technische universität dortmund

Computational Intelligence

Winter Term 2021/22

Prof. Dr. Günter Rudolph

Lehrstuhl für Algorithm Engineering (LS 11)

Fakultät für Informatik

TU Dortmund

Plan for Today

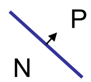
- Multi-Layer-Perceptron
  - Model
  - Backpropagation

Multi-Layer Perceptron (MLP)

What can be achieved by adding a layer?

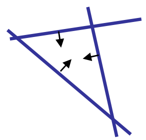
- Single-layer perceptron (SLP)

  ⇒ Hyperplane separates space in two subspaces



- Two-layer perceptron

  ⇒ arbitrary convex sets can be separated



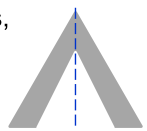
connected by AND gate in 2nd layer

- Three-layer perceptron

  ⇒ arbitrary sets can be partitioned into convex subsets,

  convex subsets representable by 2nd layer,

  resulting sets can be combined in 3rd layer

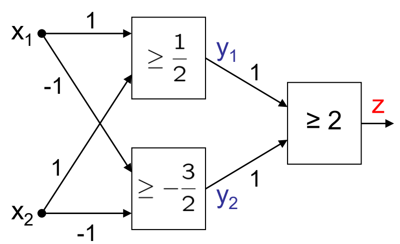


convex sets of 2nd layer connected by OR gate in 3rd layer

⇒ more than 3 layers not necessary (in principle)

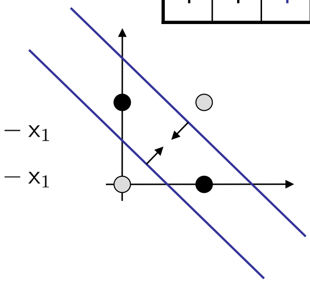
Multi-Layer Perceptron (MLP)

XOR with 3 neurons in 2 steps



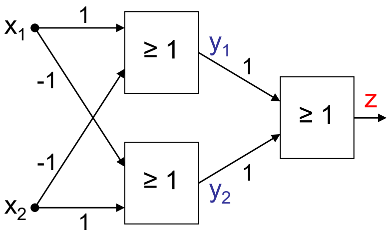
x_1	x_2	y_1	y_2	z
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

$$\left.\begin{array}{rcl} x_1 + x_2 & \geq & \frac{1}{2} \\ -x_1 - x_2 & \geq & -\frac{3}{2} \end{array}\right\}, \quad \left\{\begin{array}{rcl} x_2 & \geq & \frac{1}{2} - x_1 \\ x_2 & \leq & \frac{3}{2} - x_1 \end{array}\right.$$



convex set

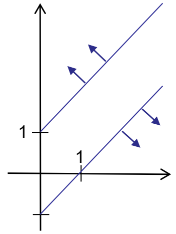
XOR with 3 neurons in 2 layers



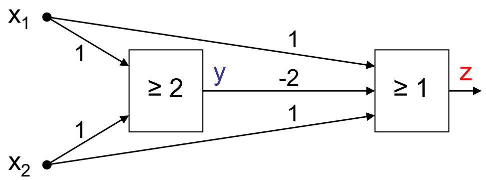
x_1	x_2	y_1	y_2	z
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	0	0	0

without AND gate in 2nd layer

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



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 <u>layered</u> 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)

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

training patters   # output neurons

⇒ leads to same solution as TSSE

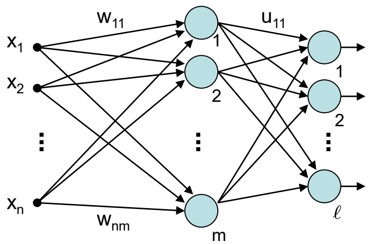
Learning algorithms for Multi-Layer-Perceptron (here: 2 layers)

idea: minimize error!

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

Gradient method

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

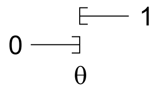


$$a(x) = \begin{cases} 1 & \text{if } x > \theta \\ 0 & \text{otherwise} \end{cases}$$

BUT:

f(w, u) cannot be differentiated!

