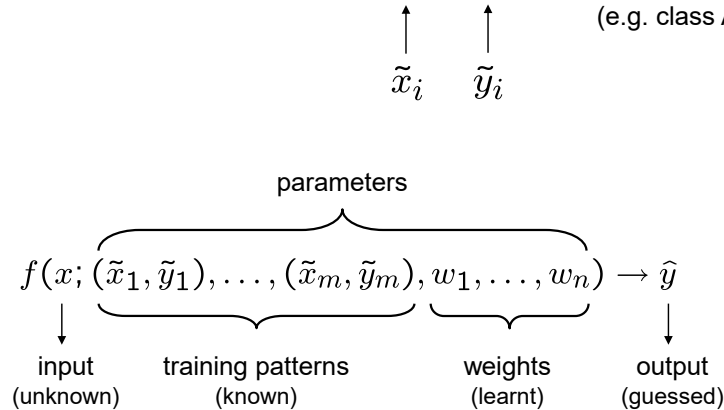
• **Evolutionary Algorithms**

individual = weights matrix

Classification

given: set of training patterns (input / output)

output = label
(e.g. class A, class B, ...)

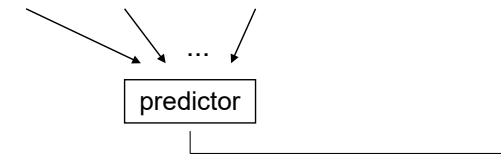


Prediction of Time Series

time series $x_1, x_2, x_3, \dots$ (e.g. temperatures, exchange rates, ...)

task: given a subset of historical data, predict the future

$$f(x_{t-k}, x_{t-k+1}, \dots, x_t; w_1, \dots, w_n) \rightarrow \hat{x}_{t+\tau}$$



training patterns:

historical data where true output is known;

$$\text{error per pattern} = (\hat{x}_{t+\tau} - x_{t+\tau})^2$$

phase I:

train network

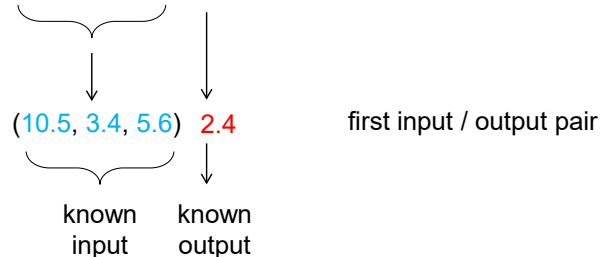
phase II:

apply network to historical inputs for predicting unknown outputs

Prediction of Time Series: Example for Creating Training Data

given: time series 10.5, 3.4, 5.6, 2.4, 5.9, 8.4, 3.9, 4.4, 1.7

time window: $k=3$

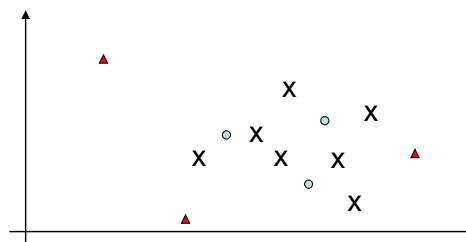


Function Approximation (the general case)

task: given training patterns (input / output), approximate unknown function

→ should give outputs close to true unknown function for arbitrary inputs

- values between training patterns are **interpolated**
- values outside convex hull of training patterns are **extrapolated**



x : input training pattern

$\circ$: input pattern where output to be interpolated

$\blacktriangle$: input pattern where output to be extrapolated