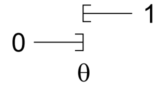
Why? → Discontinuous activation function a(.) in neuron!

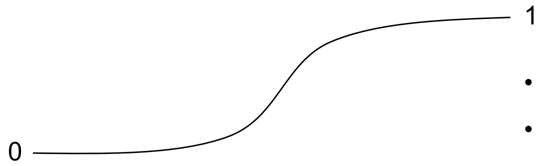


idea: find **smooth** activation function similar to original function !

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

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





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

e.g.:

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

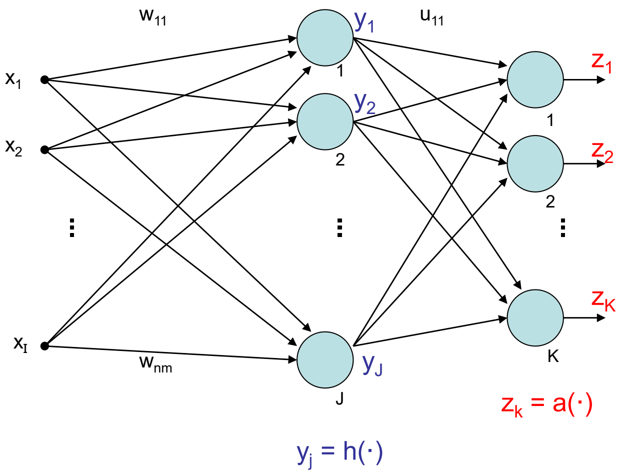
values of derivatives directly determinable from function values

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

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

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

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:

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

output of net — target output for input x

error for input x and target output z*:

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

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

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}:

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

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

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

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

factors reordered

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

Generalization (> 2 layers)

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

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

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

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:

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

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.

$\Downarrow$

- 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)

$\Rightarrow$ 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, ...)

$\tilde{x}_i \qquad \tilde{y}_i$

$$f(x; \overbrace{\underbrace{(\tilde{x}_1,\tilde{y}_1),\ldots,(\tilde{x}_m,\tilde{y}_m)}_{\text{training patterns (known)}}, \underbrace{w_1,\ldots,w_n}_{\text{weights (learnt)}}}^{\text{parameters}}) \to \hat{y}$$

input (unknown) — output (guessed)

phase I:

train network

phase II:

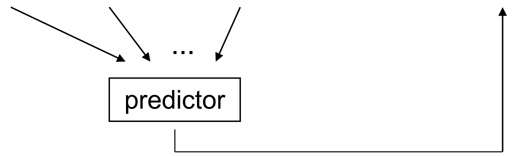
apply network to unkown inputs for classification

Prediction of Time Series

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

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

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



training patterns:

historical data where true output is known;

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

phase I:

train network

phase II:

apply network to historical inputs for predicting <u>unkown</u> outputs

Prediction of Time Series: <u>Example for Creating Training Data</u>

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

$\underbrace{(10.5, 3.4, 5.6)}_{\text{known input}}$ 2.4 (known output) first input / output pair

further input / output pairs:

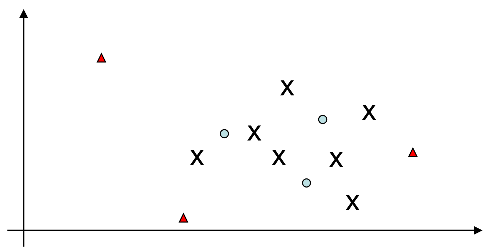
(3.4, 5.6, 2.4) 5.9
(5.6, 2.4, 5.9) 8.4
(2.4, 5.9, 8.4) 3.9
(5.9, 8.4, 3.9) 4.4
(8.4, 3.9, 4.4) 1.7

Function Approximation (the general case)

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

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

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



x : input training pattern

° : input pattern where output to be interpolated

▲ : input pattern where output to be extrapolated